EVOLVING ARTIFICIAL NEURAL NETWORKS WITH GENERATIVE ENCODINGS INSPIRED BY DEVELOPMENTAL BIOLOGY

By

Jeff Clune

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science

2010

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Evolving Artificial Neural Networks with Generative Encodings Inspired by Developmental Biology	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**123**	

ABSTRACT

EVOLVING ARTIFICIAL NEURAL NETWORKS WITH GENERATIVE ENCODINGS INSPIRED BY DEVELOPMENTAL BIOLOGY

By Jeff Clune

In this dissertation I investigate the difference between generative encodings and direct encodings for evolutionary algorithms. Generative encodings are inspired by developmental biology and were designed, in part, to increase the regularity of synthetically evolved phenotypes. Regularity is an important design principle in both natural organisms and engineered designs. The majority of this dissertation focuses on how the property of regularity enables a generative encoding to outperform direct encoding controls, and whether a bias towards regularity also hurts the performance of the generative encoding on some problems. I also report on whether researchers can bias the *types* of regularities produced by a generative encoding to accommodate user preferences. Finally, I study the degree to which a generative encoding produces another important design principle, modularity.

Several previous studies have shown that generative encodings outperform direct encodings on highly regular problems. However, prior to this dissertation, it was not known how generative encodings compare to direct encodings on problems with different levels of regularity. On three different problems, I show that a generative encoding can exploit intermediate amounts of problem regularity, which enabled the generative encoding to increasingly outperform direct encoding controls as problem regularity increased. This performance gap emerged because the generative encoding produced regular artificial neural networks (ANNs) that produced regular behaviors. The ANNs evolved with the generative encoding contained a diverse array of complicated, regular neural wiring patterns, whereas the ANNs produced by a direct encoding control were irregular.

I also document that the bias towards regularity can hurt a generative encoding on problems that have some amount of irregularity. I propose a new algorithm, called HybrID, wherein a generative encoding produces regular patterns and a direct encoding modifies

those patterns to provide fitness-enhancing irregularities. HybrID outperformed a generative encoding alone on three problems for nearly all levels of regularity, which raises the question of whether generative encodings may ultimately excel not as stand-alone algorithms, but by being hybridized with a further process of irregular refinement.

The results described so far document that a generative encoding can produce *regular* solutions. I then show that, at least for the generative encoding in this case study, it is possible to influence the types of regularities produced, which allows domain knowledge and preferences to be injected into the algorithm. I also investigated whether the generative encoding can produce *modular* solutions. I present the first documented case of this generative encoding producing a modular phenotype on a simple problem. However, the generative encoding's inability to create modularity on harder problems where modularity would have been beneficial suggests that more work is needed to increase the likelihood that this encoding produces modular ANNs in response to challenging, decomposable problems.

Overall, this dissertation paints a more complete picture of generative encodings than prior studies. Initially, it demonstrates that, by producing regular ANNs and behaviors, generative encodings increasingly outcompete direct encodings as problem regularity increases. It next documents that a bias towards regularity can harm the performance of direct encodings when problems contain irregularities. The HybrID algorithm suggests a path forward, however, by revealing that a refinement process that fine-tunes the regular patterns produced by a generative encoding can boost performance by accounting for problem irregularities. Finally, the dissertation shows that the generative encoding studied can produce modular networks on simple problems, but may struggle to do so on harder problems. The general conclusion that can be drawn from this work is that generative encodings can produce some of the properties seen in complex, natural organisms, and will likely be an important part of our long-term goal of synthetically evolving phenotypes that approach the capability, intelligence, and complexity of their natural rivals.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CCF-0643952, CCF-0750787, CNS-0751155, CCF-0820220, CCF-0523449, and CNS-0915885, the Templeton Foundation, U.S. Army Grant W911NF-08-1-0495, a Quality Fund Grant from MSU, and the DARPA FunBio program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Images in this dissertation are presented in color.

Chapter 1

Overview

A long-term goal of human engineering is to produce artifacts as complex and intelligent as the bodies and brains in the natural world. The field of Evolutionary Computation (EC) studies how natural evolution produced this complexity and implements abstractions of such principles in Evolutionary Algorithms (EAs), which are synthetic evolutionary processes that allow us to study evolution and harness its engineering capabilities. An outstanding challenge in EC is the creation of *encodings* (also called *representations*) for EAs that enable the evolution of increased levels of complexity. Encodings in EAs are the way information is stored in a genome and how that information is mapped to a phenotype. Traditional EAs use *direct encodings*, where each element in the genotype encodes an independent aspect of the phenotype. Direct encodings are limited in their ability to evolve regular and modular phenotypes because individual mutations cannot produce coordinated changes to multiple elements of a phenotype. Additionally, direct encodings do not scale well (imagine evolution searching for a genome that had to independently specify the blueprint for every cell in a blue whale). The alternative is a *generative encoding (*also called an *indirect encoding* or *developmental encoding)*, where each genomic element can influence multiple aspects of a phenotype.

Generative encodings are a relatively new area of research, and the search for the

proper way to abstract biological development is still underway. A new generative encoding was introduced in 2007 that is based on a novel abstraction of development. It is called Hypercube-based NEAT (HyperNEAT), and it incorporates a key concept from developmental biology that enables the evolution of regular brains: determining the fates of phenotypic components as a function of their geometric location. In this dissertation, I investigate the merits and costs of generative encodings using HyperNEAT as the exemplar generative encoding. I chose HyperNEAT because it is a cutting-edge generative encoding that has proven effective on a wide range of problems. It also has direct encoding controls, which is rare for generative encodings. Finally, while HyperNEAT is currently the only generative encoding to incorporate geometry, this technique requires a generative encoding and may be incorporated into many generative encodings in the future. It is therefore worthwhile to investigate the effects of this inclusion of geometric information. Overall, this dissertation contains an extensive case study comparing one generative encoding with its direct encoding controls, but I will also discuss how the results from this case study are relevant to the larger conversation about the merits and drawbacks of generative encodings versus direct encodings.

Specifically, I will show that HyperNEAT automatically exploits the regularity of problems, and increasingly outcompetes direct encoding controls as the regularity of a problem increases. I show this effect on three different problems that have tunable levels of regularity. The first two problems I specifically designed to test the exploitation of varying degrees of regularity with and without epistasis (Chapter 3). The third problem is a challenging engineering task: producing gaits for simulated quadruped robots (Chapter 4). Because most challenging problems have many different types of regularity, this ability to automatically discover and exploit regularity is a desirable attribute in an EA encoding.

I then investigate a potential downside that can result from the bias towards regularity in generative encodings: if an encoding is too biased toward producing regularities, it may be unable to create exceptions to a pattern to account for irregularities in a problem. The

results from Chapters 3 and 4 show that HyperNEAT's bias towards regularity prevents it from creating some exceptions to handle problem irregularity. In Chapter 5, I introduce a new algorithm, which is a Hybridization of Indirect and Direct encodings (HybrID), that combines the best attributes of generative and direct encodings: the generative encoding produces phenotypic regularities and the direct encoding creates necessary exceptions to those regularities. HybrID ties or outperforms HyperNEAT on a variety of regularity levels in three different problems. The success of HybrID raises the interesting question of whether the true promise of generative encodings is not as stand-alone algorithms, but in combination with a further algorithmic process that refines their regular patterns.

An interesting and novel aspect of generative encodings implemented in the style of HyperNEAT is that they can exploit the geometry of a problem because phenotypic elements are determined as a function of their geometric location. Future generative encodings may also incorporate this feature, given its effectiveness in HyperNEAT and nature. However, this technique requires the experimenter to choose a geometric representation for each problem. In Chapter 6, I describe the first extensive investigation into whether HyperNEAT is sensitive to choices of different geometric representations. I show that different geometric representations can indeed affect both the quality and kind of solutions produced by HyperNEAT. My results suggest that HyperNEAT practitioners can obtain good results even if they do not know how to geometrically represent a problem, and that further improvements are possible with a well-chosen geometric representation. My results also imply that experimenters can bias the type of solutions produced by HyperNEAT via the way a problem is represented geometrically, which enables user preferences to be injected into the algorithm. These results show HyperNEAT's sensitivity to geometry and suggest that future generative encodings that exploit geometry may behave similarly.

Chapters 3-5 document that HyperNEAT excels at producing regular Artificial Neural Networks (ANNs). In Chapter 7, I investigate whether HyperNEAT can produce modular ANNs. Modularity is important in ANNs because it can facilitate both evolvability

and learning, by making it is easier to rearrange functional subcomponents. I conducted this research on problems where I demonstrate modularity to be beneficial, and found that HyperNEAT failed to generate modular ANNs, even with modifications to it that should encourage modularity. I then performed tests on a simpler problem that requires modularity and found that HyperNEAT was able to rapidly produce modular solutions that solved the problem. I thus present the first documented case of HyperNEAT producing a modular phenotype. However, its inability to generate modularity on harder problems where modularity would have been beneficial suggests that more work is needed to increase the likelihood that HyperNEAT and similar algorithms will produce modular ANNs in response to challenging, decomposable problems.

Overall, this dissertation adds evidence to the claim that generative encodings in general, and HyperNEAT in particular, are on the path toward the goal of evolving phenotypes as complex as natural animals. However, much work remains to achieve that lofty goal. The main contribution of this dissertation is to increase our understanding of these technologies so that we may improve upon them to create the next generation of generative encodings for Evolutionary Algorithms.

Chapter 2

Background

2.1 Generative versus direct encodings

While the field of evolutionary computation has produced impressive results, the complexity of its evolved solutions pale in comparison to organisms in the natural world. One of several likely reasons for this difference is that evolutionary computation typically uses a direct encoding, where every part of the genotype encodes for a separate part of the phenotype. Given that natural organisms can contain trillions of parts (e.g. cells in the human body), a direct representation of such an organism would require a genome with at least that many separate genetic elements. We do not find such inefficient genomes in nature. An alternative is a generative encoding, where elements in a genome can be reused to produce many parts of a phenotype [47]. For example, about 25,000 genes encode the information that produces the trillions of parts that make up a human [44]. Generative encodings allow evolution to search a genotype space with far fewer dimensions than that of the final phenotype.

A further benefit of generative encodings is that the reuse of code facilitates the evolution of regular, modular, and hierarchical phenotypes, with and without variation [27]. *Modularity* is the localization of function within an encapsulated unit. In a network, mod-

ularity entails clusters of nodes with high connectivity within the cluster and low connectivity to nodes outside the cluster [28, 33]. *Regularity* refers to the compressibility of the information describing a structure, and typically involves symmetries and module repetition [33]. *Hierarchy* is the recursive composition of lower-level units [33]. Note that modularity does not require regularity, as is often assumed: the single wheel on a unicycle is a module, whereas the four wheels on a car are a regular repetition of a wheel module [33]. Without the ability to evolve phenotypes that possess regularity, modularity, and hierarchy, it may be difficult to synthetically evolve creatures as complicated as those found in nature [2, 33, 38]. A third benefit of generative encodings is that mutations in them can produce coordinated phenotypic effects (e.g., one mutation lengthening all legs proportionally). In investigations carried out so far, albeit on problems where regularity in the phenotype is advantageous, generative encodings have been found to outcompete direct encodings and produce more modular, regular and or hierarchical phenotypes, with more beneficial mutations on average than direct encoding controls [10, 15, 20, 21, 26, 40, 48]. In one experiment where tables were evolved, it is visually apparent that artifacts produced by a generative encoding are more regular, modular, and elegant than those produced by a direct encoding [24].

Broadly speaking, two types of generative encodings have attracted a lot of attention. The first type is based on rules for rewriting grammar, such as Lindenmayer Systems (L-Systems), which expand a few symbols iteratively according to a rule set [32]. A drawback to this method is that a slight change in a rule drastically affects the resultant phenotype, leading to a rugged fitness landscape. Additionally, such systems tend to be biased toward the production of overly repetitive and regular structures, since symbols tend to be rewritten deterministically (e.g., [25, 43]). The second common type of generative encoding evolves Genetic Regulatory Networks (GRNs), which typically involve simulated artificial chemistries to allow for the diffusion of proteins (e.g., [22]). Such simulation is computationally expensive, which limits the complexity of the artifacts evolved [45]. It

also requires evolution to first figure out how to produce chemical gradients that indicate where, geometrically, parts of the phenotype are situated, and then utilize such geometric information to produce phenotypes as a function of that geometry. While natural systems do just that, computational limits mean that EAs based on GRNs with chemical simulations cannot take advantage of the massive amounts of parallelism and time that enabled natural systems to produce complexity.

Generative encodings were created to enable the evolution of phenotypes that have modularity, regularity, and hierarchy, with and without variation, to the degrees seen in natural organisms. Artifacts evolved to date with generative encodings do display repeated modules or themes [26, 43], but we are still a long way off from synthetically evolving the level of complexity found in natural organisms. As such, the search for the proper abstraction of biological development continues. In the next section, I describe a recently introduced novel abstraction of development.

2.2 The HyperNEAT generative encoding

In 2007 an encoding was introduced that captures some of the power of natural encodings, but does not require the physical simulation of diffusing chemicals [45]. The encoding is called Compositional Pattern Producing Networks (CPPNs) [45]. When CPPNs encode artificial neural networks, the algorithm is called HyperNEAT, which will be described in detail below. A key idea behind CPPNs is that complex patterns (such as natural organisms) can be produced by determining the attributes of phenotypic components as a function of their geometric location. This idea is based on the knowledge that cells (or higher-level modules) in nature often differentiate into their possible types (spleen, liver, etc.) as a function of where they are situated in geometric space [2]. For example, for some insects, a segment at the anterior pole should produce antennae and a segment at the posterior pole should produce a stinger. Components of natural organisms cannot directly determine their

location in space, so organisms have evolved developmental processes that create chemical gradients and other signals that organismal components use to figure out where they are and, thus, what to become [2]. For example, early on in the development of embryos, different axes (e.g., anterior-posterior) are indicated by chemical gradients. Additional gradients signaled by different proteins can exist in the same area to represent a different pattern, such as a repeating motif. Downstream genes, such as Hox genes, can then combine repeated and asymmetric information to govern segmental differentiation. Further coordinate frames can then be set up within segments to govern intra-module patterns [2].

One of the key insights of CPPNs is that cells *in silico* can be directly given their geometric coordinates. The CPPN genome is a function that takes geometric coordinates as inputs and outputs the fate of an organismal component. When CPPNs encode two-dimensional pictures, the coordinates of each pixel on the canvas (e.g., $x = 2$, $y = 4$) are iteratively passed to the CPPN genome, and the output of the function is the color or shade of the pixel (Figure 2.1).

Each CPPN is a directed network, where each node is itself a mathematical function. The nature of the functions included can enable a variety of desirable properties, such as symmetry (e.g., a Gaussian function) and repetition (e.g., a sine function) that evolution can take advantage of. Nested coordinate frames can develop in the CPPN to compose complex coordinate frames. For instance, a sine function early in a network can create a repeating theme that, when passed into the symmetric Gaussian function, creates a repeating, symmetric motif, as demonstrated by the body segments in Figure 2.1. This process is similar to how natural organisms develop [2]. For example, many organisms set up a repeating coordinate frame (e.g., body segments) within which are symmetric coordinate frames (e.g., left-right body symmetry). Asymmetries can be generated by referencing global coordinate frames, such as the y-axis. The links that connect and allow information to flow between nodes in a CPPN have a weight value that can magnify or diminish the values that pass along them. Mutations that change these weights may, for example, give a

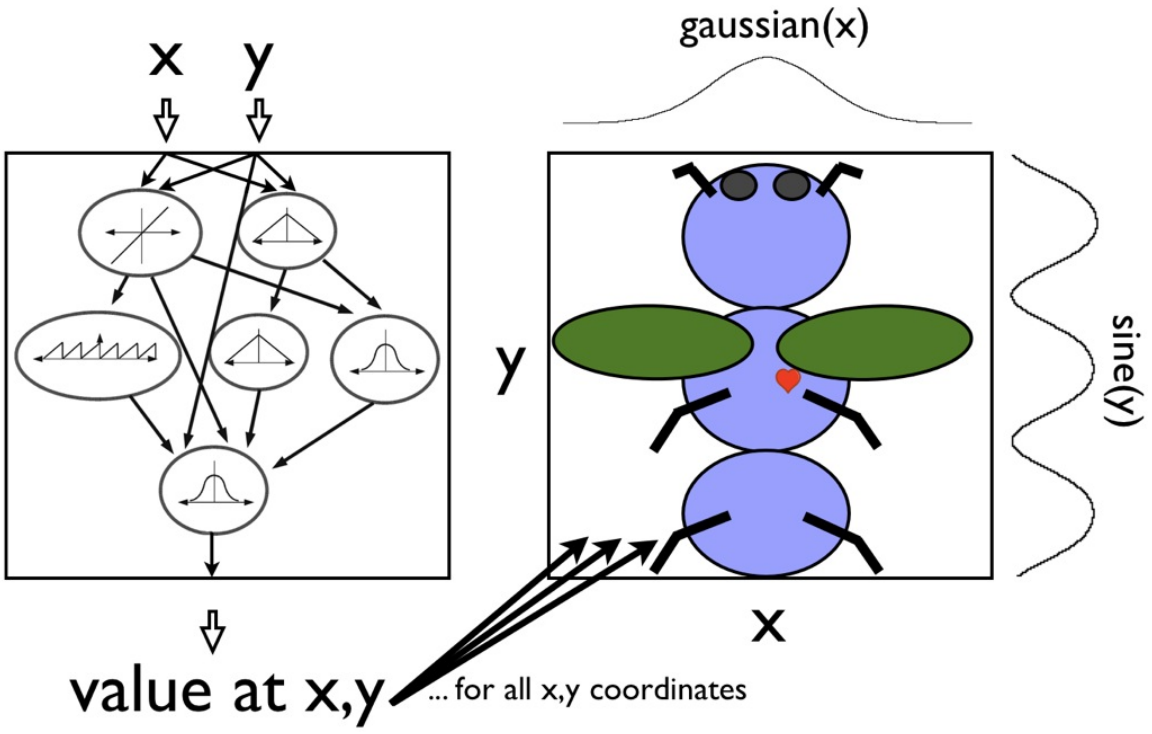

Figure 2.1: Compositional Pattern Producing Networks. CPPNs compose math functions to generate regularities, such as symmetries and repeated modules, with and without variation. This figure is adapted from Stanley (2007).

stronger influence to a symmetry-generating part of a network or diminish the contribution from another part.

One way to understand the types of forms CPPNs can produce is to evolve pictures with them, having humans perform the selection [42]. This process can generate shapes that look complex and natural (Figure 2.2). The images created demonstrate that the CPPN encoding can create forms with some of the features that generative encodings were designed to produce (e.g., regularity and modularity, with and without variation).

In the HyperNEAT algorithm, CPPNs encode for ANNs instead of pictures, and evolution modifies the population of CPPNs [46]. HyperNEAT evolves the weights for fixed-topology ANNs. Unless otherwise specified, the ANNs in the experiments in this dissertation feature a two dimensional, $m \times n$ Cartesian grid of inputs and a corresponding $m \times n$ grid of outputs. If an experiment uses an ANN with hidden nodes, the hidden nodes are

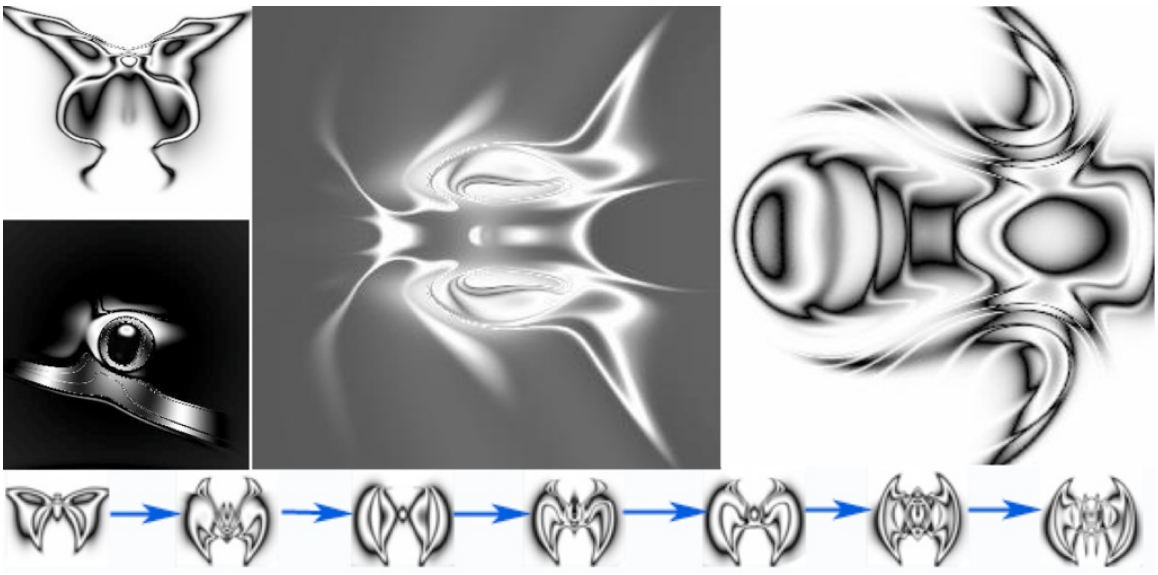

Figure 2.2: Images Evolved with CPPNs. Displayed are pictures from picbreeder.org [42], a website where visitors select images from a population evolved with the CPPN generative encoding, which is also used in HyperNEAT. The bottom row shows images from a single lineage. Blue arrows represent intermediate forms that are not pictured.

placed in their own grid between the input and output grid. Recurrence is disabled, so each of the $m \times n$ nodes in a grid has a link of a given weight to each of the $m \times n$ nodes in the proximate grid, excepting output nodes, which have no outgoing connections. Link weights can be zero, functionally eliminating a link.

The inputs to the CPPNs are a constant bias value and the coordinates of both a source node (e.g., $x_1 = 0, y_1 = 0$) and a target node (e.g., $x_2 = 1, y_2 = 1$) (Figure 2.3). The CPPN takes these five values as inputs and produces one or two output values, depending on the ANN topology. If there is no hidden layer in the ANN, the single output is the weight of the link between a source node on the input layer and a target node on the output layer. If there is a hidden layer, the first output value determines the weight of the link between the associated input (source) and hidden layer (target) nodes, and the second output value determines the weight of the link between the associated hidden (source) and output (target) layer nodes. All pairwise combinations of source and target nodes are iteratively passed as inputs to a CPPN to determine the weight of each ANN link. HyperNEAT can thus produce

patterns in link weight space similar to the patterns it produces in 2D pictures (Figure 2.2).

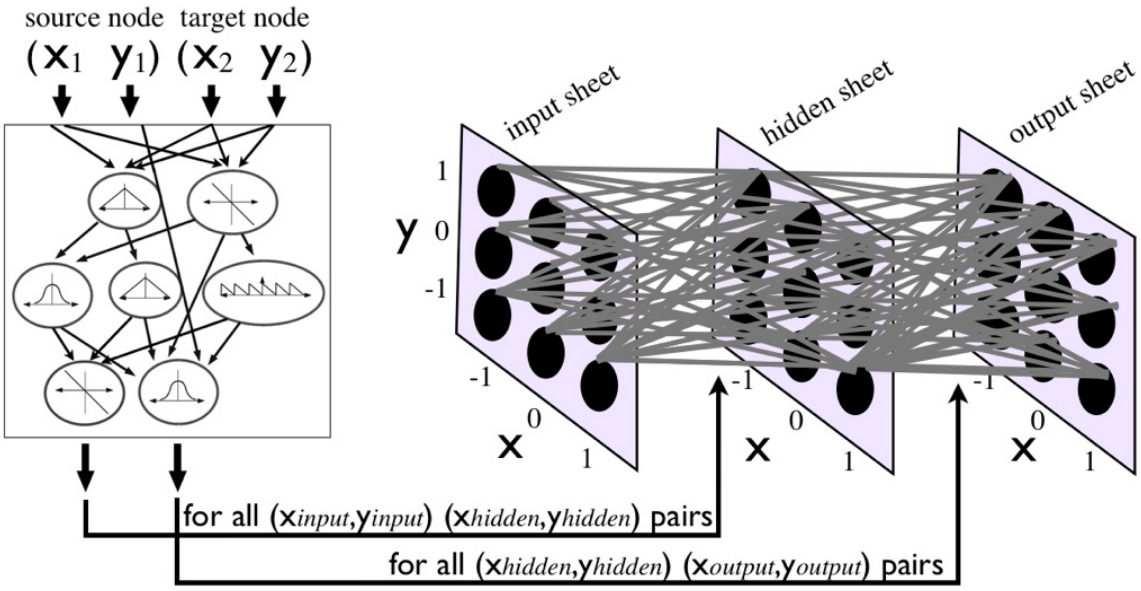

Figure 2.3: HyperNEAT Produces ANNs from CPPNs. Weights are specified as a function of the geometric coordinates of the source node and the target node for each connection. The coordinates of these nodes and a constant bias are iteratively passed to the CPPN to determine each connection weight. If there is no hidden layer, the CPPN has only one output, which specifies the weight between the source node in the input layer and the target node in the output layer. If there is a hidden layer in the ANN, the CPPN has two output values, which specify the weights for each connection layer as shown. This figure is adapted from Gauci and Stanley (2008).

An additional benefit of HyperNEAT is that it is one of the first neuroevolutionary algorithms capable of exploiting the geometry of a problem [46]. Because the link values between nodes are a function of the geometric positions of those nodes, if those geometric positions represent aspects of the problem that are relevant to its solution, HyperNEAT can exploit such information. For example, when playing checkers, the concept of adjacency (on the diagonals) is important. Link values between neighboring squares may need to be different than link values between distant squares. HyperNEAT can use adjacency to create a connectivity motif and repeat it across the board [15, 46]. Producing this type of regularity would be more difficult in other encodings, even generative ones, that do not include geometric information, because there is no easy way for such algorithms to identify

which nodes are adjacent.

Variation in HyperNEAT occurs when mutations or crossover change the CPPN function networks. Mutations can add a node to the graph, which results in the addition of a function to the CPPN network, or change its link weights. The function set for CPPNs in this dissertation includes sine, sigmoid, Gaussian, and linear functions. The evolution of the population of CPPN networks occurs according to the principles of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [47], which is described in Section 4 of this chapter. NEAT, which was originally designed to evolve neural networks, can be effectively applied to CPPNs because a population of CPPN networks is similar in structure to a population of neural networks.

The parameters for all of the experiments below follow standard HyperNEAT conventions [46] and can be found in the publications of these results [3–8].

2.3 FT-NEAT, a direct encoding control for HyperNEAT

A common direct encoding control for HyperNEAT is *Fixed-Topology NEAT* (FT-NEAT, also called Perceptron NEAT or P-NEAT when it does not have hidden nodes) [3–7, 46]. FT-NEAT is similar to HyperNEAT in all ways, except that it directly evolves each weight in the ANN independently instead of determining link weights via a generative CPPN. All other elements from NEAT (e.g., its crossover and diversity preservation mechanisms) remain the same between HyperNEAT and FT-NEAT. Additionally, the number of nodes in the ANN phenotype are the same in HyperNEAT and FT-NEAT. Mutations in FT-NEAT cannot add nodes, making FT-NEAT a degenerate version of NEAT. Recall that the complexification in HyperNEAT is performed on the CPPN genome, but that the number of nodes in the resultant ANN is fixed. The end product of HyperNEAT and FT-NEAT are thus neural network substrates with the same number of nodes, whose weights are determined in different ways.

2.4 NEAT, a second direct encoding control for HyperNEAT

While FT-NEAT is a good control for HyperNEAT because it holds the number of nodes in the ANN constant, HyperNEAT should also be compared to a cutting-edge direct encoding neuroevolution algorithm, such as regular NEAT [47].

The NEAT algorithm contains three major components [47]. Initially, it starts with small genomes that encode simple networks and slowly *complexifies* them via mutations that add nodes and links to the network. This complexification enables the algorithm to evolve the network topology in addition to its weights. Secondly, NEAT has a fitness-sharing mechanism that preserves diversity in the system and allows new innovations time to be tuned by evolution before forcing them to compete against rivals that have had more time to mature. Finally, NEAT uses historical information to perform crossover in a way that is effective, yet avoids the need for expensive topological analysis. A full explanation of NEAT can be found in Stanley & Miikkulainen (2002).

The only difference between FT-NEAT and NEAT is that hidden nodes can be added during evolution in NEAT. In those experiments in this dissertation where the optimal number of hidden nodes is not known a priori, I compare HyperNEAT to NEAT in addition to FT-NEAT.

The next chapter describes the experiments that I performed to investigate generative encodings and the results of those experiments.

Chapter 3

A generative encoding can exploit problem regularity

Note:This Chapter is an expanded version of Clune, Ofria, and Pennock 2008.

While it has been shown that generative encodings can outperform direct encodings [10, 15, 20, 21, 26, 35, 40, 46, 48], in every case the problem was highly regular or the regularity of the problem was unspecified and ambiguous. Gruau's work evolving neural nets with the generative encoding called Cellular Encoding used problem domains of bit parity or bit symmetry [20], which are both highly regular, or pole-balancing [21], where the regularity of the problem is unknown. Hornby (2002) demonstrated that a generative encoding based on L-systems outperformed a direct encoding control when applied to the perfectly regular parity problem and to evolving tables and mobile creatures (where repeating similar leg modules provided high fitness scores). The regularity of the Nothello game, a variant on Othello, from [40] is unknown. HyperNEAT has been shown to outcompete a direct encoding control on two problems that require the repetition of the same network motif [46].

These previous studies show that generative encodings do well on highly regular prob-

lems, but they raise the question of whether generative encodings achieve their increased performance on regular problems at the expense of performing poorly on problems with intermediate or low regularity. What is needed are comparisons of a generative encoding versus direct encoding controls on multiple problems as problem-regularity is decreased from high to low. Such studies should illuminate the degree to which generative encodings can make exceptions to the patterns they produce. Additionally, this type of investigation can test whether generative encodings provide any advantages over direct encodings on problems with low and intermediate amounts of regularity. Such investigations are described in the following sections.

3.1 Bit Mirroring problem

The Bit Mirroring problem is intuitively easy to understand, yet provides multiple types of regularities, each of which can be scaled independently. For each input, a target output is assigned (e.g., the input $x_1 = -1, y_1 = -1$ could be paired with output $x_2 = 0, y_2 = 1$, Figure 3.1a). A value of one or negative one is randomly provided to each input, and the fitness of an organism is incremented if that one or negative one is reflected in the target output. Outputs greater than zero are considered 1, and values less than or equal to zero are considered -1. The correct wiring is to create a positive weight between each input node and its target output and, importantly, to set to zero all weights between each input node and its non-target output nodes (Figure 3.1a). To reduce the effect of randomness in the inputs, in every generation each organism is evaluated on ten different sets of random inputs and these scores are summed to produce the fitness for that organism. The maximum fitness is thus $10n^2$, where n is the number of nodes in the input sheet. Each run lasted 2000 generations and had a population size of 500. As in Target Weights, the number of nodes does not change on this problem (additional nodes would only hurt performance), so only FT-NEAT is tested as a control.

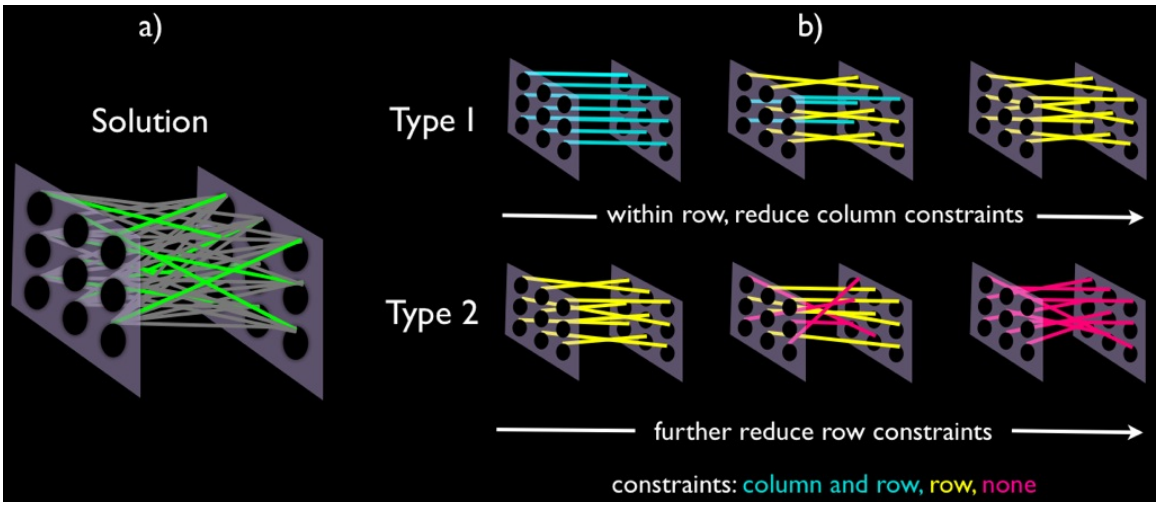

Figure 3.1: The Bit Mirroring Problem. **(a)** The correct wiring motif for the links emanating from each input node is to create a positive valued link (light green) to the correct target output node, and to turn off all other links (dark gray). **(b)** Within-column regularity (Type 1) is highest when all targets are in the same column, and can be lowered by decreasing the number of targets in the same column (by assigning unconstrained targets to columns at random). For the experiments in this paper, within-column regularity is scaled while keeping within-row regularity at its highest possible level, with all targets in the same row. Within-row regularity (Type 2) is reduced by constraining fewer targets to be in the same row. By first reducing within-column regularity, then further reducing within-row regularity, the overall regularity of the Bit Mirroring problem can be smoothly scaled from high to low. Note that the treatment with the lowest Type 1 regularity and the highest Type 2 regularity have identical constraints.

The first type of regularity in the problem is *within-column regularity*. This regularity is high when targets are in the same column, and low when there is no expectation about which column a target will be in (Type 1 in Figure 3.1b). The second type of regularity is *within-row regularity*, which is the likelihood that a target is in the same row (Type 2 in Figure 3.1b). While these two regularities are intuitively related, evolutionary algorithms must compute them independently, which means they are independent. Each of these types of regularity can be scaled by constraining a certain percent of targets to be in the same column or row, and assigning the remaining targets randomly.

The third type of regularity, called *inherent regularity*, arises from the fact that, for each node, the same pattern needs to be repeated: turning one link on and all other links off. This

type of regularity can be reduced by decreasing the number of nodes in the network, and hence the number of times that pattern needs to be repeated.

While the Bit Mirroring problem is easy to conceptualize, it is challenging for evolutionary algorithms. It requires most links to be turned off, and only a few specific links to be turned on. Moreover, links between input nodes and non-target nodes, which are likely to exist in initial random configurations and to be created by mutations, can complicate fitness landscapes. Imagine, for example, that a mutation switches the weight value on a link between an input node and its target output from zero to a positive number. The organism is now closer to the ideal wiring, but it may not receive a fitness boost if other incorrect links to that output node result in the wrong net output. The Bit Mirroring problem is useful, therefore, because it is challenging, yet its three main regularities are known, and can be independently adjusted.

Bit Mirroring Experiment 1 (BM-1): Reduce within-column regularity

A first experiment (BM-1) uses a 7×7 grid and decreases within-column regularity by reducing the percentage of inputs whose target is constrained to be in the same column. Unconstrained nodes must have the same y (row) value, but can have a different x (column) value. 10 trials were performed for each treatment lasting 2000 generations. Figure 3.2 reveals that the performance of HyperNEAT falls off as within-column regularity decreases. HyperNEAT is able to perfectly solve the problem in all but two trials of the most regular treatment, when the targets are all constrained to be directly across. As the within-column regularity decreases, the performance of HyperNEAT falls off fast. Interestingly, HyperNEAT does not benefit from the within-column regularity when 50% or fewer of its nodes are regularized in this way (only treatments with 60% or more column-constrained targets have fitnesses significantly better at a $p < 0.05$ level than fitness values from the treatment with 0% of nodes column-constrained; this and all future p values use Matlab's Mann-Whitney U-test unless otherwise specified).

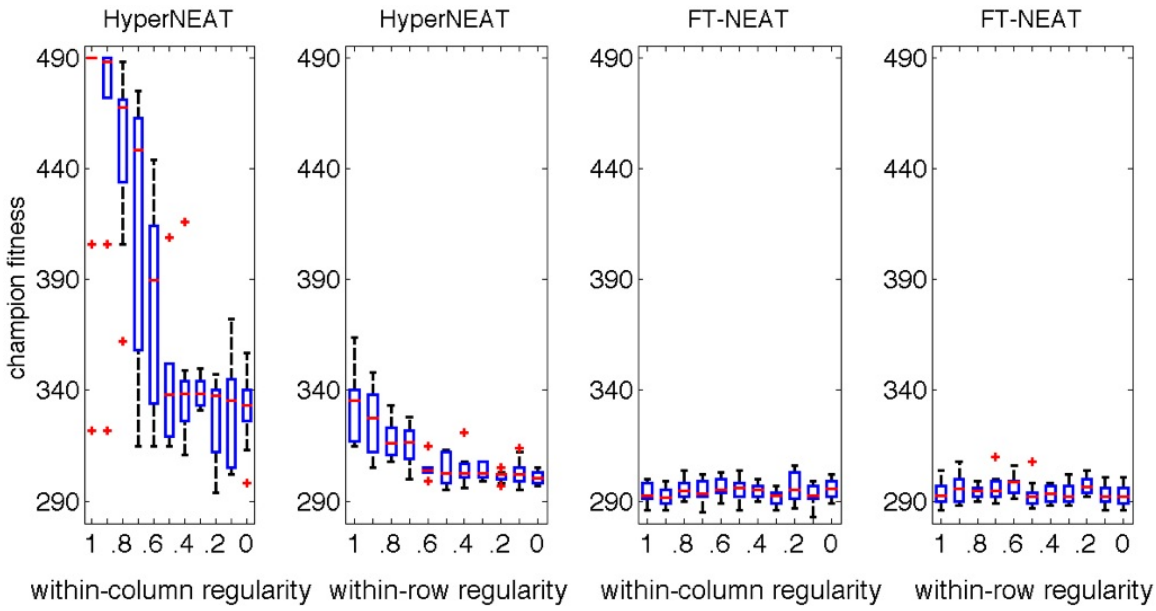

Figure 3.2: HyperNEAT and FT-NEAT on Versions of the Bit Mirroring Problem with Different Levels of Regularity. For each treatment, from left to right, within-column regularity is first decreased (left panel) and then within-row regularity is further decreased (right panel). The x-axis shows the fraction of targets constrained to columns or rows, respectively. Experiments wherein within-column regularity is scaled all have maximum within-row regularity, as described in the text.

Bit Mirroring Experiment 2 (BM-2): Further reduce row constraints

A second experiment (BM-2), which also involved 10 trials lasting 2000 generations and a 7x7 grid, scales a similar but different type of regularity by allowing all targets to be random with respect to column, but decreasing the percent that are constrained to be in the same row Figure 3.2. In a sense, BM-2 picks up where BM-1 left off (Figure 3.1). In fact, the least regular treatment from BM-1 and the most regular treatment from BM-2 have identical constraints (although different randomly generated mappings). The performance of HyperNEAT also decreases as this type of regularity is diminished (Figure 3.2). Surprisingly, the pattern of degradation is similar to BM-1; HyperNEAT no longer provides a fitness boost due to within-row regularity once that regularity falls below 60% (p only < 0.05 comparing 0% row-constrained to $>=$ 60% row-constrained treatments). While it is possible that running the BM-1 and BM-2 experiments longer would have allowed significant differences to

develop between less regular treatments, it is relevant that no significant differences were present after 2000 generations, which is a substantial number in the field of evolving neural nets. It is also interesting that the *range* of fitness values is correlated with the regularity of the problem for HyperNEAT. This variance might exist because, when regularity is present, the generative representation either discovered and exploited it, which would result in high fitness values, or it failed to fully discover the regularity, at which point its fitness more closely resembles less regular treatments.

BM-1 and BM-2 were also performed with FT-NEAT, the direct encoding control for HyperNEAT. As expected, the fitnesses produced by FT-NEAT were not affected by the regularity of the problem (Figure 3.2). None of the FT-NEAT treatments from BM-1 were significantly different from FT-NEAT treatments from BM-2 ($p < 0.05$). Furthermore, within both experiments, none of the treatments were significantly different than that experiment's 0% constrained treatment ($p > 0.05$). All HyperNEAT treatments from BM-1 do significantly better than FT-NEAT treatments from BM-1, due to both the within-row regularity present throughout and the inherent regularity of the Bit Mirroring problem. In BM-2, the within-row regularity decreases, leaving only the inherent regularity. However, presumably due to the inherent regularity of the problem, HyperNEAT still significantly outperforms FT-NEAT on all treatment conditions except for the 20% constrained treatment. Computational constraints prevented the performance of more trials, which may have eliminated this overlap in performance on the 20% constrained trial. While it is possible that running the trials for more generations would have allowed FT-NEAT to catch up to HyperNEAT on the irregular treatments in BM-2, viewing the fitness values across generations (not shown) suggests that this outcome is unlikely.

Bit Mirroring Experiment 3 (BM-3): Reduce inherent regularity (by reducing grid size)

A third experiment (BM-3) continues the comparison of Hyper-NEAT to FT-NEAT as a type of problem regularity is varied. For this experiment trials lasted 2000 generations, as before, but I conducted 40 trials per treatment due to the high variance between trials. All targets in BM-3 were random with respect to row and column, leaving only the inherent regularity in the Bit Mirroring problem. Since this inherent regularity stems from a motif that needs to be repeated for each input node ('zero out links to all outputs but one'), the number of times this motif needs to be repeated decreases with the grid size. Based solely on problem regularity, FT-NEAT should perform better in comparison to HyperNEAT as this type of regularity is decreased. Figure 3.3 reveals that this is the case. The performance of HyperNEAT degraded to, and then fell below, that of FT-NEAT as the grid size decreased. The overall decline was significant ($p < 0.05$ comparing the ratios on the 3×3 treatments to those 6×6 and greater). It is not clear why the trend was reversed on the smallest grid size. Note that BM-1 and BM-2 occurred on a 7×7 grid, where the level of inherent regularity provided HyperNEAT an advantage over FT-NEAT, even without within-column or within-row regularity.

3.2 Target Weights problem

BM-3 shows that once problems are sufficiently irregular and simple, the direct encoding FT-NEAT can outperform the generative encoding HyperNEAT. The likely explanation is that HyperNEAT is biased toward creating regular phenotypes and has trouble when the problem features mostly exceptions and little rule. However, even the 3×3 version of the Bit Mirroring problem has some inherent regularity left over. This section compares HyperNEAT to FT-NEAT on a problem that can be scaled to complete irregularity.

One way to create a completely irregular problem is to challenge evolution to pro-

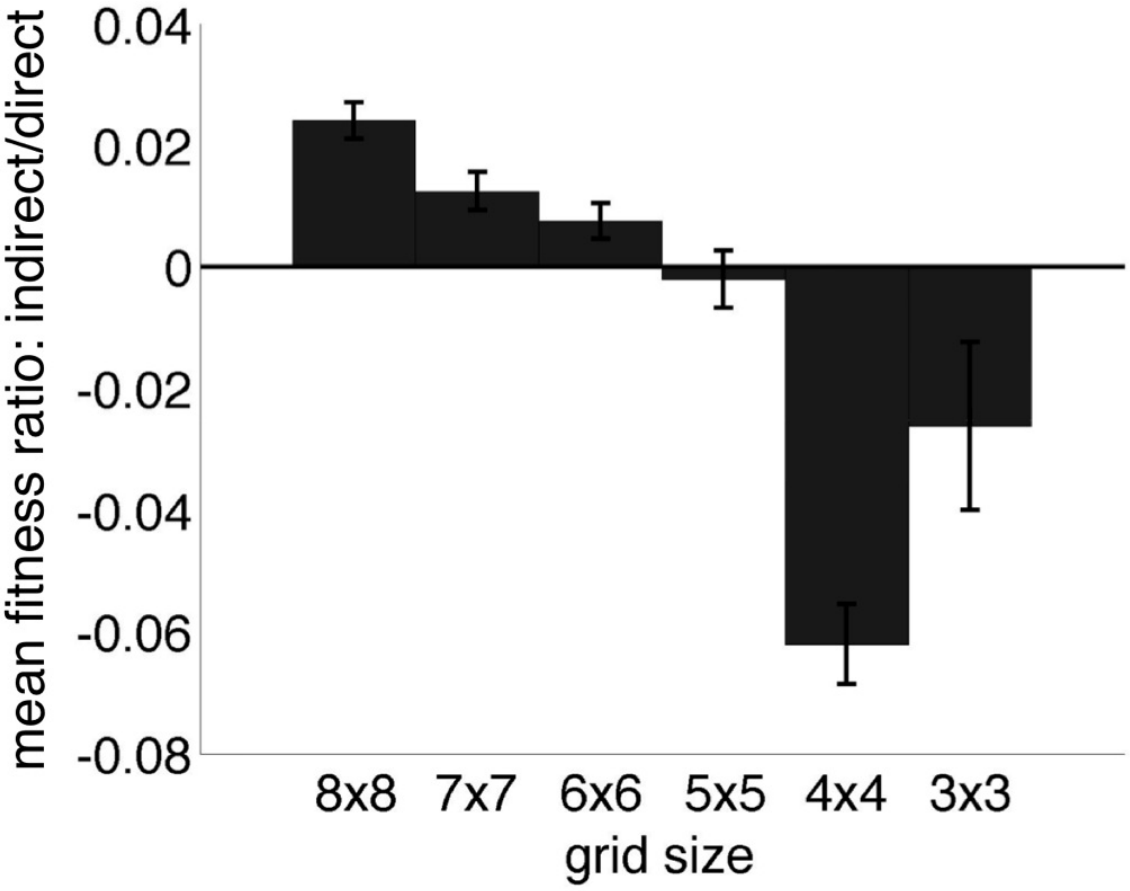

Figure 3.3: HyperNEAT Vs. FT-NEAT as the Inherent Regularity of the Bit Mirroring Problem is Decreased. Reducing the grid size reduces the amount of inherent regularity in the problem. Error bars show one standard error of the mean. Ratios are used instead of absolute differences because the allowable fitness range changes with grid size.

duce an ANN phenotype (P) that is identical to a target neural network (T), where T is completely irregular. Regularity can then be scaled by increasing the regularity of T (Figure 3.4). We call this the Target Weights problem because evolution is attempting to match a target vector of weights (recall that the number of nodes is constant, so the vector of weights in T fully describes T). Fitness is a function of the difference between each weight in P and the corresponding weight in T, summed across all weights. The lower this summed error is, the higher the fitness value. Specifically, the summed error is calculated as

$$error = \sum_{i=1}^{N} M - |P_i - T_i|, \tag{3.1}$$

where N is the number of weights, M is the maximum error possible per weight (which is 6 because weights could range from -3 to 3) P_i is the value of the ith weight in the phenotype, and T_i is the value of the ith weight in the Target ANN. To amplify the importance of small improvements, fitness is then calculated as

$$fitness = 2^{error}. \tag{3.2}$$

To scale the regularity of this problem, some randomly chosen subset of the target weight values, S, are assigned Q, a single randomly-chosen value. All remaining weight values are independently assigned a value at random. Changing the number of weights in S scales the regularity of the problem. When S is set to 0%, all of the target weight values are chosen independently at random. When S is set to 50%, half the weights have the value Q and the rest have values independently chosen at random. In the most regular version of the problem, S is set to 100% and all weights have the value Q. There are 11 treatments, with S values of $0, 10, 20...100$, and 10 runs per treatment. Target vectors are constant for each evolutionary run, but are different between runs (due to differences in randomly generated weight values, including Q). Trials last 1000 generations with a population size of 1000. The ANNs have 3×3 grids of input and output neurons. Because the number of nodes on this problem does not change, I test only FT-NEAT as a direct encoding control.

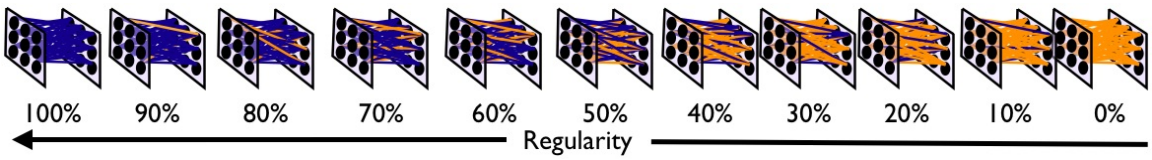

Figure 3.4: Scaling Regularity in the Target Weights Problem. Regularity is scaled by changing the percentage of weights S (which increases from right to left) that are set to Q, a single randomly-chosen value (shown as dark blue lines). The remaining weights are each set to a random number (shown as light orange lines).

The Target Weights problem is useful because it allows regularity to be scaled from zero to complete, and because the regularity of the solution is known *a priori*. It is also a simplistic problem because it has no interactions (epistasis) between link elements. Altering a given link will not change the optimal value for other links. While these attributes makes the Target Weights problem useful for investigating regularity, more challenging problems are also required to understand performance on more realistic types of problems.

The results from the Target Weights experiments (Figure 3.5) show that HyperNEAT exploited the regularity of the Target Weights problem early on, but FT-NEAT closed the gap and eventually outperformed HyperNEAT on all but the most regular treatment. This experiment provides another example of where a generative encoding can exploit intermediate amounts of problem regularity, and increasingly outperformed a direct encoding as problem-regularity rose. HyperNEAT performed better as the regularity of the problem increased, especially in early generations, where mean performance perfectly correlates with problem-regularity. Interestingly, after 1000 generations of evolution, the performance of HyperNEAT is once again statistically indistinguishable below a certain regularity threshold ($p > 0.05$ comparing the final performance of the $S = 0\%$ treatment to treatments with $S \leq 30\%$. Above that regularity threshold, however, HyperNEAT performed significantly better at each increased level of regularity ($p < 0.01$ comparing treatment with values of $S \geq 30\%$ to the treatment with an S value 10% higher). However, the difference in HyperNEAT's performance between treatments early on complicates the story of how HyperNEAT's performance plateaus below a certain regularity threshold, by making it depend on time. Fitness plots across generations from experiments BM1 and BM2 (not shown) do not tell a similar story; in those experiments the less regular treatments have similar fitness scores throughout.

In contrast to HyperNEAT, FT-NEAT was blind to the regularity of the problem: the results from different treatments are visually and statistically indistinguishable ($p > 0.05$). Early on HyperNEAT outperformed FT-NEAT on regular versions of the problem ($p < 0.01$

comparing treatments of $S \geq 60\%$ at generation 100), but at 1000 generations FT-NEAT outperformed HyperNEAT on all but the two most regular versions of the problem ($p < 0.001$). HyperNEAT outperformed FT-NEAT on the most regular version of the problem at generation 1000 ($p < 0.001$), and the algorithms are statistically indistinguishable on the $S = 90\%$ treatment ($p > 0.05$).

A further point of interest is the lack of progress HyperNEAT makes on the highly regular treatments (e.g., where 80% or 90% of the targets are repeated). While it exploits the regularity early on, HyperNEAT seems unable to make exceptions to the rule in order to encode the non-conforming link values, as evidenced by the lack of fitness improvements after the initial gains. This evidence suggests HyperNEAT is too biased towards producing regular solutions, and cannot create irregularities that would improve its performance.

3.3 Scaling concurrent regularity

Each of the previous experiments have shown how HyperNEAT and FT-NEAT perform as a *single type* of regularity is scaled from high to low. Figure 3.6 shows how HyperNEAT and FT-NEAT perform as the number of *concurrent types* of regularity is decreased. It samples from the Target Weights experiment and the three Bit Mirroring experiments. While it could have been the case that exploiting one type of regularity prevented the exploitation of others, Figure 3.6 reveals that it is possible for HyperNEAT to simultaneously exploit multiple types of regularity. It also demonstrates that the performance of HyperNEAT degrades to, then falls below, that of FT-NEAT as concurrent problem regularity decreases.

3.4 Conclusions

The experiments in this section, which cover four types of regularity from two different problems, paint a consistent picture despite some idiosyncrasies. In general, the Hyper-NEAT generative encoding showed some difficulty in making exceptions to the rules it

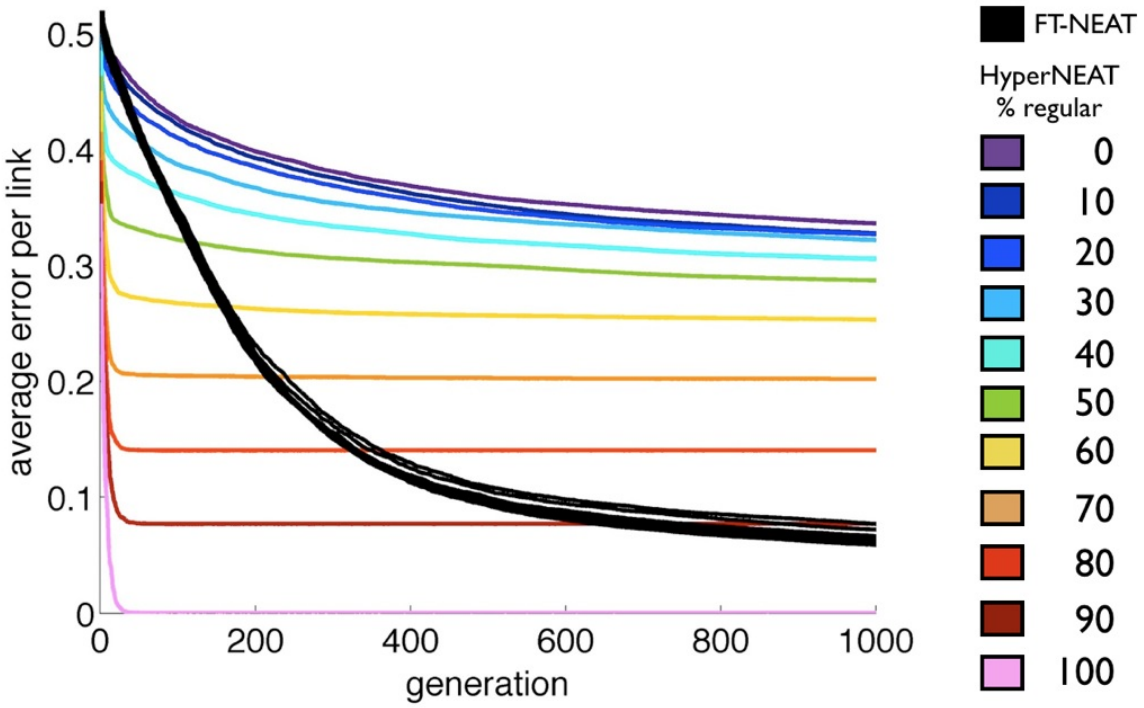

Figure 3.5: Mean Performance of HyperNEAT and FT-NEAT on a Range of Problem Regularities for the Target Weights Problem. HyperNEAT lines are colored for each regularity level, and in early generations are perfectly ordered according to the regularity of the problem (i.e., regular treatments have less error). The performance of FT-NEAT (black lines) is unaffected by the regularity of the problem, which is why the lines are overlaid and mostly indistinguishable.

discovered. Its performance decreased as problem regularity decreased. Nevertheless, the generative encoding did provide a fitness boost over its direct encoding counterpart as problem regularity increased. The generative encoding's ability to simultaneously exploit concurrent types of regularities meant that the more types of regularity, the larger the boost. However, the generative encoding could exploit a type of regularity only when the amount of regularity within that type was relatively high. This result is not obvious from theoretical considerations and has not been reported before. As I show in Chapter 4, similar conclusions hold on a challenging engineering problem. Additional work is needed to see if the conclusions drawn from this generative encoding on these problems apply to most generative encodings on many problems. Finally, while the direct encoding outperformed the generative encoding on irregular problems, it was not until the problem was relatively irreg-

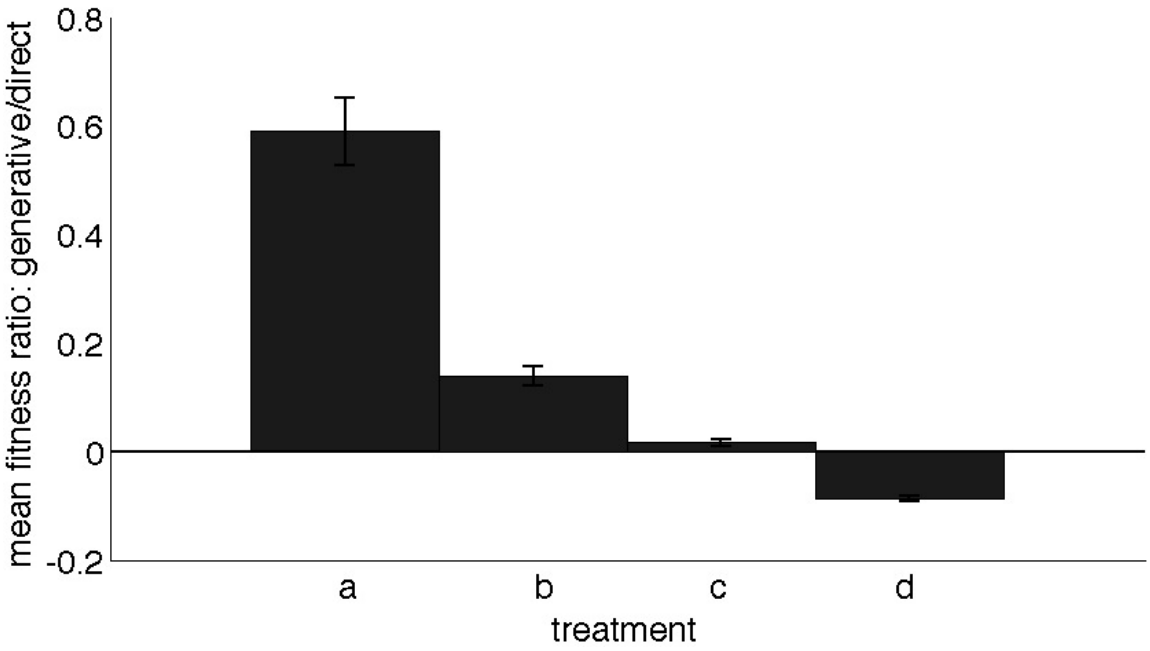

Figure 3.6: Comparison of HyperNEAT to FT-NEAT Across Experiments as Regularity is Decreased. **a**) All targets are constrained to be within the same column and row as their source on the 7x7 Bit Mirroring problem (three types of concurrent regularity). **b**) Targets are only constrained to be in the same row on the 7x7 Bit Mirroring problem (two types of concurrent regularity) **c**) Targets are randomly chosen, leaving only the inherent regularity of the 7x7 Bit Mirroring problem (one type of regularity) **d**) The Target Weights problem with all link values randomly chosen (no type of regularity).

ular that this transition occurred. FT-NEAT outperformed HyperNEAT only on the Target Weights problem and small, irregular versions of the Bit Mirroring problem. In fact, it was challenging for me to come up with problems irregular enough to provide an advantage to the direct encoding. Even the 7×7 Bit Mirroring problem, which is simple compared to real-world problems, had multiple regularities that could be exploited. It is likely that on most difficult, real-world problems, the existence of many types of regularities will provide an advantage to generative encodings. One interesting question this section raises, however, is whether the level of regularity within each type will be sufficient for a generative encoding to be able to exploit it.

The next section investigates these issues by testing HyperNEAT on a difficult real-world problem.

Chapter 4

A generative encoding can perform well on a challenging real-world problem (evolving gaits for simulated quadrupeds) by automatically exploiting problem regularity

Note: Chapter 4 is an expanded version of Clune, Beckmann, Ofria, and Pennock 2009.

The previous section showed that a generative encoding can exploit regularity on diagnostic problems that I invented because they are easy to conceptualize and because they explicitly contain regularities that can be varied. This section tests whether that generative encoding can show the same ability to automatically exploit problem regularity when the problem is more challenging, and when regularity was not explicitly built into the problem. Importantly, the problem chosen is one where previous attempts to evolve solutions to it involved the researcher manually decomposing the problem because the EA did not recog-

nize its regularity. The results from this chapter support the notion that the HyperNEAT generative encoding can deliver on its promise of evolving regular ANNs that perform well on complex problems by exploiting their regularity. Additionally, this problem provides an opportunity to study how generative encodings outcompete direct encodings on regular problems. I will show that HyperNEAT exploits problem regularity by producing regular brains with regular behaviors.

I first describe the importance of the problem and previous attempts at solving it with EAs before describing how I applied HyperNEAT to it.

4.1 The importance of producing gaits for legged robots

Legged robots are likely to play an increasingly important role in our lives in generations to come. Legged consumer robots already exist, such as the biped ASIMO and the quadruped AIBO. Both the military and industry have a variety of legged robots under development. One benefit of legged robots over their wheeled counterparts is their mobility on rugged terrain, but a major drawback is the challenge of creating controllers for them. The problem is complicated because of the number of degrees of freedom in each leg and because of the body's changing center of mass and momentum. Human engineers have had to design the majority of controllers for legged robots, which is a difficult and time-consuming process [41, 56]. Furthermore, given how sensitive gait controllers are to slight changes in the configuration of a robot, a new gait must be created each time a robot is changed, which can lead to significant delays in the prototyping stage of robotic development [27].

4.2 Previous work evolving gaits for legged robots

It is not surprising, therefore, that people have tried to automate the process of gait creation. Evolutionary computation, usually involving the evolution of neural network controllers, has been successfully used to this end [14, 19, 25, 27, 34, 43, 51, 52]. Evolved gaits are often

better than those produced by human designers; one was even included on the commercial release of Sony's AIBO robotic dog [27, 52]. However, many researchers have found that evolutionary algorithms do not perform well when challenged with the entire problem, because of the large number of parameters that need to be simultaneously tuned to achieve success [1, 13, 27, 30, 39, 51, 52]. Many of these scientists report that, while it is possible to evolve a controller to manage the inputs and outputs for a single leg, once evolution is challenged with the inputs and outputs of many legs, it fails to make progress.

One solution that has worked repeatedly is to explicitly help the evolutionary algorithm exploit the regularities in the problem. This approach involves manually decomposing the problem by, for example, evolving the controller for one leg of a quadruped and then copying that controller to every leg, with some variation in phase. Many permutations of this strategy of manually decomposing the problem have produced functioning gaits [1, 13, 27, 30, 39, 51, 52]. Another type of manual decomposition, which is often used in addition to the previous one, is to simplify the high-level problem of locomotion by breaking it into manually-defined subproblems (e.g., producing leg oscillations, not falling over, moving certain legs in synchrony, etc.), and first rewarding the simple problems, then more advanced problems, on upwards until a high-level goal, such as rapid locomotion, can be rewarded directly [34, 51, 57].

Unfortunately, both of these strategies have drawbacks. It would be better if we could completely automate the process and remove the need for human engineers to spend time decomposing the problem. Furthermore, such manual decomposition potentially introduces constraints and biases that could preclude the attainment of better solutions [36]. Finally, if we can employ algorithms that can automatically discover and exploit problem regularities, such algorithms will be able to do so for complex problems with respect to regularities of which humans might not be aware.

For these reasons, it is important to investigate algorithms that can automatically exploit regularities in problems, such as generative encodings. While it has not been the

norm, generative encodings have been used previously to evolve the gaits of legged robots. In two well-known cases, a generative encoding was used to evolve the gaits and the morphologies of robots [25,26,43]. These dual evolutionary goals complicate analysis because the creatures may not be regular, and because it is unclear if any of the demonstrated fitness advantage of generative encodings (e.g., [25]) was due to the ability to evolve regular neural networks or due to the ability to build better morphologies. In one paper with similar goals as the current work, a generative encoding and direct encoding were compared for their ability to evolve a gait for a legged robot in an attempt to see whether the generative encoding could exploit the regularity of the problem without the problem being simplified or manually decomposed [19]. However, this project used a simple model of a six-legged insect that had only two degrees of freedom per leg. Nevertheless, the work showed that a generative encoding could automatically discover the regularity of the problem and decompose it by encoding a neural submodule once and using it repeatedly. The generative encoding also outperformed a direct encoding by solving the problem faster. Unfortunately, computational limits at the time meant that such results were anecdotal and not statistically significant because so few trials could be performed.

To summarize, prior to this work it has not been shown that a generative encoding can automatically exploit problem regularity when evolving controllers for legged robots. I demonstrate in this chapter that the HyperNEAT generative encoding can achieve this objective.

4.3 Applying HyperNEAT to the Quadruped Controller problem

The quadruped robots look like tables (Figure 4.1), which is fitting because seminal work comparing generative encodings to direct encodings was performed on the evolution of static tables [24]. There are two hip joints and one knee joint. The first hip joint (HipFB)

Figure 4.1: The Simulated Robot in the Quadruped Controller Problem.

allows the legs to swing forward and backward (anterior-posterior) and is constrained to 180 degrees such that at maximum extension it is parallel with the torso. The second hip joint (HipIO) allows a leg to swing in and out (proximal-distal). Together, the two hip joints approximate a universal joint. The knee joint swings forward and backward. The HipIO and knee joints are unconstrained. Robots were evaluated in the ODE physics simulator [www.ode.org].

The ANN for this experiment features three two-dimensional, 5×4 grids forming an input, hidden and output layer (Figure 4.2). All possible connections between adjacent layers can exist, meaning that there are 800 potential links in the ANN of each organism.

The inputs to the ANN are the current angles of each of the 12 joints of the robot (described below), a touch sensor that provides a 1 if the lower leg is touching the ground and a 0 if it is not, the pitch, roll and yaw of the torso, and a modified sine wave (which facilitated the production of periodic behaviors). The sine wave function is

$$sin(t/120)\pi, \tag{4.1}$$

where t is the number of milliseconds that have elapsed since the start of the simulation.

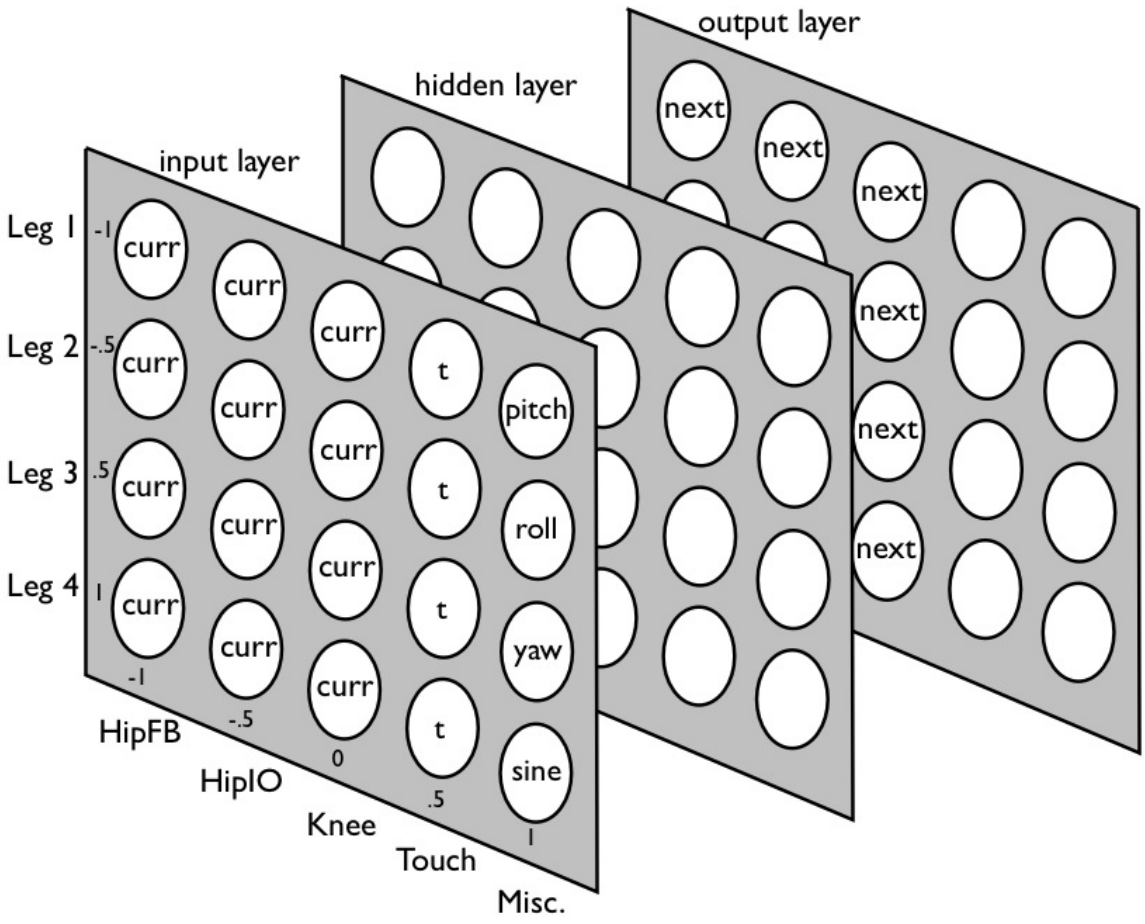

Figure 4.2: ANN Configuration for HyperNEAT and FT-NEAT Treatments. The first four columns of each row of the input layer receive information about a single leg (the current angle of each of its three joints, and a 1 or 0 depending on whether the lower leg is touching the ground). The final column provides the pitch, roll, and yaw of the torso, as well as a sine wave. Evolution determines how to use the hidden-layer nodes. The nodes in the first three columns of each of the rows in the output layer specify the desired new joint angle. The joints move toward that desired angle in the next time step as described in the text. The outputs of the nodes in the rightmost two columns of the output layer are ignored.

Multiplying by π facilitates the production of numbers that go from $-\pi$ to π, which was the range of the unconstrained joints. The constant 120 was chosen because it was experimentally found to produce fast, yet natural, gaits. While changing this constant can affect the types of gaits produced, doing so was never observed to alter any of the qualitative conclusions of this section. Preliminary tests determined that the touch, pitch, roll, yaw, and sine inputs all improved the ability to evolve fit gaits (data not shown).

The outputs of the neural network were the desired joint angle for each joint. This desired joint angle was fed into a PID controller that simulated a servo and moved the joint toward the desired angle. 50 trials were conducted for each encoding. Each trial featured a population of 150 organisms, which is common for HyperNEAT experiments [46], and lasted 1000 generations.

Each organism was simulated for 6000 time steps. Trials were cut short if any part of the robot save its lower leg touched the ground or if the number of direction changes in joints exceeded 960. The latter condition was an attempt to roughly reflect the physical fact that servo motors cannot be vibrated incessantly without breaking. The fitness of controllers was the following function of d, the maximum distance traveled:

$$fitness = 2^{d^2}. \tag{4.2}$$

The exponential nature of the function magnified the selective advantage of small increases in the distance traveled.

It is also possible to scale the regularity of the Quadruped Controller problem, which enabled me to test whether the conclusions from the preceding diagnostic problems hold on this real-world problem. Regularity in this problem can be scaled by changing the number of faulty joints. A faulty joint is one where, if an angle A is requested, the actual desired angle sent to the PID controller is A + E, where E is an error value in degrees in the range $[-2.5, 2.5]$. The value of E was chosen from a uniform random distribution in this range for each faulty joint at the beginning of a run, and was constant throughout the run. Such

errors are analogous to the inconsistencies of robotic joints produced by manufacturing processes. The more faulty joints, the less regularity there is in the problem because fewer legs behave identically. For HyperNEAT, FT-NEAT, and NEAT I performed 50 runs for experiments with 0, 1, 4, 8, and 12 faulty joints. The default version of this problem is the most regular version with 0 faulty joints, which is the version referred to throughout the rest of the dissertation unless the regularity is specified.

4.4 Comparing the performance of HyperNEAT to direct encoding controls on the Quadruped Controller problem

The data from the Quadruped Controller problem generally support the conclusions from Target Weights and Bit Mirroring. Initially, HyperNEAT outperformed both FT-NEAT and NEAT on the two most regular version of this problem, where there were 0 or 1 faulty joints (Figure 4.3, $p < 0.001$). That HyperNEAT outperformed NEAT is particularly noteworthy, given that NEAT is one of the most successful direct encoding neuroevolution algorithms. This result provides another demonstration that generative encodings can outperform direct encodings on regular problems.

As was seen on Target Weights and Bit Mirroring, HyperNEAT's performance increased with the regularity of the problem, but only above a certain threshold: the 0 faulty joint treatment significantly outperformed the 1 faulty joint treatment ($p < 0.001$), which, in turn, outperformed the 4 faulty joints treatment ($p < 0.001$) which, in turn, outperformed the 8 faulty joints treatment ($p < 0.001$), but the 8 and the 12 faulty joints treatments were statistically indistinguishable ($p > 0.05$). In contrast to Target Weights and Bit Mirroring, FT-NEAT was not blind to the regularity of this problem, although it was less sensitive to the regularity than HyperNEAT. The treatment with 0 faulty joints was statistically indis-

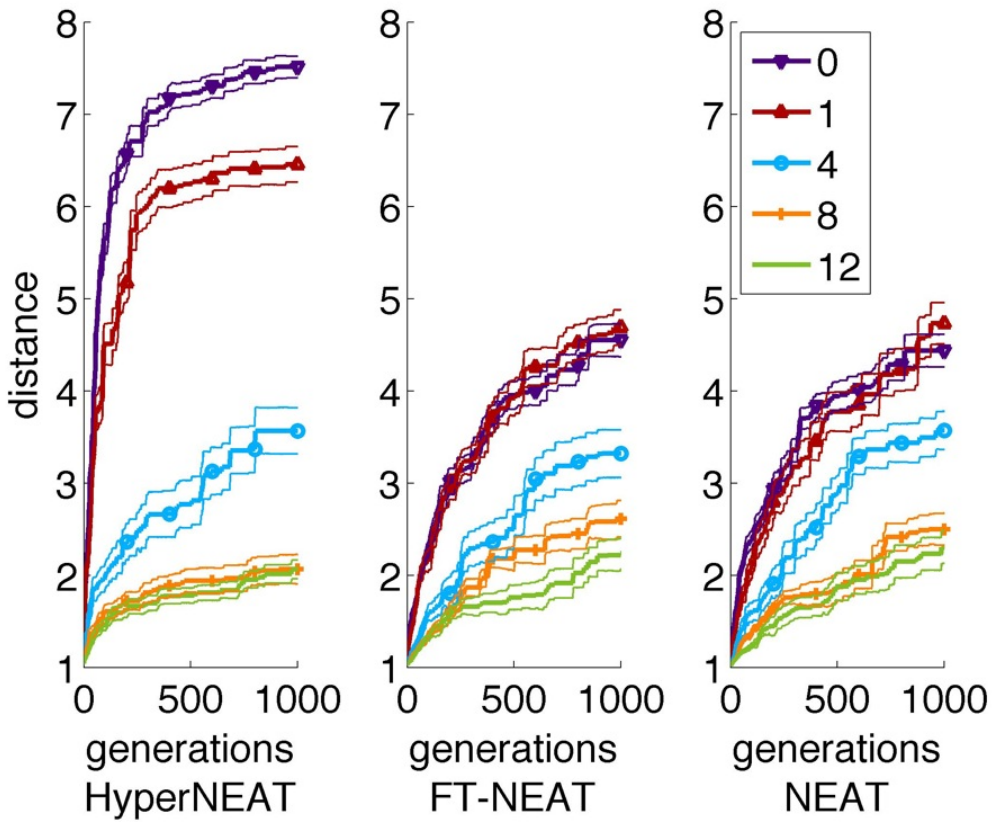

Figure 4.3: Performance of HyperNEAT, FT-NEAT, and NEAT on the Quadruped Controller Problem with 0, 1, 4, 8, and 12 Faulty Joints. Plotted for each treatment is the mean across 50 trials of the greatest distance away from the starting place arrived at by the best organism in that generation. Thin lines above and below the mean represent one standard error of the mean.

tinguishable from the 1 faulty joint treatment ($p > 0.05$), but performance on both of these treatments was higher than on the 4 faulty joints treatment ($p < 0.001$) which was, in turn, higher than the 8 faulty joints treatment. As was the case for HyperNEAT, performances for FT-NEAT on the treatments with 8 and 12 faulty joints were statistically indistinguishable ($p > 0.05$). The statistical comparisons for NEAT are the same as those for FT-NEAT.

One reason that regularity may affect the direct encodings in this problem is because link weights tend to be near the maximum or minimum allowable value, which is partly due to the fact that mutations that create links outside of this range are set to the maximum or minimum value. This method makes it more likely to see extreme link values than all

other link values, which can facilitate coordination in the joints because different joints are controlled by links with similar weights. If normal joints were controlled by links with the maximum or minimum link value, to have faulty joints behave the same as normal joints, link values would either have to be outside the allowable range, or inside the range at a non-extreme (and thus harder to set) value. Faulty joints thus increase the difficulty of the problem for generative and direct encodings, and can help explain why the direct encodings were not blind to the regularity of the problem.

Consistent with results from Bit Mirroring and Target Weights, FT-NEAT was able to outperform HyperNEAT on the Quadruped Controller problem once the regularity of the problem was sufficiently low. FT-NEAT and NEAT outperformed HyperNEAT on both the 8 and 12 faulty joints treatments. On the treatment with 8 faulty joints, the difference was significant for FT-NEAT ($p < 0.05$), but not for NEAT. On the treatment with 12 faulty joints, the difference for FT-NEAT was slightly insignificant ($p = 0.066$) but the difference for NEAT was significant ($p < 0.01$).

The difference in fitness in the first generation, which is comprised of randomly generated organisms, is interesting. HyperNEAT begins with an advantage over FT-NEAT because even randomly generated CPPNs are sometimes able to produce the coordination of legs that facilitates movement. I observed that some of the randomly generated organisms in HyperNEAT displayed impressive amounts of coordination and immediately appeared to be on the road toward rudimentary locomotion. Randomly generated FT-NEAT and NEAT organisms did not provide this impression.

Overall, these results support the conclusions from Target Weights and Bit Mirroring. The direct encodings outperform HyperNEAT when problem regularity is low, but as problem regularity increases, HyperNEAT can exploit that regularity whereas the direct encodings mostly do not. This ability to exploit problem regularity means that HyperNEAT increasingly outperforms direct encoding controls as problem regularity increases. I now investigate how HyperNEAT is able to exploit the regularity of problems, using the

Quadruped Controller problem as a case study.

4.5 Investigating why HyperNEAT outperforms direct encoding controls on the Quadruped Controller Problem

I employ three methods to understand why HyperNEAT outperforms direct encoding controls on the Quadruped Controller problem. I classify the gait behaviors by watching videos of evolved gaits, measure the joint angles during locomotion, and view ANNs of each robot. Each of these methods reveals that the brains and behaviors produced by the generative encoding are more regular than those produced by the direct encoding.

4.5.1 HyperNEAT gaits are more coordinated and general

One of the benefits of evolving behaviors in three dimensions is that human beings have a good intuitive understanding of such environments. I took advantage of these intuitions to subjectively judge the coordination of evolved gaits by watching videos of them. Specifically, I watched videos of the best organism produced in each of the 50 runs on the problem with 0 faulty joints for HyperNEAT, FT-NEAT, and NEAT. Videos of these gaits are available at http://devolab.msu.edu/SupportDocs/HyperNEAT.

The videos reveal that HyperNEAT quadrupeds tend to have a large amount of coordination among all of their legs. The HyperNEAT gaits were all extremely regular. They featured two separate types of regularity: coordination between legs, and the repetition of the same movement pattern across time. Generally, the gaits were one of two types. The first type involved a four-way symmetry where each leg moved in unison and the creature bounded forward repeatedly (Figure 4.4, top row). This gait implies that HyperNEAT is reusing neural information in a regular way to control all of the robot's legs. The sec-

ond gait type resembled a horse gallop and featured the back three legs moving in unison, with the fourth leg moving in opposite phase. This *3-1 gait* demonstrates that HyperNEAT can reuse neural information with variation, since the same behavioral pattern existed in each leg, but was inverted in one leg. The ability to produce repetition with variation is a desirable feature in genetic encodings [48].

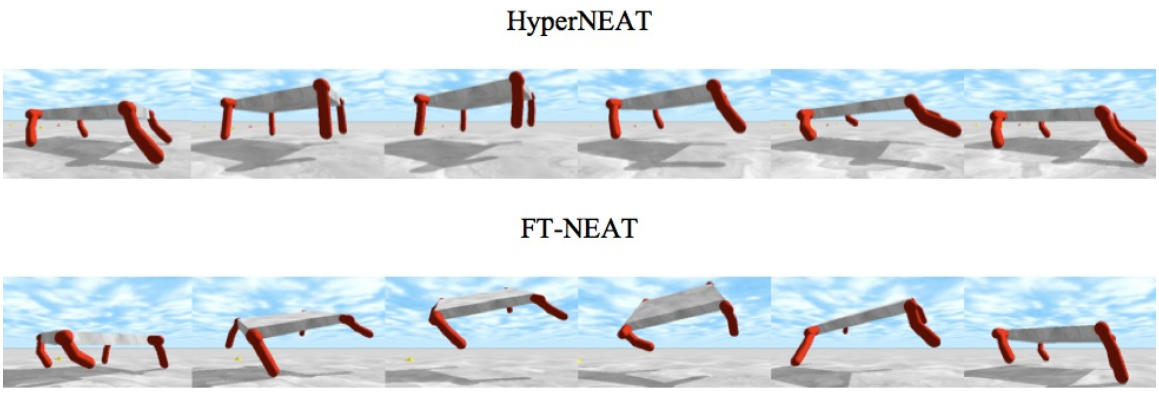

Figure 4.4: A Time Series of Images from Typical Gaits Produced by HyperNEAT and FT-NEAT. HyperNEAT robots typically coordinate all of their legs, whether all legs are in phase (as with this robot) or with one leg in anti-phase. A short sequence involving a bound or gallop is repeated over and over in a stable, natural gait. FT-NEAT robots display far less coordination among legs, are less stable, and do not typically repeat the same basic motion. NEAT gaits are qualitatively similar to FT-NEAT gaits.

Overall, the HyperNEAT gaits resemble those of running natural organisms in that they were coordinated and graceful. These observations are noteworthy because they indicate that HyperNEAT is automatically exploiting regularities on a challenging, real-world problem. This accomplishment is especially impressive given that researchers have previously needed to manually decompose legged locomotion tasks in order for evolutionary algorithms to perform well [1, 13, 27, 30, 39, 52].

The gaits of FT-NEAT and NEAT, on the other hand, were mostly uncoordinated and erratic, with legs often appearing to operate independently of one another (Figure 4.4, bottom row). A few of the best-performing gaits did exhibit coordination between the legs, and the repetition of a basic movement pattern, but most of the legs were irregular. Even

the regular gaits were not as natural and impressive as the HyperNEAT gaits, which was reflected in their lower objective fitness values. For most direct-encoding gaits, some legs flailed about, others tripped the organism, and legs often worked against each other by pushing in opposite directions. The robots frequently cartwheeled and tripped in unstable positions until they finally fell over. There was less repetition of a basic movement pattern across time, and coordination between legs tended to be infrequent and transitory.

It is interesting that there were a few examples of regular gaits produced by FT-NEAT and NEAT, which shows that it is possible for direct encodings to produce regularities. However, HyperNEAT was more consistent at producing regular gaits. All of the HyperNEAT gaits were regular, whereas only a few FT-NEAT and NEAT gaits were. It is important for algorithms to be consistent, especially when computational costs are high, so that high-quality results can be obtained without performing many runs. A test of the reliability of each encoding is to watch the median and least fit gaits of the 50 trial champions for each encoding: for HyperNEAT these gaits were coordinated and effective, whereas for FT-NEAT and NEAT they were discombobulated. In general, the videos reveal a greater gap in performance between HyperNEAT and the direct encodings than is suggested by the fitness scores, especially for all but the best runs per algorithm. Most of the direct encoding gaits do not resemble stable solutions to quadruped locomotion, whereas HyperNEAT produced a natural gait in all trials with a small variety of different solutions.

A second method for investigating how HyperNEAT was able to outperform the direct encodings is to look at the angles of the leg joints during locomotion. This technique is a different way of estimating the coordination, or lack thereof, of the different legs under each encoding. Plots of each leg's HipFB joint from the best, median, and worst runs for each algorithm corroborate the subjective opinions formed by watching videos (Figure 4.5). The legs in all HyperNEAT organisms showed a high degree of both of the two regularities mentioned above: at any point in time most legs were in similar positions (except the exception leg in the 3-1 gait, which was opposite), and a basic movement pattern was

repeated across time. The direct encoding gaits were less regular on both fronts, except for the highest-performing gaits. The median and worst gaits are representative of most of the direct encoding gaits in that there was little coordination between legs or across time (Figure 4.5). While only the HipFB joint is shown, plots of the other two joints are consistent with these results.

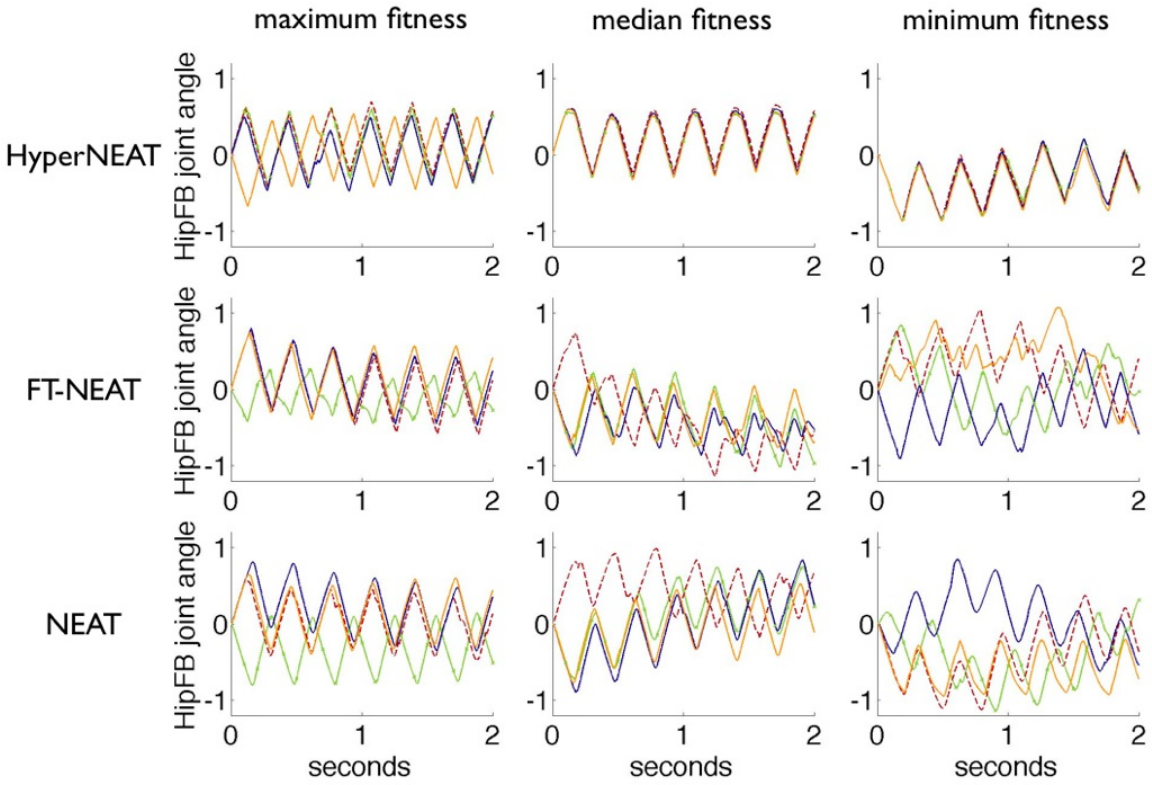

Figure 4.5: HipFB Joint Angles Observed in Robots Evolved with HyperNEAT, FT-NEAT, and NEAT. The possible range for this joint is -0.5π to 0.5π. The *Y* axis shows radians from the initial down (0) position. For clarity, only the first 2 seconds are depicted. For HyperNEAT, the best gait is an example of the 3-1 gait, where three legs are in phase and one leg is in opposite phase, which resembles the four-beat gallop gait. The other two HyperNEAT gaits are four-way symmetric, with all legs coordinated in a bounding motion (Figure 4.4). The best direct encoding gaits are mostly regular. However, the median and worst gaits, which are representative of most of the direct encoding gaits, are irregular: while some legs are synchronized, other legs prevent the coordinated repetition of a pattern.

One of the benefits of regularity can be generalization. Repeating the same basic pattern of motion is a type of regularity that is likely to generalize, because its success in one cycle makes it probable that it will be successful in the next cycle. A non-repeating sequence of

moves, however, may be less likely to generalize because its past is less likely to predict its future. The videos and joint angles suggest that the HyperNEAT gaits should generalize better than the direct encoding controls.

To test this hypothesis, I removed the time limit of 6 seconds and measured how long the evolved robots were able to move before they fell over. This analysis was conducted with the highest performing robot from all 50 runs for each encoding. Figure 4.6 reports that HyperNEAT gaits were significantly more general than FT-NEAT and NEAT gaits ($p < 0.001$). HyperNEAT gaits were the only ones, on average, that kept moving beyond the number of seconds simulated during evolution.

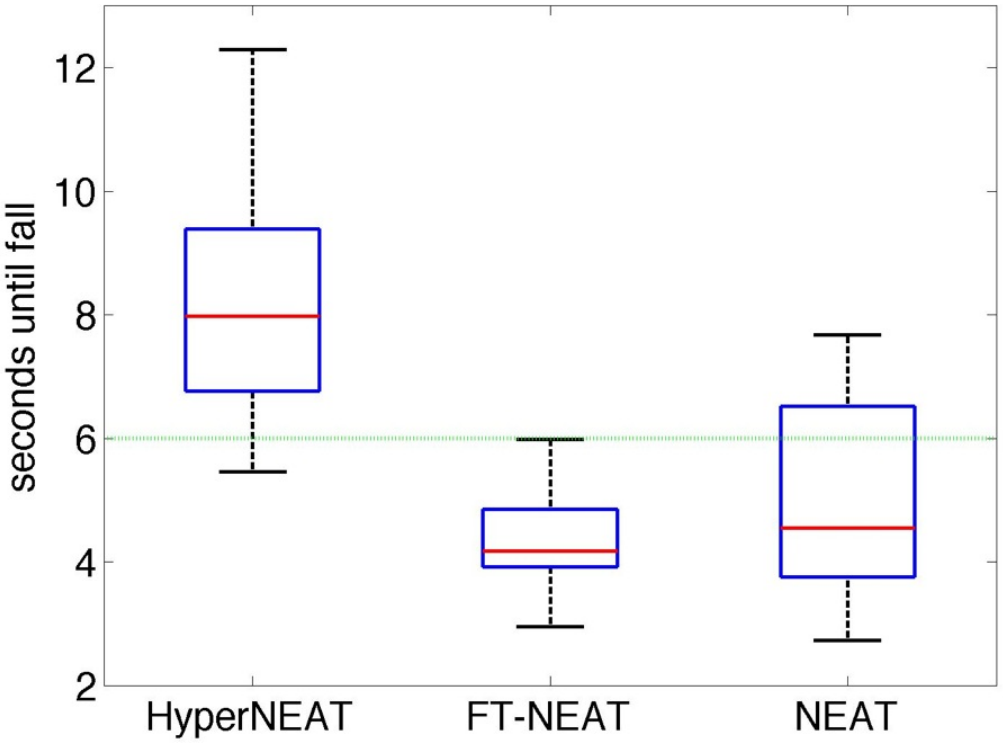

Figure 4.6: Gait Generalization. HyperNEAT gaits generalize better than FT-NEAT and NEAT gaits, which means that they run for longer before falling over. The allotted time during evolution experiments was 6 seconds (dashed horizontal line). Only the HyperNEAT gaits exceed that amount of time in these generalization tests. For clarity, outliers not shown.

4.5.2 HyperNEAT brains are more regular

I have so far focused on behavioral regularities. It is also interesting to look at the different brains (ANNs) produced by HyperNEAT and FT-NEAT. NEAT brains are not visualized because their varying numbers of hidden-node layers make comparisons difficult.

Figure 4.7 shows an example brain produced by HyperNEAT and FT-NEAT (all 50 visualizations for HyperNEAT and FT-NEAT brains can be viewed at http://devolab.msu.edu/SupportDocs/HyperNEAT). The FT-NEAT brain has no obvious regularity, and is hard to visually distinguish from an ANN with randomly generated link weights. Such irregularity existed even when the behavior produced by the brain was somewhat regular (e.g., the FT-NEAT brain depicted in Figure 4.7 was from the highest-performing FT-NEAT run, yet is irregular despite producing the relatively regular gait seen in Figure 4.5). In contrast, the HyperNEAT brain is highly regular and displays multiple different regularities. Initially, looking from the front, the links emanating from each input node repeat a pattern with inhibitory (red) links on top and excitatory (green) links below. Additionally, a clear distinction is made between the different layers of connections, at least on the top of the brain, where the input-to-hidden layer is inhibitory and the hidden-to-output layer is excitatory. An additional regularity is observable when viewing from the back in the links emanating from the outputs, where the nodes towards the bottom-left corner have a pattern of a few inhibitory links surrounded by excitatory links. Interestingly, this pattern is repeated in nearby nodes, but the strength of the pattern diffuses roughly with the distance from the bottom-left corner, such that eventually there are no remaining inhibitory links. The diffusion is faster in the y (height) dimension, and weaker in the x (width) dimension. This pattern shows that HyperNEAT is capable of producing complex geometric regularities.

A diverse set of regularities are observable in the 50 HyperNEAT champion ANNs. Some of this diversity is shown in Figure 4.8 and Figure 4.9. Left-right, top-bottom, and diagonal symmetries are produced. Additionally, exceptions (different patterns) are made

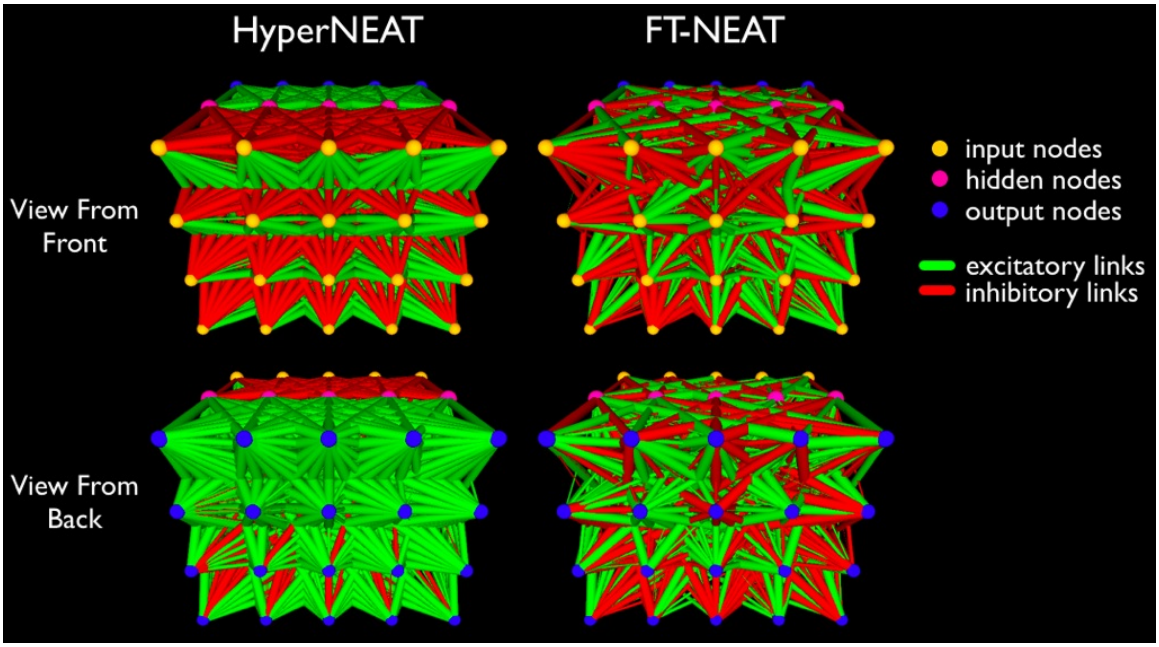

Figure 4.7: Example ANNs Produced by HyperNEAT and FT-NEAT. Views from the front (looking at the inputs) are shown in the top row, and from the back (looking at the outputs) are shown in the bottom row. The thickness of the link represents the magnitude of its weight.

for single columns, rows, and individual nodes. Some networks appear completely regular, with every viewable link having a similar value, and others feature complicated, yet still regular, patterns. Some networks have predominantly large weights (e.g., the top-left ANN in Figure 4.8), others have mainly small weights (e.g., the top-center ANN in Figure 4.8), and some have a broad range of weights. Importantly, many of these regularities include some variation. For example, the top-left ANN in Figure 4.8 has a motif that is repeated for each node, but that motif varies in regular ways (e.g., the number and strength of excitatory links emanating from nodes in the second column increases from bottom to top). Other variation is less regular, such as the exception made for a single node (Figure 4.8, bottom left), but even in this case the node itself contains a regular pattern.

This diversity of regularities demonstrates that HyperNEAT can explore many different types of regularities to solve a problem. It is also interesting that, while HyperNEAT is able to find many different regular solutions that perform and behave similarly, FT-NEAT is

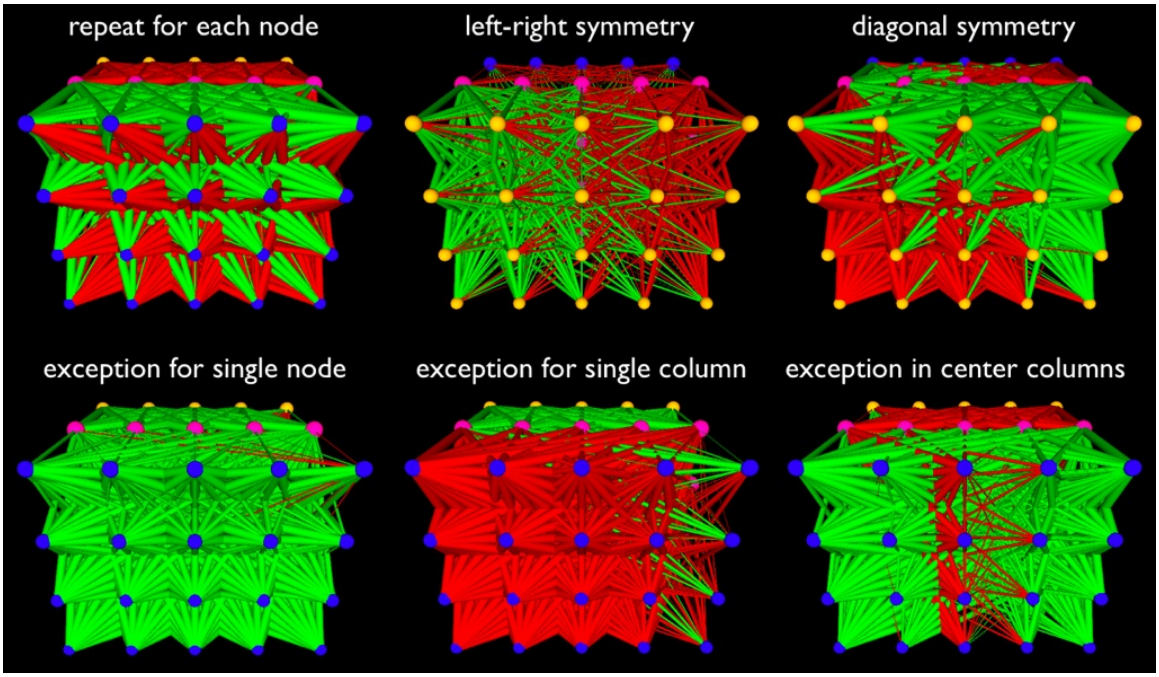

Figure 4.8: A Diverse Set of ANN Regularities Produced by HyperNEAT, with and without Variation. Figure 4.7 explains what different colors and line thicknesses represent.

unable to consistently discover *any* of these regular patterns. In other words, there are many solutions in the search space, but the regularity of those solutions means that a generative encoding frequently encounters them while the direct encoding rarely does.

As mentioned previously, HyperNEAT predominantly produces two different gaits on the Quadruped Controller problem: one with all legs in synchrony and the other with three legs in synchrony and the exception leg in opposite phase. It is sometimes possible to infer the behavior of a robot just by viewing a visualization of the regularities in its neural wiring. For example, ANNs that produce four-way symmetric gaits display similar wiring patterns for each row of outputs (Figure 4.9, top). Recall that each row of outputs controls a separate leg (Figure 4.2). For the 3-1 gaits in this experiment, the exception leg is always the front right leg, which is controlled by outputs in the top row. ANNs that produce 3-1 gaits frequently have a different pattern of weights for the top row of outputs than for the other outputs (Figure 4.9, bottom). There are no obvious differences in patterns emanating from input side of the ANNs (not shown). It is not always possible to classify gaits by

viewing ANNs alone, but in an informal experiment I made a correct prediction for most runs.

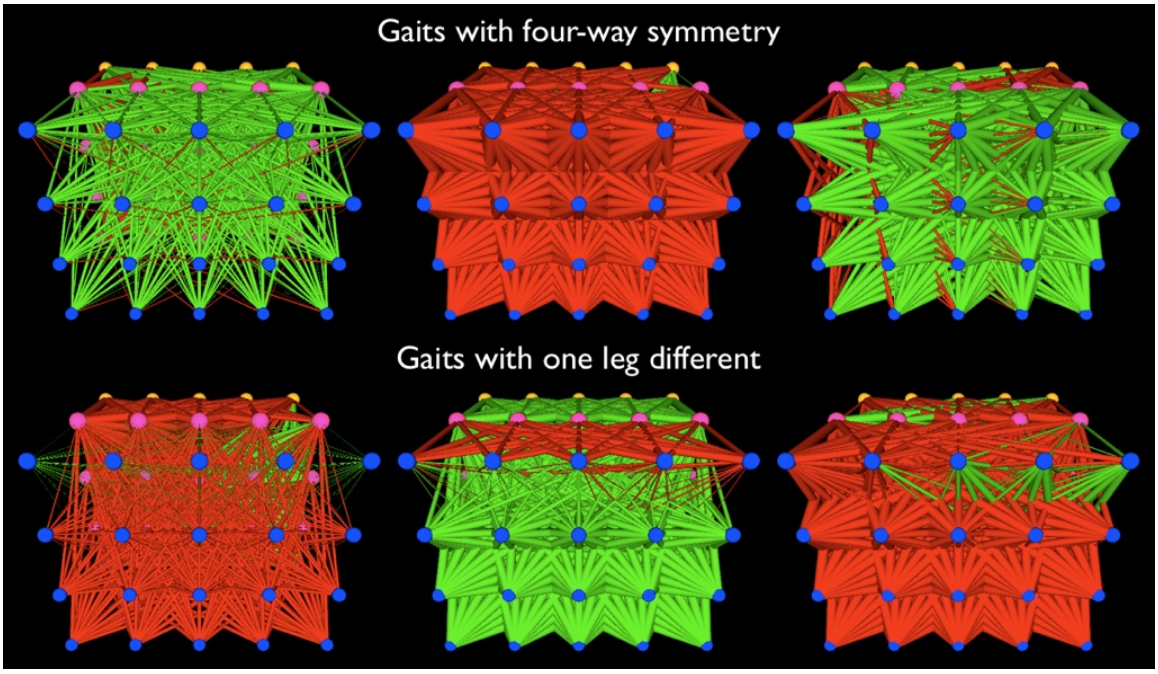

Figure 4.9: Correlating ANN Regularities to Different Behaviors. It is possible to recognize ANN patterns that produce different robotic gaits. The ANNs in the top row all generate a four-way symmetric gait. The weight patterns in these ANNs appear similar for all rows (legs are controlled by separate rows of nodes). The ANNs in the bottom row have three legs moving together and one leg in anti-phase. That exception leg is controlled by the nodes in the top row, which have a different pattern of weights than the other three rows.

It is interesting to note that in most output rows, patterns generalize to output nodes that do not control any aspect of the robot, and are therefore irrelevant to selection. Recall that in the output layer, only nodes in the first three columns (counting from the right when viewing from the back) control joints in the robot. The outputs in the last two columns are ignored (Figure 4.2). This design decision enables us to investigate whether the patterns that HyperNEAT evolves generalize to geometric areas that are not under selective pressure. In a direct encoding, one would expect that weights connected to nodes that are not under selective pressure would be random. Accordingly, these unused columns do appear random in the FT-NEAT ANN visualizations, but so do all of the columns (Figure 4.7). The patterns

for unused HyperNEAT output nodes, on the other hand, generally exhibit extensions of the patterns displayed across the entire network. This phenomenon occurrs in nearly every case, although a few networks had a different pattern in unused nodes than in other nodes within the same row (e.g., the bottom-right ANN in Figure 4.9).

This result suggests that if joints or legs were added to an evolved HyperNEAT ANN, HyperNEAT would likely produce similar behaviors in those new joints and legs as elsewhere in the robot. If this hypothesis is correct, HyperNEAT would also succeed more consistently than a direct encoding when evolving a pre-evolved ANN further to control additional legs or joints, provided that it is beneficial to have behavioral similarities between the original and newly added components. This prediction assumes that the new components are placed at geometric coordinates such that an appropriate regularity applies. This general idea has already been demonstrated to work in a domain where HyperNEAT evolved ANNs to control multiple agents of a team [9]. Such an ability to transfer skills from one task to a new task is an important goal of machine learning [50] that a generative encoding like HyperNEAT can potentially perform well at [53], but where a direct encoding is likely to perform poorly.

The previous visualizations demonstrate that HyperNEAT ANNs are regular, but that conclusion remains a subjective one. It would be better to also quantify the increased regularity in HyperNEAT ANNs versus those of FT-NEAT. Because the more regular a structure is, the less information is needed to describe it, regularity can be measured by compression [33]. One test of the regularity of a structure, then, is how much it can be compressed. I applied the standard unix gzip compression program (version 1.3.5) to text files that contained only the link values of each ANN phenotype for HyperNEAT and FT-NEAT. NEAT ANNs were not compared because the number of links in NEAT ANNs can vary, meaning that compressibility alone is not isolated. I performed this analysis only on the Quadruped Controller problem because it did not have regularity explicitly built in, as opposed to Target Weights and Bit Mirroring, where fitness scores already indicate ANN

regularity.

The gzip algorithm is a conservative test of regularity because it looks for repeated symbols, but does not compress some mathematical regularities (e.g., each link weight increasing by a constant amount). Nevertheless, gzip was able to compress HyperNEAT ANNs on the regular Quadruped Controller problem significantly more than FT-NEAT ANNs: $p < 0.001$, comparing the difference between each uncompressed and compressed HyperNEAT file, mean 4488 bytes $\pm$710 SD, and each FT-NEAT file, mean 3349 bytes $\pm$37 SD. This quantifiable result confirms the clear yet subjective observation from visually inspecting HyperNEAT and FT-NEAT brains, namely, that HyperNEAT brains are more regular.

4.5.3 HyperNEAT is more evolvable

One of the touted benefits of generative encodings is that the reuse of genetic information that produces regularity also enables coordinated mutational effects, which can be beneficial [4, 25]. It has previously been shown that a different generative encoding based on L-systems produces more beneficial mutations than a direct encoding control [25]. This section investigates whether HyperNEAT similarly tends to produce a higher distribution of fitness values in mutated offspring than direct encoding controls.

I analyzed the difference in fitness between organisms and their offspring in all cases where offspring were produced solely by mutation. While the majority of organisms were produced by crossover and mutation, this analysis isolates the impact of mutational effects. Over 1.3 million, 1.7 million, and 1.5 million organisms were produced solely via mutation for HyperNEAT, FT-NEAT, and NEAT treatments, respectively, providing a substantial sample size.

Overall, the generative encoding HyperNEAT produced a wider *range* of fitness changes than direct encoding controls (Figure 4.10). HyperNEAT also had a distribution of fitness values with a higher median than both FT-NEAT and NEAT ($p < 0.001$). While

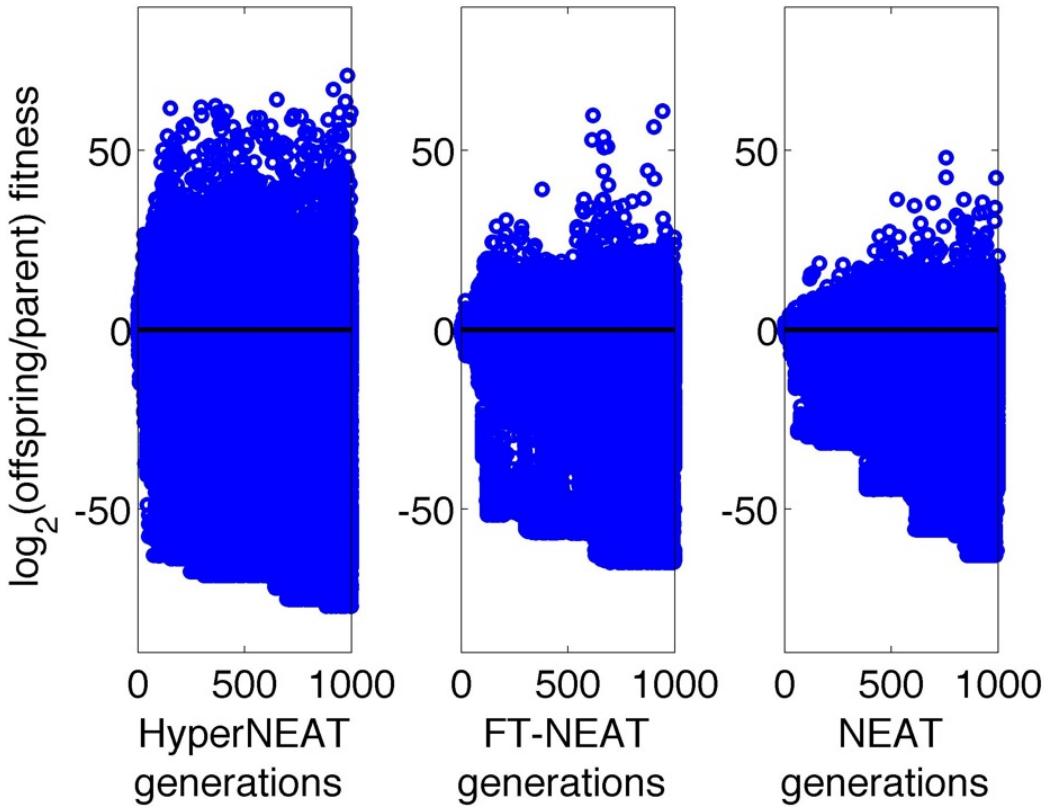

Figure 4.10: The Fitness Changes Due to Mutation in Different Encodings. Each circle represents the ratio of parent fitness over offspring fitness. Positive values indicate an offspring that is more fit than its parents, and higher numbers indicate larger fitness improvements. The inverse is true for negative numbers.

HyperNEAT had more extreme negative fitness changes, it also had more extreme positive fitness changes. For example, looking at the ratio of parent fitness over offspring fitness, 5.8% of HyperNEAT offspring had a positive value greater than 20, whereas for FT-NEAT and NEAT only 0.35% and 0.21% did, respectively (Figure 4.10). It should be noted, however, that many of these large mutational effects could be the result of a high-fitness organism suffering a mutation that makes it immobile one generation and then a compensatory mutation that approximately restores its original fitness in the next generation.

Despite the many extreme negative fitness changes, it appears that the continuous production of organisms that were much more fit than their parents fueled the success of Hyper-

NEAT over the direct encodings on this problem. That the distribution of mutations alone contributed to the success of HyperNEAT over FT-NEAT is supported by the fact that in an alternate experiment where only mutations generated offspring, HyperNEAT still outperformed FT-NEAT, and a plot of mutational effect scores looked similar to Figure 4.10 (data not shown). The magnitude of mutational effects in Figure 4.10 does change with time in both treatments because those mutant offspring that barely move from the starting place will count as a smaller fraction of the more fit parents of later generations.

4.6 Discussion and conclusions

Both objective and subjective analyses reveal that the HyperNEAT generative encoding is better at evolving quadruped controllers than direct encoding controls when the regularity of the problem is high. I demonstrated that the success of HyperNEAT arises because it produces regular brains (ANNs) and behaviors (gaits). HyperNEAT also produces a diverse set of neural wiring patterns, showing the algorithm's ability to generate many different, yet complex, patterns. Mutations to these regular patterns tended to produce more extreme fitness effects, presumably due to the coordinated and holistic nature of the mutational effects, allowing evolution to spend more time testing coordinated gaits. Additionally, the gaits produced by the generative encoding generalized better due to their increased amount of regularity. These results, along with those in Chapter 3, serve as another demonstration that generative encodings can outperform direct encodings on regular problems [19,25,46].

A second reason for HyperNEAT's success may be its unique ability to exploit geometric aspects of the problem, such as symmetry [16, 17, 46]. During this research, many symmetries were observed in HyperNEAT gaits, including four-way symmetry (all legs in unison), left-right symmetry, and front-back symmetry. Chapter 6 investigates this issue in more depth.

The results from the Quadruped Controller problem also reaffirmed conclusions drawn

from Target Weights and Bit Mirroring regarding how generative encodings compare to direct encodings as problem regularity decreases. As the regularity of the Quadruped Controller problem lessened, the performance of the generative encoding dropped to, and then fell below, the direct encoding controls. This result demonstrates that generative encodings can exploit intermediate amounts of problem regularity, and may not be the correct choice when a problem is sufficiently irregular. Additionally, the regularity of the problem did not impact HyperNEAT once it was below a certain level.

HyperNEAT has yet to be tested on a wide range of problems. To date, however, it has performed well on tasks as diverse as visual recognition [46], controlling simple multi-agent systems [11], and evaluating checkers boards [16, 17]. Additionally, the CPPNs underlying HyperNEAT produce complex, elegant, natural-looking images with symmetry and repetition [42, 45]. All of these accomplishments suggested that HyperNEAT would excel at evolving controllers for legged robots. The results in this chapter confirm that ability. HyperNEAT's success at evolving gaits for legged controllers is all the more impressive because it was able do so without the problem being manually decomposed or simplified. The encoding could handle the complexities of the entire problem because it discovered how to exploit the regularities of the problem. Based on the success reported here, which comes despite naive geometric inputs, it is not unreasonable to predict that HyperNEAT, and variations of it, will contribute significantly to the science of automating the creation of controllers for complex, yet regular, devices, such as legged robots. Moreover, the results from the Quadruped Controller problem suggest that HyperNEAT and future generative encodings based on it will be able to automatically exploit the regularities of complex, real-world engineering problems.

Chapter 5

A generative encoding can struggle to make exceptions to the rules it discovers: One remedy is an algorithm called HybrID

Note:This Chapter is an expanded version of Clune, Beckmann, Pennock, and Ofria 2009.

5.1 Motivation, overview, and background

The Target Weights and Bit Mirroring experiments both demonstrate that the HyperNEAT generative encoding performs worse as the regularity of the problem decreases. This outcome is, in part, because it is challenging for HyperNEAT to create exceptions to the rules it discovers. While we saw that HyperNEAT can make exceptions for columns or nodes (Chapter 4), even these exceptions are regular in nature. The results from the Target Weights problem (Chapter 3), however, clearly demonstrate that HyperNEAT has difficulty making changes to specific links in an irregular way. The most glaring example of this

deficit comes from the version of Target Weights where 90% of the target links were one value, and the remaining 10% of the target links were each a different random number. HyperNEAT discovered the regularity within a few generations, and then was unable to create irregular exceptions that would have produced fitness gains during the remaining hundreds of generations (Figure 3.5). The inability to make irregular exceptions to rules is likely a general problem with generative encodings, because they are biased towards the production of regular phenotypes. Direct encodings (also called *indirect encodings*), on the other hand, excel at producing irregularities, but cannot easily create regularities.

In this section, I propose a new algorithm that is a Hybridization of Indirect and Direct encodings (HybrID), which combines the benefits of both encodings. Although I present one specific implementation of HybrID, the term applies to any combination of generative and direct encodings. While I am not aware of any prior work that specifically combines direct and generative encodings, researchers have previously altered representations during evolutionary search, primarily to change the precision of values being evolved by genetic algorithms [12]. Other researchers have employed non-evolutionary optimization techniques to fine-tune the details of evolved solutions [18]. However, such techniques do not leverage the benefits of generative encodings.

I compare HyperNEAT to HybrID on the Target Weights, Bit Mirroring and Quadruped Controller problems, all of which have scalable regularity (see previous chapters for descriptions of these problems). I find that HybrID can improve performance over HyperNEAT by as much as 64%. These results recommend HybrID as a type of evolutionary algorithm that can both exploit a problem's regularities and account for its irregularities. The results also further demonstrate that HyperNEAT has difficulty making exceptions, since adding a process that adjusts the patterns HyperNEAT produces in irregular ways leads to substantial fitness gains.

5.2 The HybrID algorithm

The HybrID algorithm presented in this chapter runs HyperNEAT for a fixed number of generations and then the encoding is changed to FT-NEAT at a *switch point* (Figure 5.1). To switch, I transfer the ANN phenotypes of each individual in the HyperNEAT population to FT-NEAT genomes that are then further evolved with FT-NEAT. In Section 4 of this chapter I describe alternate HybrID instantiations. The motivation for the switch-HybrID algorithm is that the generative encoding phase will generate patterns that the direct encoding phase can then adjust to account for problem-irregularities.

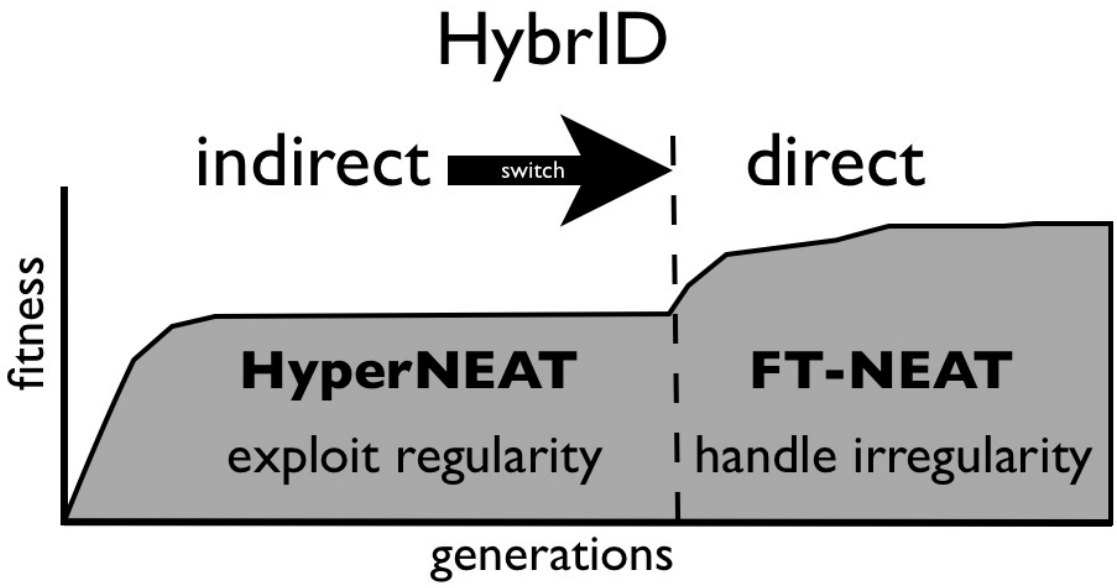

Figure 5.1: Hybridizing Indirect and Direct Encodings in the HybrID Algorithm. The HybrID implementation in this paper evolves with HyperNEAT in the first phase until a switch is made to FT-NEAT. The idea is that the generative (indirect) encoding phase can produce regular weight patterns that can exploit problem regularity, and the direct encoding phase can fine tune that pattern to account for problem irregularities. In this hypothetical example, large fitness gains are initially made by the generative encoding because it exploits problem regularity, but improvement slows because the generative encoding cannot adjust its regular patterns to handle irregularities in the problem. Fitness increases again, however, once the direct encoding begins to fine-tune the regular structure produced by the generative encoding.

5.3 Comparing HyperNEAT to HybrID: Results from the Target Weights, Bit Mirroring, and Quadruped Controller problems

I conducted 50 runs of each experimental treatment in this section, and all data plotted are averaged across them. The mutation rate per link was 0.08 for HyperNEAT and 0.0008 for FT-NEAT; preliminary experiments revealed these mutation rates to be effective for each encoding. FT-NEAT has a lower per-link mutation rate because its genome has many more mutational targets than HyperNEAT. Additional experiments showed no statistically significant increase in HyperNEAT's performance on all three problems when its mutation rate was dropped to 0.0008 at the switch point.

5.3.1 The Target Weights problem

I begin my analysis with the diagnostic problem of evolving a specific target ANN (see Chapter 3.2 for a full description). As previously discussed, HyperNEAT quickly discovered the regularity in the more regular versions of this problem, but had difficulty making exceptions to account for irregularities, even after hundreds of generations (Figure 3.5). FT-NEAT, on the other hand, was slower, but eventually performed well, in part because the problem has no epistatic interactions and thus coordinated mutational effects are not required (Figure 3.5). HybrID combined the best attributes of both encodings: it quickly discovered the regularity of the problem and, after the encoding switch at generation 100, was able to further optimize solutions by accounting for irregularities (Figure 5.2). While HybrID and FT-NEAT eventually evolved solutions of similar quality, early on HybrID did better on more regular problems and less well on less regular problems. HybrID significantly outperformed both HyperNEAT and FT-NEAT at generation 250 on the 70%, 80%, and 90% regular problems ($p < 0.01$). Earlier switch points further improved the speed at

which HybrID made progress on this problem (data not shown).

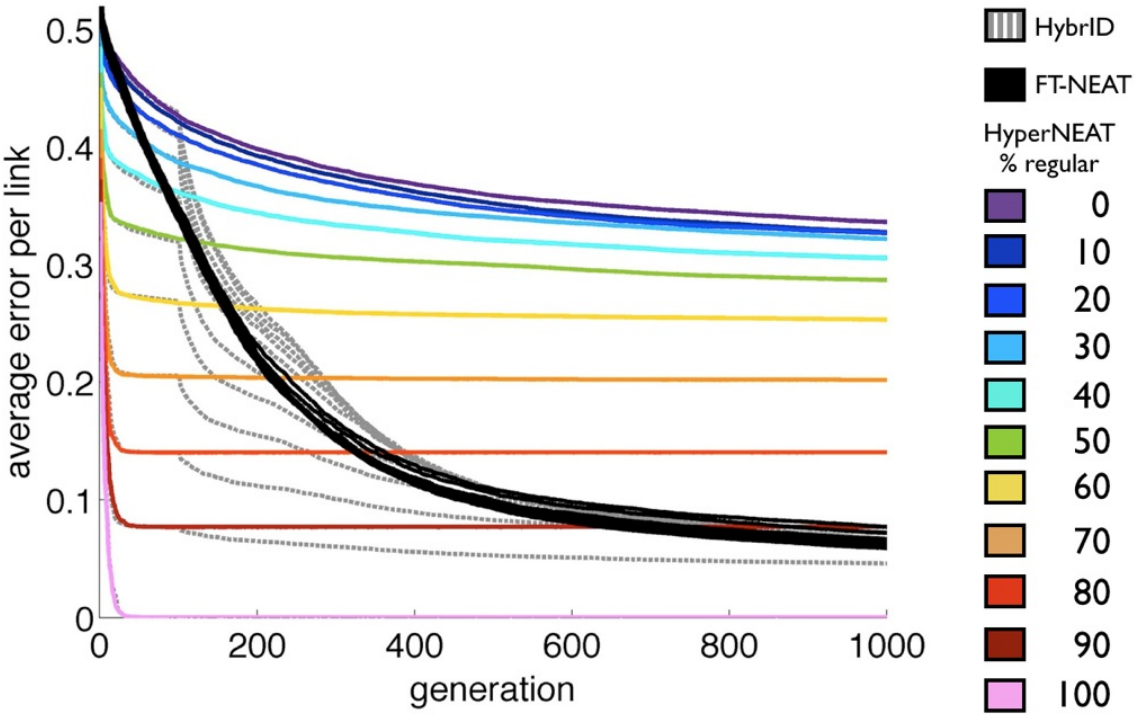

Figure 5.2: A Comparison of HyperNEAT, FT-NEAT, and HybrID on a Range of Problem Regularities for the Target Weights Problem. For each regularity level, a HybrID line (gray) departs from the corresponding HyperNEAT line (colored) at the switch point (generation 100). The performance of FT-NEAT (black lines) is unaffected by the regularity of the problem, which is why the lines are overlaid and indistinguishable. HybrID outperforms HyperNEAT and FT-NEAT in early generations on versions of the problem that are mostly regular but have some irregularities.

5.3.2 The Bit Mirroring problem

The next problem, called *Bit Mirroring* (see Chapter 3.1 for a full description), is more challenging and realistic since it has epistasis. Because I have already shown that HyperNEAT outperforms FT-NEAT on all versions of this problem (Figure 3.2), here I compare HybrID only to HyperNEAT. A population of size 500 was evolved for 5000 generations and the switch point for HybrID was at generation 2500.

The results reveal that HybrID ties HyperNEAT on the most regular versions of this

problem, and provides a significant fitness improvement over HyperNEAT on all versions of the problem that have a certain amount of irregularity (Figure 5.3). HybrID's advantage over HyperNEAT was largest on problems of intermediate regularity. The reason the gap in performance narrowed on the most irregular treatments is because the problem is difficult and both algorithms performed poorly.

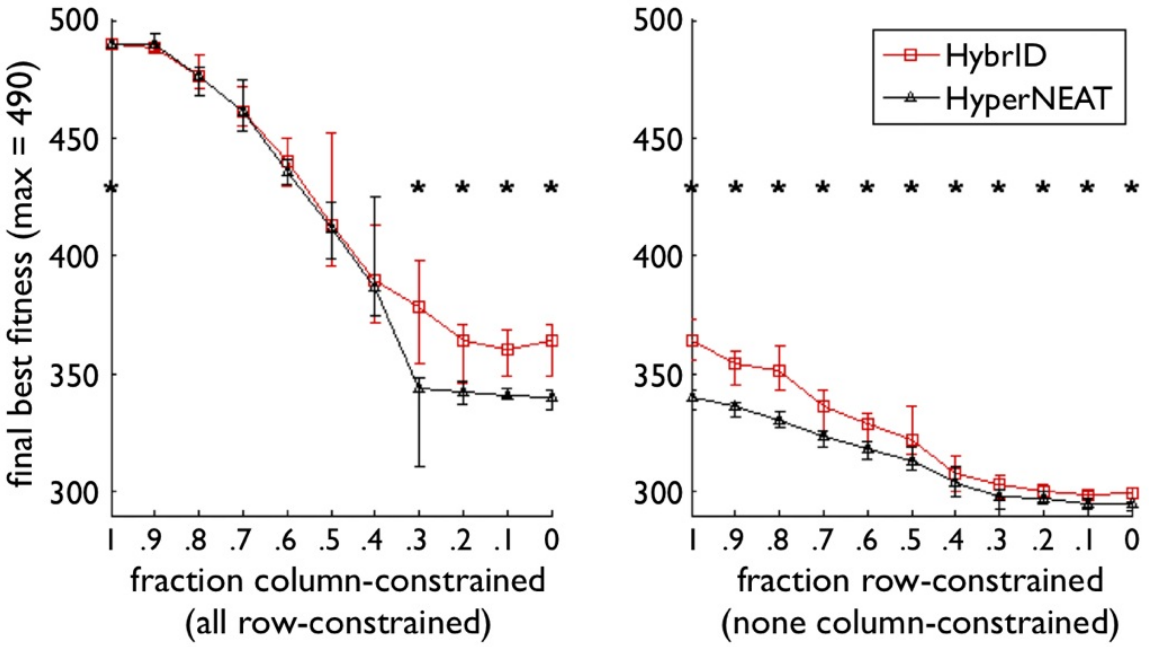

Figure 5.3: The Performance of HybrID Vs. HyperNEAT on the Bit Mirroring Problem. Regularity decreases from left to right. Plotted are median values $\pm$ the 25^{th} and 75^{th} quartiles. Asterisks indicate $p < 0.05$.

5.3.3 The Quadruped Controller problem

HyperNEAT was shown to evolve fast, natural gaits for simulated legged robots (Chapter 4). The evolved gaits, however, were extremely coordinated, with all legs often moving in near perfect synchrony (either in phase or in opposite phase). I tested HybrID on this problem to determine if it would improve fitness by facilitating the fine-tuning of aspects of the controller, such as the movements of individual legs, especially on more irregular versions of the problem (those with more faulty joints, see Chapter 4.3). I repeated the

experiment from Chapter 4 with HyperNEAT and HybrID. Experiments had a population of 150, lasted 1000 generations, and had a switch point at generation 500.

HybrID outperformed HyperNEAT on every version of the Quadruped Problem (Figure 5.4), although that difference was significant only on problems with a certain amount of irregularity ($p < 0.01$ on treatments with four or more faulty joints). HybrID increased performance over HyperNEAT by 5%, 10%, 27%, 64%, and 44%, respectively, for the treatments with 0, 1, 4, 8, and 12 faulty joints. These substantial performance improvements on the Quadruped Problem, which is a challenging engineering problem, highlight the degree to which HyperNEAT's inability to produce irregularity on its own can harm its performance. The results also demonstrate the extent to which HybrID can increase performance when it is allowed to fine-tune the regularities produced by HyperNEAT.

HybrID also outperformed both direct encoding controls on all treatments of the problem ($p < 0.05$). As reported in Chapter 4, HyperNEAT significantly outperformed both direct encodings on only the two most regular versions of the problem ($p < 0.01$). That HybrID beats the direct encodings on irregular problems underscores that it does not just act like a direct encoding on irregular problems, but instead first leverages HyperNEAT's ability to exploit available regularities and then improves upon those by accounting for problem irregularities via FT-NEAT.

It is instructive to look at how the FT-NEAT phase of HybrID changes the patterns provided to it by the HyperNEAT phase. Visualizing ANNs at the end of each HyperNEAT phase and the ANN for that same run after the FT-NEAT phase provides clues as to how HybrID generates its performance improvements (Figure 5.5). Interestingly, in all cases the FT-NEAT phase of HybrID made no major changes to the overall regular pattern produced by the HyperNEAT phase (visualizations of ANNs after the HyperNEAT and FT-NEAT phases for all 50 HybrID runs can be seen at http://devolab.msu.edu/SupportDocs/HyperNEAT). Natural selection thus maintained the regular pattern HyperNEAT generated even while that pattern was being fine-tuned by the

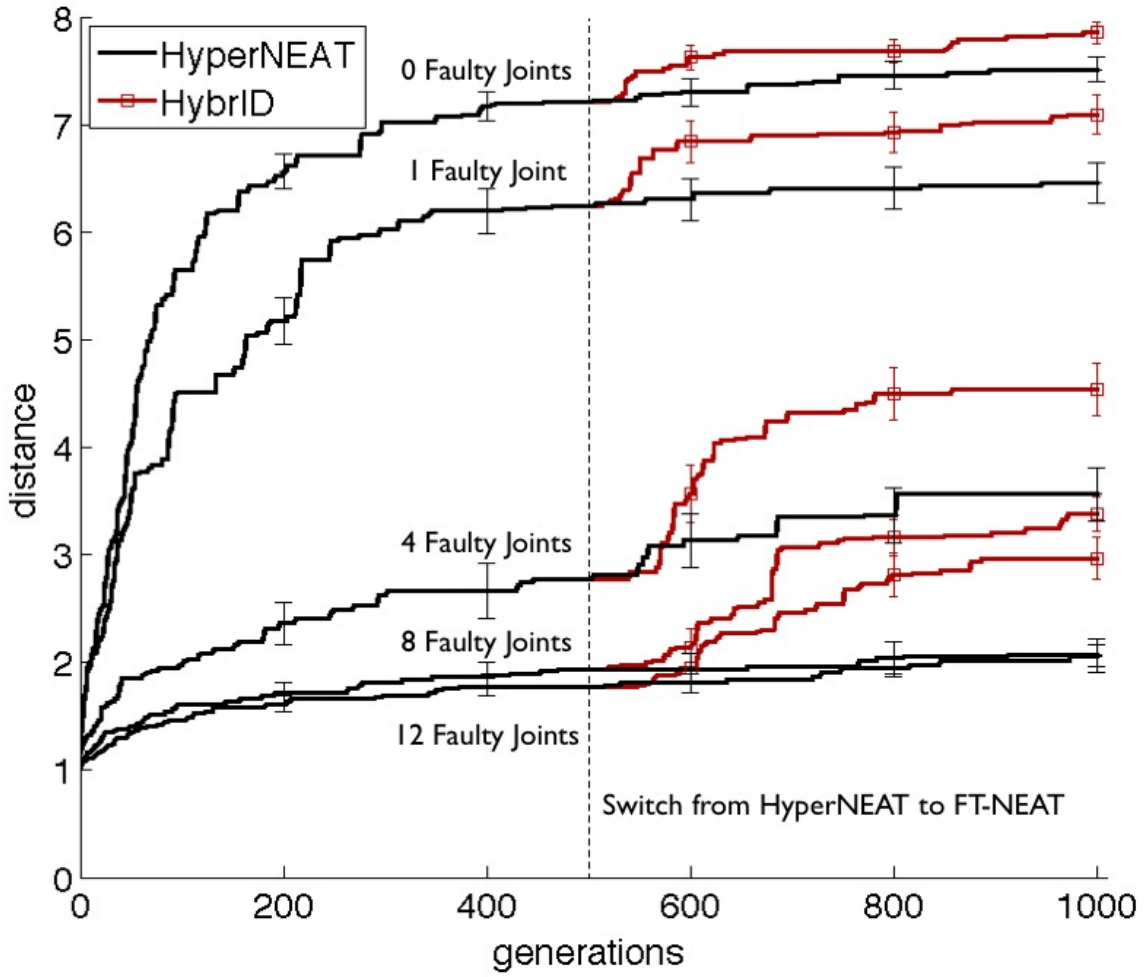

Figure 5.4: The Performance of HybrID Vs. HyperNEAT on the Quadruped Controller Problem. Error bars show one standard error of the median. HybrID outperforms HyperNEAT on all versions of the Quadruped Controller problem. The increase generally correlates with the number of faulty joints.

direct encoding.

The types of exceptions HybrID produces are different from those made by HyperNEAT alone. In many cases, only a few weights are noticeably changed by the FT-NEAT phase of HybrID, and these changes occur in an irregular distribution. For example, in the run depicted in the left column of Figure 5.5, HyperNEAT produces the regular pattern of inhibitory connections to all of the output nodes. FT-NEAT switches some of those to excitatory connections, which may have been difficult for HyperNEAT to do without changing many other weights. In another example run, depicted in the middle column of Figure 5.5,

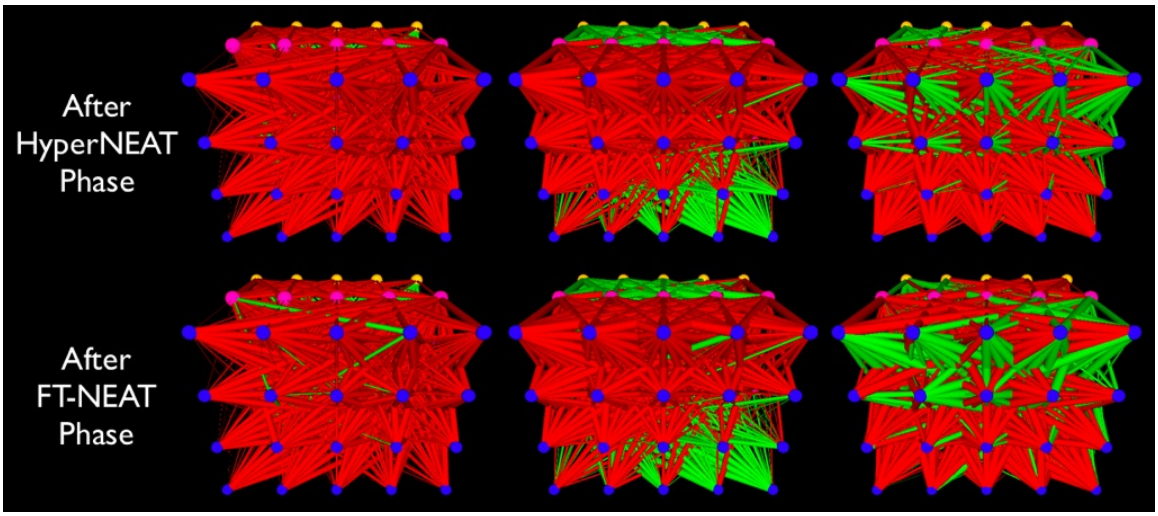

Figure 5.5: Visualizations of the ANNs Produced at the End of the HyperNEAT Phase and the FT-NEAT Phase of HybrID for Three Example Runs.

the only noticeable change FT-NEAT made is the creation of a single, strong, excitatory connection. Of course, in both cases there are subtle changes in many of the link weights that do not stand out to the human eye.

In a third example run, changes were made during the FT-NEAT phase to many different link values, yet the overall pattern remained (Figure 5.5, right column). Many of these changes were irregular, such as the links switched from excitatory to inhibitory and vice versa in the top-left node, and the few links that were made excitatory in the bottom row. What is unusual and fascinating about this run, however, is that the direct encoding made many *regular* changes. For example, most of the links in the top row proportionally increased in strength, which preserved the regular patterns. These visualizations show a rare case of a direct encoding producing coordinated phenotypic changes. It might be the case that the generative encoding discovered regularities that put this organism on the side of a hill in a fitness landscape, but climbing that hill was made difficult because mutations to the genome that increased the strength of connections in these nodes may have changed other link values and had a net negative effect on fitness. The direct encoding control does not have such constraints, and thus could increase the magnitude of all of these links. This

hypothetical explanation illustrates how natural selection can produce coordinated changes in direct encodings if it starts in a place in the fitness landscape where there is a positive fitness gradient in the same direction for many link values. Interestingly, it is unlikely that the direct encoding would have discovered this starting point without the generative encoding.

5.4 Alternate HybrID instantiations

I presented only one of many possible HybrID implementations. The HybrID in this section evolves first with a generative encoding then switches to a direct encoding, and could be called a *switch-HybrID*. Another candidate HybrID implementation would have the generative encoding produce a set of values (e.g., link weights) and the direct encoding evolve a set of offsets that modify the individual values. This *offset-Hybrid* would allow exceptions to be made while the generative encoding is still evolving. Another possibility is a *chooseOne-HybrID*, where a mutable bit can be flipped by evolution to determine whether a link value is taken from a generative encoding or a direct encoding, both of which are evolving in parallel. HybrIDs can also be made with other generative encodings, and in domains besides neuroevolution. Additionally, instead of occurring at a predefined time, the switch from generative to direct encodings, or the addition of offsets, could occur automatically after fitness has stagnated. Future investigations are required to test the efficacy of different HybrID implementations.

5.5 What HybrID teaches us about generative encodings

The work in this chapter suggests that it is difficult for generative encodings to simultaneously discover the regularity of problems and make exceptions to account for problem idiosyncrasies. At a minimum, HyperNEAT exhibits this deficiency in its present form. Theoretically, HyperNEAT should be able to make any exception required, but in practice it frequently does not.

Visualizations of the HyperNEAT ANNs revealed that HyperNEAT can create variations on patterns, and even exceptions for single nodes. However, these variations and exceptions themselves were regular, suggesting that HyperNEAT creates its exceptions by adding one regularity to another, with the result being an overall regularity with a regular variation within it. The visualizations rarely demonstrate cases where single link values were different from the prevailing pattern (Chapter 4.5.2). HybrID, on the other hand, did provide examples of such single-link exceptions (Figure 5.5). The significant boost in performance by HybrID implies that the ability to make such radical exceptions at the single-link level is important.

HybrID's performance increase over HyperNEAT, along with investigations into the different type of exceptions HybrID and HyperNEAT make, suggest that HyperNEAT can benefit from a process of refinement that adjusts individual link patterns in an irregular way. The success of this approach suggests that generative encodings may be most effective not as stand-alone algorithms, but in combination with a refining process that adjusts regular patterns in an irregular way to account for problem irregularities. While a direct encoding provides such refinement in this chapter, there are other candidate refinement processes. One intriguing possibility is that lifetime adaptation via learning can play a similar role. Lifetime learning algorithms could adjust the overall regular patterns produced by HyperNEAT to account for necessary irregularities.

It is likely that other generative encodings have similar problems to HyperNEAT with respect to accounting for problem irregularities, although additional research is required to determine the degree to which that is the case. It is an open challenge for the field to improve current generative encodings to enable the encoding of both regularities and exceptions to those regularities.

5.6 Conclusions

Many real world problems have regularities but also require exceptions to be made. It is important for evolutionary algorithms to both exploit such regularity in problems and account for their irregularities. I have shown that HybrID, a combination of generative and direct encodings, accomplishes this goal by first discovering the regularity inherent in a problem and then accounting for its irregularities. I validated the algorithm on two simpler test problems and on a more challenging, real-world problem. HybrID frequently outperformed HyperNEAT, sometimes by as much as 64%. While further research is needed to see how HybrID works with other pairs of generative and direct encodings, alternate HybrID implementations, and on additional problems, these preliminary results suggest that HybrID is an effective algorithm for evolving solutions to complex problems. The HybrID algorithm also reveals current deficiencies with HyperNEAT, which, in turn, raises possible ways to remedy HyperNEAT. The remedy this work recommends is a further process of refinement that adjusts the regularities produced by HyperNEAT to account for problem irregularities. This chapter therefore suggests a way forward for generative encodings wherein their main contribution is as the regularity-generating engine within a larger algorithm.

Chapter 6

A generative encoding can be sensitive to different geometric representations of a problem

Note: This chapter is an expanded version of Clune, Ofria, and Pennock 2009.

6.1 Motivation for studying a generative encoding's geometric sensitivity

Many problems tackled by the field of artificial intelligence have geometric regularities that are intuitively obvious to a human observer. For example, in the game of checkers the geometric concept of adjacency is important because pieces close together are likely to constrain each other. The edge squares on the top and bottom of the board are different because they can confer kingship, and all of the edge squares are important because a piece on them cannot be jumped. There are also symmetries to the game (e.g., left-right, top-bottom, and rotational). Many other AI problems, from machine vision to robotic control, also contain a plethora of geometric information. Although geometric information could

be helpful in solving these types of problems, most evolutionary algorithms do not make such geometric information available to be exploited [46]. Instead, the norm is to provide sensory information, such as whether a piece is present in a square, without also specifying the geometric coordinates associated with that square. Stripping out such information is akin to asking a human to learn to play checkers by cutting up the board into its constituent pieces and scattering them randomly on the floor [46].

The HyperNEAT generative encoding does provide geometric information to an evolutionary algorithm, and its ability to exploit the geometry of a problem has been repeatedly demonstrated (Chapter 4, Chapter 3) [11, 16, 17, 46]. The ability to inject geometric information into an evolutionary algorithm, however, necessitates that the experimenter choose how to represent that information. This requirement raises the question of how sensitive that EA is to different geometric representations of the same problem. There are aspects of some problems that have obvious geometrical representations (e.g., the Cartesian coordinate system of checkers), but other aspects of those problems may have no obvious geometric location. For example, at what geometric coordinates should the input for the current number of black pieces be placed? Other problems have many seemingly good geometric representations. For example, there are multiple ways to order the legs of a quadruped robot in two dimensions, and there are pluses and minuses for each alternative, as we will see below.

One possibility is that encodings that exploit geometry, such as HyperNEAT, are mostly immune to variation in the geometric representation. Such insensitivity would eliminate the need to spend time trying to select the most advantageous representation. This possibility would also be surprising, however, because preserving meaningful correlations in the problem's representation should aid HyperNEAT's performance. At the other extreme, it could be the case that HyperNEAT exhibits vastly different performance levels in response to even small changes in the geometric representation of a problem. If the location of such information does matter in algorithms that incorporate geometric information like Hyper-

NEAT, then users of these algorithms should be aware of that fact so they know to try a variety of configurations.

This chapter will investigate HyperNEAT's sensitivity to geometric representations by performing repeated tests of the same problem (with the same inputs and outputs), but with different representations of the geometric information. The investigation will use the Quadruped Controller problem from Chapter 4, which is appropriate for this study for two reasons. Initially, some aspects of the problem, such as the ordering of joints in each leg, have clear geometric relationships, yet other aspects of the problem do not. For example, each of the knee joints should be geometrically represented as further from the torso than the hip joints, but other inputs (e.g., the sine wave) have no obvious geometric location. Secondly, I have already shown that HyperNEAT is successful on this task with one specific geometric representation (Chapter 4), providing a baseline performance that can be compared to the performance of other geometric representations.

6.2 Previous work on this subject

I am aware of only one algorithm besides HyperNEAT that injects geometric information into an evolutionary algorithm. The *simple geometry-oriented cellular encoding* (SGOCE) is a generative encoding that evolves ANNs that control legged robots [30]. Controllers were successfully evolved with SGOCE that could walk, follow gradients, and avoid obstacles. Unfortunately, the benefit of the geometric information in the SGOCE system was not isolated and investigated. While the system as a whole performed well on the legged-robot problem, alternate geometric configurations were not tested. Additionally, the cellular-encoding method that SGOCE was based on had been previously shown to perform well on the problem of evolving controllers for legged robots without the addition of geometry [19]. Finally, the robot model was relatively simple, having only two degrees of freedom, the controller was manually decomposed into modules by the experimenter,

and staged evolution was necessary achieve high level objectives, all of which simplify the problem, but complicate analyses, compared to the robot model in the Quadruped Controller problem.

HyperNEAT has been demonstrated to exploit geometric information on a variety of tasks. It was shown to discover numerous different geometric motifs that it repeated to solve a visual discrimination task [46]. Its ability to exploit geometry was also verified in the checkers domain, where it evaluated board configurations [16, 17]. HyperNEAT also exploited the geometric representation of a team of agents to produce herding strategies [11]. For example, left-right symmetries were discovered wherein the left and right groups of agents would perform strategies that were mirror images of each other. I have also shown that HyperNEAT can exploit geometric information to produce symmetries on the Quadruped Controller problem (Chapter 4).

Only one published experiment tested HyperNEAT's sensitivity to alternate geometric representations of the same problem [46]. A simulated circular robot was rewarded for finding food. Eight food sensors and eight motor outputs were evenly distributed around the robot's center. The robot could move in the direction of any of those food sensors by activating the corresponding motor output. Two geometric configurations were tested: a *parallel* Cartesian coordinate configuration, where corresponding food sensors and motor outputs were labeled with the same x coordinates, but different y coordinates (i.e., they were in the same row, but in different columns), and a *concentric* configuration, with a polar coordinate system where corresponding food and motor nodes shared the same angle from 0 degrees. Unless extra information about the distance between nodes was provided, the parallel configuration evolved strategies that collected food faster than the concentric configuration. Even though only one experiment was conducted, and it compared only two alternate geometric representations, the fact that the geometric representation made a difference is interesting and invites a more rigorous study of HyperNEAT's sensitivity to geometric representations. Such a study is conducted in this chapter.

6.3 Experiments, results and discussion

6.3.1 Engineered versus random configurations

A test of the importance of choosing an appropriate geometric representation is to compare a human-engineered representation (Figure 4.2) to randomized representations. Such random configurations represent configurations created without intuitions about how to represent the geometric information of a problem, and could be produced by a naive engineer or algorithm. Each random configuration has the geometric locations of the inputs and outputs scrambled within their layer. For each trial, the geometric representation was randomized at the beginning of a trial and remained unchanged throughout the trial. For example, the sine input, which is located at $x = 5$, $y = 4$ in the engineered treatment (Figure 4.2), may be at (1, 1) in one randomized treatment and (3, 2) in another. An average was calculated across 50 trials, each of which had a different randomized configuration.

In this case, the human-engineered configuration significantly outperformed the average of the random configurations (Figure 6.1, $p < 0.05$). This performance difference shows that human intuitions about how to geometrically represent a problem can help HyperNEAT. These results also underscore that the performance of HyperNEAT can be significantly affected by the geometric representation. It is also instructive to compare the randomized treatment to FT-NEAT. It has been previously shown that HyperNEAT (with the engineered configuration) outperforms FT-NEAT (Chapter 4), so it is interesting to test whether a naive geometric representation lowers the performance of HyperNEAT to the level of FT-NEAT. It turns out that HyperNEAT still performs better than FT-NEAT (Figure 6.1), even with a randomized configuration ($p < 0.001$). This advantage could be due to HyperNEAT's generative ability to reuse link values, or to its ability to exploit geometric correlations that arise by chance in randomized configurations. Regardless of the reason, it is noteworthy that HyperNEAT outperforms FT-NEAT even with a naive representation of the geometric information. Unsurprisingly, FT-NEAT is not affected by a randomization of

the geometry, which for FT-NEAT changes only the ordering of the values in its encoding ($p > 0.05$, data not shown).

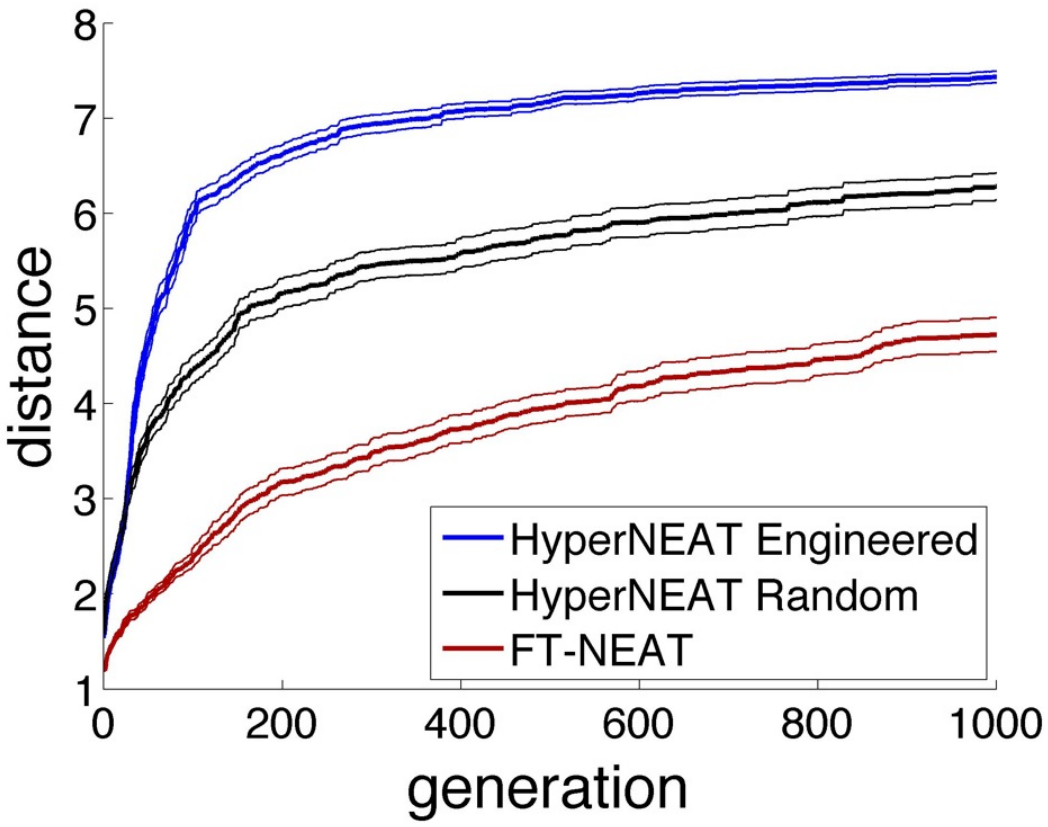

Figure 6.1: The HyperNEAT Default Configuration Vs. an Average of Randomized Configurations and Vs. a Direct Encoding Control. Thick lines show averages and thin lines show one standard error of the mean.

6.3.2 Representations in different dimensions

To date, HyperNEAT has not been tested on the same problem with geometric representations in different *dimensions*. It is an open question as to whether the problem may be easier or harder for the CPPN to solve as the dimensionality of its representation more closely approximates the true geometry of the problem, which is three-dimensional in this case. To test this hypothesis, HyperNEAT was evaluated separately with a one-dimensional (1-d), two-dimensional (2-d), and three-dimensional (3-d) representation. The 1-d treatment has

only x coordinates, the 2-d treatment has only x and y coordinates, and the 3-d treatment has x, y, and z coordinates. The coordinate values and geometric layout for each of these three treatments are shown in Figure 6.2 (1-d), Figure 4.2 (2-d), and Figure 6.3 (3-d). The number of CPPN inputs for each dimension is twice the number of dimensions, plus one for a bias. There are thus 3, 5 and 7 inputs to the CPPN, respectively, for the 1-d, 2-d, and 3-d treatments.

Interestingly, the 1-d representation performed significantly better than the 2-d and 3-d representations in the initial generations ($p < 0.05$ for generations 1-58, Figure 6.4), but the 2-d and 3-d representations soon surpassed it ($p < 0.05$ for generations 170 on). It is possible that the 1-d representation is simpler, but less powerful, making it easier to learn, but harder to achieve high performance with. More tests are needed to reveal whether this phenomenon is general to HyperNEAT on most problems, or is specific to this domain. It could be the case that the 1-d representation was hampered because it is not very accurate with respect to the actual geometry of the robot problem. For example, it is difficult to represent all of the symmetries and repetitions of the robot in 1-d.

While the 2-d representation captures more of the geometric layout of the robot than the 1-d representation, it still lacks fidelity. On the robot, the two hip joints are in the same location and the knee is further away. However, the distance between these three joints is the same in the 2-d representation. Furthermore, the 2-d representation inaccurately represents the torso as a square instead of a rectangle. Finally, the 2-d representation does not represent both the front-back and left-right symmetry of the robot. While these issues could be creatively rectified in 2-d, they disappear when providing the true dimensions of the robot in 3-d.

Even in 3-d there remain some arbitrary choices when assigning geometric coordinates to inputs and outputs. Initially, even though the two hip joints occur in the same place on the actual robot, the CPPN would be unable to distinguish them if they had the exact same geometric coordinates. This problem was avoided by slightly separating these two joints

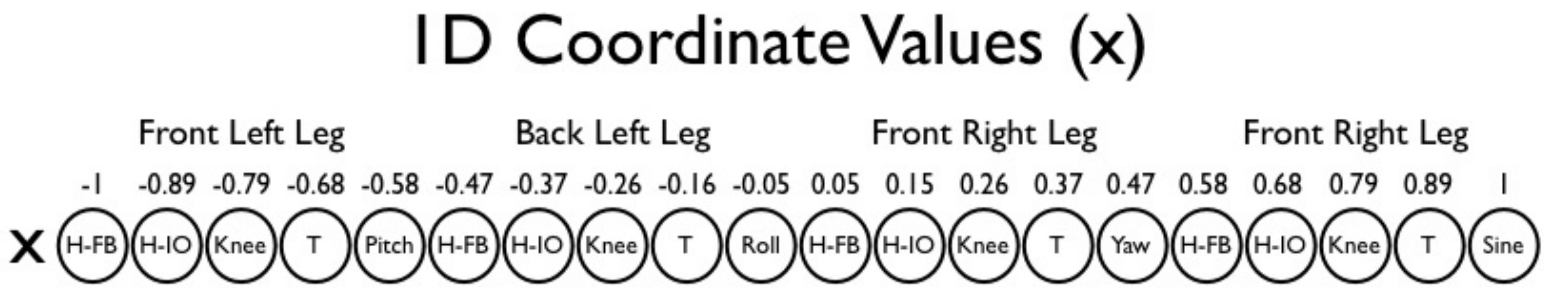

Figure 6.2: The 1-d Geometric Representation. For ease of viewing, only the node coordinates for the input layer are depicted. Those input nodes receive the current angles of the hip-InOut (H-IO), hip-FrontBack (H-FB), and knee joints, as well as a touch sensor (T) and the pitch, roll, and yaw of the torso. A sine wave is also provided to facilitate repeated movements, The numbering system is the same for hidden and output layers. The numbers shown are those fed to the CPPN when the corresponding node is the source node (or target node, for hidden or output layers) to determine the link weight between a source and target node (Figure 2.3).

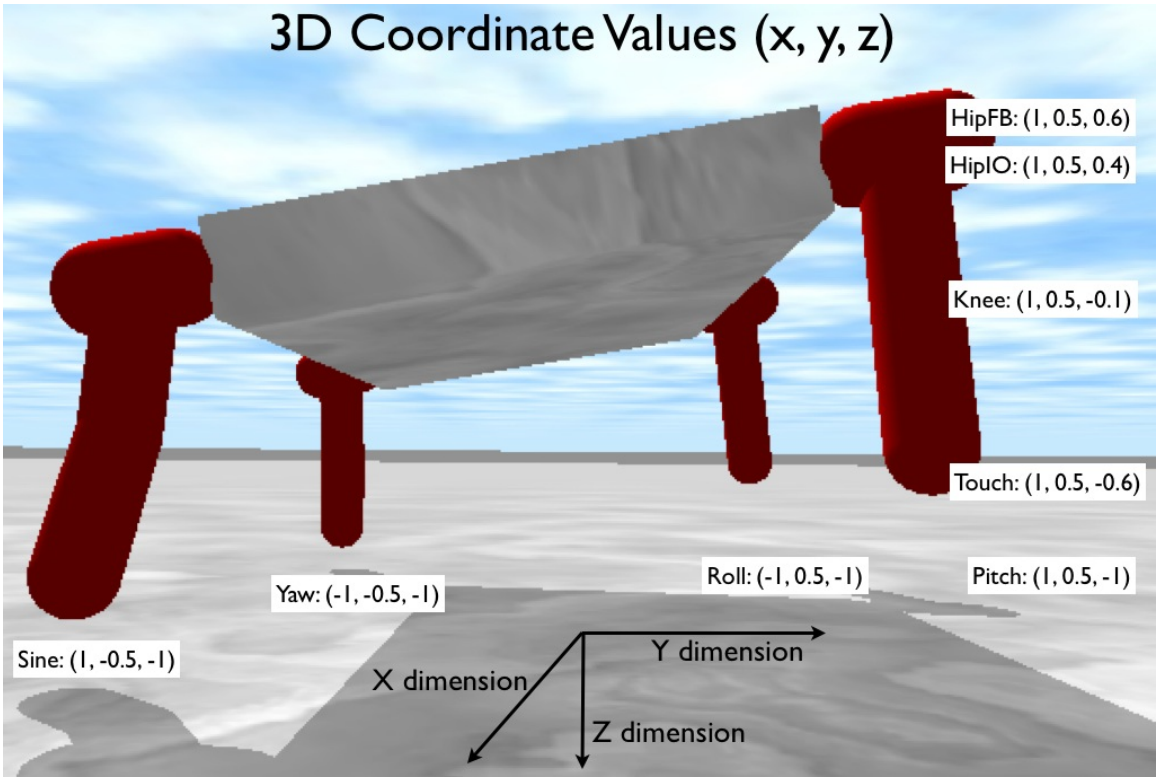

Figure 6.3: The 3-d Geometric Representation. Only the input layer node coordinates are depicted. The numbering system is the same for hidden and output layers. For three of the legs, only the roll, yaw, or sine node has its respective x, y, and z coordinates shown. The x and y coordinates for the other nodes in each of those three legs will be the same as for the node shown for that leg, but the z coordinate will change in the same manner as for the leg with all nodes shown.

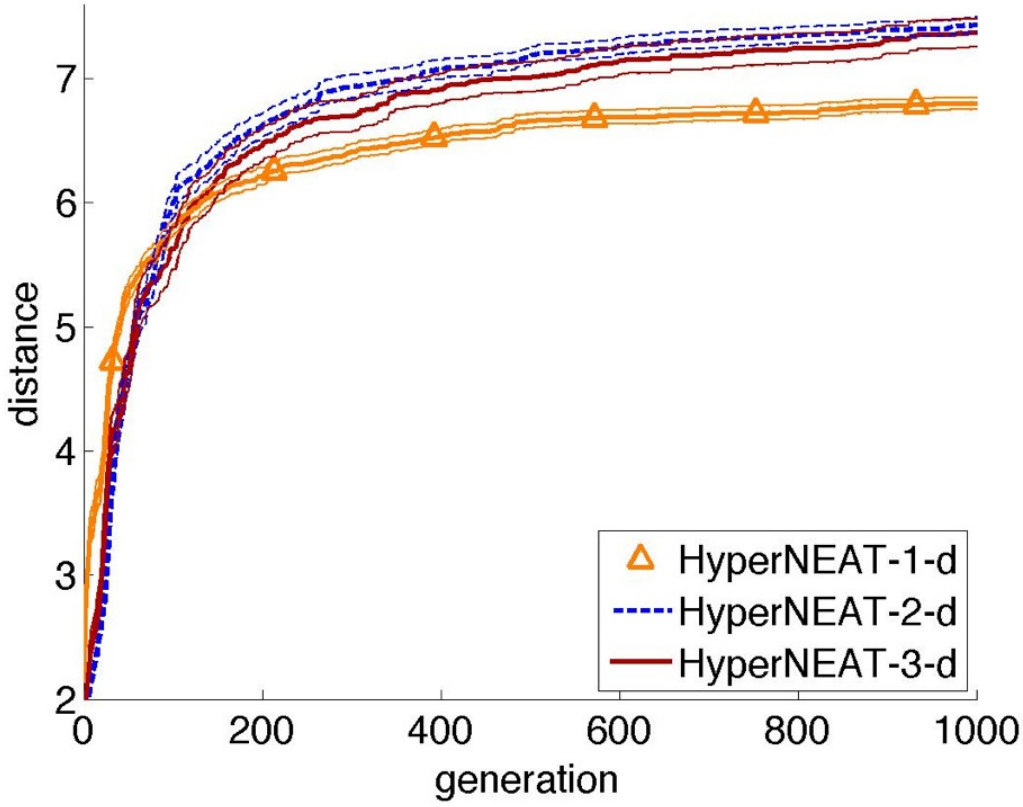

Figure 6.4: The Performance of Representations in Different Dimensions.

in the representational geometry: The HipIO joint was placed just below the HipFB joint in the z dimension. Additionally, the geometric coordinates must be determined for some information that has no meaningful geometric location. For example, where should the sine wave input go? This type of issue will always arise with HyperNEAT when dealing with information that does not belong to any geometric coordinate. The case of the pitch, roll, and yaw sensors is a bit clearer. It would be intuitive to place them in the torso, because that is what they provide feedback about, but it is not clear where in the torso they should be placed. For the purposes of this chapter, placing the pitch, roll, and yaw sensors on the torso would have made a comparison with the 2-d configuration less clean, since in the 2-d setup the pitch, roll, yaw and sine (PRYS) inputs were placed just after the touch sensor on each leg. For this reason, the PRYS inputs were kept at the distal end of each leg.

The data reveal that the 2-d and 3-d treatments performed similarly throughout the experiment, and ended up statistically indistinguishable ($p > 0.05$, Figure 6.4). Evidently, moving to a more accurate representation did not improve performance, which is consistent with a previous finding [11]. However, it is also interesting that the 3-d representation did not hurt HyperNEAT's performance. This result suggests that a user can select either a 2-d or 3-d setup depending on which is easier to implement. It is premature to extrapolate from one test in one problem domain, however, so more tests are needed to test the generality of these findings.

Comparing the engineered performance in each dimension against a randomized configuration from that dimension increases the sample size of the comparison between engineered and random configurations from 1 to 3 (although all three samples are from the same problem domain). The results, portrayed in Figure 6.5, are relatively consistent across dimensions. In all cases, the engineered configuration outperformed the random configurations ($p < 0.001$) and the random configurations outperformed FT-NEAT ($p < 0.001$). Human intuitions provided a performance boost over the random treatments of 18.6%, 18.3%, and 11.7%, respectively, for the 1-d, 2-d, and 3-d representations.

It is also noteworthy that the 2-d randomized treatment statistically ties the 3-d randomized treatment ($p > 0.05$), meaning that the CPPN does no better or worse in either treatment due to the specific set of values fed to the CPPN. However, the 1-d-randomized treatment is significantly worse ($p <.01$) compared to both the randomized 2-d and randomized 3-d treatments. This result implies that one potential explanation for why the 1-d treatment did worse than the 2-d and 3-d treatments is because the CPPN has a harder time with the input numbers in the 1-d treatment (shown in Figure 6.2), and not because the 1-d treatment is less geometrically accurate. The 1-d inputs could be harder to work with because of the specific numbers in that set, or because the numbers are close together, which could make it difficult to differentiate between them.

The data in Figure 6.5 also reveal that HyperNEAT outperforms FT-NEAT, even with

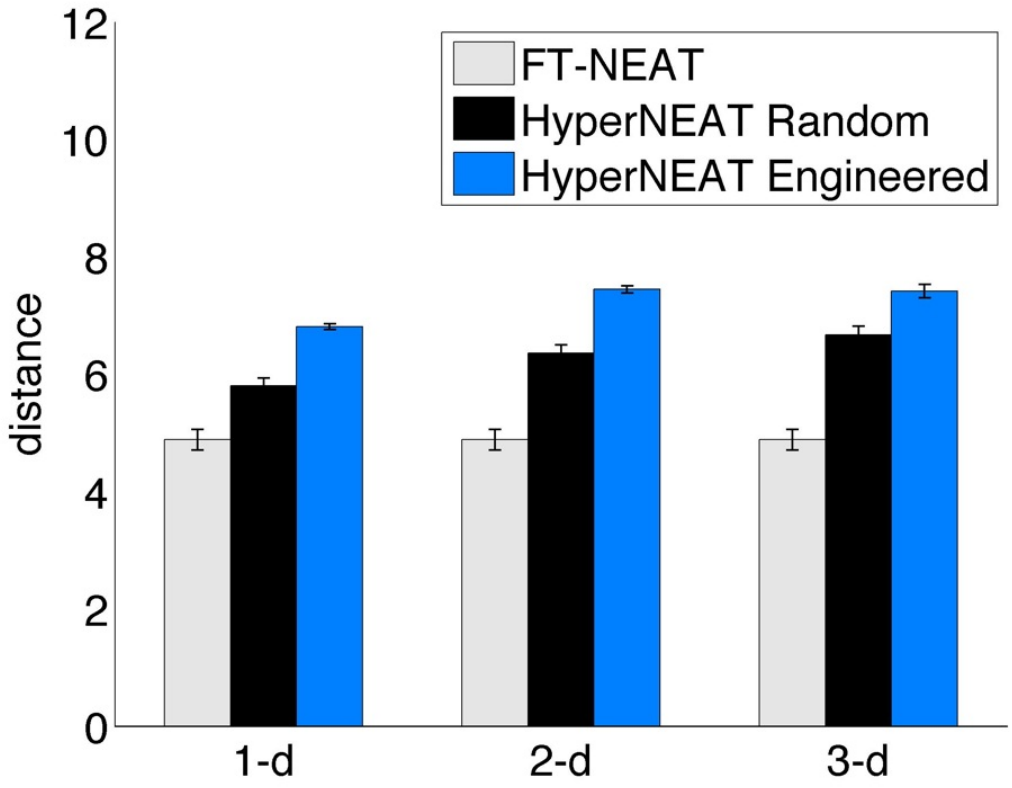

Figure 6.5: Comparing the Performances of an Engineered Configuration, Random Configurations, and FT-NEAT in Different Dimensions.

a naive, randomly chosen geometric configuration, regardless of the dimension of the representation ($p < 0.001$). It is unknown to what extent the performance difference is due to HyperNEAT's generative capabilities or to its ability to exploit even randomized geometries. Unfortunately, these two forces are intertwined and difficult to experimentally isolate.

6.3.3 Repeatedly testing random representations

In the previous experiments, the randomized treatments featured one trial for each of 50 randomized configurations. The average across these configurations was worse than the engineered configuration and better than FT-NEAT. However, in the 1-d treatment, one of

the randomized configuration trials outperformed all of the 1-d engineered trials and another randomized trial performed worse than many of the FT-NEAT trials. The variance in the results highlights the need to explore whether these configurations are inherently better or worse, or whether it was simply a stochastic idiosyncrasy during the individual trials that caused their extreme performance. To test whether certain randomized configurations might perform better than the engineered configuration or worse than FT-NEAT, 50 trials were conducted for the random configurations that performed best and worst, as well as for 25 other randomly chosen randomized configurations from the earlier experiment (Figure 6.6).

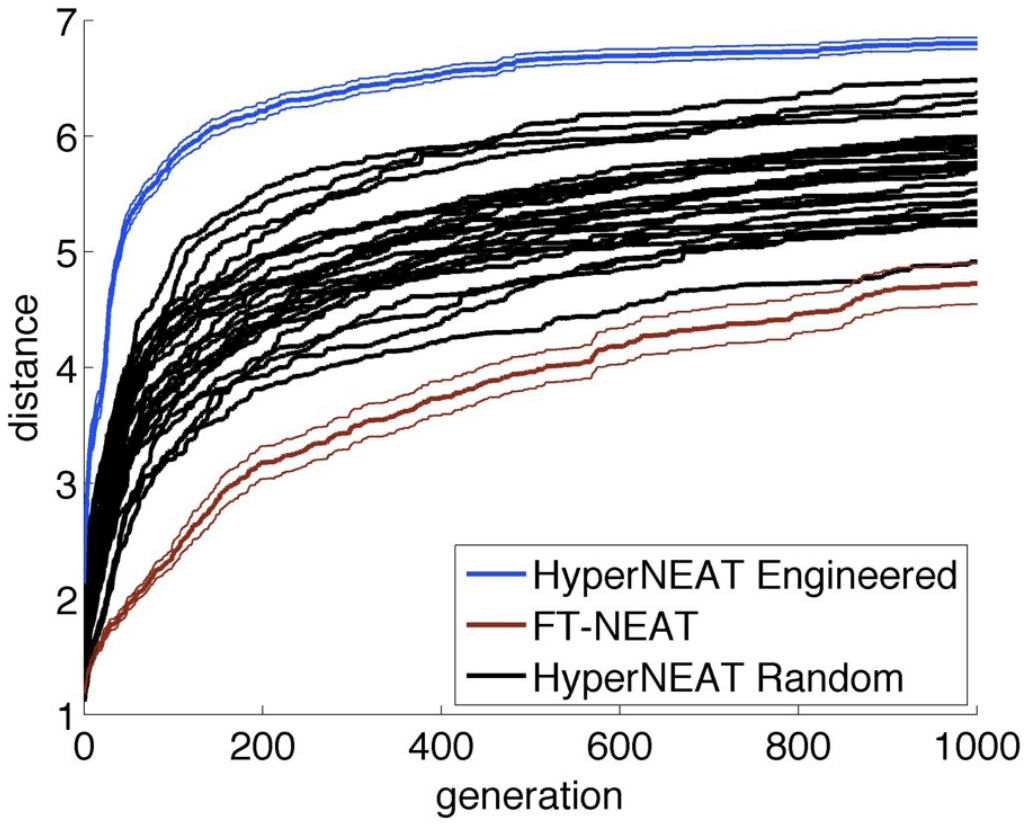

Figure 6.6: A Comparison of HyperNEAT 1-d and FT-NEAT to 27 Randomized 1-d Configurations. Each line is an average of 50 trials. For clarity, standard error bars are not shown for randomized configurations.

Averages across 50 trials for the configurations that originally performed the best and

the worst were not as extreme as the original individual trial scores produced by them. The idiosyncrasies of those single trials mattered more than any property of the configuration itself. That said, there is a substantial amount of variation between the 27 configurations tested, once again underscoring the effect that the geometric representation can have in HyperNEAT. The difference can be significant ($p < 0.001$ comparing the highest and lowest performing configurations). All of the variation in random configurations, however, was confined between the performance of FT-NEAT and HyperNEAT: every randomized treatment underperformed HyperNEAT ($p < 0.01$), and outperformed FT-NEAT ($p < 0.01$ for all but the two lowest performing randomized treatments). These data strongly recommend the selection of HyperNEAT over FT-NEAT on this problem. HyperNEAT's advantage may be because this problem is highly regular, since all legs can be correlated (Chapter 4), and because HyperNEAT increasingly outperforms FT-NEAT as problem-regularity increases (Chapter 3). It seems difficult to produce a geometric representation that performs worse than FT-NEAT. This result is surprising because it suggests that HyperNEAT can exploit regularities in any geometric representation. As such, HyperNEAT could outperform its direct encoding alternatives even if a problem has no obvious geometry (provided the problem is regular, cf. Chapter 3). Recall that NEAT and FT-NEAT performed similarly on this problem (Chapter 4), so the comparisons made here to FT-NEAT hold for NEAT as well.

No random representation outperformed an engineered representation (Figure 6.6). This outcome reinforces the fact that human intuitions about the geometry of a problem help us choose a rare subset of the possible space of geometric configurations that are high performing. Clearly, if the number of samples were increased, then, eventually, representations would be found that are equal to, and possibly better than, the engineered approach. However, those configurations may represent a tiny region of the search space that is hard to algorithmically find. The results from this case suggest the interesting possibility that human engineers may easily select high-performing representations because of our intu-

itive grasp of geometry. It would be interesting in future work to evolve the geometric locations of nodes and compare the results to human-designed configurations.

6.3.4 Comparing alternate engineered representations

In addition to comparing one engineered configuration to random configurations, it is illuminating to compare different engineered representations. Such tests are worthwhile because when arranging a configuration, some choices are difficult (because there seem to be many good alternatives) and others are arbitrary (because multiple options are seemingly equivalent). Whether it is important to investigate alternatives in both cases is addressed by comparing alternate 2-d configurations. For example, the location of the PRYS information is an arbitrary decision because, unlike the joints in each leg, the PRYS information does not have any obvious geometric location. The engineered solution places the PRYS information in the final column of a 5×4 ANN (Figure 4.2). However, it could also have been placed as an additional row in a 4×5 ANN (hereafter referred to as the *PRYS-as-row* setup).

Ideally, such arbitrary configuration details should not affect the CPPN. If it is the case that arbitrary decisions have little impact on evolution, then the designer does not need to spend time testing alternate configurations to find a better one. Unfortunately, the data show that such configurations can make a difference (Figure 6.7). The PRYS-as-row treatment does 9.7% worse than the default setup, which is statistically significant ($p < 0.001$). While it would be interesting to test additional configurations (e.g., PRYS as the first column, or as the first row), limited computational resources prevented such investigations.

Other configuration decisions within a dimension may *a priori* be expected to have a larger impact. For example, the ordering of the legs may substantially affect the quality and type of gaits evolved. If CPPNs have an easier time grouping nodes that are closer to each other, then placing certain legs next to each other in the y dimension in the 2-d setup may make it more likely for those legs to have similar neural controllers and hence have

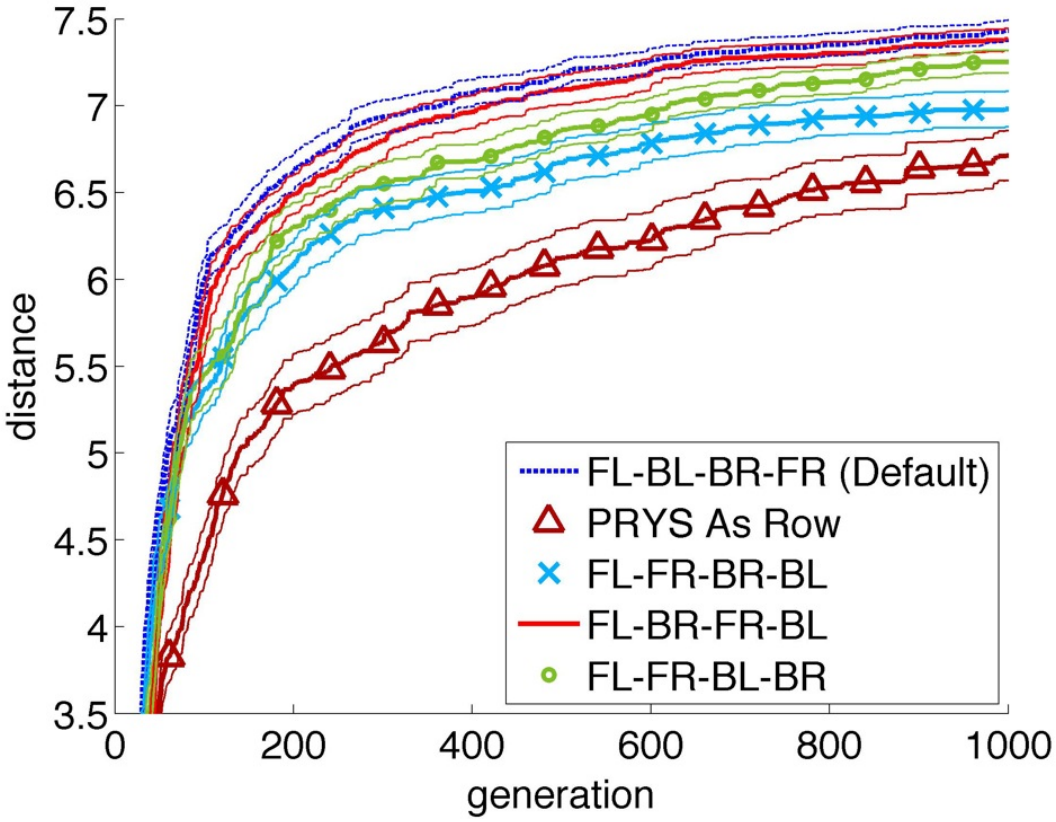

Figure 6.7: The Performance of Alternate 2-d Engineered Configurations.

coordinated movements. Thus, some leg orderings may be more likely to produce left-right symmetry than front-back symmetry, for instance, which could affect fitness scores if one type of symmetry tends to produce faster gaits.

Three alternate orderings were tested in addition to the default ordering (Figure 6.7). The performance of the default setup (FL-BL-BR-FR) is statistically indistinguishable from the FL-BR-FR-BL ordering ($p > 0.05$). However, the other two leg orderings (FL-FR-BR-BL and FL-FR-BL-BR) performed worse than the default ($p < 0.01$). These data suggest that evolution did worse when front-back symmetry was encouraged (by ordering the legs F*F*B*B*, where * is a wildcard). A more exhaustive test of different configurations is warranted, but was prevented by limited computational resources. Nevertheless, these results do conclusively show that the order in which the legs are numbered for the CPPN

can make a difference. Another way of investigating the effect of different leg orderings is to classify the gaits produced by each representation. The gait of the best controller from all 50 trials in each of the four treatments was viewed and categorized (Table 6.1). In all of the treatments, the overwhelming majority of gaits moved all four legs in synchrony. However, the exceptions to this rule within each treatment are interesting because they reflect the geometric biases of each configuration. For example, all four cases of left-right symmetry evolved in the configuration that ordered the legs *L*L*R*R. Furthermore, all seven cases of front-back symmetry were seen in the only two configurations that placed the legs in the order F*F*B*B*. It seems that it is easier for the CPPN to bisect the *y* dimension than to group legs 0 and 2 into one group and 1 and 3 into another. This 'every-other' grouping requires a more complicated function, and did not evolve in any of the best controllers. Interestingly, the configuration chosen to encourage a trot gait, where diagonal legs are in sync (FL-BR-FR-BL), evolved neither a diagonally-symmetric gait nor a gait with front-back or left-right symmetry. For some reason, possibly because the torso is inflexible, the trot gait was not employed by evolution. That, plus the difficulty of grouping the left-right legs or the front-back legs in this configuration, is probably the reason that no diagonal, left-right or front-back symmetries evolved.

Further evidence of the influence of the geometric configuration on the resultant gait can be seen by examining those gaits in which three of the legs moved in synchrony, and one leg did something different. In 23 out of 25 (92%) of these gaits, the exception leg was the last leg in the ordering. It is not surprising that it is easier for the CPPN to make one distinction (e.g., all legs less than *N*) instead of the two distinctions that are required to pick a leg out of the middle of a dimension. The two exceptions, however, prove that it is possible for the CPPN to make an exception for a middle leg. It is not clear why the CPPNs tended to single out the last leg and not the first, although this result likely occurred because making that exception was easier for the specific mathematical functions used.

It appears that the ordering of components geometrically can bias HyperNEAT's group-

	4way Sym	L-R Sym	F-B Sym	One Leg Out Of Phase			
				FL	BL	BR	FR
FL-BL-BR-FR (default)	36	4					9
FL-BR-FR-BL	47				2		1
FL-FR-BL-BR	44		3			1	
FL-FR-BR-BL	36		4		9		1

Table 6.1: The Resultant Gait Types for Different Leg Orderings. Gaits are placed into the following categories. 4way Sym(metry) (all legs in synchrony), L-R Sym (the left legs are in phase and the right legs out of phase), F-B Sym (the front legs are in phase and the back legs are out of phase), One Leg Out of Phase (three legs moved in synchrony and one is out of phase, which resembles a gallop). If two legs are motionless, they are considered in synchrony. Two gaits do not fit into these categories and are not tabulated. FL=Front Left, BL=Back Left, BR=Back Right and FR=Front Right.

ing of those components. This result, which has not been previously reported, means that a user can inject biases (desired or not) into how HyperNEAT clusters subcomponents of a problem. For example, if evolving a team of multiple agents, which HyperNEAT has been shown to do well [11], the geometric ordering could influence the types of teams selected. Imagine a game in which players have to create military units of either fast moving speedsters or slower, but more powerful, tanks. If players desired one homogenous squadron of only speedsters and a separate homogenous squadron of only tanks, they could make this outcome more likely by creating a geometric representation that ordered the pieces speedster-speedster-tank-tank. If a player wanted two heterogeneous squadrons that each had a speedster and a tank, they could make that outcome more likely with a speedster-tank-speedster-tank ordering. Importantly, the bias of any configuration is also determined by the CPPN function set, and changing it could alter the biases of any given representation.

6.4 Conclusion

This chapter shows that when evolving controllers for simulated legged robots, HyperNEAT can be sensitive to the way its geometric information is represented. HyperNEAT outperformed a direct encoding control even with randomized geometric representations. HyperNEAT's success with random configurations suggests it can perform well even if one does not know how to geometrically represent a problem. However, properly choosing a geometric configuration, which may seem intuitive to a human engineer, can provide a performance increase (10%-20% on this problem). Testing alternate engineered configurations was shown to be important for two reasons: Initially, some seemingly arbitrary decisions in the design of geometric representations can have large effects. Additionally, alternate options that *a priori* seem good for different reasons can have significantly different performance levels. In addition to quantitative fitness effects, the geometric configuration can also affect the *types* of solutions evolved, enabling engineers to bias the products of HyperNEAT evolution. HyperNEAT's sensitivity to its geometric representation is both detrimental, because work is required to optimize it, and powerful, because altering it can yield performance increases and enable engineers to shape the solutions produced. It is important to note, however, that all of the conclusions in this chapter are drawn from one problem domain. Future work is required to see whether such conclusions hold more generally. Additionally, it will be interesting to see whether these results generalize to future generative encodings that exploit geometry.

Chapter 7

Investigating modularity in a geometry-based generative encoding

Note: This chapter is an expanded version of Clune, Beckmann, McKinley, and Ofria 2010.

7.1 Motivation for studying modularity in evolved ANNs

A long-term goal in the fields of evolutionary computation, neuroevolution, and artificial life is to synthetically evolve phenotypes as complicated as those seen in the natural world. Many complex, natural organisms exhibit modularity, regularity, and hierarchy [23, 29, 38, 54,55], which are important design properties and can also increase the evolvability of these organisms [27, 28, 33, 37]. Please see Chapter 2.1 for a definition of regularity, modularity, and hierarchy.

Without the ability to evolve genotypes and phenotypes that possess these characteristics, it may be difficult to synthetically evolve creatures as complicated as those found in nature [2, 33, 38]. Modularity is especially important for neural networks, where it can improve both evolvability and learning, because modular networks can more easily be rearranged to produce new functions [28, 37]. These benefits likely explain why natural brains display a high degree of modularity, regularity, and hierarchy [23, 29, 38, 54, 55]. Designs

engineered by humans also possess these properties for the same reasons: they make it easier to design and modify complex artifacts.

Modularity, regularity, and hierarchy arise in natural organisms as a result of complex developmental processes [2, 38]. A desire to produce these design principles in synthetic evolution has led many researchers to switch from direct encodings to generative encodings (Chapter 2). It has been shown that generative encodings are capable of producing modularity, regularity, and hierarchy in phenotypes [27], and specifically can create regularity and modularity in evolved neural networks [19, 26]. These generative encodings are based on rewriting symbols, such as Lindenmayer Systems [25–27, 32], or programs that are recursively called at vertices in a graph [19]. These representations perform well in part because they strongly and explicitly bias evolution towards phenotypes with modularity, regularity, and hierarchy [27].

The HyperNEAT generative encoding (Chapter 2), which is the focus of this dissertation, was not designed to generate modularity, regularity, and hierarchy as explicitly as previous generative encodings. It is therefore important to determine the degree to which HyperNEAT produces these properties in its phenotypes. In Chapters 3-6, I demonstrated that HyperNEAT produces *regular* ANNs that exploit the regularity of problems. This chapter investigates the previously unstudied question of whether HyperNEAT produces *modular* ANNs. In the future I will investigate whether HyperNEAT can produce hierarchical ANNs. I would like to emphasize that in this chapter I focus on modularity in evolved *phenotypes*. In future work I also plan to investigate the modularity, regularity, and hierarchy of HyperNEAT *genotypes*.

To investigate modularity in HyperNEAT, I tested whether it and a direct encoding control (FT-NEAT) produced modular ANNs on a problem that has previously been shown by Kashtan and Alon [28] to generate modular ANNs with a different direct encoding neuroevolution algorithm. In contrast to those results, this problem did not encourage modularity in FT-NEAT, raising a question about the generality of Kashtan and Alon's results. I

also found that HyperNEAT did not produce modular ANNs on this problem and variants of it. I then tested whether HyperNEAT would have done better had it produced a modular ANN by imposing modularity on its ANN phenotypes. With this imposed modularity, HyperNEAT's performance improved. These results show that, irrespective of how the direct encoding performed on this problem, HyperNEAT would have done better had it produced modular ANNs. I next tested two techniques to encourage HyperNEAT to automatically produce modularity, but did not observe an increase in either modularity or performance. Finally, I conducted tests on a simplified version of the problem and found that HyperNEAT quickly was able to produce modular solutions that solved the problem. I therefore present the first documented case of HyperNEAT generating a modular phenotype, but my inability to encourage modularity on harder problems where modularity would have increased performance suggests that more work is needed to increase the likelihood that HyperNEAT and similar algorithms will produce modular ANNs in response to challenging, decomposable problems.

7.2 Motivation for, and description of, the Retina Problem

An informative test of whether HyperNEAT produces modular ANNs is to try it on a problem where modularity is known to be helpful, and in an environmental regime that has been shown to encourage modularity in a neuroevolution algorithm. Fortunately, previous research has been conducted on such a problem [28]. Kashtan and Alon demonstrated that environmental regimes that switch between problems with modularly varying goals (MVG) increase the evolution of modular phenotypic networks. MVG environments switch between tasks that have shared subproblems, but where the overall problem is solved by combining answers to these subproblems in different ways. On two different problems, Kashtan and Alon demonstrate that MVG environments produce highly modular networks. They also show that fixed goal (FG) controls that evolve to solve a single unchanging prob-

lem produce non-modular networks, even though the fixed goal was identical to one of the goals from the MVG regime and thus had the same subproblems. The MVG treatments also solved problems in an order of magnitude fewer generations. Moreover, the evolved modules of the MVG networks solved the subproblems Kashtan and Alon had designed into the overall problems. Over time, solutions evolved that allowed the modules to be reconfigured via a single or small number of mutations, thereby enabling quick adaptations from one environment to another. In subsequent work it was shown that, after an environmental change, modular networks were faster at adapting both to previously seen and novel environments: This ability to quickly adapt to new environments is made easier because of the modularity that evolved in the networks [37]. Inspired by these findings, scientists tested and confirmed that similar results hold for natural organisms: bacteria that live in changing environments have more modular metabolic networks [31, 37].

Kashtan and Alon's results are consistent with our expectations for when modularity is useful. Modularity is not necessarily helpful, and may be harmful, when designing a solution for a single, unchanging problem [33]. Modularity becomes beneficial when designs need to be changed quickly, because modules that solve subproblems can be easily reorganized [33].

Kashtan and Alon's first problem involved evolving the connections of networks of NAND gates to solve Boolean logic functions. Their second problem consisted of evolving neural networks to perform pattern recognition. I chose their second problem as the test problem in this chapter because HyperNEAT was designed to evolve neural networks.

The second problem, which I will call the *Retina Problem*, evolves a neural network to separately recognize patterns, or 'objects', on the left and right sides of an artificial retina (Figure 7.1a). The retina consists of eight pixels, four per side, which were the inputs to a neural network with sigmoid activation functions. The left four pixels (the left pane) can form 16 unique patterns, half of which are considered Left Objects. The same is true for the right four pixels (the right pane). The goal is to have the single output of the network

answer one of two Boolean logic questions: [L AND R] (true if there is a Left Object and a Right Object), or [L OR R] (true if there is a Left Object, if there is a Right Object, or both). This Retina Problem is challenging because the network must independently recognize and identify low-level patterns before processing that information to determine if a higher-level pattern is present [28]. The networks had four feed-forward layers in addition to an input layer (Figure 7.1b-c). Layer 1, which received connections from the input layer, had eight input neurons. Layers 2 and 3 were hidden layers with four and two neurons, respectively. The output layer had a single neuron.

A human engineer immediately recognizes the modularity in the Retina Problem: The left and right panes can be processed independently to determine if an object is present. The information can then be combined in either a logical AND or OR. There are non-modular ways that may be equally good at solving either problem, but such non-modularity will likely make it more difficult to switch from a network that solves one problem to a network that solves the other, where difficulty is measured by the number and magnitude of link weight changes that need to be made.

Pixels on the retina were limited to 'on' and 'off' states, represented as input values of 3.0 and -3.0, respectively. A bias neuron with a constant input of 3.0 had evolvable connections to all neurons. This feature serves a similar function to the evolvable thresholds in Kashtan and Alon's setup [28]. Outputs were considered true if they were close to 1 and false if they were close to -1. The fitness function was inversely proportional to the difference (the error) between the correct answer (1 or -1) and the network output. Specifically, the fitness function summed the error across all 256 possible input patterns and squared the result to magnify the importance of small improvements.

The specifics of the implementation in this chapter differ in certain ways from Kashtan and Alon's [28]. While the description of their model is not complete, it appears that their inputs and outputs were binary, the activation functions of their neurons were step functions with only three possible thresholds, and their link weights consisted of a small

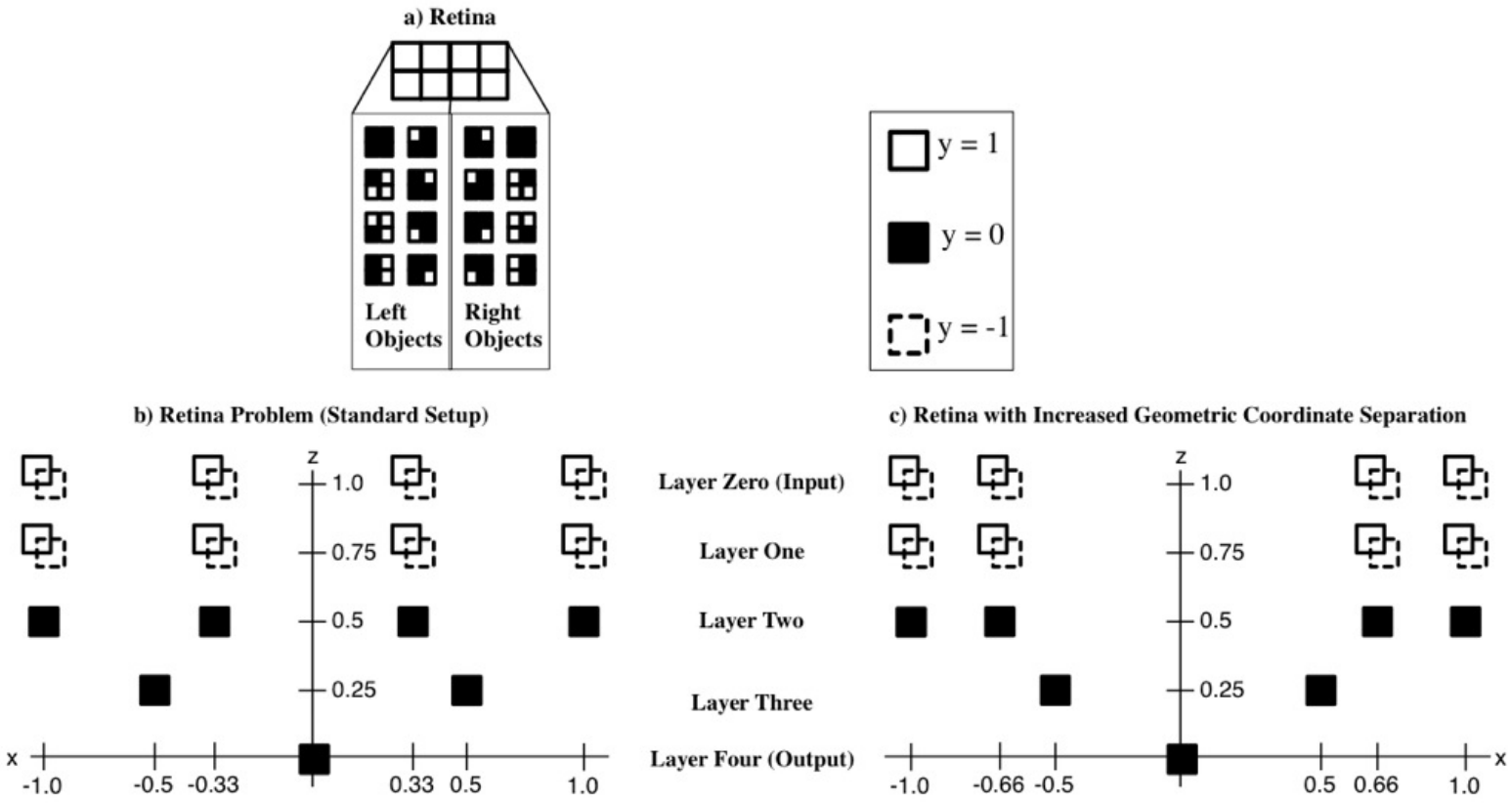

Figure 7.1: The Retina Problem. (a) The Eight-pixel Artificial Retina and the Patterns that Constitute Left and Right Objects (Adapted From Kashtan & Alon 2005). (b) The geometric representation of the ANN nodes for the Standard Setup of the Retina Problem. (c) The geometric representation for the Retina Problem with Increased Geometric Coordinate Separation. The x, y, and z coordinate values for each node are passed into the CPPN when determining the weights for connections between nodes (Figure 2.3).

set of discrete values. They evolved their networks via a standard direct encoding genetic algorithm with mutation and crossover. Their fitness was a function of the percent of correct answers provided across 100 randomly chosen input patterns. These differences, while seemingly minor, may explain the different qualitative results I observed from those of Kashtan and Alon [28].

Kashtan and Alon evolved networks in an FG regime [L AND R] and an MVG regime, wherein the rewarded task switched every 20 generations from [L AND R] to [L OR R]. They continued each evolutionary run until the networks output the correct answer for 95% of the input patterns, at which point they considered the problem solved. That took a median of 21,000 generations in the FG regime, which was nearly an order of magnitude slower than in the MVG regime, which took 2,800 generations. The MVG networks were more modular, and could adapt to an environmental change from one goal to the other in about 3 generations, often via a single mutation [28].

7.3 Experiments and results

7.3.1 Retina problem (standard setup)

I tested the performance of HyperNEAT and FT-NEAT on the Retina Problem for two MVG regimes, one that alternated between tasks [L AND R] and [L OR R] every 20 generations (MVG-20), which was the rate used by Kashtan and Alon, and another that alternated every 100 generations (MVG-100). I also conducted experiments with faster and slower rates of change, but the results were not qualitatively different (data not shown). I tested two FG regimes (FG-AND and FG-OR), one for each of the tasks. For each experimental treatment discussed in this chapter I performed 20 runs of evolution with different random number generator seeds, and report the median (bold lines) and 25^{th} and 75^{th} percentiles (thin lines). To represent fitness values, I show the percent of test cases the best organism in the population provided the correct answer for. The nature of the logic functions means

that always outputting 1 (in the OR environment) or 0 (in the AND environment) achieves a score of 75%. For this reason, it was rare to see the best organism in each generation score below 75%. Each run had a population size of 500, which is large for HyperNEAT experiments [46].

The results, presented in Figure 7.2a, reveal that HyperNEAT does not perform well on this problem. Recall that Kashtan and Alon's direct encoding achieved 95% accuracy in both the FG and MVG regimes. FT-NEAT also performed poorly (Figure 7.3), and its results were qualitatively the same as for HyperNEAT on the FG-AND, FG-OR, and MVG treatments. While Kashtan and Alon did perform evolution for many more generations, additional experiments up to 30,000 generations for both HyperNEAT and FT-NEAT revealed that longer experiments do not change the qualitative results (data not shown). More likely, the difference in absolute success has to do with the differences in the neural networks. However, alternative experiments with different selective pressures produced networks that perfectly solved the FG problems, suggesting that the difference between these results and those of Kashtan and Alon is not due to any limitation in the capability of the networks in these experiments, but is instead related to differing evolvability between the configurations.

To better understand why HyperNEAT performed poorly, I ran the same experiment, with the same number of nodes and potential links in the ANN, but where ANNs were rewarded for correctly identifying only Left Objects in the FG-AND problem. The evaluated output was taken from the left node of layer 3 (the layer just before the output layer). In this easier version of the problem, which I call Retina Left Only, HyperNEAT performed better, but still had difficulties (Figure 7.2a). These difficulties may have occurred because HyperNEAT created too many links in its substrate and therefore did not ignore the inputs from the right panel when identifying Left Objects. FT-NEAT, on the other hand, performed significantly better, with 17 of 20 treatments surpassing 95% accuracy ($p < 0.001$, Mann-Whitney U rank test, Figure 7.3).

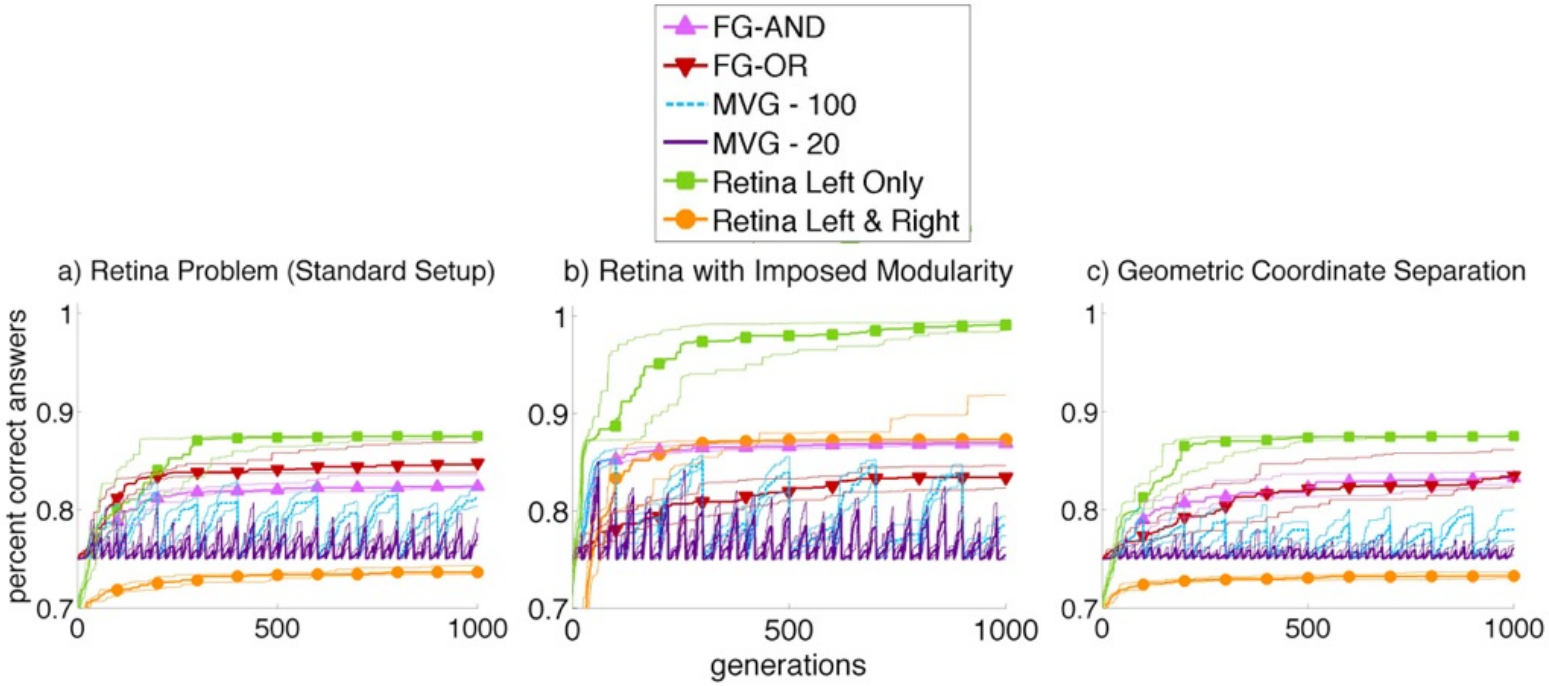

Figure 7.2: Performance Versus Evolutionary Time for HyperNEAT on (a) the Retina Problem (Standard Setup), (b) the Retina Problem with Imposed Modularity, and (c) the Retina Problem with Increased Geometric Coordinate Separation. Plotted is the percent of the 256 trials that the network output the correct answer. Medians are shown as bold lines surrounded by the 75^{th} and 25^{th} percentiles of the data. See the text for explanations of what constituted a correct answer for different variants of the problems.

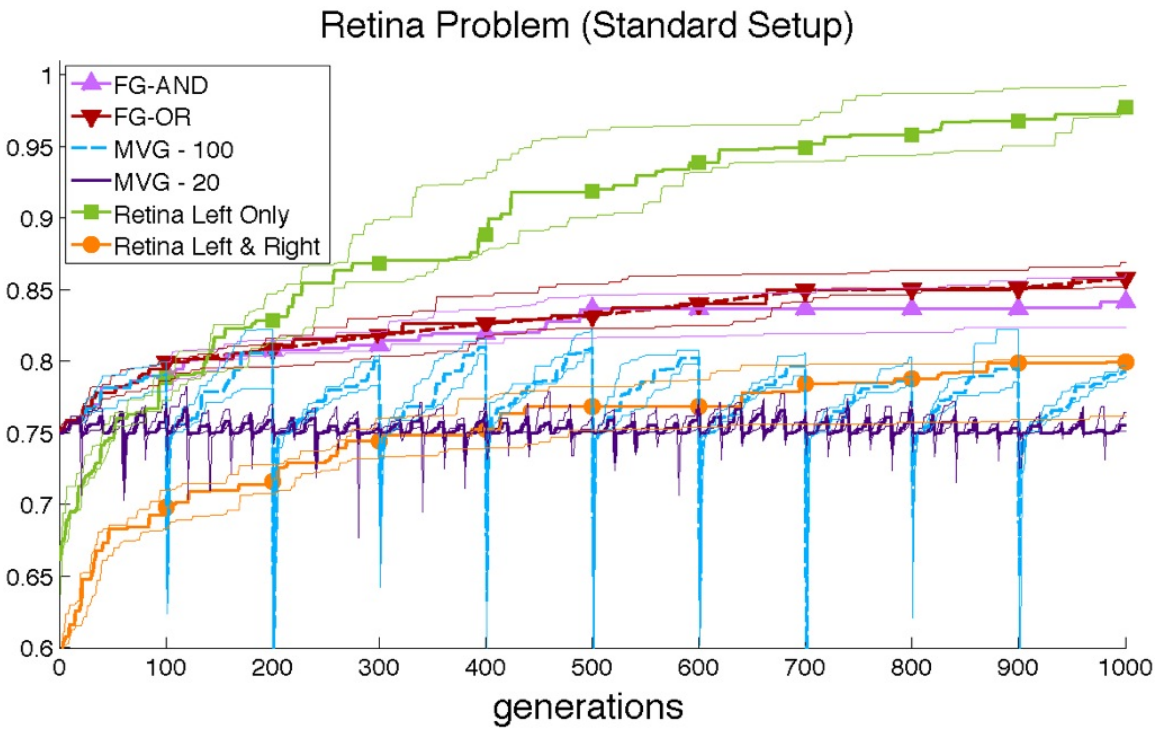

Figure 7.3: The Performance of FT-NEAT on the Standard Setup of the Retina Problem. Medians are shown as bold lines surrounded by the 75^{th} and 25^{th} percentiles of the data. Note that the y-axis scale is different than in Figure 7.2.

I conducted a similar experiment, but rewarded networks that could correctly report whether there were Left Objects and Right Objects, respectively, in the left and right nodes of layer 3. This experiment is particularly illuminating because a network must be able to first solve this task before computing the logical AND or OR of these answers, which is required for the optimal solution to the FG and MVG problems. The results of this experiment, which I call Retina Left & Right, demonstrate that HyperNEAT did worse on this task than all other versions of the problem (Figure 7.2a). That HyperNEAT was unable to independently determine the presence or absence of Left Objects and Right Objects helps explain why it did not do well on the harder tasks that require further processing this information. Of particular interest is how much better the FG-AND and FG-OR treatments performed than the Left & Right treatment. Given that HyperNEAT had difficulty independently identifying Left and Right Objects, we can infer that the strategy it employed on

the FG treatments to perform better than the Left & Right treatment did not independently process the left and right panels. HyperNEAT likely took advantage of the locally optimal shortcut of always outputting 1s or 0s, which yields fitness values of 75%. It could then have increased its performance up to the level it achieved by encoding some additional information about the problem, such as certain situations in which to provide the other output.

More important than the absolute difference between Kashtan & Alon's results and those of HyperNEAT and FT-NEAT is the qualitative difference: the MVG regimes performed worse than the FG-AND regime, which was the opposite of what occurred in Kashtan and Alon's study. This result raises questions as to the generality of Kashtan and Alon's discovery that environments with MVG will generate the evolution of modular networks. While there are differences between Kashtan and Alon's experimental setups and the one used here, the differences are relatively small and should not preclude such a seemingly general result. I will investigate what differences in the implementations led to the differing results in future work.

Despite the differences between these results and those of Kashtan and Alon, the Retina Problem still serves as a diagnostic problem regarding the ability of an algorithm to produce modularity. Because the problem can be decomposed on the left and right sides until the final layer, networks that are more modular should perform better. The case is even clearer for the Retina Left & Right problem, because the left problem is independent of the right problem. For the remainder of this chapter, I utilize the Retina Problem and variants of it to investigate HyperNEAT's ability to generate modular ANNs.

I hypothesized that HyperNEAT may have performed poorly because it was not producing modular ANNs. To test whether HyperNEAT's ANNs were modular, I counted the number of active (non-zero) links in the substrate. A modular design that processed the left and right panels separately before combining them has at most 69 of the possible 121 links in the neural network, or 57%. This number is small because there are no links

between the two modules. The degree to which the percent of links in a network is over 57% suggests the extent to which the ANN is interconnected instead of modular. This link-counting measure is a crude estimation of modularity, but it is not accurate to simply count the links between the nodes on the left and right side of the coordinate space, because it is possible for evolution to create modules that are not correlated with geographic location. In future work I plan to quantify the modularity of these networks with a more sophisticated modularity metric.

The FG regimes had a median of 94% and 96% of links active ($\pm$6% SD), respectively, for the FG-AND and FG-OR treatments. The MVG-100 and MVG-20 regimes had medians of 94% and 95% ($\pm$3%, $\pm$7% SD). For many runs, 100% of the links in the champion ANN were active, which is the lowest level of modularity possible. There were no statistically significant differences in the link percentages between the FG and MVG treatments ($p > 0.05$, Mann-Whitney U rank test). I conclude from these data that part of the reason HyperNEAT performed poorly on both the FG and MVG tasks is because it has difficulty turning off links and thus produced ANNs with low levels of modularity.

7.3.2 Retina problem with imposed modularity

I next investigated whether HyperNEAT would have done better had it discovered the left-right modularity of the problem. To test this hypothesis, all connections were disabled between the left and right sides of the network, except between layers 3 and 4. Disabling of cross-links ensured that information from the left and right panels was processed independently until it was combined in the final layer.

This *Imposed Modularity* treatment improved the performance of every treatment (Figure 7.2b, $p < 0.001$ comparing the fitnesses of the generation champions per treatment from the final generation with a Mann-Whitney U rank test) except for FG-OR, which performed worse ($p < 0.05$). The decline in FG-OR performance with imposed modularity, while slight, is counterintuitive and was anomalous compared to the results from the other

treatments. I hypothesized that this odd result may have occurred only because evolution had not yet leveled off, which was more so the case for the FG-OR treatments than the others. To test whether additional generations would make a difference, I extended the FG-OR experiments to 3000 generations, at which point the Imposed Modularity treatment outperformed the Standard Setup (the setup with results plotted in Figure 7.2a), although the difference was not statistically significant ($p > 0.05$, Mann-Whitney U rank test). Four of the extended FG-OR with Imposed Modularity runs reached at least 95% accuracy, further demonstrating that HyperNEAT is capable of solving the Retina Problem with the neural networks used in this chapter. None of the extended FG-OR runs without imposed modularity reached a fitness level of greater than 90%. It is not obvious why Imposed Modularity is less helpful on the FG-OR treatment than in the other treatments: It may be that the imposed modularity is interfering with the exploitation of a locally optimal strategy that is being used in the Standard Setup. Kashtan and Alon did not report experimenting with FG-OR, so I cannot compare my results to theirs [28].

These results confirm that HyperNEAT would have done better had it generated a modular network that independently processed the left and right panels. Interestingly, the largest effects were in two non-MVG treatments. In the Retina Just Left treatment, HyperNEAT scored nearly perfectly with imposed modularity (Figure 7.2b). This result demonstrates that the subproblems of identifying Left and Right Objects are not impossible for HyperNEAT to solve (experiments focusing just on the right side were qualitatively similar, data not shown). The imposed modularity also substantially improved the performance of the Retina Left & Right treatment. Four of these treatments achieved scores above 95%, with one at 99%, and none scored below 86%. Both the Retina Left Only and Retina Left & Right problems are thus demonstrations of problems where modularity is beneficial, but where HyperNEAT did not discover such modularity on its own. These results emphasize that HyperNEAT would perform better on some FG problems if it were better able to create modular ANNs. This is not to say that modularity is necessary, but just that in this case it

improves the likelihood of evolving a high quality solution. That four of the Left & Right treatments scored above 95% also offers additional evidence that HyperNEAT is capable of solving the Retina Problem with the neural networks used in this chapter, because putting this information together into an AND or an OR function is possible in this setup.

That imposed modularity increased performance on the Left & Right problem may help us infer why imposed modularity aided the performance of HyperNEAT on the FG-AND problem. The similarity in fitness scores between the Left & Right treatment and the FG-AND treatment with imposed modularity may indicate that the FG-AND treatment with imposed modularity did implement the globally optimal strategy of independently processing the left and right panels, albeit in an imperfect way. If so, this instance would be an interesting demonstration of how modularity helped evolution switch from a locally optimal strategy to a higher-performing and possibly globally optimal strategy. Unfortunately, it is difficult to determine if these networks did indeed correctly process the left and right panels by recording the values at the associated nodes in layer 3, because evolution can internally represent information in different ways.

7.3.3 Retina problem with fewer links

One factor that may hamper HyperNEAT's ability to create modular ANNs is that it produces too many substrate links. To test this hypothesis, the range of CPPN outputs that were converted to an ANN link weight of zero was increased, which effectively eliminates the link. For the previous experiments, CPPN outputs in the range of -0.1 to 0.1 resulted in ANN links of 0. Such a range was built into HyperNEAT to facilitate the elimination links in its ANN phenotypes [46]. I call this parameter ZeroOutHalfWidth. As the ZeroOutHalfWidth increases, a wider range of CPPN output values result in ANN link weights of zero, decreasing the expected number of ANN links.

I tested ZeroOutHalfWidth values of 0.2, 0.4, 0.6, 0.8, 0.9, 0.95 and 0.99, and compared the results to the default value of 0.1 (Figure 7.4). Altering the ZeroOutHalfWidth

parameter had little effect on fitness and did not raise fitness up to the levels observed with imposed modularity. All of the treatments with different ZeroOutHalfWidth values, but without imposed modularity, were significantly worse than the Imposed Modularity treatment on both the FG-AND and Left & Right problems ($p < 0.001$, Mann-Whitney U rank test).

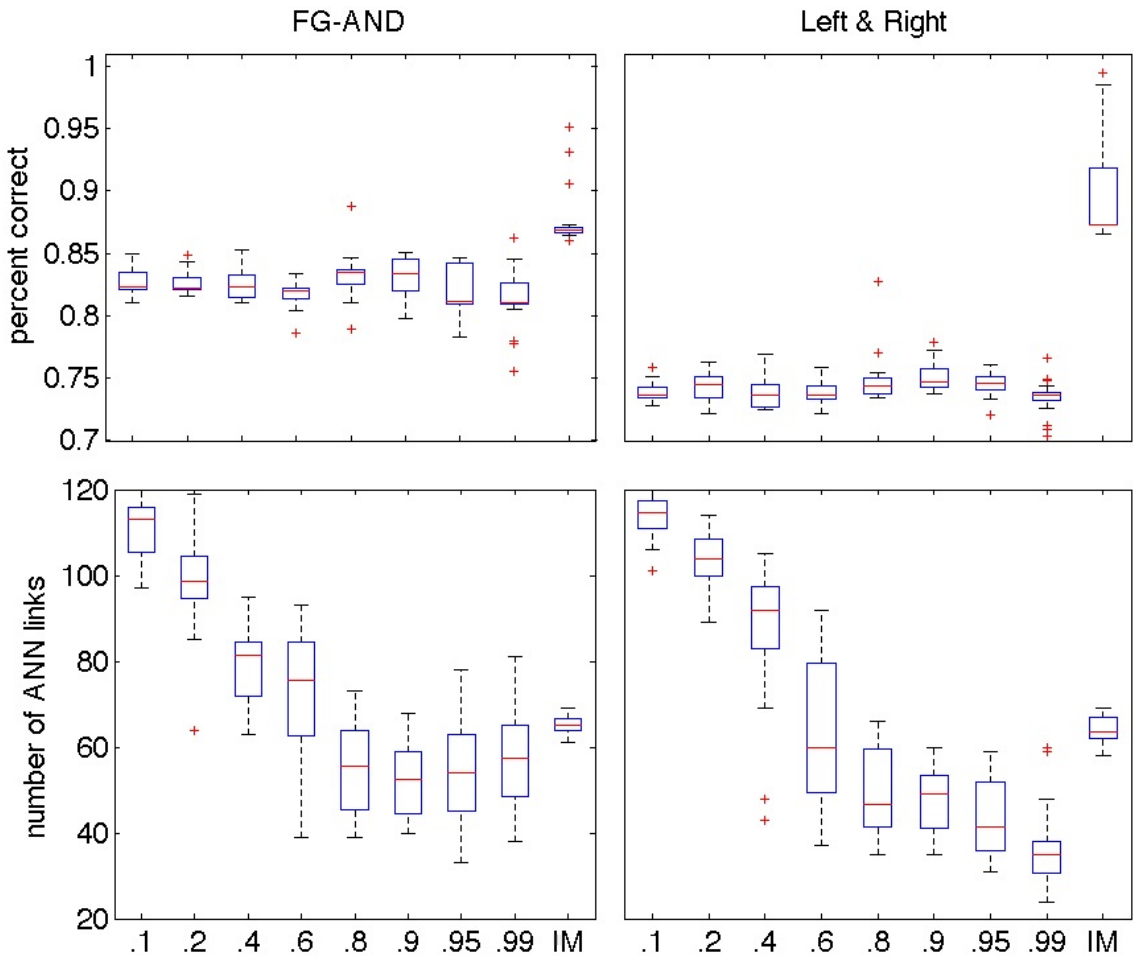

Figure 7.4: The Effect on Performance (Top Row) and the Number of ANN Links (Bottom Row) of Varying the ZeroOutHalfWidth Parameter. Each column represents a treatment with a different ZeroOutHalfWidth value (columns 1-8) or the Imposed Modularity (IM) treatment (column 9), which is shown for comparison. The midline shows the mean, the lower and upper box lines show the 25^{th} and 75^{th} percentiles, the whiskers enclose all non-outliers, and outliers are shown as plus signs.

The data reveal that higher ZeroOutHalfWidth values did reduce the number of ANN links in both problems (Figure 7.4, bottom row). Importantly, the number of ANN links for

some of the different ZeroOutHalfWidth values were roughly similar to the number of links in the Imposed Modularity treatment. Given that these treatments had similar numbers of ANN links, but performed significantly worse than the Imposed Modularity treatment, it is likely that the resulting networks were not very modular. This result suggests that it is not merely the inability to eliminate ANN links that prevents HyperNEAT from discovering modular solutions to these problems, but that HyperNEAT also has trouble controlling which links to deactivate.

Independent of its effect on modularity or fitness, the tactic of reducing HyperNEAT links by increasing the ZeroOutHalfWidth value did have the desired effect of lowering the number of ANN links. This technique may be beneficial to future HyperNEAT users that wish to reduce the number of links in HyperNEAT-generated ANNs.

7.3.4 Retina problem with increased geometric coordinate separation

Another technique that could facilitate the production of phenotypic modules in HyperNEAT's ANNs would be to make it easier for HyperNEAT's CPPNs to discriminate between the nodes on the left and right sides of the ANN. This goal can be accomplished by changing the geometric representation of the problem, which means changing the Cartesian coordinates assigned to different nodes in the ANN. Because each CPPN computes the weights of links between nodes as a function of the geometric locations of those nodes, changing these coordinate values can bias HyperNEAT toward different types of phenotypes that perform significantly differently (Chapter 6). Moreover, the intuitions human engineers have for how to geometrically represent problems can also aid the performance of HyperNEAT (Chapter 6).

This method is not guaranteed to work, however, because there are ways to create modularity that do not respect the left and right sides of the coordinate space, and this mechanism might bias the CPPN away from producing them. Nevertheless, this technique of spreading the nodes out in coordinate space could at least make it easier to adopt the left-right mod-

ularity produced by the Imposed Modularity treatment. Such left-right modularity is also likely to be the type a human engineer would apply to this problem.

This method was implemented by changing the coordinate values of the nodes from a representation that had already been designed to encourage left-right modularity (Figure 7.1b) to one that separated the left and right nodes in geometric space even further (Figure 7.1c). This geometric separation did not increase performance in any treatment compared to the Standard Setup (compare Figure 7.2a to 7.2c). Performance actually decreased slightly for the Left & Right problem, and decreased noticeably for the MVG treatments ($p < 0.05$, Mann-Whitney U rank test). Although the effect was not dramatic, this result confirms a previous finding that different geometric representations of a problem can affect HyperNEAT's performance (Chapter 6). Computational limits prevented us from testing additional geometric representations, but it is not obvious how to create a geometric layout that would substantially increase the left-right bias more than the representation I tested. Based on these tests, I believe it is unlikely that changes to the geometric representation alone will make a substantial improvement in HyperNEAT's performance on the Retina Problem.

7.3.5 Simplified Retina problem

The previous experiments in this chapter have failed to demonstrate that HyperNEAT is capable of producing modular ANNs. Instead, its ANNs tend to be more fully connected than modular ANNs would be. One explanation for these results is that HyperNEAT is simply incapable of generating modular ANNs. To test this hypothesis, I conducted experiments on a simple task that explicitly requires modularity. In this Simplified Retina Problem, there are eight inputs and two outputs (Figure 7.5). The goal of the network is to output the sum of the left four inputs in the left output, and the sum of the right four inputs in the right output. The output nodes had linear activation functions instead of sigmoid functions. The correct wiring for this task is to eliminate all connections between the left inputs and

the right output, and vice versa, creating two distinct modules. HyperNEAT was queried for all possible links between inputs and outputs, so it had to learn across evolutionary time to eliminate connections between the left and right sides. The fitness function rewarded networks that had smaller errors between the actual and outputted sum for the left and right sides.

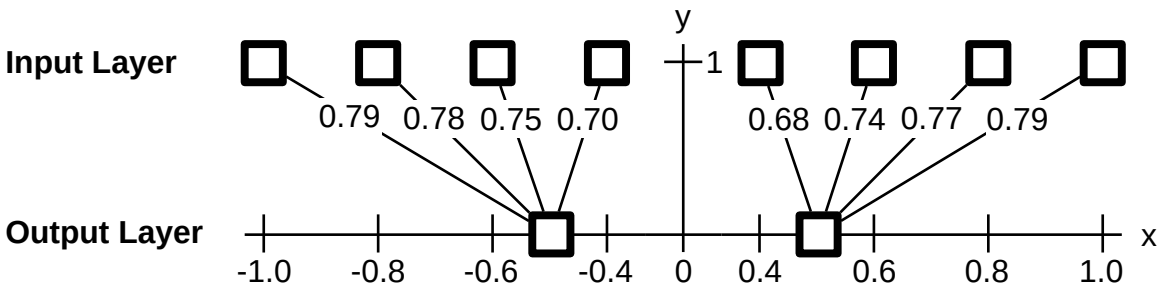

Figure 7.5: A Modular ANN Solution to the Simplified Retina Problem. Nodes (squares) are shown in their Cartesian locations. Links with a value of 0 are not shown. This champion from the end of a run has a nearly perfect fitness score because HyperNEAT created a modular ANN by deactivating links between the left and right sides.

Within 500 generations, all 20 runs had achieved near-perfect fitness scores (>98% of the maximum fitness). Additionally, in all but one run, the final champion had perfectly discovered the modularity of the problem by eliminating all links between the left and right sides. The sole run with imperfect modularity, which had the lowest fitness, had only one incorrect connection with a small weight value. The results from the Simplified Retina Problem allow us to reject the hypothesis that HyperNEAT is incapable of producing modular ANNs. This experiment provides the first documented case of HyperNEAT producing modular neural networks, albeit on a simple problem.

7.4 Discussion and conclusion

In contrast to other generative encodings that were explicitly designed to produce modularity, there is no *a priori* reason to expect HyperNEAT will produce modular networks.

Nevertheless, it would be beneficial if HyperNEAT could generate modularity when doing so would improve performance. In this chapter I tested whether HyperNEAT would generate modular ANNs on a suite of problems where both reason and experimental evidence suggest that modularity is helpful. HyperNEAT performed poorly on the problems and did not generate modular networks. Imposing modularity improved HyperNEAT's performance, indicating that HyperNEAT would have performed better had it been able to generate such modularity on its own. These results suggest that HyperNEAT has difficulty generating modularity on complex problems, although more research is necessary to determine if these results generalize to other problems.

Even with imposed modularity, HyperNEAT did not handle the MVG regime as effectively as Kashtan and Alon's direct encoding, where the MVG treatments outperformed the FG treatments. However, FT-NEAT, a direct encoding control for HyperNEAT, also did not generate modular and high-performing networks in MVG environments. These direct encoding control experiments thus suggest that the reason the MVG environments do not qualitatively differ from the FG environments for both HyperNEAT and FT-NEAT is likely explained by differences between the experiment implementations. One candidate explanation is that Kashtan and Alon's experiments have a smaller search space. For example, the link weights and thresholds seem to be discrete values with only a few options, instead of the continuous values in the experiments presented here. Mutations between a few discrete values may make it more likely that single mutations can switch between a solution to FG-OR and FG-AND. Kasthan and Alon report that networks in the MVG regimes evolved to switch between solutions to FG-OR and FG-AND with a single mutation [28]. If such a switch requires multiple mutations in the implementation used in this chapter, or a single rare mutation, evolution may be unlikely to benefit from modular phenotypes because it cannot quickly rearrange modules. I will investigate this hypothesis in future work. However, even without the benefits of reorganization (e.g., in the unchanging Retina Left & Right treatment), imposed modularity benefitted HyperNEAT, so we cannot conclude that

HyperNEAT had no incentive to produce modularity.

Despite HyperNEAT's difficulties with generating modularity on variants of the Retina Problem, I showed that it is capable of producing modular ANNs on the Simplified Retina Problem. While these results demonstrate for the first time that HyperNEAT can generate modular phenotypes, the results from variants of the more complicated Retina Problem suggest that more research is needed to understand how to generate modular ANNs via HyperNEAT on complex problems. One interesting possibility for future work is that making long connections (connections between distant nodes) rare may be a general way to encourage modularity. In natural brains, largely due to physical constraints, most connections between neurons are short, which may help explain why natural brains are so modular [49].

Given the promise of the HyperNEAT approach to evolving complex ANNs, it is worthwhile to investigate the degree to which it produces phenotypic regularity, modularity, and hierarchy, which are traits that facilitate the evolution of complexity in natural organisms. While HyperNEAT excels at producing regular phenotypes (Chapters 3-6) [16, 17], it was unknown whether it produced modular and hierarchical phenotypes. This chapter contains a preliminary study that demonstrates that HyperNEAT can generate modular phenotypes on a simple problem, but may struggle to do so on more complex problems. In my future work I will investigate how to increase HyperNEAT's ability to evolve modular phenotypes on complex problems. I will also study how HyperNEAT might be able to evolve artificial neural networks that are hierarchical.

Chapter 8

Conclusion

In this dissertation I illuminated some of the differences between generative encodings and direct encodings for evolutionary algorithms (EAs). I investigated two key properties of natural organisms that generative encodings were designed to produce in synthetically evolved phenotypes: regularity and modularity. To do so, I compared a promising new generative encoding to its direct encoding controls. This case study helps us understand the general differences between generative and direct encodings, and also reveals certain properties about a new class of generative encodings that follow nature in determining attributes of phenotypic elements as a function of their geometric location.

It had previously been shown that generative encodings outperform direct encodings on highly regular problems. It was not known, however, how generative encodings compare to direct encodings on problems with different levels of regularity. On three different problems, I showed that a generative encoding exploited intermediate amounts of problem regularity, which enabled it to increasingly outperform direct encoding controls as problem regularity increased. One of these problems was a challenging engineering problem (producing gaits for legged robots). Analyses revealed that the generative encoding outperformed the direct encoding controls on regular problems by producing regular brains (ANNs) that produced regular behaviors (gaits). The brains evolved with the generative

encoding contained a diverse array of complicated, regular neural wiring patterns, whereas the brains produced by a direct encoding control were irregular. Furthermore, all of the gaits produced by the generative encoding were coordinated, whereas those created by the direct encoding controls were mostly uncoordinated.

I also documented that the bias towards regularity can hurt a generative encoding on problems that have some amount of irregularity. I proposed a new algorithm, called HybrID, which combines the best attributes of generative and direct encodings: the generative encoding produced regularities that the direct encoding then adjusted in irregular ways to account for problem irregularities. HybrID outperformed a generative encoding alone on three problems for nearly all levels of regularity, and its performance advantage tended to be greatest on problems with some irregularity. The performance of HybrID shows that a generative encoding alone can struggle to adjust the regular patterns it produces to handle problem irregularities. HybrID further demonstrates that adding a process of refinement that fine-tunes the regular patterns produced by a generative encoding in an irregular way can substantially boost the performance of a generative encoding. HybrID's ability to improve upon the performance of a generative encoding alone raises the question of whether generative encodings may ultimately excel not as stand-alone algorithms, but by being combined with a further process of refinement. In this dissertation, that further process of refinement was a direct encoding. In future work, it would be interesting to study whether an intralife-learning algorithm could serve as this refinement process.

The results described so far in this conclusion show that a generative encoding can produce regular solutions. One interesting question is whether engineers who use generative encodings can bias the types of regularities evolution produces. I showed that, at least for the generative encoding in this case study, it is possible to influence the types of regularities produced, which allows domain knowledge and preferences to be injected into the algorithm.

In the final chapter, I turned to the question of whether this generative encoding can pro-

duce modular solutions. On a challenging problem, I found that the generative encoding struggled to produce modularity, even though it would have been beneficial to do so. On a simpler problem, however, the generative encoding frequently produced modular solutions. This work thus contains the first documented case of this generative encoding producing a modular phenotype, but its inability to create modularity on harder problems where modularity would have been beneficial reveals that more work is needed to increase the likelihood that this encoding produces modular ANNs in response to challenging, decomposable problems. In future work I plan to test different ways to encourage such modularity in this encoding. For example, I predict that making it unlikely for distant nodes to be connected should increase the likelihood of modular networks.

This dissertation contains an extensive case study focusing on one generative encoding, HyperNEAT, and its direct encoding controls. The benefit of this approach is that it allows a deep study of this particular generative encoding. The cost of this approach, however, is that it is not yet known whether similar conclusions generalize to other generative encodings. I hypothesize that many of the conclusions regarding the property of regularity will generalize. Specifically, I think that, generally, direct encodings are blind to the regularity of problems and that generative encodings can exploit problem regularity. I therefore predict that direct encoding controls will outperform generative encodings on irregular problems, and that generative encodings will increasingly outperform direct encoding controls as problem regularity increases. I am unsure whether it will be generally common that generative encodings can exploit a type of regularity only once the degree of regularity within that type is above a threshold. I also predict that engineers will be able to bias the types of regularities produced by generative encodings, but how to do that will depend on the particular encoding. The method described here, for example, will work only for those encodings that explicitly include geometric information in the algorithm. Similarly, I believe that different generative encodings will have different propensities to generate modular phenotypes. The troubles HyperNEAT had with producing modularity,

however, should apply to other algorithms that similarly make neural connections a function of geometry, unless additional constraints are added to encourage modularity. It would be worthwhile in future work to test all of these hypotheses on as many pairings of generative encodings and direct encoding controls as possible. Unfortunately, however, it is rare for a generative encoding to have a good direct encoding control.

Overall, this dissertation paints a more complete picture of generative encodings than prior studies. Initially, it analyzes the impact of the continuum between irregularity and regularity on the performance of generative versus direct encodings. This dissertation also provides an extensive analysis of a main reason generative encodings outperform direct encodings, which is their ability to exploit problem regularity. Specifically, I show that neural controllers evolved with generative encodings exploit problem regularity by creating regular brains that produce regular behaviors. This dissertation also contains the first extensive study documenting that a bias towards regularity can harm the performance of direct encodings when problems contain irregularities. A path forward is suggested, however, by the HybrID algorithm, which reveals that a process of refinement that can adjust the regular patterns produced by a generative encoding can boost performance by accounting for problem irregularities. I also showed that the types of regularities produced by a generative encoding can be biased to incorporate user preferences. Finally, this dissertation documents that the generative encoding studied can produce modular networks, but that more work is necessary to make modularity-production occur on anything but simple problems. At the highest level, this dissertation supports the idea that generative encodings are likely to play an important role in our long-term goal of synthetically evolving complex phenotypes that approach the complexity of phenotypes in the natural world.

BIBLIOGRAPHY

[1] R. Beer and J. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive behavior*, 1(1):91, 1992.

[2] S. Carroll. *Endless forms most beautiful: The new science of evo devo and the making of the animal kingdom*. WW Norton & Company, 2005.

[3] J. Clune, B. Beckmann, P. McKinley, and C. Ofria. Investigating whether hyperneat produces modular neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2010 (to appear).

[4] J. Clune, B. Beckmann, C. Ofria, and R. Pennock. Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 2764–2771, 2009.

[5] J. Clune, B. Beckmann, R. Pennock, and C. Ofria. HybrID: A Hybridization of Indirect and Direct Encodings for Evolutionary Computation. In *Proceedings of the European Conference on Artificial Life*, 2009.

[6] J. Clune, C. Ofria, and R. Pennock. How a generative encoding fares as problem-regularity decreases. In *Parallel Problem Solving from Nature*, pages 358–367. Springer, 2008.

[7] J. Clune, C. Ofria, and R. Pennock. The sensitivity of HyperNEAT to different geometric representations of a problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 675–682. ACM, 2009.

[8] J. Clune, K. Stanley, R. Pennock, and C. Ofria. On the performance of indirect encoding across the continuum of regularity. 2011 (submitted).

[9] D. D'Ambrosio, J. Lehman, S. Risi, and K. Stanley. Evolving Policy Geometry for Scalable Multiagent Learning. In *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems*, 2010 (to appear).

[10] D. D'Ambrosio and K. Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 974–981. ACM, 2007.

[11] D. D'Ambrosio and K. Stanley. Generative encoding for multiagent learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 819–826. ACM, 2008.

[12] A. Eiben, Z. Michalewicz, M. Schoenauer, and J. Smith. Parameter control in evolutionary algorithms. *Parameter Setting in Evolutionary Algorithms*, pages 19–46, 2007.

[13] D. Filliat, J. Kodjabachian, and J. Meyer. Evolution of neural controllers for locomotion and obstacle avoidance in a six-legged robot. *Connection Science*, 11(3):225–242, 1999.

[14] J. Gallagher, R. Beer, K. Espenschied, and R. Quinn. Application of evolved locomotion controllers to a hexapod robot. *Robotics and Autonomous Systems*, 19(1):95–103, 1996.

[15] J. Gauci and K. Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 997–1004. ACM, 2007.

[16] J. Gauci and K. Stanley. A case study on the critical role of geometric regularity in machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 628–633. AAAI Press, 2008.

[17] J. Gauci and K. Stanley. Autonomous evolution of topographic regularities in artificial neural networks. *Neural Computation*, 2010 (to appear).

[18] J. Grimbleby. Automatic analogue circuit synthesis using genetic algorithms. *IEE Proceedings Circuits Devices and Systems*, 147(6):319–324, 2000.

[19] F. Gruau. Automatic definition of modular neural networks. *Adaptive Behavior*, 3(2):151–183, 1994.

[20] F. Gruau and E. LIP. Genetic synthesis of Boolean neural networks with a cell rewriting developmental process. In *Combinations of Genetic Algorithms and Neural Networks, International Workshop on*, pages 55–74, 1992.

[21] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 81–89. MIT Press, 1996.

[22] A. Hampton and C. Adami. Evolution of robust developmental neural networks. In *Proceedings of Artificial Life IX*, pages 438–443, 2004.

[23] L. Hartwell, J. Hopfield, S. Leibler, and A. Murray. From molecular to modular cell biology. *Nature*, 402(6761):47, 1999.

[24] G. Hornby. Functional scalability through generative representations: the evolution of table designs. *Environment and Planning B*, 31(4):569–588, 2004.

[25] G. Hornby, H. Lipson, and J. Pollack. Generative representations for the automated design of modular physical robots. *IEEE Transactions on Robotics and Automation*, 19(4):703–719, 2003.

[26] G. Hornby and J. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002.

[27] G. Hornby, S. Takamura, T. Yamamoto, and M. Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410, 2005.

[28] N. Kashtan and U. Alon. Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences*, 102(39):13773, 2005.

[29] C. Klingenberg. Developmental constraints, modules, and evolvability. *Variations*, pages 1–30, 2005.

[30] J. Kodjabachian and J. Meyer. Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. *IEEE Transactions on Neural Networks*, 9(5):796–812, 1998.

[31] A. Kreimer, E. Borenstein, U. Gophna, and E. Ruppin. The evolution of modularity in bacterial metabolic networks. *Proceedings of the National Academy of Sciences*, 105(19):6976, 2008.

[32] A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[33] H. Lipson. Principles of modularity, regularity, and hierarchy for scalable systems. *Journal of Biological Physics and Chemistry*, 7(4):125, 2007.

[34] H. Liu and H. Iba. A hierarchical approach for adaptive humanoid robot control. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2004.

[35] J. Miller. Evolving a self-repairing, self-regulating, French flag organism. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 129–139. Springer, 2004.

[36] S. Nolfi, D. Floreano, and D. Floreano. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press, Cambridge, MA, 2000.

[37] M. Parter, N. Kashtan, and U. Alon. Facilitated variation: how evolution learns from past environments to generalize to new environments. *PLoS Computational Biology*, 4(11), 2008.

[38] R. Raff and J. Slack. *The shape of life: genes, development, and the evolution of animal form*. University of Chicago Press Chicago, 1996.

[39] M. Raibert, M. Chepponis, and H. Brown Jr. Running on four legs as though they were one. *IEEE Journal of Robotics and Automation*, 2(2):70–82, 1986.

[40] J. Reisinger and R. Miikkulainen. Acquiring evolvability through adaptive representations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, page 1052. ACM, 2007.

[41] C. Ridderstrom. Legged locomotion control—a literature survey. Technical Report TRITA-MMK, Royal Institute of Technology, Stockholm, Sweden, ISSN 1400-1179. 1999.

[42] J. Secretan, N. Beato, D. D Ambrosio, A. Rodriguez, A. Campbell, and K. Stanley. Picbreeder: evolving pictures collaboratively online. In *Proceedings of the Computer Human Interaction Conference*, pages 1759–1768. ACM, 2008.

[43] K. Sims. Evolving 3D morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.

[44] C. Southan. Has the yo-yo stopped? An assessment of human protein-coding gene number. *Proteomics*, 4(6):1712–1726, 2004.

[45] K. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.

[46] K. Stanley, D. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

[47] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[48] K. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

[49] G. Striedter. *Principles of brain evolution*. Sinauer Associates Sunderland, MA, 2005.

[50] M. Taylor, P. Stone, and Y. Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167, 2007.

[51] R. Téllez, C. Angulo, and D. Pardo. Evolving the walking behaviour of a 12 dof quadruped using a distributed neural architecture. *Biologically Inspired Approaches to Advanced Information Technology*, pages 5–19, 2006.

[52] V. Valsalam and R. Miikkulainen. Modular neuroevolution for multilegged locomotion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 265–272. ACM, 2008.

[53] P. Verbancsics and K. Stanley. Task transfer through indirect encoding. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2010 (to appear).

[54] G. Wagner. Homologues, natural kinds and the evolution of modularity. *Integrative and Comparative Biology*, 36(1):36, 1996.

[55] G. Wagner and L. Altenberg. Complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976, 1996.

[56] D. Wettergreen and C. Thorpe. Gait generation for legged robots. In *IEEE International Conference on Intelligent Robots and Systems*, pages 1413–1420, 1992.

[57] K. Wolff and P. Nordin. An evolutionary based approach for control programming of humanoids. In *Proceedings of the 3rd International Conference on Humanoid Robots*, 2003.