Honda Research Institute EU
Honda Research Institute Europe GmbH
https://www.honda-ri.de/

NEATfields: Evolution of neural fields for visual discrimination and multiple pole balancing tasks

Benjamin Inden, Yaochu Jin, Robert Haschke, Helge Ritter

2010

Preprint:

This is an accepted article published in Genetic and Evolutionary Computation Conference. The final authenticated version is available online at: https://doi.org/[DOI not available]

NEATfields: Evolution of neural fields for visual discrimination and multiple pole balancing tasks

ABSTRACT

We have developed a novel extension of the NEAT neuroevolution method, termed NEATfields, to solve problems with large input and output spaces. NEATfields networks are layered into two-dimensional fields of identical or similar subnetworks with an arbitrary topology. The subnetworks are evolved with genetic operations similar to those used in the NEAT neuroevolution method. We show that information processing within the neural fields can be organized by providing suitable building blocks to evolution. NEATfields can solve a number of visual discrimination tasks and a newly introduced multiple pole balancing task.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Learning—*Connectionism and neural nets*

General Terms

Algorithms

Keywords

neuroevolution,NEAT

Track: Generative and developmental systems

1. INTRODUCTION

1.1 Evolving artificial neural networks

Artificial neural networks are computational models of animal nervous systems and have found a wide range of successful applications, such as system control and image processing. Due to their nonlinear nature it is often difficult to manually design neural networks for a specific task. To this end, evolutionary algorithms have been widely used for automatic design of neural networks [20, 3]. An important advantage of designing neural networks with evolutionary algorithms is that both weights and topology of the neural networks can be optimized. However, a few challenges have to be addressed. For example, it is desirable that the existing function of a network should not be fully disrupted when adding new elements to the network. It is also not obvious how to recombine neural networks with arbitrary topologies, or genomes of neural networks with variable lengths.

Another grand challenge in evolving neural networks is the scalability issue: the evolution of solutions for tasks of a large dimension. This problem is particularly serious when a direct encoding scheme is used for representing the neural network, where the length of the genome grows linearly with the number of connections.

In contrast, indirect encoding of neural networks [20], in which the weights and topologies are generated using, for example, grammatical rewriting rules or grammar trees, can achieve a sublinear growth of the genome. These methods use a domain specific decompression algorithm in order to make a large phenotype from a small genotype. Typically, the class of encodable phenotypes is biased towards phenotypes that possess some kind of regularity [10], some identical or similar repeated structure. Indeed many neural networks, whether occurring in nature or in technical applications, possess repeated elements. For example, the cerebral cortex is organized into columns of similar structure. Brain areas concerned with visual processing contain many modules, in which similar processing of local features is done for different regions of the field of view in parallel. This occurs in brain locations whose spatial arrangement preserves the topology of the input [1].

Two ways of generating repeated structure can be considered: The first approach is to duplicate existing genes, which is believed to be an important mechanism in natural evolution[21]. The duplicated genes can subsequently diversify in their function. The second approach is known as co-option, where existing genes can be employed in new contexts [7]. While the first process generates modular or even redundant structures, the second can achieve decompression of genotypes into larger phenotypes. Both processes seem to be useful for the evolution of neural network topologies.

Recently, a lot of research has been performed on applying artificial embryogeny to neuroevolution [17, 8]. These methods are mainly inspired from biological mechanisms in morphological and neural development such as cell growth, cell division, and cell migration under the control of genetic regulatory networks. Here we take a different approach. We start with a direct encoding and augment it with duplication and co-option mechanisms in order to generate large phenotypes with repeated structures from small genomes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO 2010 Portland, Oregon

Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1.2 The NEAT neuroevolution method and derivatives

The NEAT method [16, 14] is a well known and competitive direct encoding method that introduces a number of techniques to successfully deal with common problems in neuroevolution. One idea is to track genes using historical markings. This is achieved by assigning a globally unique reference number to each gene once it is generated by mutation. These numbers are used to solve the permutation problem and make recombination effective. Similar to recombination in nature, genomes are aligned first, and their offspring get exactly one copy of each gene that is present in both parents. Another idea is to protect innovation through speciation. For example, if a network with a larger topology arises by mutation, initially it may not be able to compete against networks with a suboptimal but time-tested topology. By using reference numbers, the population is partitioned into species based on their phenotype similarity. The number of offspring a species has is proportional to its mean fitness. This prevents a slightly superior species from taking over the whole population, and enables innovative yet currently inferior solutions to survive. Recombination is usually only allowed to occur within a species, such that parents look rather similar to each other and the offspring look similar to its parents. Evolution starts with the simplest possible network topology in NEAT and proceeds by complexification, that is by adding neurons and connections. Neurons are only added between two connected neurons in such a way that the function of these connections changes as little as possible.

Due to the success of the method, quite a few derivatives have been developed. For example in [13], NEAT networks are used as modules and co-evolved with blueprints. Blueprints are lists of modules together with specifications on how to map module inputs and outputs on network inputs and outputs. In the MBEANN method [11], initially everything is in a module m_0. A new module is created every time a node is added that connects to at least one element in m_0. New connections are established either within a given module or between a given module and m_0. In another approach, sets of rules are evolved with NEAT-like speciation that implicitly define a neural network [12]. In HyperNEAT [2, 5], very large neural networks can be generated by using indirect encoding. Their topology is determined by the task geometry, while the connection weights are generated by giving neuron coordinates as input to another network, which is termed "compositional pattern producing network" [15] and evolved according to a slightly extended NEAT method. HyperNEAT networks can be very large and have shown impressive scaling ability. In the NEON method [9], NEAT mutation operators are used as developmental operators, and a gene can encode arbitrary numbers of operations by referring to a data pool.

1.3 NEATfields: Goals and approach

While there are already other methods for evolving large networks, NEATfields has been designed to make use of two particular features of the NEAT method: First, the genetic operators for changing network structure in NEAT were carefully designed to avoid producing redundant structures and disrupting existing functional elements. We believe that these operators are also helpful in evolving large neural networks. Second, complexification from small structures, i.e. gradual growth of the networks during the course of evolution, has been shown to be an important reason for the success of the method [16]. Therefore, exploration of the search space by gradual complexification is used as a strategy by NEATfields as well.

In order to evolve large networks with mutation operators known from direct encodings, the assumption that the input and output spaces of a task can largely be decomposed into a number of equal or similar subspaces is built into NEATfields. Many real-world tasks indeed require one or two dimensional fields of networks that do the same or similar calculations. For example, an eye or a camera provides large amounts of sensory data with a natural two-dimensional topology. Also, robots with actuated limbs often require a number of similar controllers in addition to a coordinating mechanism.

NEATfields uses small NEAT networks as elements of fields with arbitrary sizes. In contrast to some of the previously mentioned methods derived from NEAT, here the whole network architecture is specified by each individual, so there is only a single population by default. The encoding of the networks is strongly biased towards two dimensional fields, although any other network topology could evolve in principle. NEATfields starts evolution with a single field of size 1×1. If the mutation operators that are specific to NEATfields are switched off, it will reduce to a plain NEAT implementation. That means we can use it as a rather general methods to evolve solutions for tasks that are hard to solve because of being nonlinear control problems that require fast reaction, or for tasks that require internal memory. Double pole balancing and sequence recall are examples for such tasks and have been solved using NEAT implementations [16, 9]. We can also expect that these abilities of NEAT transfer to cases where many inputs and outputs are present. This will be demonstrated below.

2. METHODS

2.1 Neural networks

Like in most artificial neural networks, the activation of the neurons in NEATfields is a weighted sum of the outputs of neurons $j \in J$ to which they are connected, and a sigmoid function is applied on the activation: $o_i(t) = \tanh(\sum_{j \in J} w_{ij} o_j(t-1))$. Here, connection weights are within the range $[-3, 3]$.

A NEATfields network module (also called field element) is a recurrent neural network with almost arbitrary topology, although the used operators will ensure that no disconnected neurons exists and that there exist only one connection at most between all pairs of neurons. A field is a two-dimensional array of field elements. In special cases, the field size along one or both dimensions is 1. A complete NEATfields network consists of at least one internal field, and fields for network input and output as specified by the given task. There can be several input and output fields with different dimensions. Typically, a bias input is provided within its own input field of size 1×1. Within the NEATfields network, connections can be local (within a field element), lateral (between field elements of the same field), or global (between two fields, possibly including the input and outputs fields). For all experiments reported here, evolution starts with a single internal field of size 1×1 that is connected to all input and output fields (but in principle, it

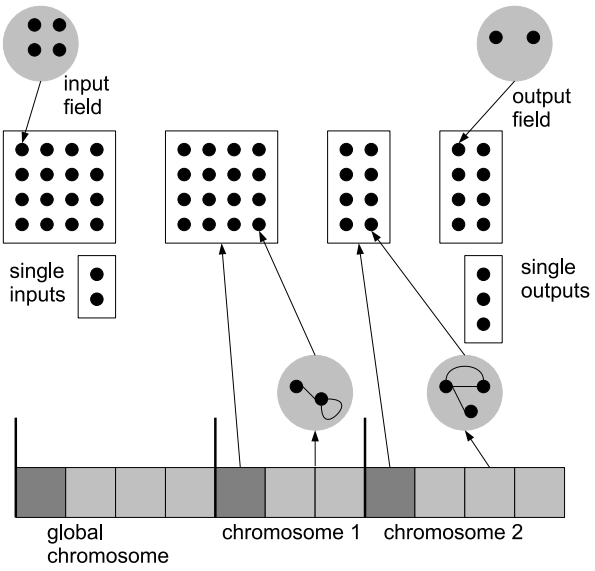

Figure 1: Encoding and architecture of NEATfields networks. A schematic representation of the genome is shown at the bottom. The gray circles contain either the input and output nodes structured as fields or the NEAT subnetworks. Within the fields, they are displayed as black dots. So each black dot in these fields stands for a whole subnetwork. Global or lateral connections are not shown here.

is possible to start with several internal fields that are connected to a subset of input and output fields each, thereby imposing some network modularity from the start). The internal field of the common ancestor contains one neuron for each different output, i.e. for each different output within the same output field element. In contrast, corresponding outputs in different elements of a field are treated as one, so if the output field is of dimension $n \times n$, for example, a neuron in the internal field will project to n^2 neurons in the output field. This number can shrink during evolution as the size of the internal field grows. The next subsection contains more information about how fields are connected.

2.2 Encoding

Each field has a corresponding genetic element (called chromosome for convenience) that encodes all its parameters (see Figure 1). The first gene in a chromosome specifies the field size. Next, there are as many node and connection genes as needed to specify one field element. All genes contain a unique reference number that is assigned once the gene is generated through a mutation. In addition, connection genes contain a connection weight, a flag indicating whether the connection is active, and the reference numbers of the source and target neurons (as well as additional data that is explained below).

There is a special "global" chromosome that contains genes coding for global connections. These genes basically contain the same information as the connection genes mentioned before, but the addressing method is slightly different to ensure that network input and output can be referenced.

If the genome contains a global connection gene for connecting fields with the same size, every field element in the target field will get a connection from the field element in the source field that has the same relative position in its field. Their connection weights are all the same because they are all derived from a single gene. If field sizes are different in a dimension, then NEATfields will still use a deterministic and topology preserving method of connecting the fields (see figure 2): if the source field is smaller than the target field, outputs are projected evenly to the target field (e.g., if it has only half the size of the target field, each field element projects to two adjacent target field elements); if the source field is larger than the target field, the target field elements get averaged input from an evenly distributed number of adjacent source field elements. In other words, the genetically specified weight is then divided by the number of connections coming to a single target field neuron from the source field neurons.

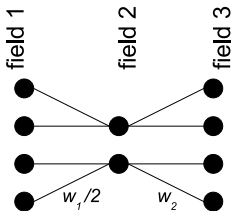

Figure 2: Global connections between elements of different fields (shown here as one dimensional) are wired in such a way that neighborhood between field elements is preserved and with automatic averaging (left) or projection (right) if fields have different sizes.

2.3 Mutation operators

2.3.1 Mutations for small subnetworks

NEATfields uses mutation operators that are very similar to those of NEAT. The most common operation is to choose a fraction of connection weights and either perturb them using a normal distribution with standard deviation 0.18, or (with a probability of 0.15) set them to a new value. The application probability of this weight changing operator is set between 0.89 and 0.94, depending on the other operators used. An operator to connect neurons is used with probability 0.02, while an operator to insert neurons is used with probability 0.001. The latter inserts a new neuron between two connected neurons. The weight of the incoming connection to the new neuron is set to 1.0, while the weight of the outgoing connection keeps the original value. The existing connection is deactivated but retained in the genome. There a two additional operators, one toggles the active flag of a connection and the other sets the flag to 1. Both are used with probability 0.01.

2.3.2 Evolving large scale topology

For evolving the large scale topology, NEATfields introduces some new operators. One operator doubles the field size along one dimension (at a probability of 0.004) and another increases the dimension by one (at a probability of 0.004). In addition, there is an operator that inserts global

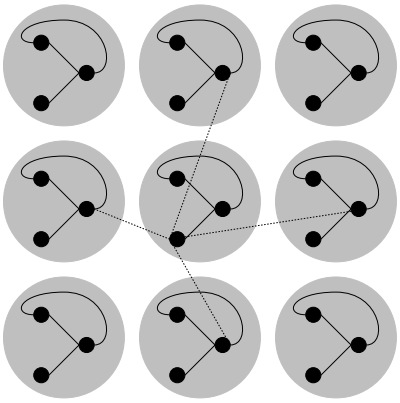

Figure 3: Lateral connections are established between a field element and its four neighbor elements (if it is not at the border). They are shown only for the central element as dotted lines here.

connections (at a probability of 0.01) and an operator that inserts a field of size 1×1 into an existing global connection(at a probability of 0.001). An existing field can also be duplicated, where all elements of the new field received new reference numbers. However, we do not use this operator here in most experiments.

2.3.3 Flow of information within neural fields

Lateral connections between field elements can enable flow of information within a neural field. Here, each field element is connected to its up to four neighbors (see figure 3). The connection is actually not between fields as such, but between neurons in the respective fields. The gene coding for a lateral connection specifies source and target neuron reference numbers just as genes coding for local connections do; it is also located in the same chromosome. But it has a lateral flag set to 1, and is created by a lateral connect operator (at a probability of 0.02).

2.3.4 Dehomogenizing neural fields

By default, corresponding connections in different field elements all have the same strength so they can be represented by one gene. The same is true for the global connections between field elements of two fields. For some tasks, it may be useful to have field elements that react slightly different to input, which can create what has been called "repetition with variation" [15]. One way to realize this is to have larger connection weights in a neighborhood of some center coordinates on the field. Here, connection weights are scaled according to $\exp(-\epsilon(\frac{distance}{field\ size})^2)$ (in our implementation, this is done separately for different dimensions), where $\epsilon = 5.0$ means that connections close to the center have a large weight and the rest will have weak weights. The center coordinates are specified in the following way: There are two eight bit values in the gene encoding a connection, one for each dimension. These are converted to two numbers between -1 and 1. If a number is between -0.15 and 0.15, the connection weights are homogeneous in the corresponding direction. This is the default for all connections. If a value is outside this range, on the other hand, then it is mapped linearly to a position between the two field borders. There is a mutation operator that (at a probability of 0.03) sets the values for a single connection gene.

In principle, a field can be completely dehomogenized by many of these connections with what we call "focal areas", but there is a faster way of doing so. The connection weights corresponding to a single gene can also be scaled by a factor that is random with respect to position in the field. Here, a random factor does not mean that the factors are randomly drawn every time the network is created. Instead, the innovation number of the gene is used as a pointer to a "random data pool" that remains constant. A similar technique has been used in previous work [9]. Technically, the innovation number is used as a seed for a common random number generator that generates the required amount of data. Of course, learning of the randomization is impossible because it is fixed, but evolution can add randomization to arbitrary many connection genes, so a desired pattern may be achieved by interaction of several available patterns. Fine tuning the dehomogenization can also be done subsequently by using the focal area dehomogenization described above.

Randomization of connection weights is specified by a flag in the connection gene. In those experiments that use this technique, we create 25% of all connections with the flag set. The flag does not mutate subsequently.

2.4 Selection methods

NEATfields uses speciation selection with variable speciation threshold like in some variants of NEAT [14, 6]. The dissimilarity between two networks is calculated as follows:

$$\begin{aligned} d &= c_n \#ref_n + c_r \#ref_c + c_w \sum \Delta w \\ &\quad + c_f \#ref_f + c_s \sum \log(1 + \Delta s_x + \Delta s_y) \end{aligned}$$

where $\#ref_n$ is the number of nodes present in just one of these networks, $\#ref_c$ is the number of connections present in just one of these networks, $\#ref_f$ is the number of fields present in just one of these networks, Δw are the connection weight differences (summed over pairs of connections that are present in both networks), the Δs are the field size differences in the x and y dimension (summed over pairs of fields present in both networks), and the c variables are weighting constants.

Using this measure, the population is partitioned into species by working through the list of individuals. An individual is compared to representative individuals of all species until the dissimilarity between it and a representative is below a certain threshold. It is then assigned to this species. If no compatible species is found, a new species is created and the individual becomes its representative. The number of offspring a species has is proportional to its mean fitness. Inside the species, the worst 60% of its members are deleted, after which uniform selection is used for the rest. Species with an offspring size greater than five also keep their best performing individual. If the maximum fitness of a species has not increased for more than 200 generations and it is not the species containing the best network, its mean fitness is multiplied by 0.01, which usually results in its extinction. Also, in order to keep the number of species in a specified range, the dissimilarity threshold is adjusted in every generation if necessary.

For the experiments reported here, we use an initial speciation threshold of 4.0, and set $c_n = 0.0$, $c_r = 1.0$, $c_w = 2.0$,

$c_f = 1.0$, $c_s = 2.0$. We use four different population sizes, each with a different target number of species: 100 (2 to 8 species), 150 (3 to 9 species), 200 (4 to 10 species), and 1000 (35 to 45 species).

2.5 Experimental Procedure

20 runs were performed for each experiment. For statistical comparisons, we use the Wilcoxon rank-sum test, where we rank the outcomes of successful runs according to the number of evaluations. All unsuccessful runs get a lower rank than the successful runs, while ranking between them is done according to the highest fitness in the final generation.

3. EXPERIMENTS AND RESULTS

3.1 Finding the large square

This task has been implemented following the description given in [5], although without the generalization and scaling tests described there. The network input is a visual field of 11×11 pixel plus bias input, while the output is also a field of size 11×11. On the input field, the networks can see two "black" squares on a "white" background. The first square is of size 3×3 pixel, while the second is just a single pixel. The task for the network is to indicate the position of the center of the large square by its highest output activation. Performance is tested in 75 episodes that are generated at the beginning of the experiment as follows: 25 positions for the small square are chosen randomly. For each position, three trials are generated by positioning the large square 5 pixels down, right or down and right. The grid is taken to be a toroid here, so if the small square is already at the lower margin of the grid, the large square will appear above it. If the larger square is divided by this procedure, it will be moved such that it appears on the grid as a single object.

Here, the network is allowed to compute for 20 time steps before the output is read. The fitness is calculated as $f = \sum_{trial=1}^{75}(200 - (x_{tgt} - x_{out})^2 - (y_{tgt} - y_{out})^2)$ to ensure positive fitness values (the maximum difference between target position and highest output position could be 10 pixels). For comparison with other approaches, the fitness can be used to compute the average distance to the correct target position, where the size of the whole field is set to 2.0×2.0. From the literature, HyperNEAT achieves an average distance of about 0.1 (and sometimes finds perfect solutions) after 250 generations using a population size of 100, while a fully connected NEAT network without structural operations used as a control achieves a distance of about 0.5 [5].

We used six different configurations for each visual discrimination task. The first ("plain") just uses NEATfields without any lateral connections or dehomogenization techniques. The second ("LC") uses NEATfields with lateral connections, but without dehomogenization. The third ("1f") is similar to the LC configuration, but with the insert field operator switched off, so just one field is used during the whole course of evolution. The fourth ("LC-F") uses lateral connections and focal area dehomogenization, the fifth ("LC-R") lateral connections and dehomogenization by randomized weights, and the sixth ("LC-FR") uses lateral connections and both dehomogenization methods. For the task in this subsection, all tasks use a population size of 100.

NEATfields in plain configuration reaches an average fitness of 14949, or an average distance of 0.16 (no perfect solutions evolved), while in the LC configuration, NEATfields reaches a perfect solution in 95% of the runs using 13325 evaluations on average. The 1f configuration solved the task in 100% of the runs using 12186 evaluations on average, which is not significantly different ($p \approx 0.35$) from LC. The LC-F configuration solved the task in 85% of the runs, the LC-R in 100% of the runs, and the LC-RF in 80% of the runs. The numbers of evaluations were similar in the successful runs for these configurations.

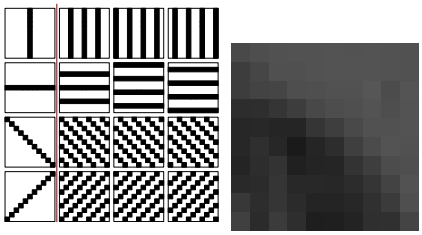

Figure 4: (left) Input patterns for three visual discrimination tasks. The first tasks uses the patterns in column 1; the second tasks uses the patterns in column 2; the third task uses the patterns in columns 2–4. (right) An example pattern for the task of distinguishing orientations of area borders in gray scale images.

Evolved solutions from the LC configuration used between 120 and 576 neurons (mean 265), and between 1028 and 4384 connections (mean 1907). Their genome consisted of between 7 and 18 genes (mean 9.1, and out of these, 3.4 code for global connections on average). All solutions had a primary field of size 12×9 at least, but in some cases up to 32×12. On average, this field consisted of 264 neurons. 1 of the 20 solutions had a second field, which contained 12 neurons. The average field element consisted of 1.2 neurons in all fields. All solutions used at least one active lateral connection gene. The smallest solution used just one neuron per field element and one lateral connection.

3.2 Distinguishing orientations of shapes and textures

The three tasks described here are tasks where dehomogenized neural fields can be expected to be useful. The same 11×11 input field is used as before (in addition, there is a bias input), but there are just 4 outputs for classification. A number of patterns are presented to the network in separate episodes. The patterns are shown in figure 4. So the first task is to map the four simple patterns to the four outputs. For the second tasks, four different textures have to be mapped to the four outputs. For the third tasks, twelve textures have to be classified according to their orientation. The output with the highest activation after 20 time steps is considered to be the classification result. The mapping that has to be learned is pre-specified, for example, the first pattern has to be mapped to the first output.

For every episode where classification is correct, the network scores 200 fitness points. For the other episodes, the network scores between 0 and 100 points depending on the difference between the maximally activated output and the output that should have been maximally activated. That value v, which can be between 0.0 and 2.0, determines the fitness score according to $f = 100 - 50v$. A population of

200 individuals is evolved for 500 generations.

For the simple patterns task, we find that the plain configuration reaches a mean fitness of 695 and a perfect solution in 10% of the runs. The LC configuration reaches a mean fitness of 700 and a perfect solution in 10% of the runs. The 1f configuration reaches a mean fitness of 700 and a perfect solution in 15% of the runs. The LC-F configuration reaches a mean fitness of 785 and a perfect solution in 80% of the runs. The LC-R configuration reaches a mean fitness of 725 and a perfect solution in 25% of the runs. The LC-FR configuration reaches a mean fitness of 785 and a perfect solution in 85% of the runs.

For the four textures task, we find that the plain configuration reaches a mean fitness of 740 and a perfect solution in 40% of the runs. The LC configuration reaches a mean fitness of 755 and a perfect solution in 55% of the runs. The 1f configuration reaches a mean fitness of 750 perfect solution in 50% of the runs. The LC-F configuration reaches a mean fitness of 745 and a perfect solution in 45% of the runs. The LC-R configuration reaches a mean fitness of 755 and a perfect solution in 55% of the runs. The LC-RF reaches a mean fitness of 760 and a perfect solution in 60% of the runs.

For the twelve textures task, no perfect solutions were found by any configuration. Mean fitness was 2060 for the plain configuration, 2088 for the LC configuration, 2107 for the 1f configuration, 2049 for the LC-F configuration, 2074 for the LC-R configuration, and 2033 for the LC-FR configuration.

The differences between the plain and the LC configuration, or between the LC and the 1f configuration, are not significant for any task here. The LC-F configuration is significantly better than LC for the four patterns task, but not for the other tasks. The LC-R configuration is indistinguishable from LC for all tasks. The LC-FR configuration is significantly better than LC on the four patterns task, and significantly worse for the twelve textures task. Note that in this and the following section, a simple t test has been applied with a significance threshold of 0.05 because the Wilcoxon rank sum test could not deal with the many runs that ended with the same maximal fitness (which made ranking impossible).

3.3 Distinguishing orientations of area borders in gray scale images

This task again uses an input field of 11×11 and 4 outputs. This time the patterns are gray scale images with two areas. There are four classes of images. The orientations of the area borders for the four classes are those in column 1 of figure 4 again. On one side of the border, color values are in the range $[0, 0.7]$, on the other side in the range $[0.3, 1]$. As these ranges overlap, classification cannot be done depending on single pixels only. Four instances are generated randomly for every class at the beginning of the run, so all individuals are tested on the same patterns in 16 separate episodes each. Again, the output with the highest activation after 20 time steps is considered to be the classification result, and the mapping to be learned is pre-specified. Fitness is calculated exactly as in the tasks described in the previous paragraph. Again, we evolve a population of 200 individuals for 500 generations.

For this task, the plain configuration reaches a mean fitness of 2771 and a perfect solution in 0% of the runs. The LC configuration reaches a mean fitness of 2815 and a perfect solution in 0% of the runs. The 1f configuration reaches a mean fitness of 2820 and a perfect solution in 0% of the runs. The LC-F configuration reaches a mean fitness of 2997 and a perfect solution in 20% of the runs. The LC-R configuration reaches a mean fitness of 2897 and a perfect solution in 10% of the runs. The LC-FR configuration reaches a mean fitness of 2964 and a perfect solution in 15% of the runs. The LC-F and LC-FR configurations are significantly better than the LC configuration, but the differences between LC and plain, or LC and 1f, or LC and LC-R configuration, are not significant (although it should be noted that LC-R could evolve perfect solutions, while LC could not).

3.4 Balancing multiple poles

Pole balancing tasks are a family of benchmark tasks, in which a nonlinear system has to be controlled and fast reactions are required. The input and output spaces are rather small. A cart can drive back and forth on a track and it has to balance either a single pole mounted on top of it by a hinge joint, or two poles, the second being one tenth the length of the first. Performance is measured by the number of time steps (a maximum is set at 100000) that the cart stays within some distance from its point of origin, and both poles do not deviate from the upright position by more than a certain angle. Neural networks get cart and pole positions as inputs, and in a simpler Markovian version, also the cart and pole velocities. A bias input is also provided. Thus we have the tasks commonly known as "single pole balancing — velocity inputs (SPV) "single pole balancing — no velocity inputs" (SPNV), "double pole balancing — velocity inputs" (DPV), and "double pole balancing — no velocity inputs" (DPNV). There is also a more difficult version of DPNV ("Anti-wiggling" DPNV) with a different fitness function, where wiggling of the poles is punished and generalization to at least 200 out of 625 starting angles is required. These tasks have been described in more detail in [19, 16].

Here we examine performance on a DPNV-based task where multiple instances of such cart-pole systems must be controlled simultaneously. The initial angles of the long poles are evenly spaced between $-1.8°$ and $1.8°$. It is expected, and can be observed, that trajectories of the cart-pole systems will diverge over time. A field of 4×4 pole systems is used by default for the pole balancing system. Each input field element provides four inputs (including a bias input), while each output field element expects one output signal as in single instance DPNV. The fitness is the number of time steps in which all systems are kept balanced. The goal is to balance all systems for 100000 time steps. A population size of 150 is used for the experiments here.

A NEAT-like NEATfields configuration (i.e., with a single internal field of size 1×1 during the whole course of evolution, no field structure in the inputs and outputs, and one neuron for each output initially, each connected to all inputs) can not solve this problem. In fact, the best solution from 20 runs could only balance the pole systems for an average of 36 time steps. The plain NEATfields configuration (i.e. without lateral connections or dehomogenization methods), however, can with probability 1.0 solve the task using 24934 evaluations on average. Evolved solutions have between 16 and 1024 neurons (mean 129, median 44) and between 96 and 6144 connections (mean 694, median 216). They use between 9 and 28 genes (mean 14.6), of which be-

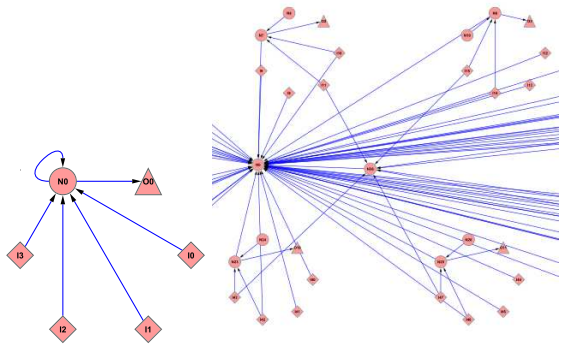

Figure 5: (left) One of the 16 identical modules of a network solving the MDPNV task. This is a completely regular network, and is the simplest solution possible. Hidden neurons are displayed as circles, input neurons as diamonds, output neurons as triangles. (right) A partial view of another network solving the MDPNV task. Four identical field elements can be seen. Some inputs are processed only via shared neurons from another field (shown in the middle). Many connections to other field elements emerge from these neurons . Hidden neurons are displayed as circles, input neurons as diamonds, output neurons as triangles.

tween 5 and 12 (mean 7.3) code for global connections. All evolved solutions use a primary field with sizes between 4×4 and 256×4. These fields contain 127 neurons on average. 9 of the 20 solutions use further fields (up to 3) that contain 4.1 neurons on average. The elements of all fields contain 1.5 neurons on average. The minimal solution that evolved several times used a field of size 4×4 that contained just one neuron with a self-connection. How other solutions use their additional fields is not intuitively obvious. These fields often contain just a single neuron that is connected to all field elements in the primary field (see Figure 5).

A '1f' configuration solves the task using 15556 evaluations on average, which is significantly better ($p < 0.01$). The insert field operation, which is left out in the '1f' configuration, creates a new field with just one neuron and inserts it in the same way as the neuron inserting operation. We also examined the influence of using the field duplication operation instead of, or in combination with, the field inserting operation on the performance (in both cases, we use the field duplication operation with a probability of 0.001). This yields success probabilities of 1.0 and 1.0, and evaluation numbers of 16851 and 18762, respectively. This is significantly better than the plain configuration in the first case ($p \approx 0.02$), but not in the second ($p \approx 0.09$).

Using a larger range of initial pole angles should make the task more difficult because of the initial faster reaction time required. If pole angles are in the range of $-3.6°$ to $3.6°$, NEATfields with a small population can find solutions with a probability of 1.0 using 22642 evaluations on average. This is not significantly different from its performance in the case of standard start angles ($p \approx 0.58$).

NEATfields also scales quite well for larger input and output spaces. It can solve a task with a 16×16 array of pole systems with a probability of 1.0 using 158955 evaluations on average using a population size of 1000.

4. DISCUSSION

The experiments described in this paper show that the NEATfields method is able to find solutions for a a number of tasks with large input and output spaces. It is interesting to know that problems like the "large square task" can be solved with two conceptually different approaches as HyperNEAT and NEATfields.

The large square task cannot be solved without lateral connections. This is understandable because field elements are more or less responsible for processing a pixel or a group of neighboring pixels, and there needs to be a way of detecting several adjacent "black" pixels.

The focal area dehomogenization (or using both homogenizations) makes NEATfields significantly better only on the four patterns task, but not on the four or twelve textures task. This can be explained because the four patterns can be distinguished by monitoring activation in some local areas of the input field. This can be done well with focal area dehomogenization. But the textures have equal activation in all areas, only the orientation is different, so focal areas are not very helpful there.

Distinguishing the gray scale images only works with dehomogenized neural fields. Focal areas are much more useful than randomization here. All in all, while both methods can dehomogenize a field, evolution, as currently set up in NEATfields, can only handle the focal areas satisfactorily.

A common pattern in the results is that those features that make solving some tasks possible will decrease performance on the tasks that are solvable without using them. If evolution has more options, it often takes the wrong way. However, we used rather small populations and few generations. It seems that a configuration with lateral connections and focal area dehomogenization (LC-F) leads to satisfactory results on most tasks examined here if a sufficient number of evaluations is allowed, so this is a reasonable setup for further experiments with NEATfields.

Whether the duplication operation is useful on the tasks tackled here remains unclear. Besides its core effect of changing the way the search space is explored, the duplication operation also influences the structural diversity within the population, and this interacts with speciation selection in a complex way. It has also been suggested that the field inserting operator is too disruptive because it can insert a field of size 1×1 between two large fields. So it might be worth trying a field inserting operator that fits the size of an inserted field to those of its neighbor fields. Regarding duplication, it has been proposed that duplication of the whole field of neurons have occurred in vertebrate brain evolution (see [18] for a discussion of this), and this operation may be useful in artificial evolution for tasks where chains of same or similar processing elements are advantageous. Perhaps the tasks examined here are still too easy to make the presence of several fields an advantage (regardless of the operator used).

The experiments on multiple pole balancing show that NEATfields can transfer the ability of NEAT to solve challenging nonlinear control problems to large input and output spaces. It could be argued that this task is in a sense trivial because the easiest solution is simply to make some identical copies of a controller for a double pole system. The task is indeed not very difficult for NEATfields because it matches

its underlying assumption so well. But it is still challenging for evolution to find the appropriate number of instances, especially while evolving the controller at the same time. Besides, we expect this problem to be quite difficult for neuroevolution methods that do not make similar assumptions. We consider multiple pole balancing as a starting point for investigating evolution of controllers for high dimensional control problems that contain regularities.

Several features of NEATfields have not yet been described in detail or examined systematically. Besides, it will be interesting to see what is necessary to find solutions for other tasks with large input and output spaces. Research work is currently conducted to apply NEATfields to several robotic tasks. We also continue to work on the twelve textures task and some other difficult visual discrimination tasks not described here. One way of solving them could be to use active vision [4]. Also of particular interest are tasks that — unlike those reported here — actually benefit from the use of several fields. We hope that with our method, the scalability of neuroevolution can be enhanced.

HyperNEAT is currently one of the most successful methods for evolving large neural networks using an indirect encoding. This method can learn arbitrary genetic architectures through the use of a neural network-like computational pattern producing network, and can discover many kinds of regularities in the phenotype autonomously [15]. These are desirable abilities, especially if a task becomes gradually more complex in long term evolution scenarios. NEATfields is more limited in this regard because its building blocks are to a large degree externally specified. However, one could extend the NEATfields method by providing other interesting building blocks, e.g. building blocks inspired by findings in neuroscience. That way, knowledge about the task can be embedded into the method, which can make evolution more efficient for a class of problems.

Acknowledgments

5. REFERENCES

[1] Mark Bear, Michael Paradiso, and Barry W. Connors. *Neuroscience: Exploring the Brain*. Lippincott Williams & Wilkins, 3rd edition, 2006.

[2] David D'Ambrosio and Kenneth Stanley. A novel generative encoding for neural network sensor and output geometry. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, 2007.

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

[4] Dario Floreano, Toshifumi Kato, Davide Marocco, and Eric Sauser. Coevolution of active vision and feature selection. *Biological Cybernetics*, 90:218–228, 2004.

[5] Jason Gauci and Kenneth Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007.

[6] Colin Green. Sharpneat. Technical report, http://sharpneat.sourceforge.net, 2006. http://sharpneat.sourceforge.net.

[7] Thomas Hansen. The evolution of genetic architecture. *Annual Reviews of Ecology, Evolution and Systematics*, 37:123–157, 2006.

[8] Simon Harding and Wolfgang Banzhaf. *Organic Computing*, chapter Artificial Development. Springer-Verlag, 2008.

[9] Benjamin Inden. Neuroevolution and complexifying genetic architectures for memory and control tasks. *Theory in Biosciences*, 127:187–194, 2008.

[10] Hod Lipson. Principles of modularity, regularity, and hierarchy for scalable systems. In *Genetic and Evolutionary Computation Conference 2004, Workshop on Modularity, Regularity and Hierarchy.*, 2004.

[11] Kazuhiro Ohkura, Toshiyuki Yasuda, Yuichi Kawamatsu, Yoshiyuki Matsumura, and Kanji Ueda. Mbeann: Mutation-based evolving artificial neural networks. In *Proceedings of the European Conference on Artificial Life*, 2007.

[12] Joseph Reisinger and Risto Miikkulainen. Acquiring evolvability through adaptive representations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007.

[13] Joseph Reisinger, Kenneth Stanley, and Risto Miikkulainen. Evolving reusable neural modules. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004.

[14] Kenneth Stanley. *Efficient Evolution of Neural Networks through Complexification*. PhD thesis, Report AI-TR-04-314, University of Texas at Austin, 2004.

[15] Kenneth Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, pages 131–162, 2007.

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

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

[18] Georg F. Striedter. *Principles of brain evolution*. Sinauer, 2005.

[19] Alexis P. Wieland. Evolving controls for unstable systems. In D. Touretzky, editor, *Connectionist Models: Proceedings of the 1990 Summer School*, 1991.

[20] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87:1423–1447, 1999.

[21] Jianzhi Zhang. Evolution by gene duplication: An update. *Trends in Ecology and Evolution*, 16:292–298, 2003.