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

Evolving neural fields for problems with large input and output spaces

Benjamin Inden, Yaochu Jin, Robert Haschke, Helge Ritter

2012

Preprint:

This is an accepted article published in Neural Networks. The final authenticated version is available online at: https://doi.org/[DOI not available]

Neural Networks 28 (2012) 24–39

Contents lists available at SciVerse ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

Evolving neural fields for problems with large input and output spaces

Benjamin Inden [a,d,*], Yaochu Jin [b], Robert Haschke [c], Helge Ritter [c]

[a] Research Institute for Cognition and Robotics, Bielefeld University, Universitätsstr. 25, 33615 Bielefeld, Germany
[b] Department of Computing, University of Surrey, United Kingdom
[c] Neuroinformatics Group, Bielefeld University, Germany
[d] Artificial Intelligence Group, Bielefeld University, Germany

ARTICLE INFO

Article history:
Received 12 November 2010
Received in revised form 17 November 2011
Accepted 7 January 2012

Keywords:
Neuroevolution
Indirect encoding
Artificial life
NEAT

ABSTRACT

We have developed an extension of the NEAT neuroevolution method, called NEATfields, to solve problems with large input and output spaces. The NEATfields method is a multilevel neuroevolution method using externally specified design patterns. Its networks have three levels of architecture. The highest level is a NEAT-like network of neural fields. The intermediate level is a field of identical subnetworks, called field elements, with a two-dimensional topology. The lowest level is a NEAT-like subnetwork of neurons. The topology and connection weights of these networks are evolved with methods derived from the NEAT method. Evolution is provided with further design patterns to enable information flow between field elements, to dehomogenize neural fields, and to enable detection of local features. We show that the NEATfields method can solve a number of high dimensional pattern recognition and control problems, provide conceptual and empirical comparison with the state of the art HyperNEAT method, and evaluate the benefits of different design patterns.

© 2012 Elsevier Ltd. All rights reserved.

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 (Floreano, Dürr, & Mattiussi, 2008; Yao, 1999). An important advantage of designing neural networks with evolutionary algorithms is that both weights and topology of the neural networks can be optimized. However, if the network topology is changed by evolution, a number of problems can arise that have to be addressed.

One problem is how to add new neurons or connections to a neural network without fully disrupting the function that it already performs. Of course, new elements that are added to the network should change its function to some degree because if there is no change at all, elements without any function could accumulate over the course of evolution, and the networks would become too large. This problem is known as the "bloat" problem in the genetic programming literature (Poli, Langdon, & McPhee, 2008). Another problem is that most evolutionary algorithms use a recombination operator to obtain the benefits that sexual reproduction provides to evolution. Ideally, recombination could combine the good features of two organisms in their offspring. However, it is not obvious what constitutes a "feature" in a neural network with an arbitrary topology, or how corresponding features in two neural networks with different topologies can be found. Similar problems exist for genomes of variable lengths in general. As discussed in the next section, the already well known NEAT method employs techniques that solve these problems satisfactorily. Our NEATfields method makes use of the same techniques.

Another challenge in evolving neural networks is the scalability issue: the evolution of solutions for tasks of a large dimension. This problem has not been addressed by the NEAT method, which typically evolves neural networks of, say, 1–50 neurons. The problem is particularly serious if, like in NEAT, a direct encoding scheme is used for representing the neural network because if every connection weight is directly encoded in the genome, the length of the genome grows linearly with the number of connections. However, the performance of evolutionary algorithms degrades with increasing genome size.

In contrast, indirect encoding of neural networks (Du & Swamy, 2006; Yao, 1999), in which the weights and topologies are

* Corresponding author at: Research Institute for Cognition and Robotics, Bielefeld University, Universitätsstr. 25, 33615 Bielefeld, Germany. Tel.: +49 0 521 106 12107; fax: +49 0 521 106 6011.
E-mail addresses: binden@techfak.uni-bielefeld.de, binden@cor-lab.uni-bielefeld.de (B. Inden), yaochu.jin@surrey.ac.uk (Y. Jin), rhaschke@techfak.uni-bielefeld.de (R. Haschke), helge@techfak.uni-bielefeld.de (H. Ritter).

0893-6080/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.neunet.2012.01.001

generated using grammatical rewriting rules, grammar trees, or other methods, can achieve a sublinear relationship between the genome size and the network size. These methods basically 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 (Lipson, 2004), i.e., some identically or similarly repeated structures. 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 (Mountcastle, 1997). 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 regions whose spatial arrangement preserves the topology of the input (Bear, Paradiso, & Connors, 2006).

A particular kind of indirect encoding methods apply artificial embryogeny to neuroevolution (Harding & Banzhaf, 2008; Stanley & Miikkulainen, 2003). These methods are mainly inspired by biological mechanisms in morphological and neural development such as cell growth, cell division, and cell migration under the control of genetic regulatory networks. By using abstractions of these processes, large neural networks can be built from small genomes.

Here we explore whether a very different kind of indirect encoding, i.e., a multilevel neuroevolution method using externally specified design patterns, can solve the scalability problem. The next two subsections discuss the NEAT method, on which our recently introduced NEATfields method (Inden, Jin, Haschke, & Ritter, 2010) is based, and present the general approach of NEATfields. In Section 2, the technical details of the method are explained. Sections 3 and 4 present experiments on using NEATfields for problems with large input and output spaces. Section 5 compares NEATfields to some other indirect encoding methods used for neuroevolution, and explains why the method is a good choice for many problem domains.

1.2. The NEAT neuroevolution method and derivatives

The NEAT method (Stanley, 2004; Stanley & Miikkulainen, 2002) is a well known and competitive neuroevolution method that introduces a number of ideas to successfully deal with the problems discussed in the previous section. One idea is to give genes an unchanging identity. This is achieved by assigning a globally unique reference number to each gene once it is generated by mutation. These numbers are used to make recombination of neural networks effective. Similar to recombination in nature, recombination in NEAT starts by aligning genomes such that corresponding genes on the two genomes match. Two genes correspond to each other if they have the same reference number. After the alignment is done, the offspring gets exactly one copy of each gene that is present in both parents. For genes that are present in one parent only, other rules are specified (not necessarily the same in all NEAT implementations) to ensure that the offspring is viable with a high probability.

Another idea introduced by NEAT is to protect innovation that may arise during evolution through a speciation technique. For example, if a network with a larger topology arises by mutation, initially it may not be able to compete against networks with a smaller topology that are at a local optimum of the fitness landscape. By using the globally unique reference numbers again, a distance measure between two genomes can be defined and used to partition the population into species. The number of offspring assigned to a species is proportional to its mean fitness. This rather weak selection pressure prevents a slightly superior species from taking over the whole population, and enables innovative yet currently inferior solutions to survive. In contrast, the selection pressure between members of the same species is much stronger in NEAT. Recombination is usually only allowed to occur within a species, such that parents look rather similar to each other and the offspring looks similar to its parents.

Another feature of NEAT is that evolution starts with the simplest possible network topology and proceeds by complexification, that is by adding neurons and connections. It makes sense to search for solutions with a small topology first because the size of the search space for connection weights increases with the network size. There is a mutation operator that adds neurons only between two connected neurons, and adjusts the connection weights such that the properties of these connections change as little as possible. This alleviates the problem of disrupting the existing function of a network.

Due to the success of the NEAT method, quite a few derivatives have been developed. The goal of these has often been to evolve larger networks than those evolved by NEAT, and/or combine the power of NEAT, which uses a direct encoding of connection weights, with ideas from artificial embryogeny. For example Reisinger, Stanley, and Miikkulainen (2004) used NEAT networks as modules and co-evolved them 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 (Ohkura, Yasuda, Kawamatsu, Matsumura, & Ueda, 2007), explicit modularity is introduced into NEAT networks. In the initial network, all neurons are in a first module m_0. A new module is created every time a neuron 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 yet another approach, sets of rules are evolved with NEAT-like speciation that implicitly define a neural network (Reisinger & Miikkulainen, 2007).

In HyperNEAT (D'Ambrosio & Stanley, 2007; Gauci & Stanley, 2007; Stanley, D'Ambrosio, & Gauci, 2009), neurons are embedded into a substrate that has an externally specified topology. For example, the substrate can be a two-dimensional plane. The placement of neurons is determined by the geometry of the given task, as well as some choices made by the user based on previous experience, while the connection weights are generated by giving neuron coordinates as input to another network, which is termed a "compositional pattern producing network" (Stanley, 2007) 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 (Inden, 2008), NEAT mutation operators are used as developmental operators, and a gene can encode arbitrary numbers of operations by referring to an external data pool. However, with the exception of HyperNEAT, these derivative methods have not been used widely, nor have they been used to evolve very large neural networks. We will compare HyperNEAT and a few other recent approaches to evolve large neural networks with NEATfields in Section 5.

1.3. NEATfields: goals and approach

Two considerations provide the motivation for the NEATfields method. The first starts with the notion that in order to make a neural network compute a given function, evolution often needs to change connection weights. In a direct encoding, the problem of changing a connection weight has a simple unimodal fitness landscape as far as the mapping from genotype to phenotype is concerned. In indirect encodings, connection weights are usually calculated using an explicitly or implicitly defined nonlinear function (e.g. the compositional pattern producing network in HyperNEAT). This function will often introduce local optima into the fitness landscape of the weight changing problem, and

therefore make the problem more difficult. Some experimental evidence that weight tuning is indeed difficult for HyperNEAT can be found in Section 5. Therefore, the starting point for the NEATfields method is to combine the direct evolution of connection weights with the evolution of higher-level genetic architecture. An interesting parallel can be found in the genetic architecture of animals: some genes code for enzymes. Changing their sequence will have direct and specific effects on the amino acid sequence of the enzyme, possibly adding new or modifying existing binding sites. Other genes code for transcription factors. Changing their sequence will change complete developmental sequences. For example, a mutation in the *Antennapedia* gene, a famous Hox gene, will cause growth of a complete leg where there would normally be an antenna in the fruit fly *Drosophila melanogaster* (Futuyma, 2005). Similarly, a multilevel neuroevolution method provides means both for changing some individual weights and for producing large-scale topological changes.

The second consideration is that while simultaneous evolution of the immediate function and the underlying genetic architecture is possible and has been demonstrated in some previous studies, it can be difficult to achieve depending on the particular task to be solved, the encoding, the way the search space is explored by the particular method, and the time scales considered (Clune, Beckmann, McKinley, & Ofria, 2010; Clune, Misevic et al., 2008; Kashtan & Alon, 2005; Toussaint, 2003). Certainly having to find a good genetic architecture from scratch is more difficult than starting with preexisting building blocks for a good architecture. Given that artificial evolution is currently severely limited regarding population sizes and numbers of generations as compared to natural evolution, starting with such building blocks may even be crucial for reaching high fitness in some domains. On the other hand, researchers in the field of neuroevolution possess some intuitions on what could be a genetic architecture with high evolvability. Such intuitions can be used to provide evolution with externally specified building recipes, or *design patterns*, as they are termed here. The evolvable parameters of these design patterns can be the higher level elements in a multilevel neuroevolution method.

Technically, the NEATfields method is an extension of the NEAT method. Two features of NEAT seem to be particularly important in this context. 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 for evolving large neural networks. Second, complexification from small structures, i.e., gradual increase of the network sizes during the course of evolution, has been shown to be an important reason for the success of the NEAT method (Stanley & Miikkulainen, 2002). Therefore, exploration of the search space by complexification is also used as a strategy by NEATfields.

The most important design pattern in the NEATfields method is a two-dimensional field of neurons or small neural networks. That way, 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. Therefore, neural networks for these tasks will often employ the same or similar subnetworks many times.

In general, two ways of generating repeated structures can be considered (see Fig. 1). Duplication of one or more existing genes can help evolution to reuse previously found functional elements.

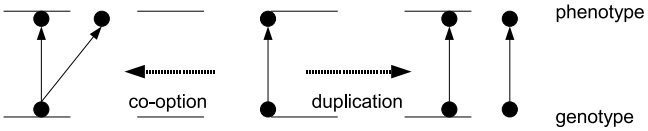

Fig. 1. Duplication versus co-option.

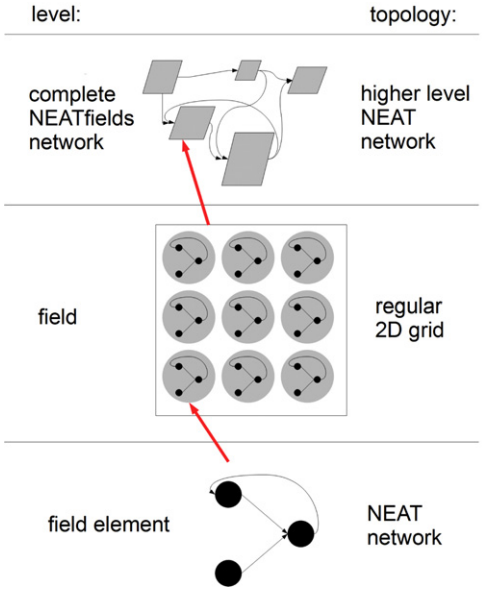

Fig. 2. The three levels of architecture in NEATfields.

Duplication is thought to have been an important mechanism in natural evolution (Soskine & Tawfik, 2010; Zhang, 2003). The duplicated elements are free to subsequently diversify in their function and employment. Although duplication can generate modular and redundant structures, it increases the genome size. The other way of generating repeated structure, called co-option, is to employ a gene or groups of genes in new contexts (Hansen, 2006). This does not increase genome size significantly. Therefore, it is a way to achieve decompression of genotypes into larger phenotypes. Both ways of generating repeated structure seem to be useful for the evolution of neural network topologies, and are used in the method presented here.

NEATfields networks have three levels of architecture (see Fig. 2). At the highest level, a network consists of a number of fields that are connected just like individual neurons are in a NEAT network. At the intermediate level, fields are collections of identical (or similar) subnetworks with a two dimensional topology. At the lowest level, these subnetworks, or field elements, are NEAT networks of individual neurons. In accordance with the idea of searching in smaller topologies first, NEATfields usually starts evolution with a single field of size 1×1. In that case, NEATfields will reduce to a standard NEAT implementation if the mutation operators that change higher-level architecture are switched off.

One issue when using externally specified design patterns is that there are many of them that could perhaps be useful for evolution. Each can be implemented in different ways, and typically has a number of parameters. This could lead to a combinatorial explosion in the design space of the neuroevolution method. However, as will be demonstrated in this paper, two strategies can prevent this explosion from becoming a real problem. The first is to assume that the same design patterns and parameters will be useful (though not necessarily optimal) for many kinds of different tasks. This is a reasonable assumption because the intuitions underlying the design patterns are derived

not so much from the demands of any particular task, but from knowledge about biological neural networks and experience from earlier approaches to the evolution or manual design of artificial neural networks. As will be shown in this article, the satisfactory performance of NEATfields networks on a number of very different tasks provides some support for this assumption. The assumption implies not only that it is unnecessary to tune parameters for each new task individually, but also that initial parameter tuning can be done using benchmark tasks that do not need much run time. The second strategy starts from the assumption that design patterns that are discovered by natural evolution subsequently often become "frozen", i.e. they cannot be changed or discarded any more. While this may prevent many optimal solutions from being found, it apparently still allows finding solutions that can solve very difficult problems sufficiently well. Similarly, we start with a number of possible alternatives for one design pattern. Once benchmark experiments have shown which one works best, we stick to this one while adding further design patterns. This process is started with those design patterns that appear to be the most simple or the most basic (as judged by experience from previous neural network research). If the NEATfields method should at some point fall behind the performance one could expect from neuroevolution, the suggested strategy would not be to re-search its whole parameter space, but to add new design patterns (and perhaps in some cases discard old ones).

Besides validating the method, the experiments in Section 3 are also designed to evaluate the usefulness of the design patterns introduced in Section 2. Parameters for the method and its design patterns have been taken from previous experience where possible. For example, the mutation rates are based on experience with NEAT and the derivative NEON method (Inden, 2008; Stanley, 2004). In other instances, different parameter settings for individual design patterns have been compared empirically. We do not report these results here, but focus on one particular parameter setting each that worked well. The unpublished experiments indicate that the NEATfields method is moderately robust to variation in many, but not all, parameters. One important exception from the method's general robustness is that the probabilities of structural mutation operators must not be too high. Because the employed selection method rewards topological innovation in the networks, too many structural mutations could lead to a runaway complexification, which would not only lead evolution into high dimensional spaces, but consume too much hardware resources.

2. The NEATfields method

2.1. Neural networks

Like in many artificial neural networks, the activation of the neurons in NEATfields is a weighted sum of the outputs of the neurons $j \in J$ to which they are connected, and a sigmoid function is applied on the activation: $o_i(t) = \tanh(\sum_{j \in J} w_{ij} o_j(t-1))$. Like in some other NEAT implementations, connection weights are constrained to the range $[-3, 3]$. There is no explicit threshold value for the neurons. Instead, a constant bias input is available in all networks.

A field element in NEATfields is a recurrent neural network with almost arbitrary topology, where almost arbitrary means the used operators will ensure that no disconnected neurons exist and that there exists 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 is a NEAT-like network where the nodes are fields. It consists of at least one internal field (internal fields are specified by the genome; see Section 2.2), and fields

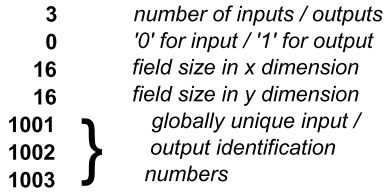

Fig. 3. For construction or mutation of a network, NEATfields gets information about input and output geometry from the task. Here the specification of a single input field is shown as an example. Specifications for several fields can simply be concatenated.

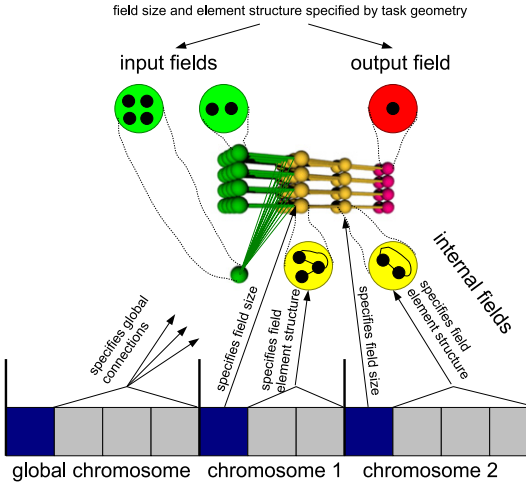

Fig. 4. Construction of a NEATfields network from its genome (bottom). The balls in the central network represent field elements. Their contents are shown in the circles above and below. In these circles, black dots represent the individual (input, output or hidden) neurons.

for network input and output (these are specified by the given task; see Fig. 3). There can be several input and output fields with different dimensions. For example, a bias input can be provided as an 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). It should be noted that connections between field elements or fields are in fact connections between individual neurons in these field elements or fields. How they are established will be described below.

The number of internal fields (as well as the size of each field) in the common ancestor are specified externally for each experiment. Usually evolution is started with a single internal field of size 1×1 that is connected to all input and output fields. So there is only one field element, and it contains one neuron for each different output. If the internal field starts with a size of 1×1, and the output field has fixed dimensions $n \times m$, then the field element in the internal field is connected to $n \cdot m$ output field elements. This number can shrink during evolution if the size of the internal field increases.

2.2. Encoding connections and other network elements in the genome

The parameters for an individual field are encoded in the genome on a corresponding chromosome (see Fig. 4). The first gene in a chromosome specifies the field size in x and y dimensions. After that node and connection genes for one field element are specified. In the current implementation, all genes have a length of 160 bits and may contain several numerical parameters and flags as well as unused space for extensions. However, each flag or numerical value within the gene basically has its own specialized mutation operator, so the genes could equally well be implemented as any

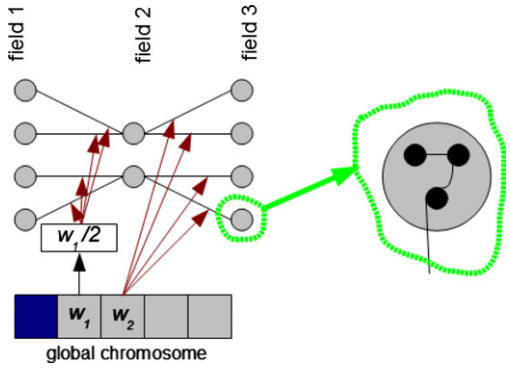

Fig. 5. An example of how the global connections between neurons in fields of different sizes (shown here as one dimensional) are created using a deterministic and topology preserving method. The genetically specified weights are automatically scaled if necessary. As shown in detail on the right side, connections go in fact to individual neurons within the field elements as specified by the connection gene.

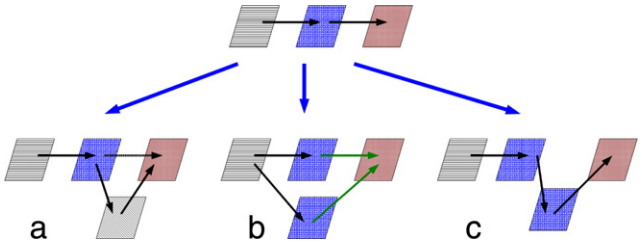

Fig. 6. Three methods to create a new field in NEATfields. (a) A completely new field (with one neuron per field element) is inserted into an existing connection. The existing connection is deactivated. This is the equivalent of the split/insert neuron mutation of NEAT on the level of global network topology. (b) A field is duplicated, and the copy is made parallel to he original, and the outgoing connection weights are halved. (c) A field is duplicated, and the copy is inserted after the original. Respective neurons in respective field elements are connected between the two copies. Note that in all cases, the connections displayed here are in fact sets of connections specified by one global connection gene.

kind of data structure as long as there are enough bits available for sufficient numerical accuracy. 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). Node genes contain additional values that are not used for the experiments described in this article.

There is a special chromosome that contains genes coding for global connections. Global connections contain reference numbers of a source and a target. These can be reference numbers either of neurons or of inputs and outputs as specified by the task description. They must be in different fields—global connections between two neurons in the same field are never created by the NEATfields method. Due to the higher level architecture of NEATfields, a neuron with a given reference number will be present n times in a field with n field elements. A single global connection gene implicitly specifies connections for all these neurons. If a global connection is between neurons in fields with the same sizes, 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 x and y dimension) in its field. Their connection weights are all the same because they are all derived from one gene. If field sizes are different in a dimension, then the fields will still be connected using a deterministic and topology preserving method (see Fig. 5): if the source field is smaller than the target field, each source field neuron projects to several adjacent target field neurons, whereas if the source field is larger than the target field, the target field neurons get input from a number of adjacent source field neurons, while the genetically specified connection weight is divided by that number. That way, field sizes can mutate without changes in the expected input signal strength.

2.3. Mutation operators

2.3.1. Mutation operators for the field element networks

The NEATfields method uses mutation operators that are very similar to those of the original NEAT implementation for evolving the contents of the field elements. 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 to 1.0 minus the probabilities of all structural mutation operators, which amounts to between 0.8815 and 0.949 in the experiments reported here. In general, structural mutations are applied rarely because they will cause the evolutionary process to operate on larger neural networks and search spaces. A structural mutation 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 where it might be reactivated by further mutations. There are two operators that can achieve this: 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 network topology on higher levels

For evolving higher level topology, NEATfields introduces some new operators. At the level of a single field, one operator doubles the field size along one dimension (with a probability of 0.001) and another changes the size (for both dimensions independently) to a random value between its current size and the size of the largest field it is connected to (with a probability of 0.005).

At the level of the complete NEATfields network, there is an operator that inserts global connections with a probability of 0.01. Another operator inserts a new field into an existing global connection with a probability of 0.0005 (higher probabilities will often lead to the evolution of unnecessarily large neural networks). The size of the new field is set randomly to some value between 1 and the larger of the sizes of the two fields between which it is inserted. This is done independently for both dimensions. These two operators correspond to operators already used to evolve the topology of the individual field elements. In addition, an existing field can also be duplicated, where all elements of the new field receive new reference numbers. The new field can either be inserted parallel to the old one (in this case, the outgoing connection weights will be halved to prevent disruption of any existing function) or in series with the old one (in this case, every neuron in every field element in the new field gets input from the corresponding neuron in the corresponding field element in the old field, while the output from the new field goes to where the output from the old field went previously). These two operators allow for reuse of previously evolved structure. The serial duplication operator is also applied on input fields, in which case an internal field is created that contains one neuron for every input in the input field. Both mutations occur with probability 0.0005. The three possible methods of creating a new field in NEATfields are depicted in Fig. 6.

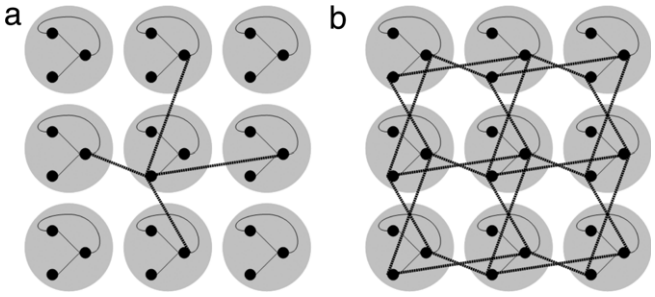

Fig. 7. Lateral connections are established between a neuron in a field element and another neuron in each of the up to four neighbor field elements. There are less neighbors if it is at the border of the field. (a) Lateral connections are only shown for the central element as thick dotted lines here for clarity. (b) All lateral connections constructed from a single gene.

2.3.3. Flow of information within neural fields

NEATfields networks can also have lateral connections between field elements of the same field. These enable flow of information within the neural field. Lateral connections are like local connections: they are between two neurons in the NEAT network that describes the field elements. However, the connection from the source neuron does not go to a neuron in the same field element, but to the corresponding neurons in the up to four neighbor field elements instead (see Fig. 7). 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. However, it has a lateral flag set to 1, and is created by a lateral connect operator (with a probability of 0.02). This is the only kind of lateral connection in the original NEATfields version (Inden et al., 2010). Further kinds of lateral connections will be introduced in Section 2.4.

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 differently to input, which can create what has been called "repetition with variation" (Stanley, 2007). One design pattern that can provide this makes connection weights larger in a neighborhood of some center coordinates on the field. Here, connection weights are scaled according to $\exp(-\epsilon(\frac{\text{distance}}{\text{field size}})^2)$ (in our implementation, this is done separately for the x and y dimensions), where $\epsilon = 5.0$ is chosen such that connections close to the center have a large weight and the rest have weak weights (see Fig. 8). 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 dimension. This is the default for all connections because they are usually created with these values set to 0.0. 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 can be called "focal areas", but another design pattern can achieve this even faster. 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 (this would introduce noise into the mapping from genotype to phenotype, and thereby reduce the information that can be gained from a single fitness evaluation). Instead, the innovation number of the gene is used as a pointer to a "random data pool" that remains constant. A similar technique was used by one of the authors in previous work (Inden, 2008). It works by passing the innovation number as seed to a common random number generator that then generates the required amount of data. Of course, searching in the random data pool is impossible because it has no structure, but evolution can add randomization to arbitrary many connection genes, so a desired pattern may be achieved by the interaction of several available patterns. Fine tuning the dehomogenization can also be done subsequently by using the focal area dehomogenization described above. In those experiments that use randomization, we set a corresponding flag in the genes of 25% of all newly created connection genes. The flag does not mutate in subsequent generations.

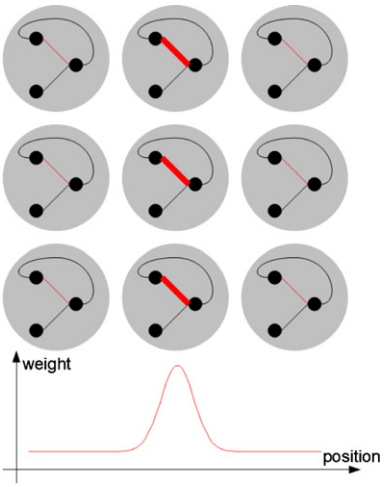

Fig. 8. Dehomogenization of neural fields by the focal areas technique. The thickness of connections here symbolizes their weights. The lower portion of the figure shows the weight scaling factor as a function of x position in the field.

2.4. Building local feature detectors

In the NEATfields method as it has been presented thus far, a field element cannot get input from several adjacent field elements in another field and process this input in arbitrary ways. It can only get input from adjacent field elements in another field if the two fields are of unequal sizes, but that input is averaged automatically by the technique for establishing global connections described above, so processing is possible in a limited way only. More sophisticated processing may be necessary for some tasks. For example, if the input field contains camera data, detection of local features such as lines or edges in a particular orientation will require the comparison of adjacent cells of the input. In principle, flow of information between adjacent field elements is possible through lateral connections. However, these are, as implemented here, homogeneous and isotropic, and only established between immediate neighbors. In this section, five different design patterns are discussed that could perhaps improve the capabilities of NEATfields in detecting local features.

Inspiration for the first two design patterns comes from earlier work on convolutional networks (LeCun, 1998). Convolutional networks are feedforward neural networks with a manually designed topology that learn by backpropagation. Learning effort is reduced by a weight sharing mechanism that is in fact quite similar to the way NEATfields produces large neural fields from small NEAT networks. Convolutional networks are composed of convolutional and subsampling layers. Neurons in convolutional layers have local receptive fields intended for the extraction of

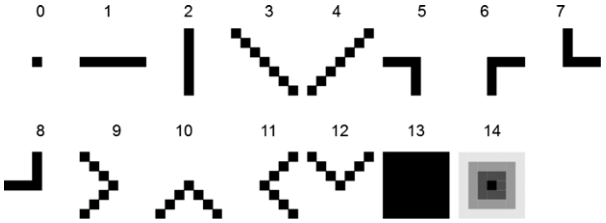

Fig. 9. Feature patterns for use with the NEATfields method. Black denotes a connection weight of 1.0, white a connection weight of −1.0, and the gray scales denote connection weights between 0.0 and 1.0.

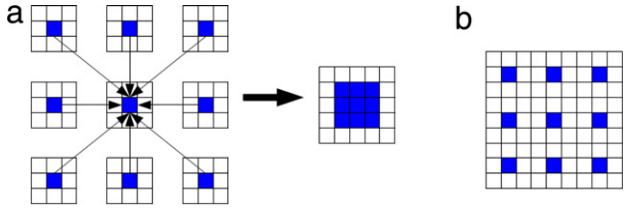

Fig. 10. Organization of receptive fields for a single field element acting as a local feature detector. (a) Superposition of overlapping receptive fields can average out or distort the pattern and is prevented. (b) Instead, non-overlapping receptive fields are tiled such that the input field is covered. This problem occurs if the target field is smaller than the source field of a global connection, and the connection uses the first or second local feature detection option described in the text.

elementary visual features. Detection of local features is what we want to do with NEATfields, too. Convolutional networks have in fact already been successfully applied to problems such as the recognition of handwritten digits.

In order to make several connections from adjacent field elements from just one connection gene, our first and second design patterns for local feature detection use an additional number on the connection gene that refers to one of at most 16 prespecified connection weight patterns. For the first design pattern, these weight patterns have been designed by hand to be useful for pattern recognition (see Fig. 9). For the second design pattern, these weight patterns are just composed of random numbers between −1 and 1. In both cases, a value from the weight pattern is multiplied by the original connection weight specified in the gene to get the weight of the connection from the field element at the respective position. The patterns have a size of 7×7. The central element of the weight pattern corresponds to the weight modification of the connection between exactly corresponding field elements. The feature detectors do not have to use an entire weight pattern. Instead, the maximal offsets between the field element positions in the source and target fields can be between 0 and 3 separately for the x and y dimensions, and are also specified on the genome. There is a mutation operator (used with probability 0.02 if local feature detectors are used) that mutates the weight pattern reference and maximal offsets of an already existing connection. Both in that case and when a new connection is created, choices of weight pattern use and sizes are made based on probabilities that are externally specified for a given experiment. In the experiments reported below, the default for method one is to use weight patterns 1–4 from Fig. 9 with equal probabilities. The default for method 2 is to use 10 random weight patterns with equal probabilities. The maximal offsets in x and y direction are 0 or 1 with equal probabilities by default. These seem to be reasonable basic settings, each using a set of complementary patterns. Comparisons with some other settings will also be provided in this article.

It should be noted that when these local feature detectors are used in a situation where the target field is smaller than the source field, the automatic projection and scaling method used in NEATfields may provide many of these feature detecting sets of connections to the same neuron such that their receptive fields overlap. This may lead to the particular pattern of connection weights being averaged out or at least distorted by superposition. It would also violate the general principle of NEATfields that there is never more than one connection between two given neurons. Therefore, NEATfields allows only one set of these connections established for any given neuron. A regular covering of the source field is created, removing all feature detecting connections that would lead to overlap of the receptive fields (see Fig. 10). Of course that means that a given input pattern cannot be detected at all positions in such a situation. But as it can be detected in some positions, there may be a path along which evolution can find better and better solutions.

The third and fourth design patterns intended for local feature detection take a very different approach. Here, a connection gene only codes for one connection. The additional space on the gene is used to specify offsets in x and y dimensions. They can be in the range $[-3, 3]$. So several otherwise identical connection genes can coexist if the offset is different. These methods use a similar mutation operator to that used by the first two design patterns. The probabilities of the different offsets are also externally specified for a given experiment. However, it should be noted that these design patterns cannot be combined with either of the other two because of the rule in NEATfields that two neurons never should be connected by more than one connection. Mixing the design patterns would make checking this condition very difficult, and would easily lead to situations were one connection gene blocks the evolution of other connection genes at that position. Of course, conflicts between the automatic connection method for global connections and a connection with a genetically specified offset will also arise if the target field is smaller than the source field. This issue is solved by multiplying the genetically specified offset with the number of source field elements that project to a given target field element to arrive at the offset as realized in the phenotype.

The third and fourth design patterns differ in the way connections arise by mutation. For the third design pattern, a single connection is generated every time. For the fourth design pattern, many connections up to some offsets in x and y dimensions are created at once when the mutation operator is applied. The probabilities for different maximal offsets (including the standard offset 0) are again externally specified like in the first and second design patterns.

The fifth design pattern intended for local feature detection is inspired by computational neuroscience and in particular by the work of Mouret, Doncieux, and Girard (2010), which is discussed in more detail in Section 5. In their method, fields of neurons can be connected in an all-to-all fashion. Ultimately we intend to implement this for NEATfields as well. However, this method is not compatible with the way fields are usually connected in NEATfields, therefore the implementation effort is quite high. Here we examine a related and simpler design pattern where all-to-all connections are just possible within fields, that is, as two new types of lateral connections. Compared to the lateral connections introduced earlier, these lateral connections allow a faster flow of information between field elements that are not adjacent. In experiments where the method is used, the lateral connection operator produces a standard lateral connection with probability 0.5, and the two all-to-all lateral connection types with a probability of 0.25 each. In the first all-to-all lateral connection type, the connection weights are all the same. In the second type, they are scaled according to $\exp(-5\frac{d_x^2+d_y^2}{s^2})$, where d_x and d_y are the distances in x and y dimension, and s is the field size in the larger of the two dimensions.

2.5. Selection methods

NEATfields uses speciation selection with variable speciation threshold like in some variants of NEAT (Green, 2006; Stanley, 2004). The dissimilarity between two networks is calculated as follows:

$$d = c_n \#\text{ref}_n + c_r \#\text{ref}_c + c_w \sum_{(i,j)\in CC} |w_i - w_j| + c_f \#\text{ref}_f + c_s \sum_{(i,j)\in CF} \log(1 + |s_i^x - s_j^x| + |s_i^y - s_j^y|)$$

where $\#\text{ref}_n$ is the number of nodes present in just one of these networks, $\#\text{ref}_c$ is the number of connections present in just one of these networks ($\#\text{ref}_n$ and $\#\text{ref}_c$ are only counted in fields that are present in both networks, otherwise the excess nodes and connections are just ignored), $\#\text{ref}_f$ is the number of fields present in just one of these networks, CC is the set of all pairs of connection genes with the same reference number and w are their respective weights, CF is the set of pairs of fields with the same reference number and s^x and s^y are the field sizes in the x and y dimension, 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 $c_n = 0.0, c_r = 1.0, c_w = 1.0, c_f = 1.0, c_s = 1.0$. We use three different population sizes, each with a different target number of species: 100 (2–8 species), 150 (3–9 species), and 1000 (35–45 species).

The selection method as described so far was originally designed for small neural networks, where growth occurs by addition of single neurons. In NEATfields, some mutation operators double the size of the network. We found in preliminary experiments that while evolution works well with large networks, in a very small fraction of runs networks become so large that the physical resources of the computer are exhausted. To prevent the simulation from crashing or becoming very slow in such a situation, we set the fitness of networks that exceed absolute limits of the numbers of nodes or connections to 0.0. The limits are obviously implementation and hardware specific. Here we set them to 10 000 nodes and 100 000 connections. These numbers are orders of magnitude above what we expect to be necessary to solve the tasks described in the next sections.

2.6. Evaluation

For all statistical comparisons in this article, we use Wilcoxon's 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. We perform 20 runs for each experiment.

3. Pattern recognition with evolving neural fields

3.1. General setup of experiments

The experiments reported in this section are designed to answer two questions. What kinds of problems with large input and output spaces can the NEATfields method solve? And what particular design patterns might enable it to do so? For evaluating design patterns, we follow the incremental approach outlined in Section 1.3. For this purpose, we use the following configurations.

- The "basic" configuration uses NEATfields without any design patterns for lateral connections, dehomogenization techniques, or local feature detectors.
- "LC" uses basic NEATfields together with the design pattern for lateral connections.

By comparing the performance of the basic and LC configurations, we can evaluate the design pattern for lateral connections.

- "LC-F" adds the focal area dehomogenization design pattern to LC.
- "LC-R" adds the random dehomogenization by design pattern to LC.
- "LC-FR" adds both dehomogenization design patterns to LC.

By comparing LC-F, LC-R, and LC-R to the LC configuration (and to each other), we can evaluate the two proposed design patterns for dehomogenization.

- "LFD1" adds the first proposed design pattern for local feature detection (externally specified simple connection patterns) to LC-F.
- "LFD2" adds the second proposed design pattern for local feature detection (externally specified random connection patterns) to LC-F.
- "LFD3" adds the third proposed design pattern for local feature detection (single connections with offset) to LC-F.
- "LFD4" adds fourth proposed design pattern for local feature detection (a whole set of single connections with offset created simultaneously) to LC-F.
- "LFD5" adds the fifth proposed design pattern for local feature detection (all-to-all lateral connections) to LC-F.

By comparing LFD1, LFD2, LFD3, LFD4, and LFD5 against LC-F, we can evaluate the proposed design patterns for local feature detection.

The population size is 100 in the "large square" task, and 1000 in all other tasks. The "large square" task is evolved for at most 250 generations, the others for at most 1000 generations.

3.2. Finding the large square

This task has been implemented following the description given in (Gauci & Stanley, 2007). 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 $f = \sum_{\text{trial}=1}^{75} (200 - (x_{tgt} - x_{\text{out}})^2 - (y_{tgt} - y_{\text{out}})^2)$ 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 is also 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 (Gauci & Stanley, 2007). This task is interesting because it shows whether NEATfields can reach the same performance level as HyperNEAT on a task with large input and output spaces, and because information from different positions in the input field must be integrated to decide whether the small or the large square is present.

3.3. Distinguishing orientations of shapes and textures

In this section, four similar simple pattern recognition tasks are introduced. For the first task, an 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. Like in the previous task, the network is allowed to compute for 20 time steps before its outputs are read. The output with the highest activation is considered to be the classification result. The mapping that has to be learned is prespecified, so, 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 $f = 100 - 50v$. The patterns are shown in Fig. 11(a). This task should be easily solvable with dehomogenization techniques because each pattern is present in different areas of the visual field.

The second task uses the four patterns in Fig. 11(b) instead. This may be less easily solvable because all patterns have black dots in all areas of the visual field. But as the orientations of the black lines are different, local feature detectors may be useful to solve the task.

The third task uses six patterns as shown in Fig. 11(c) that have to be classified according to their two different orientations. Here there is only one output. A full fitness of 200 is awarded in a trial if the distance between the correct output (1 or −1) and the network output is less than 0.5. Otherwise, the fitness is proportional to the distance. This task cannot be solved by just looking at a single point anywhere on the visual field because there is a pattern from each class that includes that particular point. However, it could be solved by looking at combinations of pixels, either by using very finely tuned dehomogenization, or by using local feature detectors that are sensitive to a particular orientation.

The fourth task shows a small diagonal bar in two possible orientations as shown in Fig. 11(d). It is displayed on an input matrix of size 7×7, where it can be in any position provided that it fits onto the matrix completely. Therefore, the center of the pattern must be at least 2 pixels away from the border of the visual field. This leaves 9 possible positions, so there are 18 trials altogether. The fitness function is identical (up to multiplicative constants) to that of the previous task. Because of the many different positions tested, one could expect that local feature detectors sensitive to a particular direction would make the task easier than just dehomogenization.

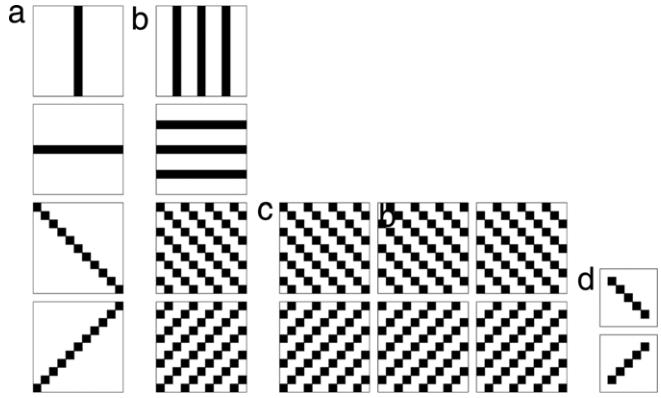

Fig. 11. Input patterns for the "four patterns", "four textures", "six textures", and "two orientations" tasks as described in the text.

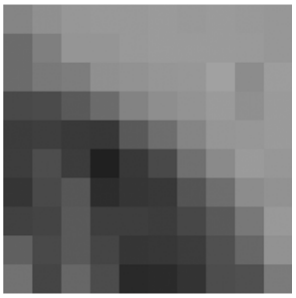

Fig. 12. An example gray scale image for the "area borders" task. The contrast of the image has been greatly enhanced for better visibility.

3.4. 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 (see Fig. 12 for an example). There are four classes of images. The possible orientations of the area borders are those in Fig. 11(a) 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 prespecified. Fitness is calculated exactly as for the tasks with four outputs described in the previous paragraph. 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 $f = 100 - 50v$.

3.5. Results

Results are listed in Table 1. Surprisingly, even the basic NEATfields configuration can sometimes find solutions for four of the tasks. As an examination of evolved networks reveals, this is mostly caused by the ability of NEATfields to create a series of connected fields of different sizes. The automatic projection method that NEATfields applies for fields of unequal sizes makes information flow between different positions within the fields possible. However, the performance of this basic configuration on most tasks is not satisfying.

Table 1
NEATfields performance on pattern recognition tasks. The upper number is the fraction of successful runs, while the lower number is the average number of evaluations in successful runs. Abbreviations for configurations are explained in Section 3.1. The configuration LFD1 is highlighted because it is recommended as standard for future experiments based on these results.

Configuration		Large square	Four patterns	Four textures	Six textures	Two orientations	Area borders
Basic		0.0 –	1.0 235 429	1.0 78 216	0.15 660 732	0.0 –	0.2 657 334
Lateral connections	LC	1.0 4017	1.0 258 737	1.0 79 104	0.45 624 854	0.1 791 945	0.2 592 440
Dehomogenization	LC-F	1.0 4252	1.0 100 665	1.0 81 466	1.0 480 523	1.0 303 863	1.0 346 011
	LC-R	1.0 4193	1.0 128 115	1.0 69 234	0.9 614 967	0.65 548 612	0.3 519 417
	LC-FR	1.0 5265	1.0 75 551	1.0 74 171	0.9 424 237	1.0 296 500	0.95 339 569
Local feature detection	LFD1	1.0 4384	1.0 73 585	1.0 62 376	1.0 256 898	1.0 250 888	1.0 349 396
	LFD2	1.0 4884	1.0 70 610	1.0 70 436	1.0 234 233	1.0 240 203	1.0 321 885
	LFD3	0.0 –	1.0 103 566	1.0 72 012	0.95 655 039	0.95 491 157	0.95 383 624
	LFD4	1.0 4191	1.0 79 702	1.0 67 562	0.75 509 000	1.0 213 798	1.0 417 501
	LFD5	1.0 5929	1.0 91 387	1.0 87 018	1.0 394 509	1.0 392 036	0.95 453 480

Using the lateral connections design pattern (LC) improves performance significantly for the large square ($p < 10^{-10}$) and the six textures ($p < 0.001$) tasks, while the difference for the other tasks is not significant. The large square task can be solved perfectly. So lateral connections are a useful design pattern for NEATfields and are for good reasons enabled in all other configurations discussed now.

Next, we compare the configurations that use dehomogenization design patterns against LC. Using focal area dehomogenization (LC-F) significantly increases performance for the four patterns ($p < 10^{-7}$), six textures ($p < 0.001$), two orientations ($p < 10^{-10}$), and area borders ($p < 10^{-6}$) tasks, while it does not significantly influence the performance on the other tasks. Random dehomogenization (LC-R) improves performance significantly on the four patterns ($p < 10^{-5}$), six textures ($p < 0.01$), and two orientations ($p < 10^{-6}$) tasks, while there is no significant change for the other tasks. Combining both methods (LC-FR) improves performance significantly on the four patterns ($p < 10^{-9}$), six textures ($p < 0.05$), two orientations ($p < 10^{-10}$), and area borders ($p < 10^{-5}$) tasks, but does not change performance significantly on the other tasks. Using LC-FR instead of LC-F only has no significant effects. All in all, both dehomogenization design patterns can be used to make neural networks respond differently to input at different positions on the visual field, which is obviously often very useful, but they may also make it more difficult for evolution to exploit the symmetry inherent in a task. The more irregular random dehomogenization design pattern is more difficult for evolution to use.

Next, we compare configurations using each one of the five proposed design patterns for local feature detection to the LC-F configuration. LFD1 significantly increases performance on the four patterns ($p < 0.05$) and six textures ($p < 0.001$) task, but does not produce significantly different results for the other tasks. LFD2 performs significantly better on the four patterns ($p < 0.05$) and six textures ($p < 0.001$) tasks, with no significant differences elsewhere. LFD3 performs significantly worse for the large square ($p < 10^{-7}$), six textures ($p < 0.05$), and two orientations ($p < 0.05$) tasks, again without significant differences on other tasks. LFD4 does not perform significantly different on any task. LFD5 performs significantly worse on the large square ($p < 0.05$) and area borders ($p < 0.05$) tasks, but does not perform significantly different on any other task. Therefore, it can be concluded that using design patterns for local feature detection where a number of connections are generated at once, and represented in one gene (LFD1, LFD2), does improve performance on some pattern recognition problems. In contrast, using single connections to connect adjacent positions in different fields (LFD3, LFD4) does not improve performance and sometimes even decreases performance. All-to-all lateral connections (LFD5) within fields do not yield an advantage over lateral connections between neighbors here. The question whether this design pattern works well in combination with one of the other local feature detection design patterns, or may be useful for other kinds of tasks, is left for future work.

As mentioned in Section 2.4, there could be interference between connections between unequal positions on different fields on the one hand, and the automatic connection method for fields of different sizes on the other hand, when the field size grows during evolution. To investigate whether this causes serious problems for the NEATfields method, we performed additional experiments (results not shown) where the size of the internal field had already been made equal to the size of the input field in the LFD1 and LC-F configurations. We found that this modification resulted in a significant advantage for LFD1 on most tasks, but not for LC-F. However, the difference in performance was not large, so it can be concluded that fields can grow in size reasonably well during evolution even if a local feature detection design pattern is used.

Finally, as already mentioned, HyperNEAT achieves an average distance from the true target position of about 0.1 in the large square task, whereas a fully connected NEAT network without structural operations achieves a distance of about 0.5. The results in Table 1 reveal that almost all tested NEATfields configurations reach an average distance of 0.0, i.e., they solve the task perfectly.

It can be concluded from the experiments in this section that the NEATfields method can find solutions for high dimensional pattern recognition problems. Regarding the use of design patterns, the LFD1 and LFD2 configurations seem to be a good choice for a number of different problems, while the LC-F configuration seems to work almost equally well. Therefore, these configurations will be used for subsequent experiments.

3.6. The effect of different feature detector patterns on performance

In the experiments from the previous subsection, the local feature detection method 1 (LFD1) was always used with patterns 1–4 from Fig. 9. We also compared some other settings against this default setting on the "six textures" and "two orientations" tasks. These tasks were chosen because, as shown in the previous section, they seem to benefit most from using local feature detection methods. The following variants of the LFD1#F configuration were considered.

1. A configuration using patterns 5–8 instead of patterns 1–4. These patterns feature edges instead of straight lines.
2. A configuration using patterns 1, 13, and 14 with equal probabilities. These patterns combined allow various forms of "center versus surround" calculations.
3. A configuration only using pattern 14, but with size 7×7 instead of 3×3 as in the default.
4. A configuration only using pattern 14, where all sizes between 1×1 and 7×7 can be used with equal probabilities.
5. A configuration using the default patterns 1–4, but where all sizes between 1×1 and 7×7 can be used with equal probabilities.

The fraction of successful runs was 1.0 in all configurations. Variant 3 was slightly but significantly worse than variants 2 and 5 on the two orientations task, whereas variant 4 was slightly but significantly worse than variant 2 on the six textures task. Apart from these exceptions, there were no significant differences found for either task in pairwise comparisons of the outcomes. It seems that for the tasks examined here, evolution can work with all of these preselected sets of patterns almost equally well. How best to structure the local feature detectors in general or for other specific tasks remains a topic for future research.

3.7. Generalization and scaling to larger visual fields

We also evolved solutions for a visual field of size 21×21 using the LC-F configuration. The method scales well for the large square (100% success, 6124 evaluations on average), four patterns, (100% success, 97 308 evaluations), four textures (100% success, 98 468 evaluations), and area borders (100% success, 198 713 evaluations) tasks. It scales less well for the six textures task (90% success, 660 133 evaluations). It cannot find any solutions for the two orientations task even on an 11×11 visual field (the LFD1 configuration occasionally finds a solution, but performance is not good). One factor that may contribute to an explanation of this performance is that unlike the other tasks, this task also has the number of trials increase with increasing size of the visual field. This is because all possible positions of the stimulus within the inner part of the visual field are tested. In the case of an 11×11 visual field, an evaluation consists of 98 trials. While it may be possible to build successful NEATfields networks for this task, there may not exist a path towards such solutions that has a fitness increase at (almost) every step. This is a hypothesis that will be studied in more detail in future work. So far, it seems that true position-invariant recognition of the simple patterns studied here remains beyond reach with the current version of NEATfields.

Like Gauci and Stanley (2007), we also examined generalization and scaling performance for the "large square" task. We evaluated the final champions from the LC-F configuration on 75 randomly chosen new input configurations each. All 20 champions solved their new tasks without any error. The good generalization performance can be explained by the fact that a basic NEATfields network with lateral connections is aligned very well to this particular task. Regarding scaling, Gauci and Stanley (2007) achieved scaling to visual fields of sizes 33×33 and 55×55 with HyperNEAT by just increasing the substrate resolution without any further evolution. This is not possible with NEATfields. Nevertheless, we took the champions from the LC-F configuration again and evolved them to solve the problem with a 33×33 input field. This took 4428 evaluations on average, which is not significantly faster ($p \approx 0.21$) than evolving directly for the 33×33 input field (5677 evaluations). It remains to be seen whether incremental evolution yields any advantage for more complex tasks.

4. Evolving coordinated reaching movements for segmented arms

This task is somewhat similar to the "octopus arm" task recently studied by other researchers (Koutník, Gomez, & Schmidhuber, 2010; Woolley & Stanley, 2010), but results are not directly comparable because of the very different physical properties of the arm used here. The task involves control of an arm consisting of s segments of identical length. The length of the whole arm is set to 1.0 arbitrary length units, and it is situated in a two dimensional rectangular plane that extends from $(-1.0, -1.0)$ to $(1.0, 1.0)$. Initially, the arm is stretched out horizontally such that its tip is at $(1.0, 0.0)$. There is a controllable joint at the beginning of each segment. Joint limits are $[-\frac{\pi}{2}, \frac{\pi}{2}]$ except for the first segment, which is attached to the origin of the coordinate system and has joint limits of $[-\frac{3}{4}\pi, \frac{3}{4}\pi]$. These joint limits are designed to prevent the arm from reaching around the negative x axis, which could lead it into local optima if it approached a target from the wrong side. Each individual is tested on n different goal positions in separate episodes. The goal positions are chosen randomly within a radius of between 0.2 and 0.95 from the origin, in a direction chosen from the interval $[-\frac{5}{8}\pi, \frac{5}{8}\pi]$. They are the same for all individuals in a given run of the experiment. An episode is considered successful if the tip of the arm is within 0.01 length units of the goal position. At every time step, the individual gets a fitness of $\frac{4-d^2}{20}$, where d is the distance between the tip and the goal. If the tip hits the goal, the episode is terminated, and a fitness of $2 \cdot t_{\max} - t + 1$ is added, where t is the current time step and $t_{\max} = 100$ is the number of time steps before the episode is terminated without success. The final fitness value of an individual is obtained by summing up all fitness contributions over all episodes, and squaring the result to increase the selection pressure.

For this reaching task, the neural network gets input from two fields. The first field is of size 1×1 and provides a bias input and the distance between the arm tip and the goal position in x and y dimensions (scaled to $[-1, 1]$). The second field is of size $1 \times s$ and provides the current joint angles. There is also an output field of size $1 \times s$, the contents of which are interpreted as desired joint angles. For a network output $o_i \in [-1, 1]$, the current joint angle is changed by an amount of $o_i \cdot \frac{\pi}{20}$. The joint angle remains unchanged if the joint would be pushed beyond its limits by that action. If the movement during one time step would result in a collision between different segments, or push a segment out of the angular range of $[-\frac{3}{4}\pi, \frac{3}{4}\pi]$ from the origin of the coordinate system, the movement is canceled and the position of the whole arm remains unchanged.

For this task, we start evolution with an internal field of size 1×1. We use the NEATfields method without any design patterns for local feature detection (this configuration was termed LC-F in the preceding section). A population of 1000 individuals is evolved for at most 1000 generations. In the first scenario, the arm consists of 10 segments, and there are 5 targets. In that case, a perfect solution is found in 60% of the runs, while the mean number of reached targets is 4.5. In the case where there are 10 different targets, a perfect solution is found in 25% of the runs, and the

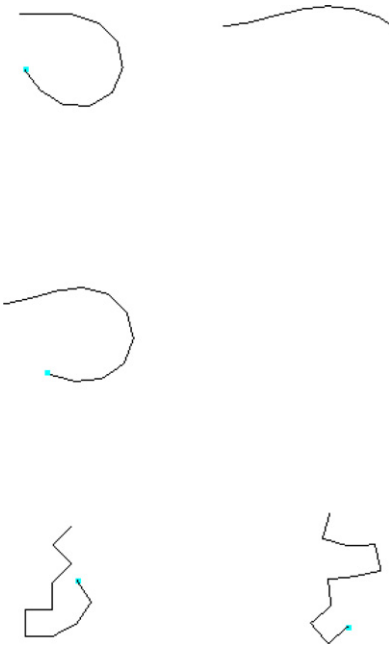

Fig. 13. (Upper two rows) Example end poses for the reaching task with NEATfields. (Lower row) Example end poses for the reaching task with plain NEAT.

mean number of reached targets is 8.9. An arm consisting of 20 segments was also used. In that case, there are 65% perfect runs and 4.6 reached targets on average for the 5 targets setup, and 20% perfect runs and 8.8 reached targets on average for the 10 targets setup. This shows that the task difficulty does not increase with increasing number of segments. NEATfields is able to exploit the inherent regularity in the task. A plain NEAT configuration, on the other hand, achieved no perfect solution and an average of 3.0 out of 5 targets with an arm consisting of 10 segments, and no perfect solution and on average 2.6 out of 5 targets with an arm consisting of 20 segments. Therefore, the ability of NEATfields to exploit regularity is essential for solving this task. The end poses shown for scenario 1 in Fig. 13 also show that NEATfields uses a strategy of repetition with variation to control joint angles: neighboring joint are driven to similar, but not exactly identical angles (they start with the same angle and get the same kind of input, so the differences come from slight differences in the individual field elements). By contrast, NEAT solves the control problem in a completely different way for different joints.

We also measured the generalization performance of the winners of all runs in new scenarios where 1000 targets were given. In the 10 segment arm setup, the controllers that were trained on 5 targets reached 366 ± 34 targets, while those that were trained on 10 target reached 479 ± 26 targets. The difference was significant ($p < 0.05$). In the 20 segment arm setup, the controllers that were trained on 5 targets reached 350 ± 43 targets, while those trained on 10 targets reached 500 ± 21. Again, the difference was significant ($p < 0.05$). The differences between the setups with identical target number, but different number of segments, were not significant. These results show that the learned reaching strategy is not very general, but becomes more general if more training examples are used.

A manually designed network topology for this task would probably use an internal field of size 10×1. In contrast, the smallest evolved successful network used a single internal field of size 3×1, with one neuron in each field element. However, most solutions used fields of size 10×1 or larger. It should be noted that the problem is not solvable using an internal field of size 1×1 because this would set all joints to the same angle, which in turn would make it difficult or impossible to reach many of the specified positions.

The experiments on reaching in this section show that NEATfields can be used for high dimensional control problems. A challenging aspect of this reaching task is that the controlled elements are not independent of each other: the arm can only reach a target if all segments are appropriately coordinated. NEATfields can deal with that. Besides, the experiments show that the NEATfields method can find solutions with topologies that are smaller than manually designed topologies.

5. Comparison to other methods for evolving large networks

One of the earliest methods to evolve large neural networks was Cellular Encoding (Gruau, 1994; Gruau & Whitley, 1993; Gruau, Whitley, & Pyeatt, 1996). This method starts construction of a neural network with a single neuron, and executes instructions from its tree-like genome. Upon cell division, daughter cells inherit the construction program, but each have their own instruction pointer. This allows for reuse, as does an explicit recursion instruction that sets the instruction pointer back to the root of the program. Cellular encoding was used to evolve some impressively large networks for the bit parity problem. On the other hand, its performance on pole balancing was very bad compared to later approaches. While it was easy for evolution to build large network structures with Cellular Encoding, fine tuning the connection weights was obviously not. Cellular encoding was also combined with learning rules for the connection weights. Some other early approaches on evolving large networks relied on matrix rewriting for evolving the topology of a neural network, and considered learning as most important mechanism for setting connection weights (e.g. Sendhoff & Kreutz, 1999).

HyperNEAT (D'Ambrosio & Stanley, 2007; Gauci & Stanley, 2007; Stanley et al., 2009) is the most well known recent method to evolve large neural networks. In HyperNEAT, the number and position of neurons on a substrate is prespecified, while a NEAT-like compositional pattern producing network (CPPN) that gets the neuron coordinates as inputs specifies whether neurons are connected and what the weight of their connection is. Because the CPPN is evolved with the NEAT method, we might expect that HyperNEAT can find good solutions to a wide range of problems from different classes just like NEAT can. In addition, it can evolve very large networks because the CPPN can compute connection weights for arbitrary many neurons without having to become larger. On the other hand, there are a number of characteristics of the NEAT method that are no longer present in HyperNEAT. First, search does not proceed by complexification of the phenotype in HyperNEAT because the number of neurons is prespecified. Second, because it is an indirect encoding, mutations will likely not affect connection weights in the same way as in a direct encoding. In particular, inducing slight and locally restricted changes in connection weights may be much more difficult for evolution. NEATfields, on the other hand, both allows slight and local changes in connection weights and searches by complexification. Does that mean that NEATfields can solve problems that HyperNEAT cannot solve?

A literature search reveals that HyperNEAT has already been used to solve a variety of tasks, among them some on-line control tasks. For example, it has been used to control the wheels of car-like agents in a simulated environment (Drchal, Koutník, & Snorek, 2009). It has also been used to select predefined macro actions in the Keepaway subtask of the Robocup setup (Verbancsis & Stanley, 2010). The Keepaway task has already been used as a benchmark by some reinforcement learning methods, and is known to be challenging because of the large state space, the partial observability of the states, and the noise applied to

sensors and actuators. HyperNEAT has also been used to control robots walking on four legs. In the particular setup reported, the outputs of the HyperNEAT network are used as desired joint angles for the robot, and a P controller is used to approach these angles. In addition, a sine wave is provided as input to the network to ease the evolution of regular gaits (Clune, Beckmann, Ofria, & Pennock, 2009). While performance on the multilegged walking task was best with a hand designed input and output geometry, HyperNEAT still outperformed a direct encoding when the geometry was randomized (Clune, Ofria, & Pennock, 2009). In another experiment, HyperNEAT was used to control a heterogeneous team of hunters in a simulated environment. The movement of each hunter was determined by a speed output and two direction outputs (D'Ambrosio & Stanley, 2008). HyperNEAT also can learn to control reaching movements of a simulated "octopus arm", which consists of many similar segments that change their size depending on the contractions of their internal muscles (Woolley & Stanley, 2010). On the other hand, it was also shown that as the regularity of a problem decreases, HyperNEAT may eventually perform worse than a direct encoding. This was done using two tasks: in a bit mirroring task, the network had to copy the input value on one particular input position to the output at another position. Performance of HyperNEAT decreased as more and more positions changed from an identity mapping to a random mapping. In the second task, the networks were rewarded not for the output they produced, but for their weights being close to a target weight. Again, performance decreased as the pattern of target weights became less regular (Clune, Ofria, & Pennock, 2008). These experiments are relevant to the general question of how well indirect encodings can fine tune connection weights.

Here we focus on two kinds of problems. Problems that require fast and accurate control and therefore will need fine tuning of the connection weights, and tasks that require the evolution of memory and therefore will benefit from evolution of recurrent connectivity.

HyperNEAT and NEATfields are first compared on some simple pole balancing tasks. Pole balancing tasks are a family of benchmark tasks that require fast control of a nonlinear system. 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 100 000) 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 previously (Stanley & Miikkulainen, 2002; Wieland, 1991). While indirect encodings like HyperNEAT have obviously not been designed for these kinds of problems, which have small input and output spaces, and require few neurons and connections to be solved, pole balancing tasks certainly require fine tuning of connection weights. The difficulties of the different versions of these tasks are well known from previous studies. Early neuroevolution methods, as well as several other machine learning methods, were not able to solve the more difficult varieties of these tasks (Stanley & Miikkulainen, 2002). Pole balancing here serves as a simple substitute for higher dimensional problems that will also require such precise tuning of connection weights.

We will also compare the two neuroevolution methods on bit sequence recall tasks as introduced in Inden (2008). Because recall of bit sequences is only possible with recurrent connections or chains of connected neurons, these tasks are useful for examining how a method can evolve the network topology. In principle, HyperNEAT can do this by switching on and off connections in the substrate as long as enough neurons are present there. On the other hand, NEATfields uses NEAT-like complexification for evolving network topology. Again, we use this task as a substitute for high dimensional problems where evolution of structure is also required. It has been shown previously that a full NEAT implementation can solve these problems, while, as in the case of the more difficult pole balancing varieties, it cannot solve them if speciation selection is substituted by simple standard selection methods (Inden, 2008). This is a clear indication of the difficulty of these problems.

Neural networks tested for sequence recall get a signal input and a bias input, and have one output. A single trial proceeds as follows. Within a period of 10 time steps each, the signal input is set to either 1 or -1 during the first 5 time steps, and to 0 afterwards. After the whole sequence of bits has been presented to the network (we use sequence lengths of $n = 1 \ldots 3$ here), the network is run for another n periods. At the end of each period, the output is read, so in the end the relevant network output is again a sequence of length n. The goal is to produce a complement sequence for the input (e.g. "$1\ -1\ -1$" for the input "$-1\ 1\ 1$"). Fitness in a given trial is proportional to how close the actual output is to the complement sequence. To make the task easier, a network output is considered identical to the respective part of the target sequence if they differ by not more than 0.5. The total fitness is summed over all trials, and a task is considered solved if the fitness of the best network differs from the maximum obtainable fitness by not more than 0.01.

We use our own simple HyperNEAT implementation for these experiments. It uses NEATfields with all features specific to NEATfields disabled (i.e. as an implementation of basic NEAT) as a pattern generating network. The pattern generating network gets only three inputs: bias, source neuron number, and sink neuron number. For the small networks we want to generate, we only need a one dimensional topology—there is not much geometry to exploit in the tasks as studied here. The output of the pattern generating network determines the connection weights between any two neurons in the same way as in the original HyperNEAT: if the absolute value is below 0.2, no connection is created; otherwise, the connection is scaled to a value in the range $[-3, 3]$. We use the following transfer functions in the pattern generating network with equal probabilities: hyperbolic tangent, sine, Gaussian, and absolute value. Each neuron in the substrate is connected to all substrate neurons (including itself) and to all inputs. For a task with n outputs, the first n neurons of the substrate are considered as output neurons, while the other neurons are hidden neurons.

Results have been obtained for a configuration that only allows feedforward connections in the pattern generating network, and for another configuration that allows recurrent connections as well. The pattern generating network is run for 20 time steps. Running it for 10 or 30 time steps produces results that look very similar. Exactly the same mutation and selection parameters as for the basic NEATfields configuration are used. The substrate for pole balancing experiments consists of a single neuron, while for sequence recall tasks, it consists of 5 neurons (see Fig. 14). Similar results were obtained for pole balancing using 2 neurons, and for sequence recall using between 1 and 6 neurons.

The results are presented in Table 2. They show that NEATfields with all standard design patterns (i.e., a LC-F configuration) does not lose much of the performance of the underlying NEAT method and can solve all problems. In contrast, our simple HyperNEAT implementation does not do well on the more difficult variants of pole balancing and sequence recall.

Table 2
NEATfields and HyperNEAT performance (fraction of successful runs and mean number of evaluations for successful runs) on pole balancing and sequence recall tasks. The population size is 150. The meaning of the abbreviations for pole balancing varieties can be found in the main text.

	NEATfields reduced to NEAT	Full NEATfields (1)	HyperNEAT without recurrence	HyperNEAT with recurrence (2)	Are (1) and (2) indistinguishable?
SPV	1.0 294	1.0 258	1.0 180	1.0 165	$p < 0.01$
SPNV	1.0 5134	1.0 4519	1.0 16 626	0.95 17 061	$p \approx 0.7$
DPV	1.0 5920	1.0 6289	0.0 –	0.35 58 993	$p < 10^{-7}$
DPNV	1.0 7908	1.0 19 970	0.05 73 500	0.1 64 800	$p < 10^{-6}$
AWDPNV	1.0 28 571	1.0 23 957	0.0 –	0.0 –	$p < 10^{-10}$
1 bit	1.0 1994	1.0 1696	1.0 5850	1.0 11 250	$p \approx 0.4$
2 bit	1.0 27 396	1.0 27 377	0.85 397 412	0.65 259 462	$p < 10^{-5}$
3 bit	1.0 261 736	0.9 277 248	0.0 –	0.0 –	$p < 10^{-10}$

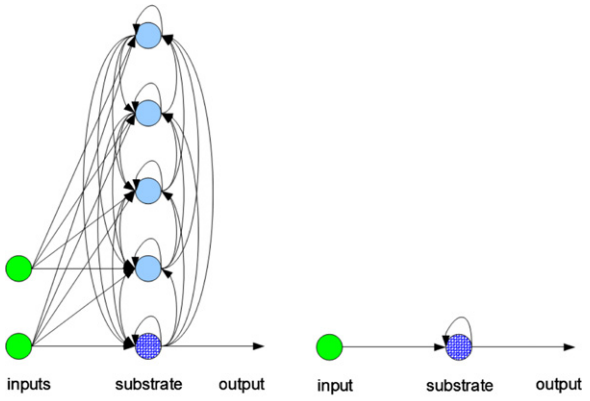

Fig. 14. (a) HyperNEAT substrate for sequence recall tasks. (b) Substrate for pole balancing tasks.

The results we got for HyperNEAT can be interpreted in a number of different ways. It could be argued that there exist configurations that enable HyperNEAT to perform well on these tasks, but we have not found them yet. Given the consistent decrease in performance with increasing difficulty of the tasks observed in our simulations, and the fact that the tested configurations work well when used with basic NEAT and NEATfields, we consider this possibility an unlikely one. A more plausible explanation is that it is difficult to fine tune connection weights using this indirect encoding. In principle, HyperNEAT could learn a good genetic architecture (i.e., the way in which phenotype variables are correlated) for these tasks, but for the small neural networks we consider here, this requires a lot of extra overhead. Besides, as mentioned before, learning a genetic architecture and a difficult task at the same time may cause some interference. The fact that HyperNEAT does not search through the neural network space by complexification may also explain the bad performance, especially on the sequence recall tasks.

It was shown in Section 4 that NEATfields can find solutions with a network topology that is smaller than a manually designed network topology. This can be an advantage of NEATfields over HyperNEAT because smaller networks require less resources. In addition, the experiments in this section show that compared to HyperNEAT, NEATfields may be more useful for tasks that are both high dimensional and require fine tuning of connectivity and weight within subspaces of the input and output. Recent research on extending HyperNEAT, on the other hand, could lead to versions of HyperNEAT that perform better on such tasks. For one, HyperNEAT has already been combined with lifetime learning rules (Risi & Stanley, 2010), and it could be argued, as already mentioned, that lifetime learning can do the fine tuning. Besides, an approach to evolve the HyperNEAT substrate topology as well as the weights using a single CPPN has been published recently (Risi, Lehman, & Stanley, 2010), although there is not yet much information available on how topology changes during the course of evolution using that approach, or whether tasks like those described above can be solved.

Another recent approach called HybrID (Clune, Stanley, Pennock, & Ofria, 2009) extends HyperNEAT by using both a direct and an indirect encoding. In the presented experiments, networks are evolved using HyperNEAT for some generations to discover regularities in the problem. Then the evolved network is transformed into a NEAT network, and further evolved with the weight-changing mutation operators of the NEAT method. So like NEATfields, HybrID can fine tune connection weights. But it does so in a different way. The NEATfields method respects higher level structures (fields and field elements) when tuning connection weights, whereas HybrID does not know anything about higher level regularities in the second phase of evolution. Another difference is that the NEATfields method can change topology and tune connection weights simultaneously, while the HybrID version presented by Clune, Stanley et al. (2009) does so in successive phases. However, as the authors mention, in principle there could be a HybrID version that evolves the direct and the indirect encoding simultaneously. Yet another difference is that the presented HybrID version does not evolve network topology. In principle, it could do so by using the above mentioned variant of HyperNEAT that does evolve the substrate topology. How well this works has not yet been explored to our knowledge.

In general, the ability to learn arbitrary genetic architectures makes HyperNEAT a very promising method, especially if the task becomes gradually more complex in long term evolution scenarios. NEATfields is more limited in this regard because its design patterns are externally specified (although their respective parameters can evolve). However, one can always extend the NEATfields method by providing other interesting design patterns inspired by neuroscience research. That way, one could perhaps make the study of limited domains within neuroevolution research more tractable than with methods that are very general and can learn almost anything, but may need too much time to do so.

Finally, two more recent methods also approach the problem of evolving large neural networks in interesting ways. One method is to evolve Fourier series coefficients and apply inverse Discrete Cosine Transformation on them to determine the entries of the connection weight matrix of a neural network (Koutník et al., 2010). This has the advantage that regular matrices can be generated using a few coefficients, but less regular matrices can also be generated when using more coefficients. However, the topology of the network remains fixed, and this approach does not exploit any regularity present in the input or output space. The method is used to evolve solutions to an octopus arm task, to an abstract ball throwing task, and to pole balancing, including a non-Markovian version of double pole balancing. A kind of coevolutionary algorithm is used to evolve the coefficients. The performance of that method on the presented tasks is very impressive.

The method that seems to be most similar to ours has been presented simultaneously with an initial publication on NEATfields (Inden et al., 2010). In that method (Mouret et al., 2010), a NEAT-like method is used to evolve a network. The nodes of this network can represent neural fields. The neurons of a field have identical properties. There are three types of connections in that method: one-to-one connections, one-to-all connections, and one-to-all connections where the connection weights decrease with distance between the positions of two neurons in their respective fields. With this simple "toolbox", networks can be built that are very similar to those of NEATfields. Whereas a field in NEATfields contains an array of small NEAT networks, it contains an array of just one neuron in the other method. Speaking in NEATfields terminology, all connections are global connections in that method. So the method of Mouret et al. (2010) has less mutation operators and less parameters than NEATfields, which is good. On the other hand, the way fields are set up in NEATfields leads to some kind of explicit modularity and allows to create connections within the modules with a probability different from the probability of creating global connections. NEATfields also allows for dehomogenization of neural fields. Another difference between the methods is that NEATfields uses lateral connections and local feature detectors for information flow between different positions in the field(s), while Mouret et al. (2010) use the two kinds of one-to-all connections. Furthermore, as it was presented there, the method of Mouret et al. (2010) (unlike NEATfields) uses one dimensional fields that cannot take advantage of the natural two dimensional topology that e.g. visual data often has. Finally, they use a more complex neuron model than NEATfields (currently) does. Of course, many of these different features could in principle be used with both methods. The method of Mouret et al. (2010) so far was used in an action selection task inspired by conventional neural network models of the basal ganglia.

6. Conclusions

The experiments described in this paper show that the NEATfields method is able to find solutions for tasks with large input and output spaces. It can both fine tune connection weights and search for a suitable network topology by complexification. As it can solve both pattern recognition tasks on a simple visual field and demanding control tasks, the next step is to evolve integrated perception–action systems for robotic tasks. This is one direction of future work on the NEATfields method. We have recently published experiments that use the NEATfields method to evolve controllers for multilegged locomotion and multisensory integration (Inden, Jin, Haschke, & Ritter, 2011a, 2011b). The results of these experiments, as well as the results reported here, make us confident that the NEATfields method is a good choice for performing evolutionary robotics experiments with real world (i.e., high dimensional) sensory input.

In comparing our method with some other indirect encoding methods, we have argued that the NEATfields method has some unique and useful properties among these methods. By providing evolution with useful design patterns, the NEATfields method makes it possible to arrive at solutions for some real-world like challenges in a reasonable amount of time. We have shown that all design patterns currently used in the default configuration of the method (fields, lateral connections, dehomogenization, and local feature detectors) are beneficial for solving at least some of the tasks presented here, which justifies their use in further experiments, where the demands on network structure and information flow may be unknown in advance and difficult to predict. A question for further research is what other design patterns may prove useful to neuroevolution. There is a rich source of inspiration here in the neuroscience literature. In particular, finding good ways of organizing lifetime learning in large neural networks by evolution is an important goal. Furthermore, besides issues of representation and operators, other aspects of artificial evolution merit more attention. For example, it seems plausible to us that the potential of some design patterns provided in the NEATfields method cannot be fully exploited for pattern recognition because there is no path towards using them that can be found by the evolutionary algorithm that is a part of the NEATfields method. Recent work indicates that rethinking the concept of selection in evolutionary algorithms may be essential to complement the progress that has been achieved on aspects of genetic representations and mutation operators for large networks (Lehman & Stanley, 2010; Mouret, 2009).

One aspect of the NEATfields method that may be disturbing is that, unlike most of the other methods mentioned in Section 5, it is not so much based on a single ingenious idea, but instead provides an ever growing abundance of design patterns. We have already commented on how to avoid a combinatorial explosion of parameter space in Section 1.3. It could be added here that (unlike physics) biology, and genetics in particular, is not governed by a few simple principles, but by an immense abundance of various patterns. Therefore it seems fully justified to explore such an approach on the way towards more powerful neuroevolution methods.

Acknowledgments

Benjamin Inden gratefully acknowledges the financial support from Honda Research Institute Europe for the project "Co-Evolution of Neural and Morphological Development for Grasping in Changing Environments". This work was started by Benjamin Inden while still at Max Planck Institute for Mathematics in the Sciences, Leipzig, Germany; support by Jürgen Jost is gratefully acknowledged. Pole balancing code was adapted from the NEAT implementation by Kenneth Stanley. This article also benefited from the comments of several anonymous reviewers.

References

Bear, M., Paradiso, M., & Connors, B. W. (2006). *Neuroscience: exploring the brain* (3rd ed.). Lippincott Williams & Wilkins.

Clune, J., Beckmann, B. E., McKinley, P. K., & Ofria, C. (2010). Investigating whether HyperNEAT produces modular neural networks. In *Proceedings of the genetic and evolutionary computation conference*.

Clune, J., Beckmann, B. E., Ofria, C., & Pennock, R. T. (2009). Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Proceedings of the IEEE congress on evolutionary computing*.

Clune, J., Misevic, D., Ofria, C., Lenski, R. E., Elena, S. F., & Sanjuán, R. (2008). Natural selection fails to optimize mutation rates for long-term adaptation on rugged fitness landscapes. *PLoS Computational Biology*, *4*, e1000187.

Clune, J., Ofria, C., & Pennock, R. T. (2008). How a generative encoding fares as problem-regularity decreases. In *Proceedings of the international conference on parallel problem solving from nature*.

Clune, J., Ofria, C., & Pennock, R. T. (2009). The sensitivity of HyperNEAT to different geometric representations of a problem. In *Proceedings of the genetic and evolutionary computation conference*.

Clune, J., Stanley, K. O., Pennock, R. T., & Ofria, C. (2009). HybrID: a hybridization of indirect and direct encoding for evolutionary computation. In *Proceedings of the European conference on artificial life*.

D'Ambrosio, D., & Stanley, K. (2007). A novel generative encoding for neural network sensor and output geometry. In *Proceedings of the genetic and evolutionary computation conference*.

D'Ambrosio, D., & Stanley, K. O. (2008). Generative encoding for multiagent learning. In *Proceedings of the genetic and evolutionary computation conference*.

Drchal, J., Koutník, J., & Snorek, M. (2009). HyperNEAT controlled robots learn how to drive on roads in simulated environment. In *Proceedings of the 2009 IEEE congress on evolutionary computation*.

Du, K.-L., & Swamy, M. N. S. (2006). *Neural networks in a softcomputing framework*. Springer-Verlag.

Floreano, D., Dürr, P., & Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, *1*, 47–62.

Futuyma, D. J. (2005). *Evolution*. Sinauer Associates.

Gauci, J., & Stanley, K. (2007). Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the genetic and evolutionary computation conference*.

Green, C. (2006). SharpNEAT. URL: http://sharpneat.sourceforge.net.

Gruau, F. (1994). Neural network synthesis using cellular encoding and the genetic algorithm. Ph.D. Thesis.

Gruau, F., & Whitley, D. (1993). Adding learning to the cellular development of neural networks: evolution and the Baldwin effect. *Evolutionary Computation*, *1*, 213–233.

Gruau, F., Whitley, D., & 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*.

Hansen, T. (2006). The evolution of genetic architecture. *Annual Review of Ecology, Evolution, and Systematics*, *37*, 123–157.

Harding, S., & Banzhaf, W. (2008). *Organic computing*. Springer-Verlag (Chapter). Artificial development.

Inden, B. (2008). Neuroevolution and complexifying genetic architectures for memory and control tasks. *Theory in Biosciences*, *127*, 187–194.

Inden, B., Jin, Y., Haschke, R., & Ritter, H. (2010). Neatfields: evolution of neural fields. In *Proceedings of the conference on genetic and evolutionary computation*.

Inden, B., Jin, Y., Haschke, R., & Ritter, H. (2011a). Evolution of multisensory integration in large neural fields. In *Tenth international conference on artificial evolution*.

Inden, B., Jin, Y., Haschke, R., & Ritter, H. (2011b). Exploiting inherent regularity in control of multilegged robot locomotion by evolving neural fields. In *Third world congres on nature and biologically inspired computation*.

Kashtan, N., & Alon, U. (2005). Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences*, *102*, 13773–13778.

Koutník, J., Gomez, F., & Schmidhuber, J. (2010). Evolving neural networks in compressed weight space. In *Proceedings of the genetic and evolutionary computation conference*.

LeCun, Y. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 2278.

Lehman, J., & Stanley, K. O. (2010). Revising the evolutionary computation abstraction: minimal criteria novelty search. In *Proceedings of the genetic and evolutionary computation conference*.

Lipson, H. (2004). Principles of modularity, regularity, and hierarchy for scalable systems. In *Genetic and evolutionary computation conference 2004, workshop on modularity, regularity and hierarchy*.

Mountcastle, V. B. (1997). The columnar organization of the neocortex. *Brain*, *120*, 701–722.

Mouret, J.-B. (2009). Novelty-based multiobjectivization. In *Proceedings of the workshop on exploring new horizons in evolutionary design of robots, 2009 IEEE/RSJ international conference on intelligent robots and systems*.

Mouret, J.-B., Doncieux, S., & Girard, B. (2010). Importing the computational neuroscience toolbox into neuro-evolution—application to basal ganglia. In *Proceedings of the genetic and evolutionary computation conference*.

Ohkura, K., Yasuda, T., Kawamatsu, Y., Matsumura, Y., & Ueda, K. (2007). MBEANN: mutation-based evolving artificial neural networks. In *Proceedings of the European conference on artificial life*.

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). A field guide to genetic programming. Published via. http://lulu.com. URL: http://www.gp-field-guide.org.uk.

Reisinger, J., & Miikkulainen, R. (2007). Acquiring evolvability through adaptive representations. In *Proceedings of the genetic and evolutionary computation conference*.

Reisinger, J., Stanley, K., & Miikkulainen, R. (2004). Evolving reusable neural modules. In *Proceedings of the genetic and evolutionary computation conference*.

Risi, S., Lehman, J., & Stanley, K. O. (2010). Evolving the placement and density of neurons in the HyperNEAT substrate. In *Proceedings of the genetic and evolutionary computation conference*.

Risi, S., & Stanley, K. O. (2010). Indirectly encoding neural plasticity as a pattern of local rules. In *Proceedings of the 11th international conference on simulation of adaptive behavior*.

Sendhoff, B., & Kreutz, M. (1999). A model for the dynamic interaction between evolution and learning. *Neural Processing Letters*, *10*, 181–193.

Soskine, M., & Tawfik, D. S. (2010). Mutational effects and the evolution of new protein functions. *Nature Reviews Genetics*, *11*, 572–582.

Stanley, K. (2004). Efficient evolution of neural networks through complexification. Ph.D. Thesis, report AI-TR-04-314. University of Texas at Austin.

Stanley, K. (2007). Compositional pattern producing networks: a novel abstraction of development. *Genetic programming and evolvable machines*, 131–162.

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

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

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

Toussaint, M. (2003). The evolution of genetic representations and modular adaptations. Ph.D. Thesis. Ruhr-Universität Bochum.

Verbancsis, P., & Stanley, K. O. (2010). Transfer learning through indirect encoding. In *Proceedings of the genetic and evolutionary computation conference*.

Wieland, A.P. (1991). Evolving controls for unstable systems. In D. Touretzky (Ed.), *Connectionist models: proceedings of the 1990 summer school*.

Woolley, B. G., & Stanley, K. O. (2010). Evolving a single scalable controller for an octopus arm. In *Proceedings of the 11th international conference on parallel problem solving from nature*.

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

Zhang, J. (2003). Evolution by gene duplication: an update. *Trends in Ecology and Evolution*, *16*, 292–298.