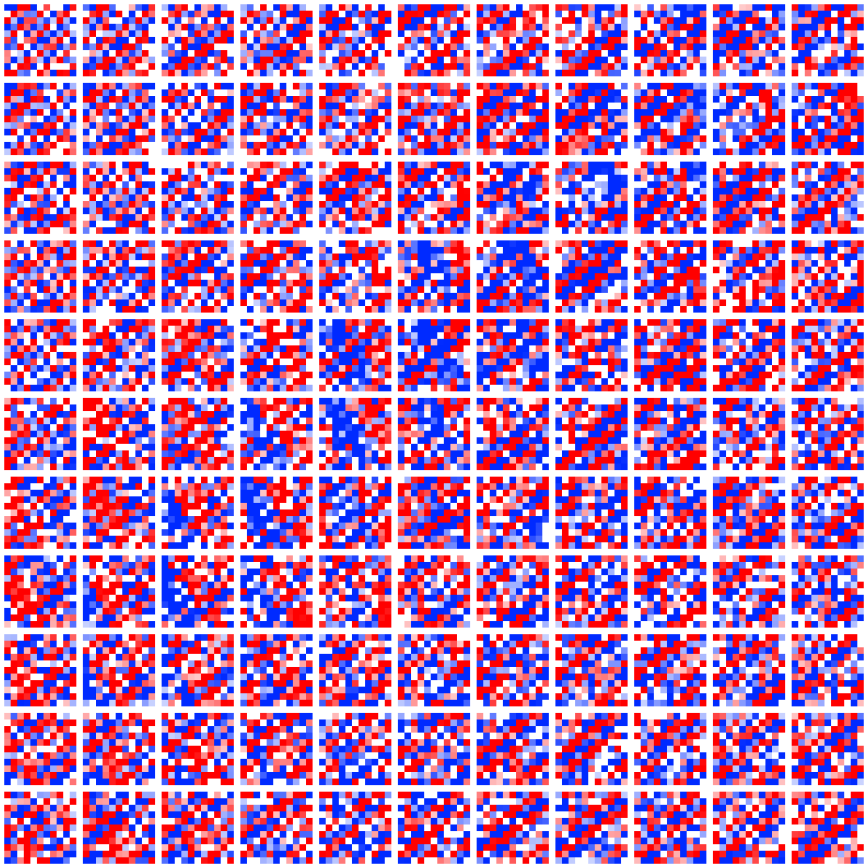
An Empirical Analysis of HyperNEAT



Thomas van den Berg

<thomas.g.vandenberg@gmail.com>

Supervisor: Shimon Whiteson

This thesis is submitted in partial fulfillment of the requirements for the University of Amsterdam's Master of Science Degree in Artificial Intelligence.

August 21, 2013

Abstract

HyperNEAT is a popular indirect encoding method for evolutionary computation that has performed well on a number of benchmark tasks. This thesis reexamines a selection of those tasks in an attempt to learn more about the underlying reasons for HyperNEAT's performance. First, we determine the fewest hidden nodes a genotypic network needs to solve these tasks. Our results show that they can all be solved with at most one hidden node. This means that they are in fact easy tasks. We make the task more difficult to see whether HyperNEAT can fulfill its promise of evolving complex, regular patterns in challenging tasks. HyperNEAT's performance is significantly impacted by this increase in difficulty, and in many cases it can be considered to have failed the difficult task. We hypothesize that *fracture* in the problem space, known to be challenging for regular NEAT, is even more so for HyperNEAT. In an experiment where we control for fracture, we show that HyperNEAT's performance decreases quickly as fracture increases. We connect these results to earlier experiments on the effects of *irregularity*, and show that it is likely that irregularity is an extreme form of fracture. Most importantly, we are still left with little evidence showing that HyperNEAT can optimize complex networks when those are necessary.

Contents

Chapter 1

Introduction

Optimization methods try to select the best element or parameters from a set of candidate solutions to a problem. Introducing randomness into this search process can accelerate progress. Methods that do this are *stochastic* optimization methods. Within this field are methods inspired by the process of natural selection, known as *genetic algorithms* (GAs) (Goldberg, 1989). These methods *evolve* a set of candidate solutions, repeatedly selecting and reproducing the ones with the highest *fitness*, as given by the *fitness function*, which maps candidates to a fitness value. This process of iterative optimization continues until a sufficiently fit solution has been found or until the available computational resources have been exhausted.

Reinforcement-learning tasks are among the optimization problems that can be solved with GAs (Moriarty et al., 1999). In these tasks, an agent learns how to take *actions* in an *environment* to maximize some notion of cumulative *reward*. It does this by somehow representing a *policy*: a mapping from the *state* the robot is in to the (optimal) action. To illustrate this: a chess-playing agent might learn what the best move (action) is for every board position (state) in order to win the game (for which it is rewarded). When using a GA to solve a reinforcement-learning problem, the candidate solutions represent such policies. Most of the popular evolutionary methods for reinforcement-learning tasks are based on *neuroevolution* (Yao, 1999), i.e., they evolve a neural network which serves as a controller or other policy representation.

In many task domains, the problem of *scalability* arises: we would like to evolve more complex networks to perform larger or more difficult tasks. The size of the evolved networks has a profound effect on the efficacy of the GA, because it relies on random mutations to optimize each element of the genotype (c.q. the network). At some point, directly optimizing large genotypes becomes an insurmountable task. Imagine evolving a controller for a millipede robot; intuitively, though there are many legs, we would like to encode the essential walking motion only once, and not have to discover the beneficial mutations separately for each leg. Methods that implement this abstraction are often called *indirect encodings*.

Indirect encodings or *generative developmental systems* (GDSs) (Astor and Adami, 2000; Hornby et al., 2001; Lindenmayer, 1968; Stanley and Miikkulainen, 2003) were designed to mitigate this problem of scalability. Unlike traditional direct encodings in evolutionary computation, in which the genotype is identical

or isomorphic to the phenotype, indirect encodings employ a more complex process to *develop* the phenotype from the genotype. They hold enormous promise for evolutionary computation because they enable the reuse of genes, i.e., each gene can be expressed in multiple places throughout the phenotype. As a result, not only do they encourage phenotypes with the regular structure often needed for complex tasks, they also have the potential for greater scalability, since the dimensionality of the phenotype can be far larger than that of the genotype (Stanley and Miikkulainen, 2003).

However, getting indirect encodings to work in practice has proven challenging, and the best way to represent the mapping from genotype to phenotype has long been unclear. Fortunately, substantial progress has been made in the last few years; most notable is the advent of *HyperNEAT* (Stanley et al., 2009), which has quickly become the most popular indirect encoding method for neural networks. In contrast to previous indirect encoding methods, which build the phenotype temporally, in step-by-step fashion (Astor and Adami, 2000; Hornby et al., 2001; Lindenmayer, 1968), HyperNEAT "captures the essential properties of natural developmental encoding without implementing a process of development" (Stanley, 2006). In particular, HyperNEAT uses a genotypic *compositional pattern producing network* (CPPN) to draw the connectivity of the phenotypic *substrate network* on the inside of a hypercube. By exploiting a geometric representation of the space in which the substrate resides, HyperNEAT "can produce spatial patterns with important geometric motifs that are expected from generative and developmental encodings and seen in nature." (Stanley et al., 2009).

Since its introduction, HyperNEAT has enjoyed a number of empirical successes. It performed translation-invariant detection of objects (Stanley et al., 2009; Coleman, 2010) and discovered a controller for food-gathering (Stanley et al., 2009) and line-following (Buk et al., 2009; Drchal et al., 2009) robots. HyperNEAT also evolved the walking behavior of a quadruped robot (Clune et al., 2009a; Risi and Stanley, 2013), controllers for robot keepaway (Verbancsics and Stanley, 2010), produced an evaluation function for checkers (Gauci and Stanley, 2008), and simultaneously evolved controllers for multiple agents using a single genotype (D'Ambrosio and Stanley, 2008; D'Ambrosio et al., 2010).

The research in this thesis is split up into three main research questions. First, we examine the question of why HyperNEAT has been successful on the existing benchmarks. The hypothesis is that though the tasks themselves exhibit a certain complexity and regularity, the CPPNs required to solve them do not. To test this we compare the performance of HyperNEAT to that of a variant limited to zero or one hidden nodes, which is only able to generate the most trivial patterns. It turns out that these variants indeed perform as least as well as unrestricted HyperNEAT. This means that these tasks do not provide any evidence that HyperNEAT can discover complex representations for difficult tasks.

Second, we try to answer the question of whether HyperNEAT is able to solve more difficult benchmarks tasks. We do this by increasing the difficulty of each of the earlier problems, while retaining the regular structure. We contrast HyperNEAT with an alternative—simpler—connectivity hypercube method that does not depend on a CPPN to generate the weights. On almost every task, HyperNEAT is outperformed by this method, and on most, HyperNEAT can be considered to have failed the task. Almost none of these experiments show Hy-

perNEAT outperforming a variant restricted to a single hidden node, so those still provide no evidence that HyperNEAT is able to optimize complex CPPNs.

Finally, we look at fracture as a possible source of difficulty for HyperNEAT. It has already been shown that problem spaces with high fracture are difficult for NEAT: the method on which HyperNEAT is based (Kohl and Miikkulainen, 2009). We relate the concept of fracture to that of irregularity—in the form of noise—used in (Clune et al., 2011). Our results suggest that HyperNEAT suffers badly from irregularity, as it is an extreme form of fracture. We then introduce a generative definition of *fracture* to create a problem space with discontinuous variation. Our results in these experiments show that HyperNEAT suffers even from a mild degree of fracture, possibly because it requires quite a complex CPPN to represent.

Overall, we found that the examined benchmark tasks were easy after the dimensionality reduction performed by HyperNEAT's representation. While it is an impressive feature of HyperNEAT's encoding, these tasks do not provide evidence for its capability of optimizing complex genotypes. Our experiments on more difficult tasks support this picture: on most of them HyperNEAT performed poorly and did not outperform a variant restricted to a single hidden node. Fracture might be a factor contributing to HyperNEAT's difficulty, as it was shown to negatively impact performance in our final experiments.

The rest of this thesis is structured as follows. Chapter 2 gives an overview of related methods and an introduction into the methods actually used in the rest of the thesis. Chapter 3 familiarizes the reader with some of the work that has been done that is closely related to the research in this thesis. Chapter 4 explains the newly introduced methods in detail. Chapters 5, 6, & 7 contain our experiments and their results. Chapter 8 treats some limitations of our experiments and suggests possibilities for expanding this work and mitigating the hypothesized problems with HyperNEAT. Finally, we present our conclusions in Chapter 9.

Chapter 2

Background

2.1 Genetic Algorithms

As mentioned in the introduction, there are many evolutionary methods for optimization problems. These *genetic algorithms* (GAs) have in common that they employ mutation, selection, and reproduction to iteratively refine a population of candidate solutions (Goldberg, 1989). Individuals in the population that perform better at the target problem have a higher probability of making it to the next generation. Though they are based on the same principles—relying on the presence of a fitness function—there is a huge variation in evolutionary methods, including distribution-based methods (Larrañaga and Lozano, 2002; Rubinstein and Kroese, 2004), algorithms for preserving diversity (Holland, 1975; Mahfoud, 1995), and advanced methods for selecting which individuals are allowed to reproduce (Heidrich-Meisner and Igel, 2009).

Among the optimization problems to which evolutionary methods are applied are reinforcement-learning tasks (Moriarty et al., 1999), where the candidate solutions represent policies for performing the task. The fitness is of each policy is determined by the average reward it obtains in a number of Monte Carlo trials in the task. In other words, the task is treated as a black box, returning only the effectiveness of a given policy as a whole; the algorithm then uses this information to search for an optimal policy. In contrast to *temporal difference* (TD) methods (Sutton and Barto, 1998, chap. 6), these kind of policy-search methods do not utilize any of the intermediate information that is present in the task, such as the states the individual passes through, the actions it chooses, and the intermediate reward it receives. This might put evolutionary methods at a disadvantage (Sutton and Barto, 1998). However, exactly when this information is misleading, e.g. because of partial observability, evolutionary methods can outperform TD methods (Whiteson et al., 2010). There are many more occasions of GAs outperforming TD methods (Whitley et al., 1994; Stanley and Miikkulainen, 2002; Gomez et al., 2008) and they remain a particularly popular class of optimization methods. A nice overview of using evolutionary methods for reinforcement learning is given by Whiteson (2012).

2.2 Neuroevolution

Many of the most popular GAs for reinforcement learning use a *neural network* to model the policy. In so-called *neuroevolution*, individuals in the population are genotypes describing neural networks. These networks are evaluated by simulating their activation in a specific task and looking at the outcome. In the most generic approach to neuroevolution (Yao, 1999), a population of networks is randomly initialized, and the fitness of each network is evaluated in the task. The best performing networks are selected through any of a number of selection methods (Goldberg and Deb, 1991). Then, they are mutated and optionally mated through crossover, resulting in the next generation. It is also possible to use *estimation of distribution algorithms*, such as *covariance matrix adaptation evolution strategy* (CMA-ES) (Hansen et al., 2003), to optimize neural networks for reinforcement-learning tasks (Igel, 2003). The latter effectively optimizes the weights of networks with a fixed size. Other successful methods are able to subject both the topology and the weights of the network to mutation (Fullmer and Miikkulainen, 1992; Gruau et al., 1996; Braun and Weisbrod, 1993; Stanley and Miikkulainen, 2002), allowing networks of appropriate sizes and complexity to be evolved.

2.3 NEAT

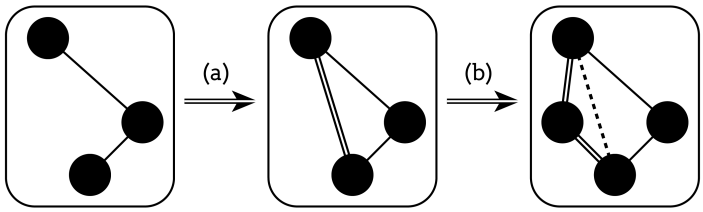


Figure 2.1: An illustration of the two mutation operators that allow complexification in NEAT: adding connections (a) and adding nodes (b). A new node is always added in the middle of an existing connection (the dashed line), which is then disabled.

Though somewhat complex to implement, one of the most successful methods for neuroevolution is *NeuroEvolution of Augmenting Topologies* (NEAT) (Stanley and Miikkulainen, 2002). It is one of the methods that evolve not only the weights but also the topology of the network, by way of mutation operators that add new nodes and new connections. At the core of NEAT is the principle of *complexification*: it starts with with a population of random minimal topologies, and adds complexity gradually through mutations, thereby keeping the search space small in early generations. NEAT adds *historical markings* to keep track of the genealogy of its individuals: during selection and reproduction, individuals with similar historical markers are put in the same *species*, wherein they share fitness. Because larger species have more individuals to share fitness with, speciation gives a relative advantage to smaller species even if their performance is worse, helping to preserve diversity in the population.

2.4 Indirect Encoding

NEAT and many other algorithms for neuroevolution employ *direct encodings*, i.e., the genotype and phenotype are isomorphic and the process for generating the latter is a simple translation step. In *indirect encoding* methods or *generative developmental systems* (GDSs), the phenotype is *generated* or *developed* from the genotype (Gruau, 1994; Stanley and Miikkulainen, 2003). The main goal of these is to be able to encode a much larger phenotype in a small genotype, thus keeping the search space small. An often made comparison is the one between the size of our human genome versus the number of neurons and connections in the human brain. The latter far exceeds the former, which suggests that our genotype does not directly encode the structure of our brains (at least the part of the structure of our brain that is determined by nature as opposed to nurture). This is assumed to be possible when the phenotype contains a lot of redundant or repetitive structures that can be efficiently represented by a set of generative rules. One classic indirect encoding method uses *L-systems* (Lindenmayer, 1968): formal grammars for recursively generating complex structures. These have been successfully used for generating both the bodies and brains for robots, outperforming direct encodings in (Hornby and Pollack, 2002). Another classic indirect encoding method: *cellular encoding* (Gruau, 1994) evolves graphs describing the construction of a phenotypical neural network. As an attempt to unify these methods where the phenotype is a neural network, (Stanley and Miikkulainen, 2003) proposes a taxonomy for such systems.

2.5 HyperNEAT

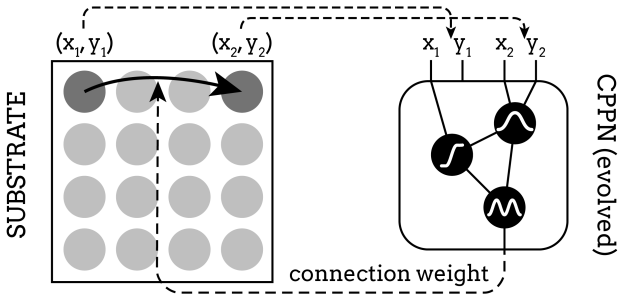


Figure 2.2: HyperNEAT evolves CPPNs that generate connection weights for the substrate networks applied to the task. The geometric locations of each pair of substrate nodes are fed to the CPPN to yield the weight of the connection between those nodes.

HyperNEAT is the indirect encoding extension of NEAT (Stanley et al., 2009). Instead of applying the evolved network directly to the task, HyperNEAT uses it as a *compositional pattern producing network* (CPPN). This CPPN is the genotype that defines the connections for a—possibly much larger—*substrate* network, which is then actually applied to the task. The CPPN is evolved by NEAT but the mutation operators are extended to allow different *node types*.

Regular NEAT generates networks containing only sigmoid nodes, but HyperNEAT uses an expanded set usually containing a sine, a bounded linear function, a Gaussian, a sigmoid, and an absolute value function. A substrate network is constructed by querying the CPPN for the connection weight between each pair of nodes in the substrate, with those nodes' Cartesian coordinates supplied as input. Thus, the CPPN is a compound function of pairs of node coordinates: $\text{CPPN}(x_1, y_1, x_2, y_2) = \text{weight}(n_1, n_2)$ (see Figure 2.2). In this way, HyperNEAT tries to exploit the fact that—in many tasks—a human with some domain knowledge can place substrate nodes in a space such that there is *regularity* in the desired connection weights. For example, it is natural to place the nodes representing a checkers board in a rectangular grid, so that HyperNEAT can exploit the fact that the board has two sides belonging to opposite players (symmetry), and that pieces can move to adjacent spaces throughout the board (repetition) (Gauci and Stanley, 2008). For more details on the HyperNEAT algorithm we refer to Stanley et al. (2009).

Chapter 3

Related Work

3.1 HyperNEAT Successes

As mentioned in the introduction, HyperNEAT has successfully solved a number of tasks. In (Drchal et al., 2009; Buk et al., 2009), it was used to evolve controllers for simple robots on a line-following task. HyperNEAT enabled the robots to stay on the line and avoid bumping into each other by staying on one side of the road. In (Verbancsics and Stanley, 2010), controllers are successfully evolved for Keepaway and Knight Joust tasks. Additionally, they show that a novel bird's eye view problem representation combined with HyperNEAT's encoding allows for effective transfer of knowledge from the Knight Joust task to—and between between different versions of—the Keepaway task. D'Ambrosio and Stanley (2008); D'Ambrosio et al. (2010) show that HyperNEAT makes it possible to evolve a single genotype that generates individual policies for multiple cooperating agents in a predator-prey task. These individual policies are different slices taken from the same substrate evolved by HyperNEAT. They significantly outperform a team of agents that homogeneously employs a single evolved policy. Moreover, they show that it is possible to generate policies for a larger team of agents using the genotype evolved for a smaller team by simply taking extra substrate slices. Gauci and Stanley (2008) use HyperNEAT to successfully exploit the geometric information in the checkers game. In their research, HyperNEAT outperforms both a naive direct encoding (NEAT) and a version of NEAT that starts with a large number of handmade geometry-related inputs. Clune et al. (2009a) show that HyperNEAT generates walking gaits for a four legged robot more efficiently than a direct encoding. They suggest that the generative aspect of HyperNEAT allows it to tweak the phenotypical network in a "holistic" manner, affecting each leg's behavior simultaneously.

3.2 Limitations

Besides the previously mentioned success stories, there is also a great deal of ongoing research into the limitations of HyperNEAT and their causes. First off, in (Clune et al., 2010), it was investigated whether HyperNEAT generates modular solutions for problems that would benefit from it. Their results show that this was not always the case, except for the simplest benchmark.

Attempts to encourage HyperNEAT to generate modularity by further separating the locations of proposed modules in the substrate or by increasing the threshold for creating connections were not successful. The problem seems to be that HyperNEAT tends to generate uniform and dense connection patterns, something that is consistent with our findings in Chapter 6. HyperNEAT-LEO (Verbancsics and Stanley, 2011) was designed to mitigate that problem by decoupling the weights from the connectivity pattern. This approach improved performance only when the CPPN structure generating the connectivity pattern was initially handcrafted using domain knowledge. We tried to see if using HyperNEAT-LEO improved results in our experiments, the results of this are in Appendix B.

Other research showed that decreasing the *regularity* of the required solutions increases the difficulty for HyperNEAT, such that a direct encoding eventually performs better (Clune et al., 2008, 2011). They introduce the *target weights* task to gain full control over the phenotype that a method has to generate. Similarly, Clune et al. (2009b) examine the sensitivity of HyperNEAT to different arrangements of substrate nodes, showing that random configurations, by introducing irregularity, negatively impact performance, even though HyperNEAT still outperforms a direct encoding in some tasks where irregularity is introduced. In Chapter 7, we relate these effects to the concept of problem space *fracture*, which has been shown to be a critical factor in the performance of regular NEAT (Kohl and Miikkulainen, 2009). Our experiments regarding fracture confirm those of previous experiments showing that irregularity decreases performance, but also show that HyperNEAT fails even at moderate levels of fracture, which correspond to very low levels of irregularity.

3.3 Target-Based Tasks

Figure 3.1: A very informal spectrum of fitness function strictness. The interactive evolution of Picbreeder (Secretan et al., 2011) belongs on the left. The target based experiments in (Woolley and Stanley, 2011; Clune et al., 2011) and in our Chapter 7 are represented all the way on the right. Our other experiments use regular fitness functions, and fall somewhere in the middle.

Often the success of evolutionary methods depends highly on the characteristics of the fitness function. Intuitively, evolution progresses more slowly if the maxima in the 'fitness landscape' are sparse. In a sense, this makes the fitness function "tight"; there are few good solutions, and they are hard to find. Some evolutionary methods deal better with this than others: to name just one example, CMA-ES (Hansen et al., 2003) adapts the covariance of the population distribution, possibly creating a wide search when it is needed. For Hyper-

NEAT, Woolley and Stanley (2011) demonstrate that the CPPNs generated under a "loose" fitness function (humans selecting which images they prefer) could not be reproduced by NEAT under a "tight" fitness function (minimizing distance to one specific image generated under the loose fitness function). These results can be put at either end of an informal spectrum of fitness function strictness (Figure 3.1). Our results in Chapter 7 are consistent with those—both are on the tight end of the spectrum—as we also show that HyperNEAT consistently fails when a specific nontrivial regularity is required in a *target-based* task. What (Woolley and Stanley, 2011) shows is that evolution failed on a fitness function that is *much* harder than the previously published success (the loose fitness function) because it limits the solution to a specific target phenotype. By contrast, we show that several of the previously published successes themselves require only trivial regularity and that making them even a *little* bit harder, without limiting them to a specific target phenotype, already induces failure (Chapter 6). These results give us some indication about HyperNEAT's performance in the middle of the spectrum.

3.4 Extensions to HyperNEAT

To overcome some of HyperNEAT's weaknesses, there have been a number of extensions to the method. (Buk et al., 2009) replaces the NEAT part of HyperNEAT with genetic programming, which improved performance on the line following task; this approach is similar to our wavelet baseline (see Chapter 7) but was not tested on more difficult problems in that paper. A method named HyperNEAT-ES is introduced in (Risi et al., 2010; Risi and Stanley, 2011). This extension makes the placement and density of neurons in the substrate amenable to evolution by HyperNEAT. It does so by looking at where the information is located in the connectivity hypercube, and adding enough nodes to express that information. Initially this didn't improve task performance, but the follow-up paper improves the algorithm to allow for a finer search of node locations and shows HyperNEAT-ES outperforming regular HyperNEAT in a maze navigation task. (Suchorzewski and Clune, 2011) introduces DSE, a cellular encoding method that incorporates programs that take arguments from the geometric coordinates, like in HyperNEAT. DSE outperformed HyperNEAT on a number of tasks, including one where the objective was to generate a modular network, and a visual discrimination task.

The method introduced in (Koutník et al., 2010; Gomez et al., 2012) has some components similar to HyperNEAT and the wavelet method we introduce in this thesis. By encoding the connectivity in the frequency domain, the size of the generated networks is independent from the size of the genotype. Searching through the frequency domain at different network complexities allows the method find small genotypes that still solve the different benchmarks.

Chapter 4

Methods

4.1 Limited Hidden Node HyperNEAT

To investigate the difficulty of the tasks in Chapter 5, we introduce a variant of HyperNEAT that is limited to a maximum number of hidden nodes in the CPPN. If this number is reached, HyperNEAT evolution continues as normal, but mutations that add nodes are no longer executed. We use this method only as a diagnostic baseline: if the task can be solved by such a restricted variant, that establishes an upper bound on the complexity of the task. Such an analysis is useful because many of the tasks to which we apply HyperNEAT were introduced within the field of GDS and not much is known yet about their complexity. In the rest of this thesis, we sometimes refer to these restricted variants as M0HN or M1HN for brevity.

To get an intuition of the amount of complexity that these restricted variants are capable of generating, exemplary patterns are shown in Figures 4.1 and 4.2. These patterns were generated by a fully connected CPPN with random weights and node types. In these figures, the connectivity of a one-dimensional substrate network is visualized as a two-dimensional image. To connect this to a functional example: a network where each node is connected to its neighboring nodes is visualized in Figure 4.1 on the right.

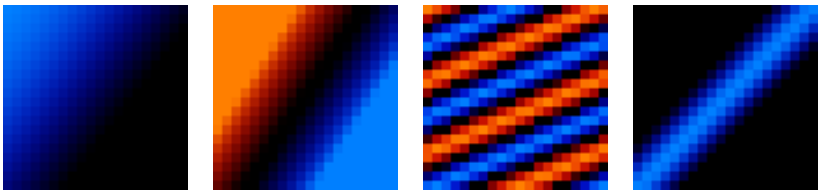


Figure 4.1: Examples of connectivity patterns of a one-dimensional substrate generated by a CPPN with no hidden nodes.

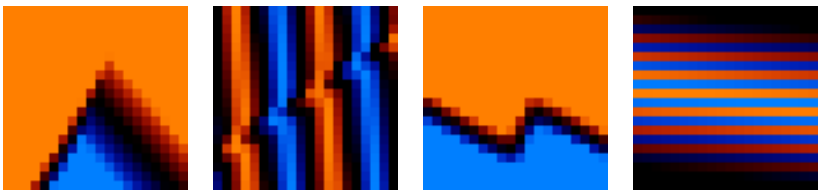


Figure 4.2: Examples of connectivity patterns of a one-dimensional substrate generated by a CPPN with a single hidden node.

4.2 Wavelet Encoding

In this thesis we introduce a new method—the "wavelet" method—and apply it to the more difficult problems of Chapter 6. Increasing the difficulty of the tasks might cause HyperNEAT to fail to solve them, so we need a way to verify that it is still possible to find a solution phenotype at all, i.e., that we haven't made the tasks *too* difficult. Applying the wavelet method does exactly that because it is able to solve all of the tasks in Chapter 6. A reason for its good performance might be the fact that it was designed to deal with *fracture* (Chapter 7). However, the goal of this thesis is not to contribute or advocate for this new method; this baseline is used only to establish that the task variants are not too difficult for *any* method.

This method builds the phenotype by adding a number of component *wavelets*. Wavelets have a frequency and a *location*, so this method can add features more specifically to demarcated parts of the phenotype, which aids in generating fractured patterns. The wavelet method is similar to HyperNEAT because it builds the connectivity in a substrate network using the geometric coordinate of the substrate nodes. Both HyperNEAT—because it uses sine nodes—and the wavelet method are somewhat similar to the method introduced in (Koutník et al., 2010), because they can perform a search in the frequency domain. However, what sets the wavelet method apart is the ability to *localize* these oscillations.

The wavelet method does not employ a CPPN to generate the connectivity, but instead computes connection weights using a sum over a number of *wavelet* basis functions. A wavelet can be seen as a brief oscillation: it is periodic, but the amplitude starts at zero, increases to a maximum, and then decreases. *Gabor wavelets*, often used in computer vision for image analysis, are among the simplest, as they are the product of a sine and a Gaussian (Daugman et al., 1985; Lee, 1996). In the wavelet method, the connectivity C of nodes located at (x_1, y_1) and (x_2, y_2), i.e., the weight of their connection, is:

$$C((x_1, y_1, x_2, y_2) = \sum_{i=0}^{N} W_i \left(\begin{bmatrix} x_1 & y_1 & x_2 & y_2 & 1 \end{bmatrix} \right),$$

The vector of node coordinates is projected to 2 dimensions using a matrix:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \mathbf{x} \cdot \mathbf{a_i} \\ \mathbf{x} \cdot \mathbf{b_i} \end{bmatrix} = \begin{bmatrix} \mathbf{a_{i0}} & \mathbf{a_{i1}} & \mathbf{a_{i2}} & \mathbf{a_{i3}} & \mathbf{a_{i4}} \\ \mathbf{b_{i0}} & \mathbf{b_{i1}} & \mathbf{b_{i2}} & \mathbf{b_{i3}} & \mathbf{b_{i4}} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ 1 \end{bmatrix}$$

So W_i, the i-th wavelet, is:

$$W_i(\mathbf{x}) = \alpha_i \cdot G(\mathbf{x} \cdot \mathbf{a_i}, \mathbf{x} \cdot \mathbf{b_i}, \sigma_i).$$

Here, G is the real part of a single Gabor wavelet (see Figure 4.3):

$$G(x, y, \sigma) = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \cos\left(2\pi x + \frac{\pi}{2}\right),$$

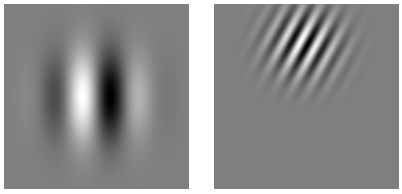


Figure 4.3: Left: a Gabor wavelet $G(x, y, \sigma)$ for x and y between $[-2, 2]$ and $\sigma = 0.5$. Right $G(x', y', \sigma)$ with transformed x' and y' and $\sigma = 1.5$. Lighter colors indicate higher output values.

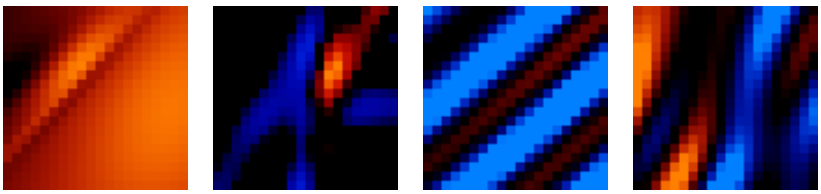


Figure 4.4: Examples of connectivity patterns of a one-dimensional substrate generated by the wavelet method. For each of these, three wavelets were added to new genotypes and then mutated three times.

Thus, each wavelet has the following parameters that must be optimized: $\alpha, \mathbf{a}_0, \ldots, \mathbf{a}_n, \mathbf{b}_0, \ldots \mathbf{b}_n, \sigma$. This is done with a simple evolutionary method employing two kinds of mutations. Either a new wavelet is added with probability 0.1, or, for each wavelet, the parameters are perturbed with probability 0.3. When a new wavelet is added, it is initialized with a random value from a normal distribution ($\mathcal{N}(0, 0.1)$) for all parameters. When a wavelet is perturbed, $\mathcal{N}(0, 0.1)$ is added to each parameter. *Tournament selection* is used to spawn a new generation. In tournament selection a small sample is repeatedly taken from the population, and the best individual from this sample is added to the next generation until the desired number of offspring is reached. The size of the

sample determines the selection pressure: a larger sample size increases selection pressure, and a sample size of one corresponds to completely random selection. We used a sample size of three, which offers a good trade-off between preserving variation and efficiency. Each of the individuals in this new generation is then mutated, except for the champion of the last generation, which is carried over unmodified.

In some tasks to which HyperNEAT is applied, the substrate has more than two layers of nodes, and thus requires more than one layer of connections. Often, the connectivity for each of these connection layers is generated by a separate output node in the CPPN. Because the wavelet method does not use a CPPN, it evolves a separate 'layer' of wavelets for each set of connections in the substrate, each of which is subject to the mutations above. So, in contrast to HyperNEAT, the weights for these different layers are necessarily *uncorrelated*. To get an idea of the kind of patterns that the wavelet method generates, the connectivity for a number of randomly determined genotypes is shown in Figure 4.4. In these figures, the connectivity of a one-dimensional substrate network is visualized as a two-dimensional image.

Chapter 5

Complexity

In this chapter, we examine the complexity of several tasks in which HyperNEAT has previously succeeded, where complexity is defined as the minimum number of CPPN hidden nodes required. To do so, we apply the limited hidden node variant of HyperNEAT (Chapter 4.1) to each task. To establish an upper bound on a task's complexity, we set the limit to different levels.

There are many existing implementations of NEAT and HyperNEAT, and there are slight differences in the way they implement these complicated methods. Additionally, each of these existing implementations includes—and is set up for—different benchmark tasks. We built our own implementation (van den Berg, 2013), in order to include as many benchmark tasks as possible within the same codebase. For the walking gait task, we used the existing implementation that was used for (Clune et al., 2011). It is unavoidable that our implementation of HyperNEAT differs from existing ones. To verify that these differences do not qualitatively affect our conclusions we performed an analysis which is reported in Appendix C.

We tried to match the parameters of each of the experiments in this chapter to the parameters of the original experiment, an overview of the most important settings is given in Table 5.1.

	Vis. Discr.	Line Foll.	Checkers	Walking Gait
Pop. size	100	100	120	150
Generations	200	100	30	1000
P(add node)	0.03	0.03	0.03	0.03
P(add conn.)	0.1	0.1	0.1	0.05
P(mutate weight)	0.8	0.8	0.8	0.8
Node types	Linear, Bounded, Sine, Gaussian, Sigmoid (logistic function)			Bounded, Sine, Gaussian, Sigmoid (hyperbolic tangent)
CPPN output node type	Any	Any	Any	Sigmoid

Table 5.1: Parameter settings used for each of the experiments in Chapters 5 & 6.

5.1 Visual Discrimination

The visual discrimination task, one of the first to show HyperNEAT's effectiveness (Stanley et al., 2009), is designed to mimic the way the same pattern can be recognized equivalently at different places throughout the visual field. This is a kind of regularity: there should be a similar response for each location at which the pattern appears and therefore similar substrate weights at each of these locations. The input to the task is an 11×11 image, containing a 3×3 *target square*, and a 1×1 *distractor square*. The goal is to "point out" the target square by giving the highest activation to the node corresponding to its center. Fitness is inversely proportional to the distance between the node with the highest activation and the target square's center. The substrate is a *sandwich network*: an input layer connects directly to an output layer, both corresponding to the image size. The original result showed that HyperNEAT can evolve a connectivity pattern for this sandwich network that solves the task (Stanley et al., 2009).

We employ the variant of this task that uses the location *deltas* (Stanley et al., 2009), so the CPPN receives an additional input of $\delta_x = x_1 - x_2$ and $\delta_y = y_1 - y_2$. In our experiment, the distractor object is placed randomly, in contrast to the original experiment, which placed it at a fixed location relative to the target square. Random placement ensures each target square location is paired with *different* distractor square locations during fitness evaluation, making the task more difficult.

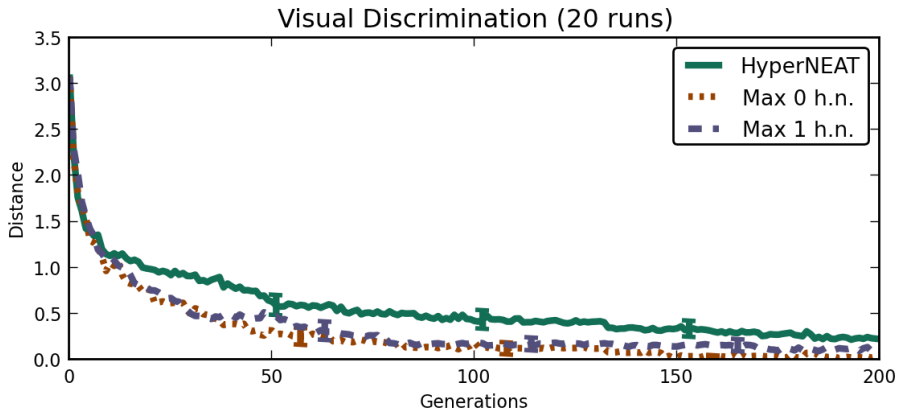


Figure 5.1: Average distance to target location of generation champions on visual discrimination task.

Figure 5.1 displays our results in this task, which show that performance does not degrade when HyperNEAT is limited to a CPPN *without* hidden nodes. This result is surprising, since a CPPN without hidden nodes can only compute a single function over a linear combination of its inputs, i.e., the solution connectivity pattern is of the form $f(w_1x_1 + w_2x_2 + w_3y_1 + w_4y_2 + w_5\delta_x + w_6\delta_y)$, where f is any of the allowed node function types in HyperNEAT.

Examination of the solutions reveals that they mostly rely on the fact that the target square has a larger mass and thus a higher imprint on the substrate activation. To accumulate this higher activation in the target location, each

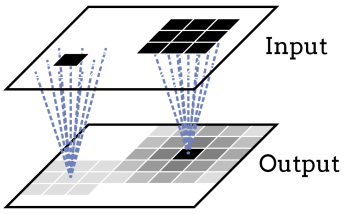


Figure 5.2: Solution to the visual discrimination task, showing incoming connections for two sample nodes. The target and distractor squares are activated and darker values indicate higher activation.

input node need only be connected to adjacent nodes in the output layer, as shown in Figure 5.2. Thus, the task can be solved by any CPPN that gives high values to connections if $x_1 - x_2 \approx y_1 - y_2 \approx 0$. A function such as gauss$(w_1x_1 + w_2x_2 + w_3y_1 + w_4y_2)$ meets this requirement when $w_1 \approx -w_2$ and $w_3 \approx -w_4$. The function gauss$(w_5\delta_x + w_6\delta_y)$ meets the requirement for positive values of w_5 and w_6. These are exactly the functions that nearly all the champions compute in runs where no hidden nodes are allowed. Hence, solving this problem requires optimizing at most four weights, as suggested in (Stanley et al., 2009). This kind of regularity, where connections are strong between adjacent nodes, is one of the most trivial, with only degenerate functions, e.g., outputting an equal weight for *every* connection, being simpler. By contrast, final champions of regular HyperNEAT runs used on average 11 hidden nodes to compute a similar function, suggesting substantial superfluous complexity.

5.2 Line Following

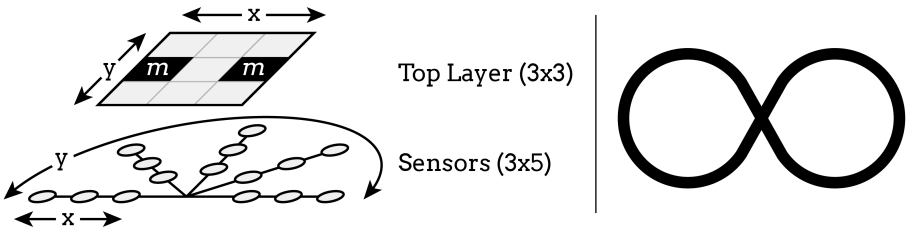


Figure 5.3: Left: substrate layout of the robot in the line following task. The black nodes marked 'm' control the wheel motors. Right: the figure-eight road that the robot must follow.

In the line following task, a simulated wheeled robot must drive on a flat terrain with zones of different friction (e.g. roads and grass) (Drchal et al., 2009; Buk et al., 2009). The goal is to maximize average speed by staying on the figure-eight road, where the friction is lowest. The robot has a number of sensors mounted on antennae that can detect the type of terrain under them (see Figure 5.3). Two independently controlled wheels enable driving and steering. The task has two forms of regularity. First, the robot is symmetrical as the

antennae are laid out radially and the wheels are opposite each other. Second, the five antennae are identical in length and response function.

The physics simulation was built on the Chipmunk physics engine (Lembcke, 2011), and the rules of the task were taken from the original experiment (Buk et al., 2009).

The substrate consists of two layers: a bottom layer containing all the sensor inputs and a top layer containing hidden nodes and the output nodes that control the wheels. The locations of the 5×3 sensor nodes are given in polar coordinates, i.e., the y-input to the CPPN is determined by which antenna the sensor is on, and the x-input by its distance from the center. The top layer only has 3×3 nodes and the output nodes are embedded in this layer. The nodes that control the wheels are on the mid-left and mid-right, determining the speed of the left and right wheel respectively (see Figure 5.3). The other nodes in the top layer serve as hidden nodes. The CPPN has three output nodes, which specify connections from the bottom to the top layer, internal connections in the top layer, and the top layer bias, respectively.

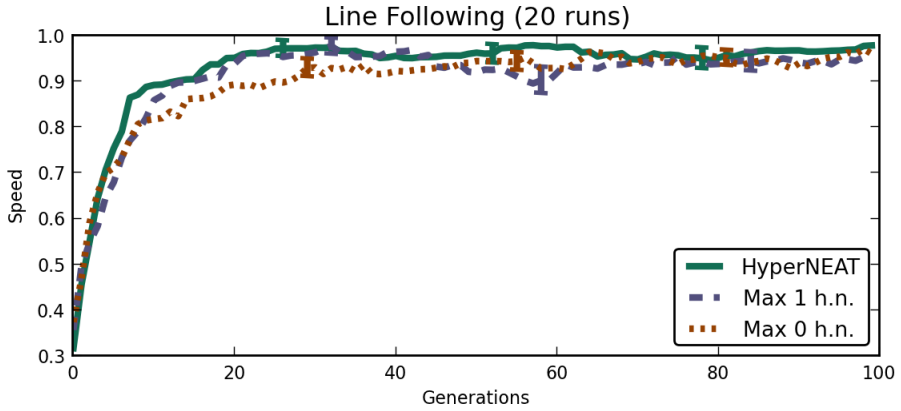


Figure 5.4: Average distance covered by generation champions on the line following task.

Figure 5.4 shows our results in this task. Again, performance does not suffer from a restriction in the number of CPPN hidden nodes. By contrast, the champions at the end of the regular HyperNEAT runs had an average of about 12 hidden nodes. A comparison to so a so-called *Braitenberg vehicle* (Braitenberg, 1986) can reveal why this task is so easy. Such a vehicle has a simple wiring pattern, yet exhibits complex emergent behavior. One of the simplest of these is a wheeled robot with two light sensors connected, respectively, to two wheels. This robot steers away from the light by sending high activation to the wheel on the same side as the sensor on which the light falls. When examining the evolved networks, they seem to exhibit a similar behavior: a high friction area observed on either side causes a high activation to input nodes on that side; by sending that high output to the wheel on that side, the robot steers in the *other* direction, away from it. Thus, the apparent symmetry in the task does not need to be exploited explicitly, i.e., though the task shows substantial regularity, solving it requires only very simple regularity: as in the visual discrimination task, nodes need only be connected to nodes that are nearby, which is trivial.

5.3 Walking Gait

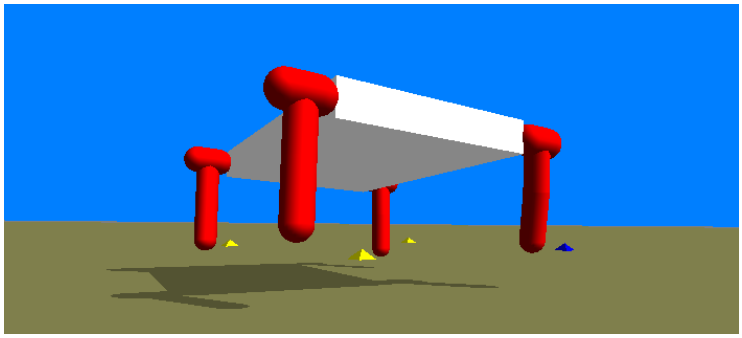


Figure 5.5: Rendering of the walking gait simulation.

In the walking gait task, a four-legged table-shaped robot (Figure 5.5) simulated in a physics engine has to walk as far as possible (Clune et al., 2009a). Since finding stable and energy efficient gaits for walking robots is famously challenging, HyperNEAT's success on this task is considered a significant result. Each leg has three joints: two in the hip that allow anteroposterior and lateral motion of the upper leg and one in the knee for flexing of the leg. Values on the substrate's output layer determine target angles for each of the 3×4 joints. The input layer specifies: the current angle of each joint, the angle of the body (roll, pitch, and yaw), whether each leg is touching the ground, and a time-based sine signal. In addition to the input and output layer, the substrate has a 3×5 hidden layer.

The task is regular because the legs have identical structure and likely need to behave similarly. One might think that the walking behavior also displays *temporal* regularity due to the repeated motion. However, this periodicity is provided externally by the sine input.

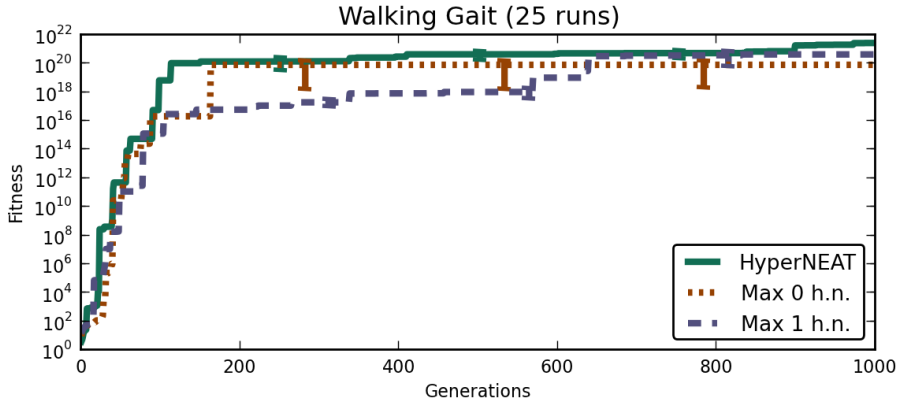


Figure 5.6: Average fitness of generation champions on the walking gait task.

Figure 5.6 displays our results in this task, which show the restricted vari-

ants can solve this task by the end of the run. By contrast, the champions at the end of the regular HyperNEAT runs had an average of nearly 12 hidden nodes. Analyzing the solutions reveals that moving all joints in synchrony with the supplied sine pulse results in a rigid hopping behavior, which is effective at covering distance. For the genotypes without hidden nodes, 17 out of 25 generated such a behavior. With unrestricted HyperNEAT, 6 out of 25 runs generated this kind of behavior. For either method, in almost all of the other runs the *front-right leg* moves in antiphase to the other three.

To generate the synchronized hopping behavior, the joint nodes in the substrate need only be uniformly linked to the sine input. Thus, only the CPPN inputs relating to the input coordinates of each connection are relevant, i.e., CPPN(x_1, y_1, x_2, y_2) should be high for a specific value of x_1, y_1, and low for all others, but the value of x_2, y_2 is irrelevant. A function such as gauss$(x_1 - w_1, y1 - w_2)$ accomplishes this, and requires optimizing only two weights. It is not surprising that the restricted CPPNs generate those trivial solutions, but apparently HyperNEAT generates solutions that are not more effective, albeit using more nodes.

5.4 Checkers

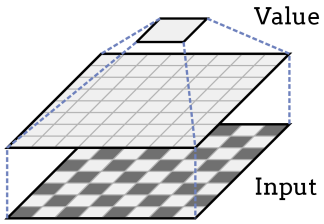


Figure 5.7: Illustration of the layout of the checkers substrate.

In the checkers task (Gauci and Stanley, 2008), HyperNEAT's goal is to evolve a good static *board evaluation function* for checkers, in order to beat an opponent with a fixed heuristic. Both the evolved player and its opponent use a heuristic search (minimax with alpha-beta pruning) to find good moves. The lookahead depth of the search is fixed, and the board evaluator is consulted at the search tree leaves. The opponent, *Simple Checkers* (Fierz, 2002), utilizes a static evaluation function based on a combination of piece counts and positions. For example, it assigns a higher value to pieces on the edge of the board and it gives favorable values to piece exchanges if the current player is at an advantage. Our experiments use a version of this opponent that was reimplemented in Python, available in the PEAS package (van den Berg, 2013). Different from the original task, all evolved players play against a non-deterministic version of this opponent: this opponent picks the *second-best* move found by minimax 5% of the time. This is done to obviate the need for a separate training and generalization test and to prevent ties when the optimal move for both players would lock them in a cycle of repeating moves.

Checkers poses a highly regular task because the players symmetrically oppose each other and because the abilities of each piece are identical throughout

the board. Figure 5.7 shows the setup of the substrate, with a single output node for the board value and a single hidden layer. The CPPN in this task has two output nodes: one for determining the connectivity from the input to the hidden layer, and one for the connections to the output node. For the evolved player, the board is evaluated by feeding the two-dimensional board directly into the generated network, with values of 0.5 and 0.75 for black men and kings respectively, and -0.5 and -0.75 for white pieces. The fitness is a sum over the piece count differences during the last 100 turns of the game, evaluated using the following equation:

$$100 + 2m_e + 3k_e + 2(12 - m_o) + 3(12 - k_o)$$

where m_e and k_e are the number of men and kings for the evolved player, playing as black, and m_o and k_o the number of pieces for the opponent, playing as white. If the evolved player wins the game, a bonus fitness of 30,000 is added.

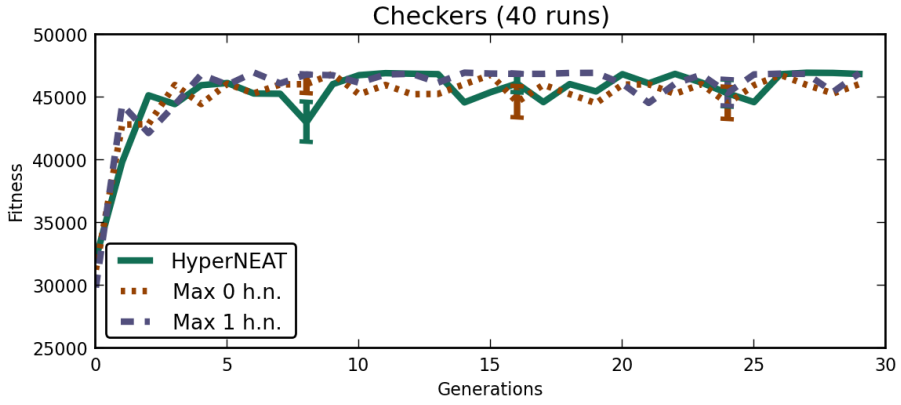


Figure 5.8: Average fitness of generation champions on the checkers task.

The results in 5.8 show that the restricted variants can solve the task using few or no hidden nodes, whereas HyperNEAT uses about 4 on average after 30 generations. Another surprising aspect of these results is that the evolved players learn to win even more quickly than in the original experiment from (Gauci and Stanley, 2008), which reports 8.2 generations on average. We verified that the resulting networks are indeed able to defeat deterministic and non-deterministic variants of the original opponent. This suggests that our implementation of HyperNEAT might be slightly more efficient.

Analyzing why exactly this task is so easy is not trivial, as the evaluation function can rely on very small numerical differences in its output that are hard to visualize. Examining the evolved substrate networks shows that many of them compute a function with a continuous variation in piece values throughout the board, suggesting that their operation might be a kind of "weighted" piece counter. A thorough examination of the networks that HyperNEAT generates to solve the checkers task can be found in (Gauci and Stanley, 2010). Though their solution networks are generated by unrestricted HyperNEAT, some of the same principles might apply. To give a lower bound on the complexity of this

task: a pure piece counter with the same lookahead depth does not defeat the opponent. Whatever the explanation might be, the fact remains that this task, too, is an easy one: it can be solved by a CPPN without hidden nodes.

The experiments in this chapter have provided us with useful information about the complexity of a number of tasks that HyperNEAT was applied on. We have shown that these tasks were in fact easy, and that they provide little support for HyperNEAT's ability to generate complex patterns. In order to assure that such benchmarks actually contribute evidence for a method's effectiveness, it is important to repeat this kind of analysis—using a restricted baseline—on more existing benchmark tasks, as well as include it with newly introduced benchmarks.

Chapter 6

Scalability

The results presented in the previous chapter show that the considered tasks can be solved with trivial CPPNs. On the one hand, this is a success for HyperNEAT since the point of indirect encodings is to perform dimensionality reduction so that large phenotypes can be evolved using small genotypes. On the other hand, it seems likely that, even after maximal dimensionality reduction, many realistic tasks require evolving nontrivial genotypes; these results show that previously published HyperNEAT successes on the considered tasks do not confirm its ability to do so. In this chapter, we present results of experiments designed to fill this gap. For each task, we applied HyperNEAT to the simplest extension that we could devise, in order to increment task difficulty only slightly.

HyperNEAT's parameters here are identical to those in the last chapter. We did not optimize them for these new tasks, so that our results would be informative with regard to the robustness of HyperNEAT to different task setups.

In addition, we compare the performance of HyperNEAT on these variations to that of the wavelet baseline algorithm that we devised. Note that the goal of this thesis is not to contribute or advocate for this new method. On the contrary, this baseline is used only to establish that the task variants are not too difficult for *any* method: if HyperNEAT fails at the task variant but the baseline algorithm succeeds, then we know the results indicate a limitation of HyperNEAT rather than an excessively difficult task.

6.1 Visual Discrimination

The original visual discrimination task could be solved by spreading out activation across the substrate and relying on the larger mass of the target square to increase activation of the correct output. To make this task more difficult, we turn it into a true *shape discrimination* task, where the target and distractor objects differ only in form, not size. Regularity is preserved because the same objects still need to be recognized across the entire visual field. In addition, the target and distractor objects are now mirror images of each other: they are both triangles created by diagonally slicing the original target square. The center location is still at the center of the bounding square, as shown in Figure 6.1.

Figure 6.2 shows our results for this task variation. To get an impression of

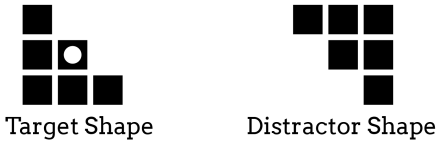


Figure 6.1: Shapes used in the difficult visual discrimination task. The center is indicated by the dot.

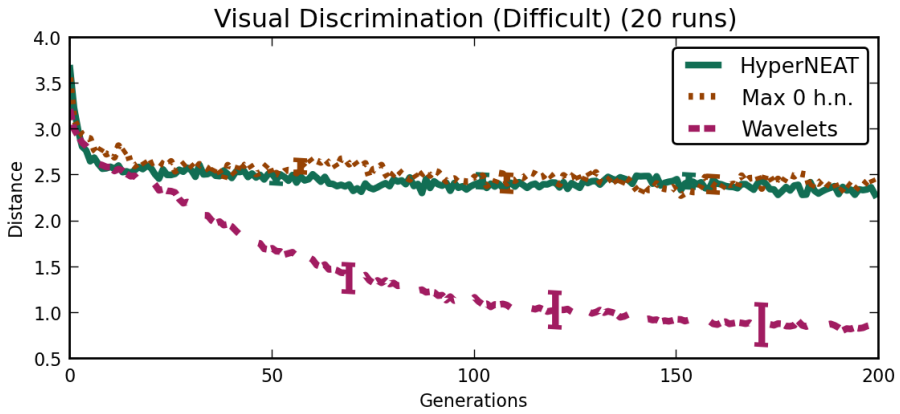


Figure 6.2: Average distance to target of generation champions on the difficult visual discrimination task.

the qualitative performance of the methods: the average distance to the center node is 3.4. This is the score obtained when the network always activates the node at the center of the visual field, regardless of input. When the generated substrate does not discriminate between the shapes, and simply picks the center of mass of the objects combined, a strategy similar to that in the previous experiment, the average distance is about 2.5. HyperNEAT performs slightly better than that but again fails to substantially outperform the variant forbidden to use hidden nodes[1]. The wavelet baseline method performs much better: its average distance to the target object is close enough that we can say it is surely discriminating between the two shapes. This indicates that the task is indeed solvable, but HyperNEAT fails to do so. Note, however, that a different HyperNEAT implementation, given a larger population, more generations, and optimized parameter settings, was able to perform better on this task (Stanley et al., 2013).

6.2 Line Following

This section describes another experiment where we increase the difficulty of a task to see whether HyperNEAT is still able to generate effective solutions. The increased difficulty in the line following task stems from a discrepancy between

[1]Results for the restricted variant vary when using different settings for HyperNEAT, an analysis is included in Appendix C.

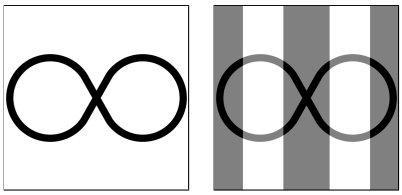


Figure 6.3: Left: the friction values of the terrain. Right: the *observation* in the difficult version of the line following task. The 'grass' in the dark zones and the road in the light zones both have a value of 0.5.

the observation and the actual friction value of the terrain. The road still looks darker than its surroundings, but the actual values are not the same everywhere, as shown in Figure 6.3. To solve this task, the substrate has to *compare* sensor values in order to find out whether they correspond to the road. Regularity is still present, as the layout of the robot did not change.

In addition, we introduced an extension to the fitness function because we found that the robots would sometimes evolve effective behaviors that are *not* line following. This also occurred in the experiments in (van den Berg and Whiteson, 2013). Without this extension, we found that most robots oscillate in the center region, without following the loops. When this is the case, HyperNEAT sometimes outperforms the restricted baseline and the wavelet method. An analysis of the different behaviors found in line following, the settings under which they occur, and the performance of the methods can be found in Appendix A. To encourage true line following for the experiment in this section, fitness is only awarded if the robot covers new ground that it has not visited recently.

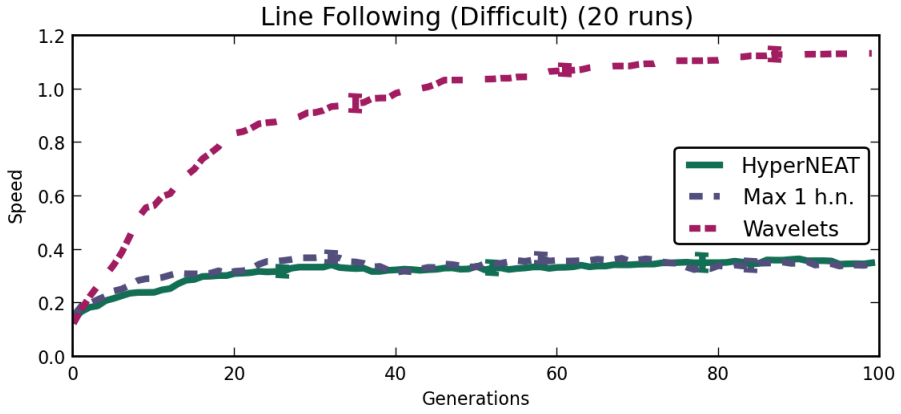


Figure 6.4: Average distance covered by champions on the difficult line following task.

Our results for this experiment are shown in Figure 6.4. Aside from HyperNEAT, we show performance of the wavelet baseline and HyperNEAT limited to a single hidden node. In an earlier experiment on the easy line following task,

we found that a single hidden node was needed to obtain performance similar to unrestricted HyperNEAT, which is why we show that variant here. Both regular HyperNEAT and the variant restricted to one hidden node perform equally poorly on this task, while the wavelet method attains a substantially higher speed. Failure in this task is hard to define, but it is clear that HyperNEAT's performance is greatly impacted by the increased difficulty.

6.3 Walking Gait

In the original walking gait task, many of the evolved networks produce a hopping behavior where all legs move together. This is efficient because each hop propels the robot a large distance. However, in real robots, actuators are typically much less powerful compared to the robot's weight. To create a more realistic simulation where hopping is *not* efficient, we increased the mass of the torso in the physics engine from 10 to 100 units. This decreases the relative force exerted by the joint motors, with two consequences. First, hopping is not as effective as the robot is unable to hop very far. Second, the robot is less stable, as it cannot always absorb the full impact of landing on a single leg, making balance more important. The regularity of the task remains unaltered.

Figure 6.5 shows our results in this task. As expected, the HyperNEAT robots cover substantially less distance than in the original task. Analyzing the resulting gaits reveals that HyperNEAT still generates hopping gaits with similar outputs for every joint angle controller. While this makes sense for the anteroposterior joints, the lateral joints *also* receive a target angle, moving them sideways during the hop. This leaves the robot unbalanced and, because the joint motors have limited power, it is more likely to fall over. The evolved gaits that do not fall over exhibit a careful—thus slow—hopping gait.

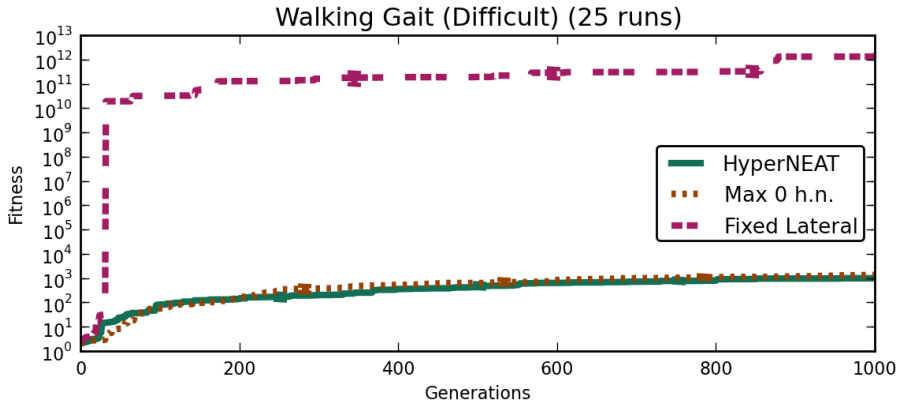


Figure 6.5: Average distance covered by champions on the difficult walking gait task.

Of course, since the heavier torso will affect even good walking gaits, it is important to determine whether better performance in this task variation is possible. Unfortunately, due to limitations in the software implementation created in (Clune et al., 2009a), we were unable to implement the baseline

wavelet method in this task. Instead, we constructed an alternative task-specific baseline as follows. HyperNEAT evolves the controller but, in order to generate more stable gaits, the lateral joints are fixed by setting the target angle to zero after the substrate has been queried for an output.

As shown in Figure 6.5, this baseline greatly outperforms regular HyperNEAT. In principle, regular HyperNEAT should be able to evolve CPPNs that generate exactly this behavior. In fact, doing so requires only removing the connections to all of the lateral weights, as shown in Figure 6.6. The failure of HyperNEAT to do so demonstrates that it does not account for this exception and that sideways motion of the lateral joints is an adverse effect of an overly simple solution. Because the exempted connections form a continuous region, it is likely that a *fractured* solution—as explained in Chapter 7—could benefit HyperNEAT's performance.

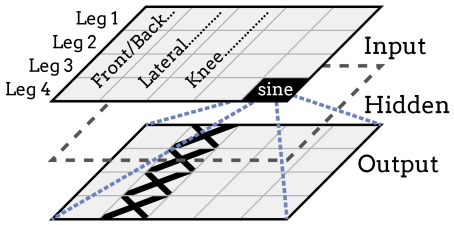


Figure 6.6: Better solutions to the difficult walking gait problem are obtained if lateral joint nodes are not connected, as indicated by the crosses. HyperNEAT could theoretically isolate the column with those nodes but fails to do so.

Risi and Stanley (2013) show that HyperNEAT can successfully generate controllers for simulated robots with a variable leg length. This experiment might be similar to ours because the longer leg lengths in that experiment might make the robot comparably top-heavy. However, it is hard to relate HyperNEAT's success to our experiment because they use a modular substrate configuration that utilizes continuous time activation. Additionally, we do not know whether their task is actually more difficult because they did not compare performance to a baseline.

6.4 Checkers

The regular version of the checkers task shown in Chapter 5 could be solved by simple CPPNs and in very few generations. Because it is not entirely clear what made the task easy in that case, it is not clear either how to raise the complexity. However, there is one very natural way to increase the difficulty of the *game* due to the fixed minimax search depth. We can increase the lookahead depth of the opponent or decrease that of the evolved player, both of which will lead to a more difficult task. Our first experiment decreases the lookahead of the evolved player to 2, from 4.

The results of this experiment are shown in Figure 6.7. The average fitness values shows quite some variance, in part due to the randomization of the opponent. Therefore we also report a more informative metric: the number

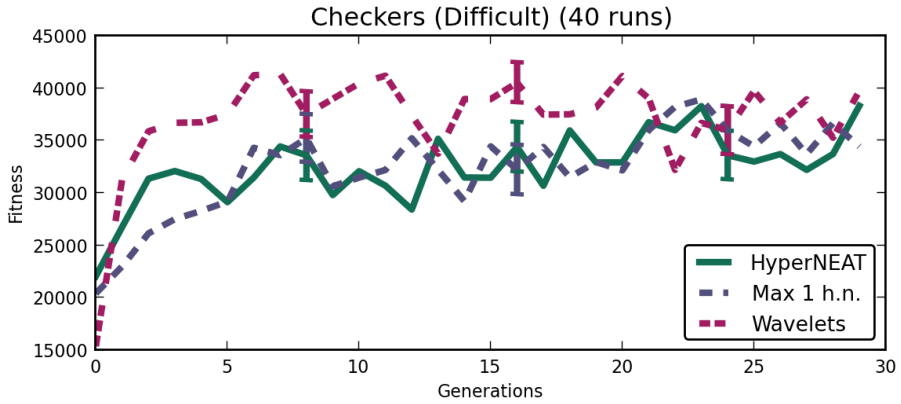


Figure 6.7: Average fitness of champions on the difficult checkers task.

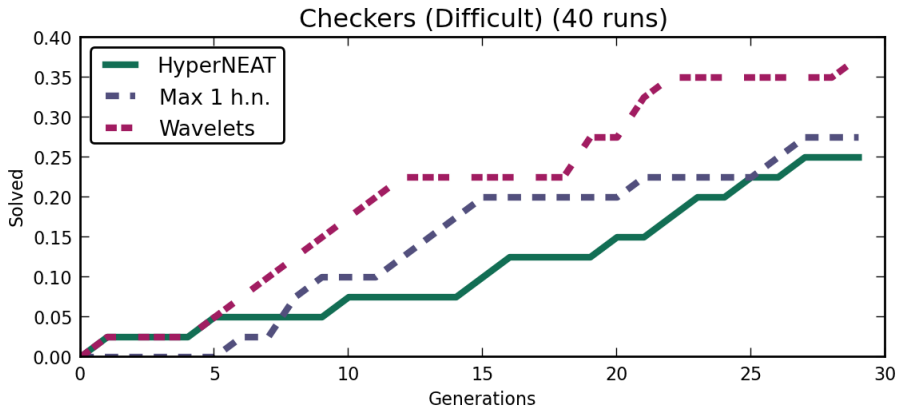


Figure 6.8: Portion of runs producing a *solution* to the difficult checkers task.

of runs that have *solved* the task (divided by the total number of runs). A genotype is said to have solved the task if it can beat the randomized opponent three times in a row, indicating a robust behavior. This is shown in Figure 6.8. There is some variation in the speed at which the different methods solve the task, but each of the methods still finds a solution. HyperNEAT solves this—presumably—more difficult task, but since the restricted variant finds a solution equally fast, there is no indication that the task is in fact more complex. Again, this experiment does not provide any evidence that HyperNEAT can effectively optimize topologies of more than one CPPN hidden node. Intuitively, when continuing to increase the difficulty, the restricted variant will no longer be able to create a solution, whereas unrestricted HyperNEAT hopefully still might. To see whether that is the case, we ran the same experiment with many different combinations of lookahead depths for both the evolved player and the opponent. The most extreme case here is a lookahead of 1 for the evolved player, and a lookahead of 5 for the opponent.

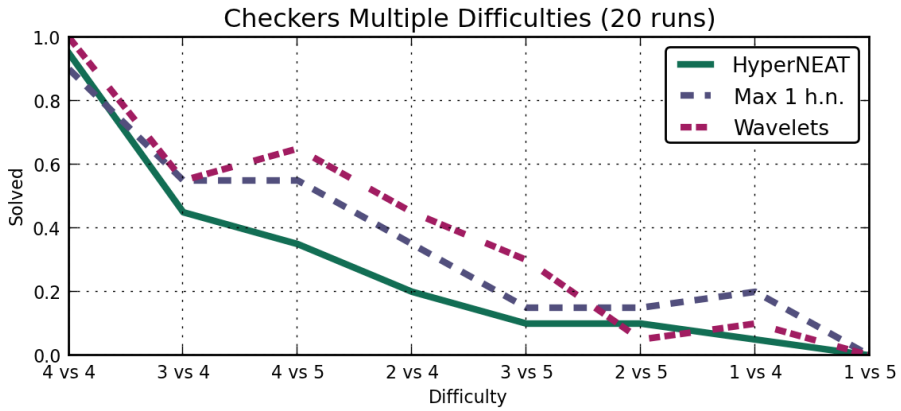


Figure 6.9: Portion of runs producing a *solution* to each of the increasingly more difficult checkers tasks. Difficulty is indicated as "lookahead of evolved player vs lookahead of opponent". The settings are sorted by HyperNEAT's performance on them.

The results of this last experiment are shown in Figure 6.9. We see a gradual decline in performance of all methods, which means that we have succeeded in scaling the difficult of this task. Amazingly, even when evolving a player with a lookahead of 1 against the opponent with a lookahead of 4, some of the runs of every method still come up with a solution. As an illustration, in an example game by one of the champions of a successful run of the wavelet method in this scenario, the evolved player used an average of 8 board evaluations per turn, whereas the opponent used 419. Beyond this difficulty level, none of the methods succeed at finding a solution. On the one hand, it is consistent with our earlier results that HyperNEAT does not outperform the restricted baseline. On the other hand, we do not know whether it is even possible to outperform this baseline, because none of the methods find a solution in the hardest scenario. So, unfortunately, this experiment gives us no evidence about the ability of either method to evolve complex networks.

This chapter has shown that HyperNEAT's performance is impacted when difficulty is increased beyond the level of the existing benchmark tasks. That effect is to be expected, but application of the wavelet baseline shows that it *is* possible to attain better performance in most problems. Additionally, in the experiments in this chapter, HyperNEAT does not effectively make use of hidden nodes in the CPPN to find more complex solutions.

Chapter 7

Fracture

In this chapter, we examine the hypothesis that *fracture* in the problem space, which is known to be challenging for the original NEAT method (Kohl and Miikkulainen, 2009), might even be more problematic for HyperNEAT. For regular NEAT, fractured problems are defined as those with "a highly discontinuous mapping between states and optimal actions" (Kohl and Miikkulainen, 2009). An example of a fractured space is shown in Figure 7.1. To quantify this notion, the authors express it in terms of *function variation* in the output of the network generated by NEAT, measured as the sum of the differences of adjacent values on a sampled version of the function. For example, for a one-dimensional function, if $X_{0,\ldots,n}$ is a set of n sample points on the domain of function f, the fracture is $\sum_{i=0}^{n-1} |f(X_i) - f(X_{i+1})|$. For multidimensional functions, this becomes rather more complicated, as is explained in detail in the original paper.

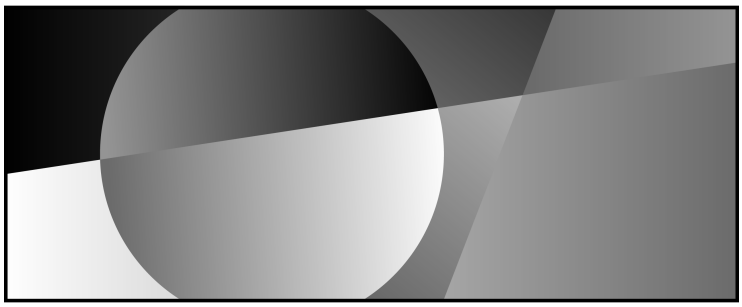


Figure 7.1: A highly fractured space as defined for NEAT. Even though there are regions with gradual change, there are many borders with discontinuous transitions.

This definition, however, is not appropriate for a CPPN because even the simplest CPPNs can include a sine node which, at the right frequency, would cause maximal function variation. To more accurately capture the essence of fracture as the degree to which regions of the generated function have to be treated differently, we propose an alternative definition of fracture for indirect encodings. Because nodes in HyperNEAT can actually emit repeated patterns,

we define fracture as a *discontinuous variation of patterns*. If we are free to construct the target function that the CPPN is to approximate, we can formalize this as the number of contiguous and convex *regions* in the target function with different patterns (see Figure 7.2). Because it is hard to distinguish between regions and patterns in an arbitrary problem, we use this only as a *generative* definition: delineating regions in problems for which we construct the target function.

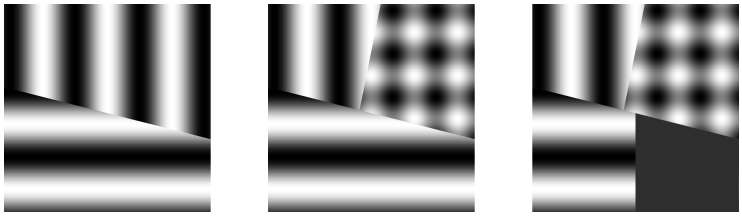


Figure 7.2: Increasingly fractured spaces as defined for HyperNEAT.

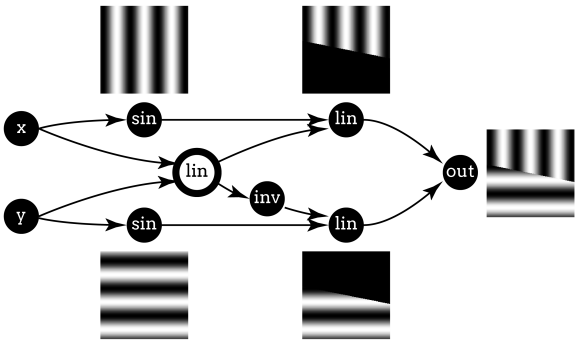


Figure 7.3: A topology for combining two patterns into a fractured substrate. The outlined node forms a linear combination of the coordinate vector that emits a 'mask' indicating whether the given (x, y) is in the target region. Regions are 'zeroed out' by subtracting this value from the pattern, since values below some threshold are truncated. Only then can the different patterns be summed.

The reason we hypothesize that fracture can be difficult for HyperNEAT is that many nodes are needed to divide the substrate in such regions. Though the CPPN can evolve nodes for generating different patterns, some mechanism is needed to select which pattern applies in which region. An 'indicator node' is needed that emits a signal when the given (x, y) is inside the region; this alone might require some complexity since HyperNEAT needs multiple nodes to compute a nonlinear combination of the coordinates. Even if there is an indicator node, an extra mechanism is needed to apply a pattern *only* to that region, because the CPPN lacks a multiplication operator that could zero out the pattern outside of the region. For example $\text{gauss}(x) \cdot \sin(x)$ would apply a sine only in the region that the Gaussian indicates, but it is exactly this

multiplication that a CPPN can not perform.

There is a workaround but it is somewhat complex. Figure 7.3 illustrates the workaround for a substrate divided along a single infinite line. It is possible that there are simpler CPPNs that output patterns exhibiting the characteristics of fracture. Therefore, the previous figure provides only an *upper* bound on the number of nodes needed to create a single arbitrary fractured region. Even more nodes are needed to isolate finite convex regions, and concave regions must be split up into multiple convex ones. Since CPPNs with only some of the nodes required for this workaround are likely to perform poorly, we hypothesize that it is difficult for HyperNEAT to evolve such workarounds incrementally. In contrast, positioning features is exactly what the wavelet method does well. Every wavelet by definition has a target location to which it is applied, in which it draws a (repeated) pattern.

7.1 Target Weights

To test our hypothesis about the effects of *fracture* on the performance of HyperNEAT, we use the target weights task (Clune et al., 2011). The goal of this task to generate a specific weight pattern in the substrate. It was originally used to show that performance of HyperNEAT degrades as *irregularity* increases. In that experiment, irregularity was increased by adding noise: an increasing percentage of randomly chosen weights are each assigned a different random value. This can be seen as an extreme form of fracture, since a unique value has to be assigned to every region consisting of a single connection weight, unless adjacent connections form contiguous (patterned) regions by chance. The substrate in the original experiment consists of 3^4 connections, so even at 10% noise, approximately eight new regions are introduced.

Therefore, if our hypothesis is correct, we should expect that HyperNEAT would perform poorly even at low noise levels. However, it is unclear how to determine what 'poor' performance is, since the original experiment did not compare to any baselines, but only compared across noise levels. To remedy this, we generate a linear least-squares solution for each problem instance, the coefficients of which are the node coordinates that are also input to the CPPN. Note that this is a very weak baseline, as a linear solution cannot cope with any of the irregularity or fracture. The data shown for the least-squares solution is the average of the performance of the least-squares solutions for all of the target patterns used for every method. Like the original experiment, we used a population of 500 individuals over 500 generations.

Figure 7.4 shows our results, which reproduce the original experiment but add the HyperNEAT variant restricted to zero hidden nodes, the wavelet baseline method, and the linear least-squares solution. To be consistent with the least-squares solution, we base the fitness function on the sum of squared errors between the generated substrate and the target weights. In addition, for brevity, we show only performance of the champion of the last generation of each run, grouped by noise level. These results show that HyperNEAT is never doing much better than a linear solution, even at the lowest noise level, and is therefore uniformly unable to account for the aberrant regions. The wavelet baseline method does substantially outperform the linear solution. Results from the original experiment (Clune et al., 2011) seem to indicate that performance

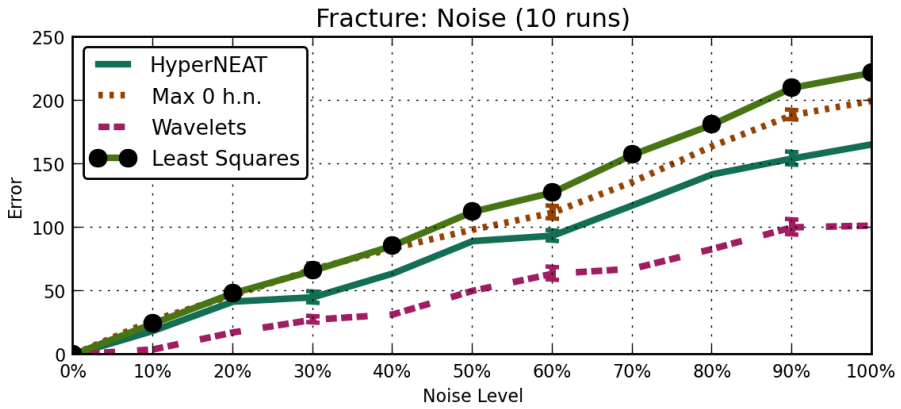


Figure 7.4: Average error of final generation champions in the target weights task with increasing noise.

gradually declines as irregularity increases. The increasing magnitude of the error in our experiment seems to confirm this trend, but comparison to the baselines sheds new light on the original: HyperNEAT has actually already failed at the lowest noise level, since even this level might have introduced more fracture than it can cope with.

7.2 Target Weights with Bisection

To determine what level of fracture HyperNEAT can cope with, we devised a second target weights experiment. Instead of assigning a random value to single weights, we divide the substrate into regions along straight lines, each time bisecting previous regions, as illustrated in Figure 7.5. Each of these regions is then assigned a single random weight. We decrease the dimensionality of the target matrix from 3^4 to 8^2, and continue splitting until each of the 8×8 connections consists of its own region. The result is that fracture increases gradually, enabling us to test fracture levels lower than that of the lowest noise level in the original experiment.

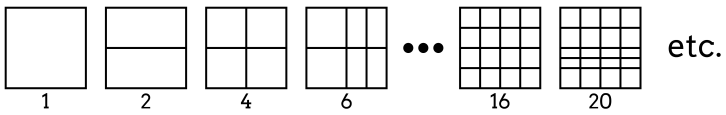


Figure 7.5: Iterative addition of fracture.

Figure 7.6 shows the results, again comparing to the linear least-squares baseline. These results show that, even at very low levels of fracture, HyperNEAT is unable to substantially improve on a linear solution. The wavelet baseline again performs significantly better. Together, these results confirm our hypothesis that fracture can be highly problematic for HyperNEAT.

A caveat of the experiments in this chapter is that they are target-based tasks, as mentioned in Chapter 3.3. They are set up in this way because our

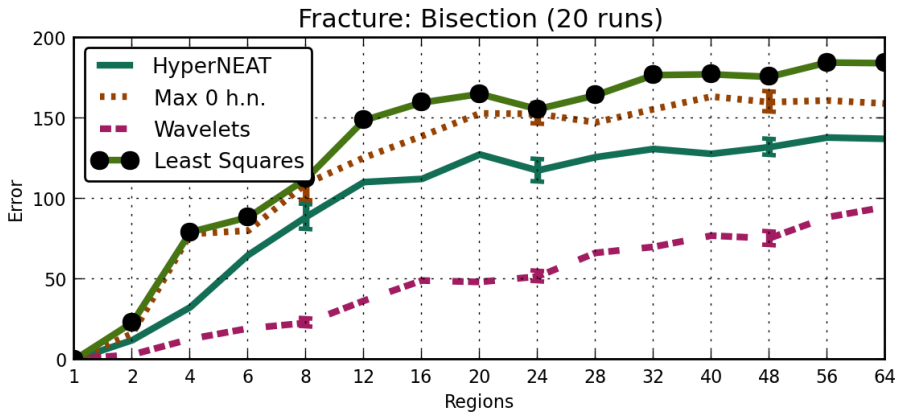


Figure 7.6: Average error of the final generation champions in the target weights task with increasing bisection.

definition of fracture is a generative one and because our experiments are a follow up to the experiment by Clune et al. (2011), which was also target-based. Woolley and Stanley (2011) report that HyperNEAT is not able to generate certain patterns under target-based conditions, and our results are in a sense consistent with theirs. However, our experiments include much simpler patterns and we were disappointed by how quickly HyperNEAT's performance decays compared to the baselines. The experiments in this section show that fracture can be a contributing factor to HyperNEAT's performance, and that it has a substantial negative effect when HyperNEAT is used on target-based tasks. We give a reinterpretation of the experiment done by Clune et al. (2011) and show that HyperNEAT's efficacy is impacted much earlier than their results might have suggested.

Chapter 8

Discussion & Future Work

The results presented in this paper show that the success of HyperNEAT on the tasks we considered does not provide much support for the hypothesis that "symmetry, imperfect symmetry, repetition, and repetition with variation...are compactly represented and therefore easily discovered" (Stanley et al., 2009).

On the one hand, on the simple versions of the considered tasks, HyperNEAT succeeds because the connectivity hypercube encoding represents simple regularity in such a way as to make the problem solvable by a trivial CPPN. The fact that HyperNEAT succeeded where it did should not be disregarded: the connectivity hypercube representation that it introduced has made these tasks much easier than they would be under a direct encoding.

On the other hand, on only slightly harder versions of these tasks, HyperNEAT does not enable the easy discovery of the more elaborate—but still regular—phenotypes these tasks require, as it is not able to evolve the corresponding genotypes. This result is somewhat surprising, since incrementally evolving complex solutions is supposed to be NEAT's specialty. However, it seems that doing so in HyperNEAT is more difficult than in regular NEAT, though more research is needed to determine with certainty why that is. When new nodes are introduced in regular NEAT, they initially do not influence the network output, because they are inserted into an existing connection. In contrast, in HyperNEAT, they are assigned a random function type, which can have an immediate disruptive effect on the output, thereby frustrating incremental evolution of topologies. However, more research is needed to reconcile this with the fact that the number of hidden nodes in HyperNEAT's CPPNs does increase as fitness does.

A recent reaction (Stanley et al., 2013) to the research in this thesis (published earlier in (van den Berg and Whiteson, 2013)) presents results claimed to contradict our findings. They show reimplemented experiments for the visual discrimination and line following task. In their reimplemented version of the difficult line following task, they show that HyperNEAT can successfully learn to follow the line. It is unclear how this would compare to our results, since the scaling of the robot's speed can vary significantly in different settings. In the visual discrimination domain, they show that optimizing parameters can be beneficial to HyperNEAT's performance. Additionally, they show that—given many more generations—HyperNEAT can learn to perform better on the difficult visual discrimination task, and outperform the restricted variant. They

show that some runs achieve perfect performance, and average performance is such that most runs are probably actually distinguishing between the target and the distractor shape. However, using more generations means doing more fitness evaluations—essentially making the task easier—so those results do not show that HyperNEAT has solved a more difficult task. The discrepancy between HyperNEAT and the restricted variant on the this task might seem to contradict our findings, but the restricted variant used in their experiments is actually more limited than ours (see Appendix C for an analysis of their 'canonical' variant). The report also points out that there are two examples of HyperNEAT generating effective walking behavior for different—possibly more adverse—body morphologies. One of these uses a modular substrate configuration that utilizes the continuous time activation (Risi and Stanley, 2013). In the other the CPPN is directly used to output target angles for the joints in a new setup called *single-unit pattern generator* (SUPG) (Morse et al., 2013); it is debatable whether this is still an indirect encoding. As such, though they are walking gait experiments, they are hard to compare to our setup. Additionally, we cannot establish whether these tasks are actually more difficult, because a restricted baseline has not been applied to either one.

There is some evidence that *novelty search* could overcome some limitations of HyperNEAT (Morse et al., 2013). However, applying novelty search requires some domain knowledge to design an effective *behavior function* (Kistemaker and Whiteson, 2011).

A caveat of our results in Chapter 7 is that, since each target weights task requires a specific phenotype to solve, they consider only tight fitness functions like those in (Woolley and Stanley, 2011). This is a necessary limitation due to the generative nature of our definition of fracture. There is some evidence that HyperNEAT can cope with modest fracture (two regions) given looser fitness functions (Coleman, 2010; Stanley et al., 2013). However, in our experiments, HyperNEAT could not solve the harder walking gait task, which also has a looser fitness function, even though it requires generating a pattern with only three regions (see Figure 6.6). More research is needed into the interplay between the tightness of the fitness function and HyperNEAT's ability to cope with fracture, though doing so is difficult since these parameters cannot be directly controlled.

Of course, a key limitation of all our results is that they consider only some of the tasks on which HyperNEAT has succeeded. Thus, another important avenue for future work is to repeat this sort of analysis on these other tasks. While we can only speculate about HyperNEAT's behavior on such tasks, there are hints to suggest it is in at least some cases consistent with what we report here. Another task for which such an analysis could be done is keepaway (Verbancsics and Stanley, 2010), for which performance is competitive with other leading methods but the comparison is confounded by the use of a novel "bird's eye view" representation. It may be that this representation obviates the need to discover a nontrivial CPPN. Other HyperNEAT successes may be more substantial, e.g., it solved a harder visual discrimination task somewhat similar to ours in Chapter 6 (Coleman, 2010), evolved behaviors for teams of robots (D'Ambrosio et al., 2010, 2011), and discovered more intricate walking gaits for real robots (Yosinski et al., 2011). However, whether these tasks actually require complex solutions is not known, because no restricted baseline was applied.

To better cope with fracture, it might be beneficial to introduce nodes with a radial basis function, creating "RBF-HyperNEAT", corresponding to "RBF-

NEAT" (Kohl and Miikkulainen, 2009), which was shown to handle fracture better than regular NEAT. From there it is a small step to implement nodes emitting *wavelets*, which —given the results in this paper— could be an effective extension. Alternatively, the wavelet baseline method considered here itself showed promising initial performance and could be used as a starting point for a more robust indirect encoding method.

Most of all, though, this thesis shows that it is necessary to verify that benchmark tasks actually require complex solutions by applying an appropriate baseline. A continued analysis of existing benchmarks, and inclusion of such an analysis in future research will help to establish the actual power of examined GDS methods.

Chapter 9

Conclusion

This thesis did an empirical analysis of HyperNEAT's performance, positing fracture as a possible critical factor. We examined the difficulty of a number of tasks on which HyperNEAT has succeeded and found that all of these tasks are easy: they can be solved with at most one hidden node and require generating only trivial regular patterns. A key lesson from this research is that we need to verify whether a task requires a complex solution before we use it as evidence that a method is capable of solving complex problems. We examined how HyperNEAT performs when these tasks are made harder and found that HyperNEAT's performance decays quickly: it fails to solve all variants of these tasks that require more complex—but still regular—solutions. Finally, we examined the role of problem space fracture and found that, in our experiments, HyperNEAT is unable to cope with even modest levels of fracture. Together, these results suggest that the capacity of HyperNEAT to capture complex regularities may be less than was previously supposed and hence new methods may be needed to finally fulfill the promise of indirect encodings.

Bibliography

Astor, J. and Adami, C. (2000). A developmental model for the evolution of artificial neural networks. *Artificial Life*, 6(3):189–218.

Braitenberg, V. (1986). *Vehicles: Experiments in synthetic psychology.* MIT press.

Braun, H. and Weisbrod, J. (1993). Evolving neural feedforward networks. In *Proceedings of the Conference on Artificial Neural Nets and Genetic Algorithms*, pages 25–32.

Buk, Z., Koutník, J., and Šnorek, M. (2009). NEAT in HyperNEAT substituted with genetic programming. *Adaptive and Natural Computing Algorithms*, pages 243–252.

Clune, J., Beckmann, B., McKinley, P., and Ofria, C. (2010). Investigating whether HyperNEAT produces modular neural networks. In *GECCO*, pages 635–642.

Clune, J., Beckmann, B., Ofria, C., and Pennock, R. (2009a). Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Congress on Evolutionary Computation*, pages 2764–2771.

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

Clune, J., Ofria, C., and Pennock, R. (2009b). The sensitivity of HyperNEAT to different geometric representations of a problem. In *Conference on Genetic and Evolutionary Computation*, pages 675–682. ACM.

Clune, J., Stanley, K., Pennock, R., and Ofria, C. (2011). On the performance of indirect encoding across the continuum of regularity. *Transactions on Evolutionary Computation*, 15(3):346–367.

Coleman, O. (2010). Evolving neural networks for visual processing. *B.S. Thesis.*

D'Ambrosio, D. and Stanley, K. (2008). Generative encoding for multiagent learning. In *GECCO*, pages 819–826.

D'Ambrosio, D. B., Lehman, J., Risi, S., and Stanley, K. (2010). Evolving policy geometry for scalable multiagent learning. In *AAMAS*, volume 1, pages 731–738.

D'Ambrosio, D. B., Lehman, J., Risi, S., and Stanley, K. O. (2011). Task switching in multirobot learning through indirect encoding. In *IROS*, pages 2802–2809.

Daugman, J. et al. (1985). Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *Journal of the Optical Society of America A*, 2:1160–1169.

Drchal, J., Koutník, J., and Snorek, M. (2009). HyperNEAT controlled robots learn how to drive on roads in simulated environment. In *CEC*, pages 1087–1092.

Fierz, M. (2002). Simple checkers. http://arton.cunst.net/xcheckers/.

Fullmer, B. and Miikkulainen, R. (1992). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In *Proceedings of the First European Conference on Artificial Life*, pages 255–262.

Gauci, J. and Stanley, K. (2008). A case study on the critical role of geometric regularity in machine learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, pages 628–633.

Gauci, J. and Stanley, K. O. (2010). Autonomous evolution of topographic regularities in artificial neural networks. *Neural computation*, 22(7):1860–1898.

Goldberg, D. (1989). Genetic algorithms in optimization, search and machine learning. *Addison Wesley*, 905:205–211.

Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Urbana*, 51:61801–2996.

Gomez, F., Koutník, J., and Schmidhuber, J. (2012). Compressed network complexity search. In *Parallel Problem Solving from Nature-PPSN XII*, pages 316–326. Springer.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965.

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

Gruau, F., Whitley, D., and Pyeatt, L. (1996). 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.

Hansen, N., Müller, S. D., and Koumoutsakos, P. (2003). Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation*, 11(1):1–18.

Heidrich-Meisner, V. and Igel, C. (2009). Hoeffding and Bernstein races for selecting policies in evolutionary direct policy search. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 401–408. ACM.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

Hornby, G., Pollack, J., et al. (2001). Body-brain co-evolution using L-systems as a generative encoding. In *GECCO*, pages 868–875.

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

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In *CEC*, volume 4, pages 2588–2595. IEEE.

Kistemaker, S. and Whiteson, S. (2011). Critical factors in the performance of novelty search. In *GECCO*, pages 965–972.

Kohl, N. and Miikkulainen, R. (2009). Evolving neural networks for strategic decision-making problems. *Neural Networks*, 22:326–337.

Koutník, J., Gomez, F., and Schmidhuber, J. (2010). Searching for minimal neural networks in fourier space. In *Proc. of the 4th Conf. on Artificial General Intelligence*. Citeseer.

Larrañaga, P. and Lozano, J. A. (2002). *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer.

Lee, T. (1996). Image representation using 2D Gabor wavelets. *Transactions on Pattern Analysis and Machine Intelligence*, 18(10):959–971.

Lembcke, S. (2011). Chipmunk 2D physics engine. https://github.com/slembcke/Chipmunk2D.

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

Mahfoud, S. (1995). Niching methods for genetic algorithms. *Urbana*, 51(95001).

Moriarty, D. E., Schultz, A. C., and Grefenstette, J. J. (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276.

Morse, G., Risi, S., Snyder, C. R., and Stanley, K. O. (2013). Single-unit pattern generators for quadruped locomotion. In *GECCO*.

Risi, S., Lehman, J., and Stanley, K. (2010). Evolving the placement and density of neurons in the HyperNEAT substrate. In *GECCO*, pages 563–570. ACM.

Risi, S. and Stanley, K. (2011). Enhancing ES-HyperNEAT to evolve more complex regular neural networks. In *GECCO*, pages 1539–1546. ACM.

Risi, S. and Stanley, K. O. (2013). Confronting the challenge of learning a flexible neural controller for a diversity of morphologies. In *GECCO*. ACM.

Rubinstein, R. Y. and Kroese, D. P. (2004). *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer.

Secretan, J., Beato, N., D'Ambrosio, D., Rodriguez, A., Campbell, A., Folsom-Kovarik, J., and Stanley, K. (2011). Picbreeder: A case study in collaborative evolutionary exploration of design space. *Evolutionary Computation*, 19(3):373–403.

Stanley, K. (2006). Exploiting regularity without development. In *Proceedings of the AAAI Fall Symposium on Developmental Systems*.

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

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

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

Stanley, K. O., Clune, J., D'Ambrosio, D. B., Green, C. D., Lehman, J., Morse, G., Pugh, J. K., Risi, S., and Szerlip, P. (2013). CPPNs effectively encode fracture: A response to critical factors in the performance of HyperNEAT.

Suchorzewski, M. and Clune, J. (2011). A novel generative encoding for evolving modular, regular and scalable networks. In *GECCO*, pages 1523–1530.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. The MIT Press.

van den Berg, T. (2013). Python evolutionary algorithms (PEAS). https://github.com/noio/peas/tags.

van den Berg, T. and Whiteson, S. (2013). Critical factors in the performance of HyperNEAT. In *GECCO 2013: Proceedings of the Genetic and Evolutionary Computation Conference*.

Verbancsics, P. and Stanley, K. (2010). Evolving static representations for task transfer. *The Journal of Machine Learning Research*, 11:1737–1769.

Verbancsics, P. and Stanley, K. (2011). Constraining connectivity to encourage modularity in HyperNEAT. In *GECCO*.

Whiteson, S. (2012). Evolutionary computation for reinforcement learning. In Wiering, M. and van Otterlo, M., editors, *Reinforcement Learning: State of the Art*, pages 325–358. Springer, Berlin, Germany.

Whiteson, S., Taylor, M., and Stone, P. (2010). Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 21(1):1–35.

Whitley, D., Dominic, S., Das, R., and Anderson, C. W. (1994). *Genetic reinforcement learning for neurocontrol problems*. Springer.

Woolley, B. and Stanley, K. (2011). On the deleterious effects of a priori objectives on evolution and representation. In *GECCO*, pages 957–964.

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

Yosinski, J., Clune, J., Hidalgo, D., Nguyen, S., Zagal, J., and Lipson, H. (2011). Evolving robot gaits in hardware: the HyperNEAT generative encoding vs. parameter optimization. In *ECAL*, volume 8, page 12.

Appendix A

Line Following Task Settings

During an extended analysis of the line following task, we found that the performance of different methods is highly sensitive to the settings of this task. In addition, the evolved *behavior* varies strongly in different setups. This appendix gives an overview of that analysis.

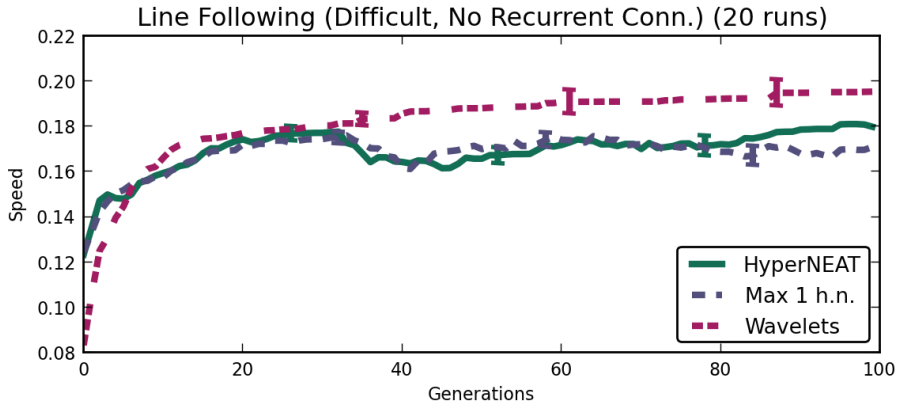


Figure A.1: Average distance covered by champions on the difficult line following task from (van den Berg and Whiteson, 2013), with recurrent connections disabled.

The results for the original difficult line following task—first presented in (van den Berg and Whiteson, 2013)—are shown in Figure A.1. In this experiment recurrent connections in the top layer of the substrate are disabled by flushing activation on each timestep. This makes the task exceedingly difficult: the right activation has to be accumulated directly into the motor nodes. Both regular HyperNEAT and the variant restricted to one hidden node perform equally poorly on this task, while the wavelet method attains a moderately higher speed.

Disabling recurrent connections is not a very common setup, so we ran a

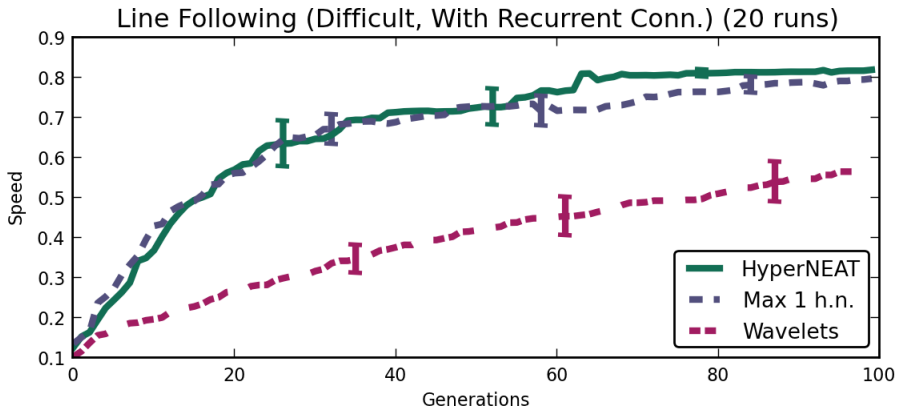


Figure A.2: Average distance covered by champions on the difficult line following task when recurrent connections are enabled.

new set of experiments with these connections enabled. The results of this are shown in Figure A.2. Here we see that the HyperNEAT variants are at a clear advantage, with the wavelet method only slowly catching up. We looked at the behavior of the evolved solutions to find a reason for this inversion, and we found that most of them exhibit an oscillating motion across the center of the figure-eight road (Figure A.3).

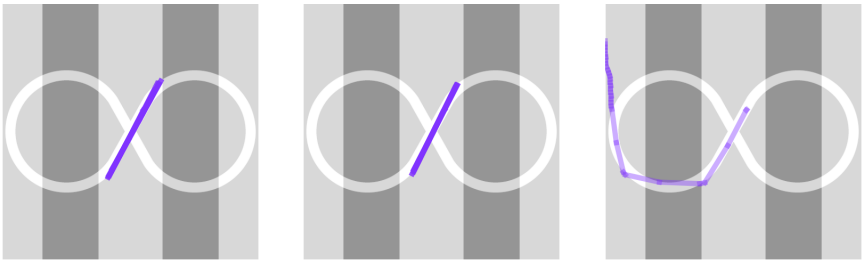


Figure A.3: Example behavior in the line following task with recurrent connections enabled. The left two images show behaviors similar to virtually every solution by each method. The right image shows a different behavior evolved by a few of the wavelet runs.

This behavior does not give us an explanation for why the relative performance is suddenly different from our other experiments, but it does show us that this task does not live up to its name: true line following behavior is not necessarily rewarded, and the methods exploit that fact. We implemented two features in order to discourage the oscillating behavior. The first consists of decreasing the torque of the robots' motors: this makes reversing much harder to do, and makes coasting relatively more attractive. The second consists of modifying the fitness function to reward fitness only when the robot is in a region that it has not visited recently. The field is divided into 32×32 square cells, and the robot is rewarded only when it stays in the same cell, or when it

visits a new cell that is not part of the 20 cells it visited last. The amount of fitness score rewarded is not changed: it is equal to the distance covered. We also picked out another unintended source of difficulty: the force to the wheels was applied from a global coordinate system, not relative to the robot. This yields the final experiment that we used for Chapter 6 (shown in Figure 6.4), where the wavelet method attains a substantially higher speed. Presented in Figure A.4 are some example behaviors that were evolved in this experiment, confirming that the robots actually learn to follow the line. Some evolutionary epochs find behaviors where the robot only follows one of the loops of the figure-eight, but this is not undesired as long as they do so efficiently.

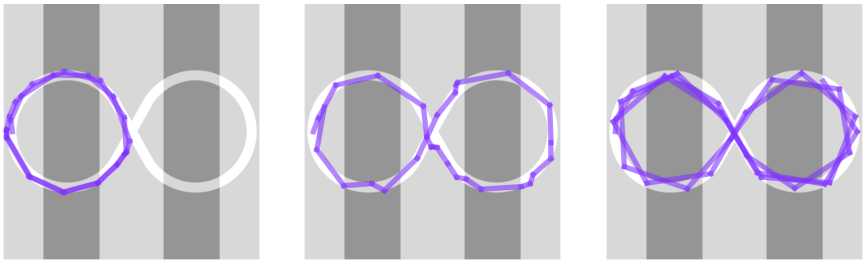


Figure A.4: Example behavior in the line following task where covering new ground is rewarded. The images show respective behaviors for unrestricted HyperNEAT, the variant limited to a single hidden node, and the wavelet method.

At this point, we have introduced quite a few variable settings for this task. In Chapter 6 we showed the one that most embodies the spirit of the task: learning to follow a line with the added difficulty of an altered observation. However, in order to convince the reader that we have not cherry picked this setup to support our hypotheses, we show the final performance for every combination of these settings in Figure A.5. The figure contains some interesting features. First, we see that in most setups the wavelet method's performance is better than or equal to HyperNEAT's. In most setups where performance is equal, it is bad for all methods, indicating that these might be nigh impossible tasks. For example, the combination of applying the wheel force globally (F_A) and rewarding newly covered ground yields dramatic performance (2, 6, 10, and 14). This is likely because the former makes oscillating much more attractive, whereas the latter tries to prevent it. Second, quite notably, there are two setups (5 and 13) where the HyperNEAT variants outperform the wavelet method. One of these setups (13) is the previous experiment shown in Figures A.2 & A.3. They share the combination of recurrent connections, global wheel force, and rewarding total travel distance. These factors might be beneficial to oscillating behavior, which HyperNEAT might learn quicker. Finally, there is one setting (11) where HyperNEAT somewhat outperforms the restricted variant. This is an interesting case: we have seen that it is possible to evolve a restricted CPPN that runs the whole figure eight when rewarding ground covered. The restricted variant also achieves this in *most* of the runs of this variant, but not as often as HyperNEAT does. It might be that rewarding ground covering creates better stepping stones in the fitness function, and that the lack of these somehow impacts the restricted variant more than HyperNEAT.

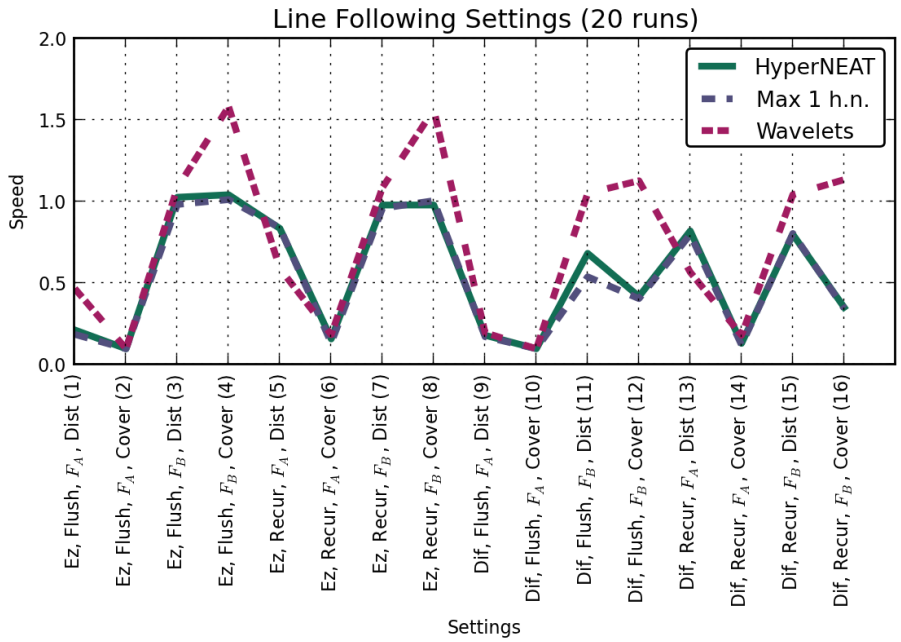


Figure A.5: An overview of the final performance of the methods on every combination of settings for the line following task. The axis is labeled as follows: whether the task is easy (Ez) or difficult (Dif), whether recurrent connections are enabled (Recur) or activation is flushed (Flush), whether the wheel force is applied from a local coordinate frame (F_A) or not (F_B), and finally whether fitness corresponds to total distance traveled (Dist) or robots are encouraged to cover new ground (Cover).

Appendix B

HyperNEAT-LEO

Standard HyperNEAT uses the CPPN to generate connection weights for each pair of nodes in the substrate. Ordinarily, when the output of the CPPN is below a certain threshold, connections are omitted. However, research has shown that HyperNEAT has trouble generating sparse connection patterns (Clune et al., 2010). HyperNEAT-LEO (Link Expression Output) was designed to overcome this limitation (Verbancsics and Stanley, 2011). The key change is that an additional output node in the CPPN is designated to control whether or not a connection is created at all. This *link expression output* can share some structure with the weight expression output or it can utilize an entirely separate part of the CPPN, leaving it up to evolution to determine how much coupling there is between the weights and the connectivity pattern.

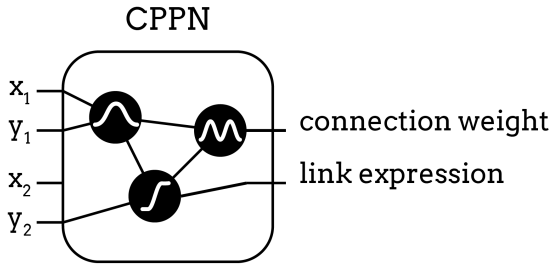


Figure B.1: An extra output is added to the CPPN that can determine whether a connection is added. The added connection is then given a weight according to the original weight output. As the illustration shows, the link expression output can share any amount of structure with the regular output.

We included HyperNEAT-LEO in two of the experiments from Chapter 6: visual discrimination and line following. We did not seed the CPPN with a specific topology for the link expression output as in (Verbancsics and Stanley, 2011). Instead, the CPPN starts with no hidden nodes and the link expression output is only connected to the input nodes, just like the regular weight output. Results of this experiment are shown in Figures B.2 & B.3. In both experiments, using HyperNEAT-LEO gives some improvement over regular HyperNEAT. A

possible explanation could be that HyperNEAT-LEO helps deal with low degrees of fracture. The link expression output can directly indicate a region for which connections are expressed. HyperNEAT-LEO might more easily model patterns that have one disconnected region and one (patterned) connected region. For example, it is possible that HyperNEAT-LEO can find a simple representation that disconnects the lateral joints in the walking gait problem (Figure 6.6); unfortunately, we couldn't implement this extension in that task domain.

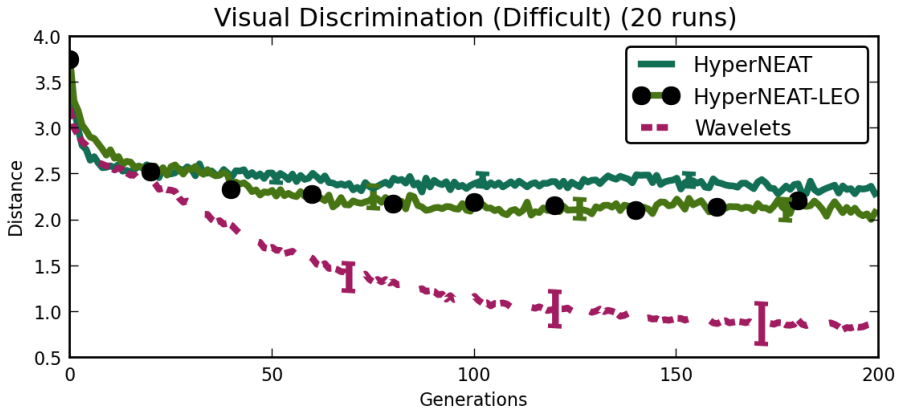


Figure B.2: Average distance from the target by champions on the difficult visual discrimination task.

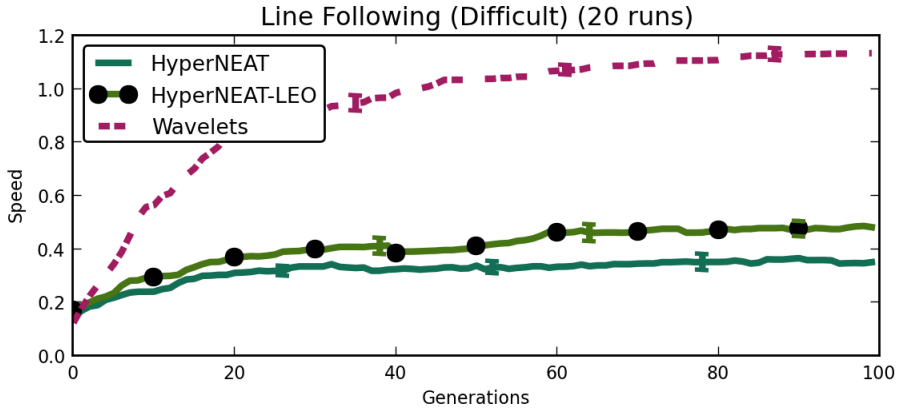


Figure B.3: Average distance covered by champions on the difficult line following task.

Appendix C

Canonical HyperNEAT

There are some minor implementation differences between our version of HyperNEAT and the 'canonical' version, as pointed out to us in an email exchange by Ken Stanley. We did an experiment to verify that these differences do not upset our conclusions. This experiment sheds some additional light on the complexity required to solve the visual discrimination task. First, we list the differences between the version of HyperNEAT used in our experiments and the canonical version.

- In our implementation, on mutation, there is a 20% chance for a node to change its activation function type. The activation functions in canonical HyperNEAT are fixed after the node is created and given a random type.
- The *output* node in our implementation is initially set to any of the special functions allowed by HyperNEAT, not just a (signed) sigmoid function. It is also subject to the type mutation described above.
- In our implementation, incoming activation of every node is multiplied by 4.9. This is an attribute of each node and it can be mutated by the genetic algorithm. This value is used to scale sigmoid functions in many implementations of NEAT. Canonical HyperNEAT lacks this multiplier.
- The sigmoid activation function in our implementation is the unsigned logistic function, instead of the signed hyperbolic tangent function.
- The set of activation functions in our implementation includes an unbounded linear function, whereas canonical HyperNEAT does not.
- Our implementation uses fitness boosts for young genomes and fitness penalties for old genomes, the same way NEAT does. It was pointed out to us that this mechanism is no longer used in recent versions of HyperNEAT.

In PEAS (van den Berg, 2013), these differences are all user settings, and the defaults are currently set to correspond to canonical HyperNEAT.

We applied a canonical version of each of the HyperNEAT variants to the visual discrimination task, the results are shown in Figure C.1. Performance of the restricted variants is heavily impacted by these changes, whereas unrestricted HyperNEAT shows slight improvement. Two of the differences outlined

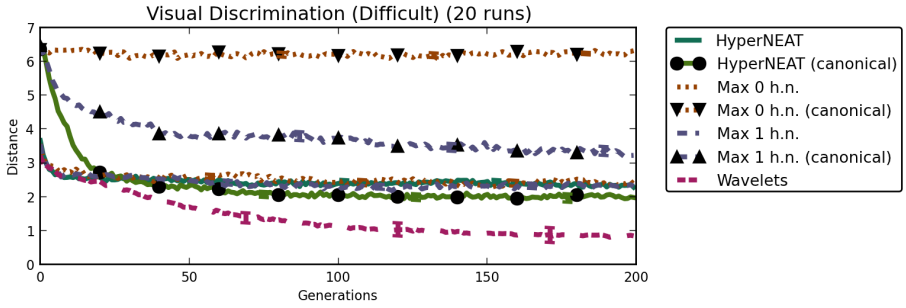


Figure C.1: Performance of the canonical version of the restricted HyperNEAT variants. The wavelet method is shown as a reference.

above are enough to form a hypothetical explanation for these discrepancies: the fact that we allow the *output* node to have a special activation type and that we subject activation types to mutation.

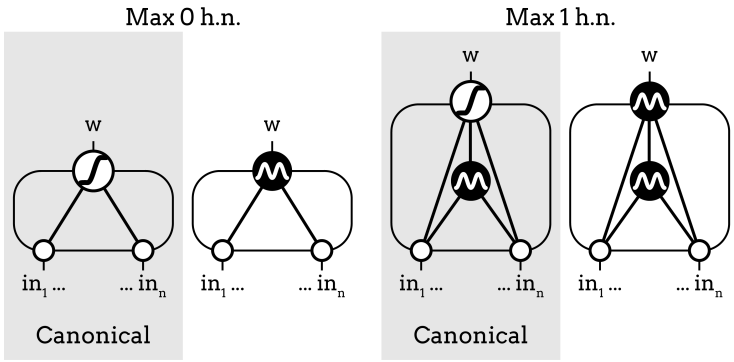


Figure C.2: Illustration of the difference in complexity of genotypes in the canonical and in our version of the restricted HyperNEAT variants. A black node indicates that a node can be assigned any activation type. The white outlined nodes indicate the type is always a sigmoid.

The fact that the canonical variant without hidden nodes does not improve at all is unsurprising. It does not have access to *any* activation function besides a sigmoid, so it cannot even compute the solution for the easy task which utilizes a Gaussian (Chapter 5.1). Canonical M1HN still shows progress at the end of the epoch; it is likely that it will catch up to the performance of our variants of M0HN/M1HN in the end, since we know that M0HN attains this performance with a strictly simpler CPPN (Figure C.2). The reason for slower convergence of canonical M1HN might be the fact that in canonical HyperNEAT function types are fixed as soon as a node is randomly created. With room for only one hidden node some individual lineages will be 'doomed' as soon as they add a hidden node with the wrong type, because there is no way to remove this node or add a new node with the correct type. Small genotypes have so few parameters that

it might be beneficial if all of them—including activation function type—remain subject to mutation, whereas mutating the activation type in larger genotypes might have a larger disruptive effect. Finally, there is the phenomenon that performance of every canonical method is worse at the start of the epoch. An explanation for this might be that the canonical variants all start with a fixed activation type output node (c.q. a sigmoid), they have to add a hidden node before they can utilize the expressiveness of any of the special types.

Overall, though these results are interesting in and of themselves, they do not contradict the results in the rest of this thesis. They still provide no strong evidence that HyperNEAT can benefit from evolving complexity beyond a single hidden node.