# HyperNTM: Evolving Scalable Neural Turing Machines through HyperNEAT

Jakob Merrild, Mikkel Angaju Rasmussen, and Sebastian Risi

IT University of Copenhagen, Denmark
{jmer, mang, sebr}@itu.dk

**Abstract.** Recent developments in memory-augmented neural networks allowed sequential problems requiring long-term memory to be solved, which were intractable for traditional neural networks. However, current approaches still struggle to scale to large memory sizes and sequence lengths. In this paper we show how access to an external memory component can be encoded geometrically through a novel HyperNEAT-based Neural Turing Machine (*HyperNTM*). The indirect HyperNEAT encoding allows for training on small memory vectors in a bit vector copy task and then applying the knowledge gained from such training to speed up training on larger size memory vectors. Additionally, we demonstrate that in some instances, networks trained to copy nine bit vectors can be scaled to sizes of 1,000 *without further training*. While the task in this paper is simple, the HyperNTM approach could now allow memory-augmented neural networks to scale to problems requiring large memory vectors and sequence lengths.

**Keywords:** Neural turing machine; HyperNEAT; neuroevolution; indirect encoding

## 1 Introduction

Memory-augmented neural networks are a recent improvement on artificial neural networks (ANNs) that allows them to solve complex sequential tasks requiring long-term memory [1,2,3]. Here we are particularly interested in Neural Turing Machines (NTM) [3], which augment networks with an external memory tape to store and retrieve information from during execution. This improvement also enabled ANNs to learn simple algorithms such as copying, sorting and planning [2].

However, scaling to large memory sizes and sequence length is still challenging. Additionally, current algorithms have difficulties *extrapolating* information learned on smaller problem sizes to larger once, thereby bootstrapping from it. For example, in the copy task introduced by Graves et al. [3] the goal is to store and later recall a sequence of bit vectors of a specific size. It would be desirable that a network trained on a certain bit vector size (e.g. eight bits) would be able to scale to larger bit vector sizes without further training. However, current machine learning approaches often cannot transfer such knowledge.

Recently, Greve et al. [4] introduced an *evolvable* version of the NTM (ENTM), which allowed networks to be trained through neuroevolution instead of the gradient descent-based training of the original NTM. This combination offered some unique

advantages. First, in addition to the network's weights, the optimal neural architecture can be learned at the same time. Second, a hard memory attention mechanism is directly supported and the complete memory does not need to be accessed each time step. Third, a growing and theoretically infinite memory is now possible. Additionally, in contrast to the original NTM, the evolved networks were able to perfectly scale to very long sequence lengths. However, because it employed a direct genetic encoding (NEAT [5]), which means that every parameter of the network is described separately in its genotype, the approach had problems scaling to copy tasks with vectors of more than eight bits.

To overcome this challenge, in this paper we present an indirectly encoded version of the ENTM based on the Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) method [6]. In HyperNEAT the weights of a neural network are generated as a function of its geometry through an indirect encoding called compositional pattern producing networks (CPPNs), which can compactly encode patterns with regularities such as symmetry, repetition, and repetition with variation [7]. Neurons are placed at certain locations in space, allowing evolution to exploit topography (as opposed to just topology) and correlating the geometry of sensors with the geometry of the brain. This geometry appears to be a critical facet of natural brains [8] but lacks in most ANNs. HyperNEAT allowed large ANNs with regularities in connectivity to evolve for high-dimensional problems [6,9].

In the approach introduced in this paper, called *HyperNTM*, an evolved neural network generates the weights of a main model, *including how it connects to the external memory component.* We show that because HyperNEAT can learn the geometry of how the ANN should be connected to the external memory, it is possible to train a CPPN on a small bit vector size and then scale it to larger bit vector sizes *without further training*.

While the task in this paper is simple it shows – for the first time – that access to an external memory can be indirectly encoded, an insight that could now also directly benefit indirectly encoded HyperNetworks trained through gradient descent [10]. Additionally, an exciting future opportunity is to combine our HyperNTM approach with recent advances in evolutionary strategies [11] and genetic programming [12], to solve complex problems requiring large external memory components.

## 2 Background

This section reviews NEAT, HyperNEAT, and Evolvable Neural Turing Machines, which are foundational to the approach introduced in this paper.

### 2.1 Neuroevolution of Augmenting Topologies (NEAT)

The HyperNEAT method that enables learning from geometry in this paper is an extension of the NEAT algorithm that evolves ANNs through a *direct* encoding [5,13]. It starts with a population of simple neural networks and then *complexifies* them over generations by adding new nodes and connections through mutation. By evolving networks in this way, the topology of the network does not need to be known a priori; NEAT searches through increasingly complex networks to find a suitable level of complexity.

The important feature of NEAT for the purpose of this paper is that it evolves *both* the network's topology and weights. Because it starts simply and gradually adds complexity, it tends to find a solution network close to the minimal necessary size. The next section reviews the HyperNEAT extension to NEAT that is itself extended in this paper.

### 2.2 HyperNEAT

In direct encodings like NEAT, each part of the solution's representation maps to a single piece of structure in the final solution [14]. The significant disadvantage of this approach is that even when different parts of the solution are similar, they must be encoded and therefore discovered separately. Thus this paper employs an *indirect* encoding instead, which means that the description of the solution is compressed such that information can be reused. Indirect encodings are powerful because they allow solutions to be represented as a *pattern* of parameters, rather than requiring each parameter to be represented individually [15,16,17,18]. HyperNEAT, reviewed in this section, is an indirect encoding extension of NEAT that is proven in a number of challenging domains that require discovering regularities [19,16,6]. For a full description of HyperNEAT see Gauci and Stanley [16].

In HyperNEAT, NEAT is altered to evolve an indirect encoding called *compositional pattern producing networks* (CPPNs [7]) *instead* of ANNs. CPPNs, which are also networks, are designed to encode *compositions of functions*, wherein each function in the composition loosely corresponds to a useful regularity.

The appeal of this encoding is that it allows spatial patterns to be represented as networks of simple functions (i.e. CPPNs), which means that NEAT can evolve CPPNs just like ANNs. CPPNs are similar to ANNs, but they rely on more than one activation function (each representing a common regularity) and are an abstraction of biological development rather than of brains. The indirect CPPN encoding can compactly encode patterns with regularities such as symmetry, repetition, and repetition with variation [7]. For example, simply by including a Gaussian function, which is symmetric, the output pattern can become symmetric. A periodic function such as sine creates segmentation through repetition. Most importantly, *repetition with variation* (e.g. such as the fingers of the human hand) is easily discovered by combining regular coordinate frames (e.g. sine and Gaussian) with irregular ones (e.g. the asymmetric x-axis). The potential for CPPNs to represent patterns with motifs reminiscent of patterns in natural organisms has been demonstrated in several studies [20,7,21,22].

The main idea in HyperNEAT is that CPPNs can naturally encode *connectivity patterns* [16,6]. That way, NEAT can evolve CPPNs that represent large-scale ANNs with their own symmetries and regularities. Formally, CPPNs are *functions* of geometry (i.e. locations in space) that output connectivity patterns whose nodes are situated in $n$ dimensions, where $n$ is the number of dimensions in a Cartesian space. Consider a CPPN that takes four inputs labeled $x_1, y_1, x_2$, and $y_2$; this point in four-dimensional space *also* denotes the connection between the two-dimensional points $(x_1, y_1)$ and $(x_2, y_2)$, and the output of the CPPN for that input thereby represents the weight of that connection (Figure 1). By querying every possible connection among a pre-chosen set of points in this manner, a CPPN can produce an ANN, wherein each queried point is a

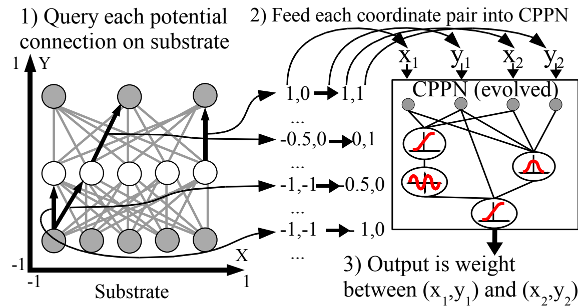



Fig. 1: Hypercube-based Geometric Connectivity Pattern Interpretation. A collection of nodes, called the *substrate*, is assigned coordinates that range from $-1$ to $1$ in all dimensions. (1) Every potential connection in the substrate is queried to determine its presence and weight; the dark directed lines in the substrate depicted in the figure represent a sample of connections that are queried. (2) Internally, the CPPN (which is evolved) is a graph that determines which activation functions are connected. As in an ANN, the connections are weighted such that the output of a function is multiplied by the weight of its outgoing connection. For each query, the CPPN takes as input the positions of the two endpoints and (3) outputs the weight of the connection between them. Thus, CPPNs can produce regular patterns of connections in space.

neuron position. Because the connections are produced by a function of their endpoints, the final structure is produced with *knowledge* of its geometry.

In the original HyperNEAT, the experimenter defines both the location and role (i.e. hidden, input, or output) of each such node (more advanced HyperNEAT variations can infer the position of hidden nodes from the weight pattern generated by the CPPN [23,21]). As a rule of thumb, nodes are placed on the substrate to reflect the geometry of the task [19,6]. That way, the connectivity of the substrate is a function of the task structure. How to integrate this setup with an ANN that has an external memory component is an open question, which this paper tries to address.

### 2.3 Neural Turing Machines

The recently introduced Neural Turing Machine (NTM) is a neural network coupled with an external memory component [2,3]. The neural network controller determines what is written to and read from the memory tape. At each time step, the ANN emits a number of different signals, including a data vector and various control inputs. These signals allow the NTM to focus its read and write heads on different parts of the external memory. The write heads modify the tapes content and the information from the read heads is used as input to the ANN during the next time step. At the same time, the ANN receive input from its environment and can interact with it through its outputs.

While the original NTM was trained through gradient descent, the recently introduced evolvable variant (ENTM) uses NEAT to learn the topology and weights of the ANN controller [4] (Figure 2). That way the topology of the network does not have to be defined a priori (as is the case in the original NTM setup) and the network can

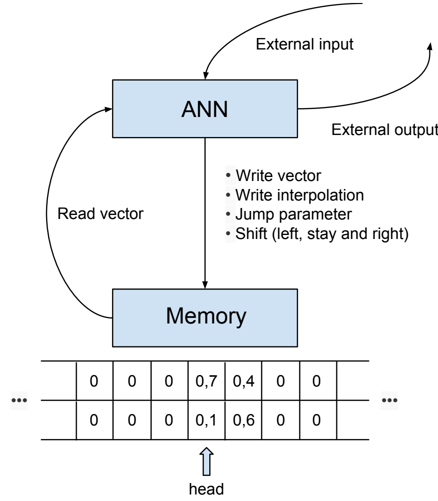



Fig. 2: Evolvable Neural Turing Machines. Shown is the activation flow between the ANN and its external memory. Extra ANN outputs determine the vector to be written to memory and the movement of the read and write heads. The ANN receives the content of the current memory location as input at the beginning of the next time-step. In addition to the NTM specific inputs and outputs, the ANN also has domain dependent inputs and outputs.

grow in response to the complexity of the task. As demonstrated by Greve et al., the ENTM often finds compact network topologies to solve a particular task, thereby avoiding searching through unnecessarily high-dimensional spaces. Additionally, the ENTM was able to solve a complex continual learning problem [24]. Because the network does not have to be differentiable, it can use hard attention and shift mechanisms, allowing it to generalize perfectly to longer sequences in a copy task. Additionally, a dynamic, theoretically unlimited tape size is now possible.

The ENTM has a single combined read/write head. The network emits a write vector $w$ of size $M$, a write interpolation control input $i$, a content jump control input $j$, and three shift control inputs $s_l$, $s_0$, and $s_r$ (left shift, no shift, right shift). The size of the write vector determines the size of each memory location on the tape. The write interpolation component allows blending between the write vector and the current tape values at the write position, where $M_h(t)$ is the content of the tape at the current head location $h$, at time step $t$, $i_t$ is the write interpolation, and $w_t$ is the write vector at time step $t$: $M_h(t) = M_h(t-1) \cdot (1 - i_t) + w_t \cdot i_t$.

The content jump determines if the head should be moved to the location in memory that most closely resembles the write vector. A content jump is performed if the value of the control input exceeds $0.5$. The similarity between write vector $w$ and memory vector $m$ is determined by:

$$s(w, m) = \frac{\sum_{i=1}^{M} |w_i - m_i|}{M}.$$

At each time step $t$, the following actions are performed in order: (1) Record the write vector $w_t$ to the current head position $h$, interpolated with the existing content according to the write interpolation $i_t$. (2) If the content jump control input $j_t$ is greater than $0.5$, move the head to location on the tape most similar to the write vector $w_t$. (3) Shift the head one position left or right on the tape, or stay at the current location, according to the shift control inputs $s_l$, $s_0$, and $s_r$. (4) Read and return the memory values at the new head position to the ANN controller.

## 3 Approach: Hyper Neural Turing Machine (HyperNTM)

In the novel HyperNTM approach introduced in this paper, the CPPN does not only determine the connections between the task related ANN inputs and outputs but also how the information coming from the memory is integrated into the network and how information is written back to memory. Because HyperNEAT can learn the geometry of a task it should be able to learn the geometric pattern of the weights connecting the external memory component to the neural network.

In this paper the HyperNTM approach is applied to the copy task, which was first introduced by Graves et al. [3]. In the copy task the network is asked to store and later recall a sequence of random bit vectors. At the start of the task the network receives a special input, which denotes the start of the input phase. Afterwards, the network receives the sequence of bit vectors, one at a time. Once the full sequence has been presented to the network, it receives another special input, signaling the end of the input phase and the start of the output phase. For any subsequent time steps the network does not receive any external input.

In summary, the network has the following inputs. *Start:* An input that is activated when the storing of bit vectors should begin. *Switch:* An input that is activated when the recitation of the stored sequence should start. *Bit vector input:* The bit vector that should be stored in memory. *Memory read input:* The memory vector at the current position in memory. The network has the following outputs. *Bit vector output:* The bit vector that the network outputs to the environment. During the input phase this output is ignored. *Memory write output:* The memory vector that should be written to memory. *TM controls:* TM specific control outputs: Jump, interpolation, and three shift controls (left, stay, and right).

### 3.1 Copy Task Substrate

The HyperNEAT substrate for the copy task is shown in Figure 3. The substrate is designed such that the bit vector input nodes share $x$-coordinates with the memory vector write nodes and vice versa with memory vector read nodes and bit vector output nodes. Furthermore, the switch input shares its $x$-coordinate with the jump output, thus encouraging the network to jump in memory when it should start reciting. In this paper, the size of the memory vector equals the bit vector size. Furthermore, none of the substrates contain hidden nodes as it has been shown previously that it is possible to solve the copy task without any hidden nodes [25].

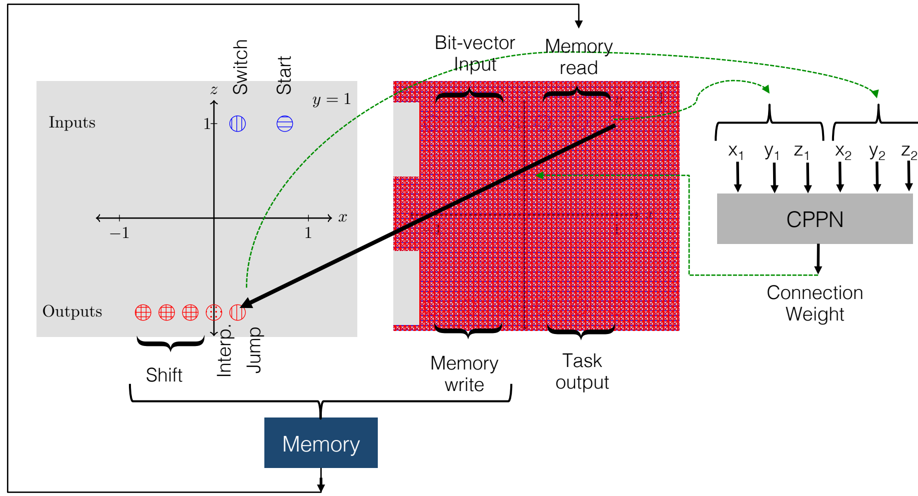



Fig. 3: In the HyperNTM approach the CPPN does not only determine the connectivity between the task related inputs and outputs but also how information is written/read from memory. This figure depicts the HyperNTM substrate for the copy task. All inputs are in $z = 1$ and all outputs in $z = -1$. The plane on the left shows all nodes with $y = 1$, which are the start/switch inputs and the TM controls. Notably the $x$-coordinate is the same for the switch input and the jump control output. The plane on the right shows the nodes in $y = -1$, which are the bit vector and memory vector input and outputs. Bit vector input nodes share $x$-coordinates with memory vector write nodes, while memory vector read nodes share $x$-coordinates with bit vector output nodes. The potential connections between all pairs of inputs and outputs are queried for by the CPPN. The dark line shows an example of such a potential connection that connects an input in the $y = -1$ plane with an output in the $y = 1$ plane.

Following Verbancsics and Stanley [26], in addition to the CPPN output that determines the weights of each connection, each CPPN has an additional step-function output, called the *link-expression output* (LEO), which determines if a connection should be expressed. Potential connections are queried for each input on layers $y = 1$ and $y = -1$ to each output on layers $y = 1$ and $y = -1$. The CPPN has an additional output that determines the bias values for each node in the substrate. These values are determined through node-centric CPPN queries (i.e. both source and target neuron positions $xyz$ are set to the location of the node whose bias should be determined).

### 3.2 Scaling

A particularly intriguing property of HyperNEAT is the fact that, because it is able to capture the particular domain geometry, the number of inputs and outputs in the substrate can theoretically be scaled without further training [27,28,29].

In this paper the substrate reflects the domain geometry of the copy tasks (Figure 3), which means the number of inputs and outputs on the $y = -1$ layer can be scaled

dependent on the size of the copy task bit vector. Networks that can store larger bit vectors can be generated without further evolution by requerying the same CPPN at higher-resolutions (Figure 8). When rescaled, neurons are uniformly distributed in the $x$ interval $[-1.0, -0.2]$ for bit vector inputs and memory write vector and in the interval $[0.2, 1.0]$ for the memory read vector and bit vector output.

# 4 Experiments

A total of three different approaches are evaluated on bit-sizes of 1, 3, 5, and 9. **HyperNTM**, a NTM based on HyperNEAT, is compared to one evolved by **NEAT**, and a **Seeded HyperNTM** treatment that starts evolution with a manually designed CPPN seed that encourages locality on both the $x$- and $y$-coordinates (Figure 7a). A similar locality seed has been shown useful in HyperNEAT to encourage the evolution of modular networks [26]. This locality seed is then later adjusted by evolution (e.g. by adding/removing nodes and connections and changing their weights).

The fitness function follows the one in the original ENTM paper [4]. During training the network is given a sequence of random bit vectors with sequence lengths between 1 and 10, which it then has to recite. The network is tested on a total of 50 random sequences. Fitness is determined by comparing the bit vectors recited by the network to those given to it during the input phase; for every bit vector the network is given a score based on how close the output from the network corresponds to the target vector. If the two bit vectors have a match $m$ of at least $25\%$, fitness is increased by: $f = \frac{|m-0.25|}{0.75}$, otherwise the network is not awarded for that specific bit vector. The fitness for a complete sequence is the sum of the fitness values for each bit vector normalized to the length of the sequence. Thus final fitness scores are in the interval [0, 1] and reward the network for gradually getting closer to the solution, but do not actively reward the network for using the memory to store the presented sequence.

## 4.1 Experimental Parameters

For the NEAT experiments, offspring proportions are $50\%$ sexual (crossover) and $50\%$ asexual (mutation). We use $98.8\%$ synapse weight mutation probability, $9\%$ synapse addition probability, and $5\%$ synapse removal probability. Because previous results [4] have demonstrated that the task can be solved without any hidden nodes, node addition probability is set to a relatively low value of $0.05\%$.

The code is build on a modified version of SharpNEAT v2.2.0.0[1], which is an implementation of NEAT and HyperNEAT made in C# by Colin Green. Our code is available from: `https://github.com/kalanzai/ENTM_CSharpPort`. This NEAT implementation uses a complexity regulation strategy for the evolutionary process, which has proven to be quite impactful on our results. A threshold defines how complex the networks in the population can be (here defined as the number of genes in the genome and set to 10 in our experiments), before the algorithm switches to a simplifying phase, where it gradually reduces complexity.

[1] http://sharpneat.sourceforge.net/

For the HyperNTM experiments the following parameters are used. Elitism proportion is 2%. Offspring generation proportions are 50% sexual (crossover) and 50% asexual (mutation). CPPN connection weights have a 98.8% probability of being changed, a 1% change of connection addition, and 0.1% change of node addition and node deletion. The activation functions available to new neurons in the CPPN are Linear, Gaussian, Sigmoid, and Sine, each with a 25% probability of being added.

Parameters for both NEAT and HyperNEAT have been tuned through prior experimentation. Both methods run with a population size of 500 for a maximum of 10,000 generations or until a solution is found.

## 5 Results

Figure 4a shows the mean champion fitness over $10,000$ generations for each of the different approaches and bit vector sizes. While NEAT performs best on smaller bit vectors, as the size of the vector grows to 9 bits, the seeded HyperNTM variant outperforms both NEAT and non-seeded HyperNTM. The numbers of solutions found (i.e. networks that reach a training score $\geq 0.999$) for the different bit vector sizes are shown in Figure 4b. For bit size 1 all approaches solve the problem equally well. However, as the size of the bit vector is increased the HyperNTM configuration with the locality seed performs best and is the only method able to find any solutions for size 9.

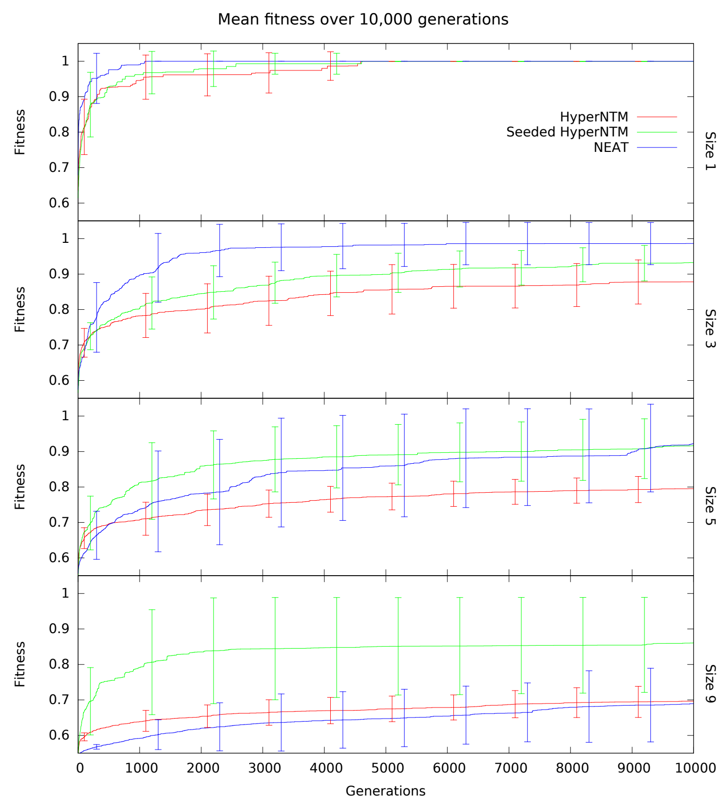
**Testing Performance.** To determine how well the champions from the last generation generalize, they were tested on 100 random bit vector sequences with sequence lengths varying randomly from 1 to 10 (Figure 5). With 1 and 5 bit vectors there is no statistical difference between either treatment (following a two-tailed Mann-Whitney U test). On size 3, NEAT performs significantly better than the seeded HyperNTM ($p < .0001$). Finally, seeded HyperNTM performs significantly better than NEAT on size 9 ($p < .00001$). The main conclusions are that (1) while NEAT performs best on smaller bit vectors it degrades rapidly with increased bit sizes, and (2) the seeded HyperNTM variant is able to scale to larger sizes while maintaining performance better.

**Generalizing to longer sequences.** We also tested how many of the solutions, which were trained on sequences of up to length 10, generalize to sequences of length 100. The training and generalization results are summarized in Table 1, which shows the number of solutions for each of the three approaches, the average number of generations it took to find a solution, and how many of those solutions generalized to sequences of length 100. For all three methods, most solutions generalize perfectly to sequences that are longer than the sequences encountered during training.

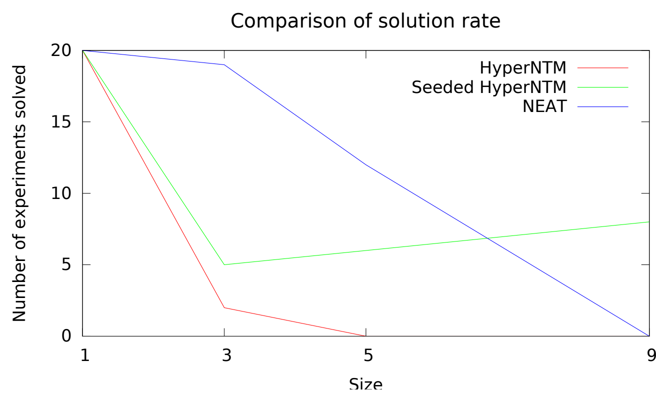
### 5.1 Transfer Learning

To test the scalability of the Seeded HyperNTM solutions, champion genomes from runs which found a solution for a given size were used as a seed for evolutionary runs of higher sizes. The specific runs and which seeds were used can be seen in Table 2.

Because the number of solutions found varied between the different sizes (see Table 1), the scaling experiments were not run exactly 20 times. Instead, the number of




(a)




(b)

Fig. 4: (a) Shown are mean champion fitness for the different treatments and bit sizes, averaged over 20 independent evolutionary runs. Error bars show one standard deviation. The number of solutions found by each configuration for the four different bit vector sizes are shown in (b).

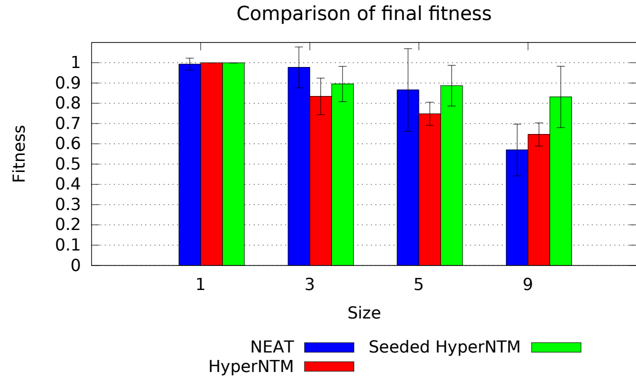



Fig. 5: The mean testing performance of the champion networks from the last generation. In contrast to NEAT, the seeded HyperNTM approach is able to better maintain performance with an increase in bit vector size. Error bars show one standard deviation.

runs was the smallest number above or equal to 20 which allowed for each champion to be seeded an equal number of times, e.g. if there were 6 solutions 24 runs were made; 4 runs with the champion from each solution. Figure 6 shows a comparison of HyperNTM seeded with the locality seed and seeded with champion genomes of smaller sizes on the size 9 problem. HyperNTM yielded significantly better results when seeded with size 5 and 3→5 champions compared to starting with the locality seed ($p < .001$), but not when seeded with the champion from size 3 ($p > 0.05$).

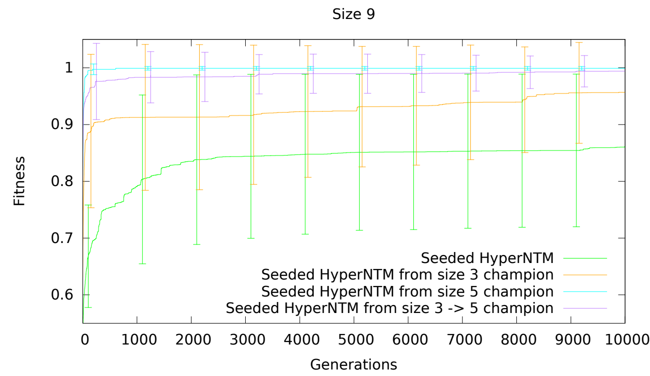



Fig. 6: Comparison of training performance on bit size 9 with the locality seed and with champion seeds from smaller sizes.

Table 1: Training and Generalization. Shown are the number of solutions ($\#sol$) that were able to solve sequences of up to length 10 during training, together with the average number of generations it took to find those solutions ($gens$) and standard deviation ($sd$). The number of those solutions that generalize ($\#gen$) to sequences of length 100 during testing are also shown.

| | Size | #sol. | #gen | gens. | sd. |
|---|---|---|---|---|---|
| Seeded HyperNTM | 1 | 20 | 20 | 1055.8 | 1147.4 |
| | 3 | 5 | 5 | 4454.8 | 3395.8 |
| | 5 | 6 | 5 | 2695.5 | 2666.5 |
| | 9 | 8 | 5 | 1523.25 | 2004.1 |
| HyperNTM | 1 | 20 | 20 | 1481.45 | 1670.8 |
| | 3 | 2 | 2 | 4395.5 | 388.2 |
| | 5 | 0 | 0 | N/A | N/A |
| | 9 | 0 | 0 | N/A | N/A |
| NEAT | 1 | 20 | 19 | 281 | 336.3 |
| | 3 | 19 | 19 | 2140.5 | 1594.4 |
| | 5 | 12 | 11 | 3213.4 | 2254.2 |
| | 9 | 0 | 0 | N/A | N/A |

### 5.2 Scaling without further training

The champions from runs which found a solution were tested for scaling to larger bit vector sizes without further evolutionary training (i.e. new input and output nodes are created and queried by the CPPN but no evolutionary optimization is performed; see copy task substrate Section 3.1). Each genome was tested on 50 sequences of 100 random bit vectors of size 1,000. Some of the champions found using only the LEO size 9 configuration scaled perfectly to a bit-size of 1,000 without further training, as seen in Table 3. The main results is that it is possible to find CPPNs that perfectly scale to any size. The fact that evolution with HyperNEAT performs significantly better when seeded with a champion genome which solved a smaller size of the problem, together

Table 2: Transfer Learning. Seeds $X \rightarrow Y$ refer to champions from a run of size $Y$ which were seeded with a champion from a run of size $X$. Networks evolved under these treatments were then tested on larger bit vector sizes without further training.

| Seed | Size | #sol | #general | gens. | sd. |
|---|---|---|---|---|---|
| 3 | 5 | 12/20 | 6 | 434.1 | 574.6 |
| 3 | 9 | 16/20 | 2 | 1150.3 | 2935.9 |
| 5 | 9 | 23/24 | 11 | 58.9 | 129.2 |
| 3→5 | 9 | 23/24 | 8 | 588 | 1992 |
| 5 | 17 | 18/20 | 12 | 258.9 | 301.3 |
| 9 | 17 | 24/24 | 12 | 89.3 | 235.9 |
| 5→9 | 17 | 20/20 | 17 | 36.25 | 47.5 |
| 9 | 33 | 23/24 | 17 | 94 | 217 |
| 9→17 | 33 | 24/24 | 19 | 456.5 | 2002.4 |

Table 3: Scaling using LEO without further evolution.

| Size | # of champions | # which scaled to 1000 |
|---|---|---|
| 9 | 8 | 2 |
| 9 → 17 | 24 | 7† |

† 6 of these can be traced back to the 2 champions from size 9 which scaled perfectly.

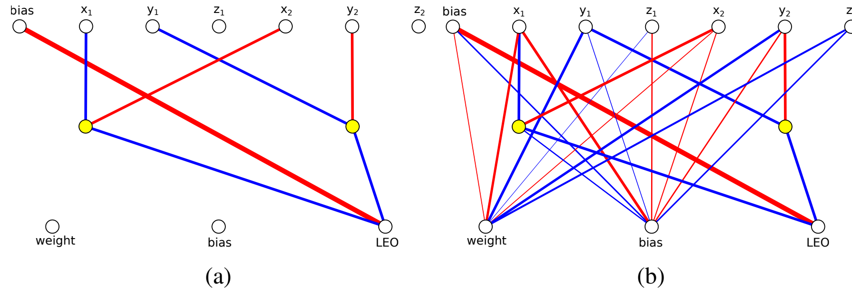



Fig. 7: (a) The manually designed CPPN seed that is used to promote locality on the $x$ and $y$ axes. (b) A champion trained on the size 9 problem, which was able to scale without further evolution to size 1,000. Blue connections have a positive weight, while red connections have a negative weight.

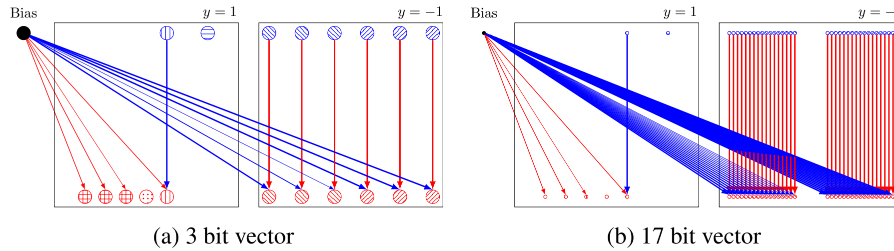
with the fact that evolution sometimes finds solutions which scale without further training, demonstrates that the HyperNTM approach can be used to scale the dimensionality of the bit vector in the copy task domain.

### 5.3 Solution Example

Here we take a closer look at one of the champion genomes (trained on bit vector size 9), which was able to scale perfectly to the size 1,000 problem (Table 3). Figure 7 shows a visualization of the champion genome, as well as the locality seed from which it was evolved. The champion genome does resemble the seed but also evolved several additional connections that are necessary to solve the problem.

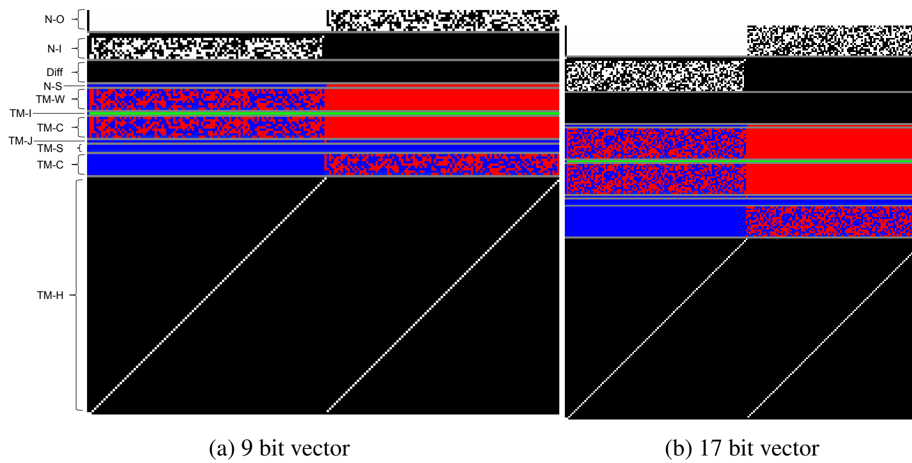
Figure 8 shows two ANNs for different bit vector sizes generated by the same CPPN, which is shown in Figure 7b. It can be seen that for non-bias connections to be expressed, the source and destination nodes have to be located in the same position on both the $x$ position and $y$ layer in the substrate. These results suggest that the locality encouraging seed works as intended.

To further demonstrate the scalability of this evolved CPPN, memory usage for bit vector sizes of 9 and 17 are shown in Figure 9. Both generated NTMs solve the task perfectly, continually performing a left shift while writing the given input to memory. When reaching the delimiter input that signals the start of the recall phase, the networks perform a content-jump to the original position and continue shifting left while reading from the memory, reciting the correct sequence through the network's outputs.




(a) 3 bit vector (b) 17 bit vector

Fig. 8: Networks produced by the champion CPPN (Figure 7b) for bit sizes 3 (a) and 17 (b. The CPPN discovered a connectivity pattern that only expresses non-bias connections in which the source and destination nodes are located in the same position on both the $x$ position and $y$ layer in the substrate.




(a) 9 bit vector (b) 17 bit vector

Fig. 9: Recordings of the activities for bit sizes 9 (a) and 17 (b) networks. Both networks are produced by the same CPPN. The different rows show the activity of the network and the state of the memory over time. After the input sequence is given to the network it has to recite the sequence of random binary vectors. In more detail, row **N-O** shows the output produced by the ANN. **N-I** is the bit vector input to the network. **Dif** is the difference in produced and expected output (i.e. how well the network recited the sequence given to it); here the section is black since the network reproduces the sequences perfectly. **TM-W** is the write vector, and **TM-I** write interpolation. **TM-C** shows the content of the tape at the current head position after write. **TM-J** is the content jump input, **TM-S** the three shift values, and **TM-R** the read vector. **TM-H** shows the current head position that the TM is focused on when writing (left) and reading (right).

# 6 Conclusion

This paper showed that the indirect encoding HyperNEAT makes it feasible to train ENTMs with large memory vectors for a simple copy task, which would otherwise be infeasible to train with an direct encoding such as NEAT. Furthermore, starting with a CPPN seed that encourages locality, it was possible to train solutions to the copy task that perfectly scale with the size of the bit vectors which should be memorized, without any further training. Lastly, we demonstrated that even solutions which do not scale perfectly can be used to shorten the number of generations needed to evolve a solution for bit vectors of larger sizes. In the future it will be interesting to apply the approach to more complex and less regular domains, in which the geometry of the connectivity pattern to discover is more complex. Additionally, combining the presented approach with recent advances in evolutionary strategies [11] and genetic programming [12], which have been shown to allow evolution to scale to problems with extremely high-dimensionality is a promising next step.

# References


1. Sukhbaatar, S., Weston, J., Fergus, R., et al.: End-to-end memory networks. In: Advances in neural information processing systems. pp. 2440–2448 (2015)
2. Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J., et al.: Hybrid computing using a neural network with dynamic external memory. Nature 538(7626), 471–476 (2016)
3. Graves, A., Wayne, G., Danihelka, I.: Neural turing machines. CoRR abs/1410.5401 (2014), http://arxiv.org/abs/1410.5401
4. Greve, R.B., Jacobsen, E.J., Risi, S.: Evolving neural turing machines for reward-based learning. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. pp. 117–124. GECCO '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2908812.2908930
5. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation 10(2), 99–127 (2002)
6. Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based encoding for evolving large-scale neural networks. Artificial life 15(2), 185–212 (2009)
7. Stanley, K.O.: Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines 8(2), 131–162 (2007)
8. Sporns, O.: Network analysis, complexity, and brain function. Complexity 8(1), 56–60 (2002)
9. Clune, J., Stanley, K.O., Pennock, R.T., Ofria, C.: On the performance of indirect encoding across the continuum of regularity. IEEE Transactions on Evolutionary Computation 15(3), 346–367 (June 2011)
10. Ha, D., Dai, A., Le, Q.V.: Hypernetworks. arxiv preprint. arXiv preprint arXiv:1609.09106 2 (2016)
11. Salimans, T., Ho, J., Chen, X., Sutskever, I.: Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864 (2017)
12. Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O., Clune, J.: Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567 (2017)

13. Stanley, K.O., Miikkulainen, R.: Competitive coevolution through evolutionary complexification. J. Artif. Int. Res. 21(1), 63–100 (Feb 2004), http://dl.acm.org/citation.cfm?id=1622467.1622471
14. Floreano, D., Dürr, P., Mattiussi, C.: Neuroevolution: from architectures to learning. Evolutionary Intelligence 1(1), 47–62 (2008)
15. Bongard, J.C.: Evolving modular genetic regulatory networks. In: Proceedings of the 2002 Congress on Evolutionary Computation (2002)
16. Gauci, J., Stanley, K.O.: Indirect encoding of neural networks for scalable go. In: Schaefer, R., Cotta, C., Kołodziej, J., Rudolph, G. (eds.) Parallel Problem Solving from Nature, PPSN XI: 11th International Conference, Kraków, Poland, September 11-15, 2010, Proceedings, Part I. pp. 354–363. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15844-5_36
17. Hornby, G.S., Pollack, J.B.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life 8(3) (2002)
18. Stanley, K.O., Miikkulainen, R.: A taxonomy for artificial embryogeny. Artificial Life 9(2), 93–130 (2003)
19. Clune, J., Beckmann, B.E., Ofria, C., Pennock, R.T.: Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC-2009) Special Session on Evolutionary Robotics. IEEE Press, Piscataway, NJ, USA (2009)
20. Secretan, J., Beato, N., D'Ambrosio, D.B., Rodriguez, A., Campbell, A., Stanley, K.O.: Picbreeder: Evolving pictures collaboratively online. In: CHI '08: Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems. pp. 1759–1768. ACM, New York, NY, USA (2008)
21. Risi, S., Stanley, K.O.: A unified approach to evolving plasticity and neural geometry. In: Neural Networks (IJCNN), The 2012 International Joint Conference on. pp. 1–8. IEEE (2012)
22. Cellucci, D., MacCurdy, R., Lipson, H., Risi, S.: 1D printing of recyclable robots. IEEE Robotics and Automation Letters 2(4), 1964–1971 (2017)
23. Risi, S., Stanley, K.O.: An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. Artificial life 18(4), 331–363 (2012)
24. Lüders, B., Schläger, M., Korach, A., Risi, S.: Continual and one-shot learning through neural networks with dynamic external memory. In: European Conference on the Applications of Evolutionary Computation. pp. 886–901. Springer (2017)
25. Greve, R.B., Jacobsen, E.J., Risi, S.: Evolving neural turing machines for reward-based learning. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference. pp. 117–124. ACM (2016)
26. Verbancsics, P., Stanley, K.O.: Constraining connectivity to encourage modularity in hyperneat. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation. pp. 1483–1490. GECCO '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2001576.2001776
27. D'Ambrosio, D.B., Lehman, J., Risi, S., Stanley, K.O.: Evolving policy geometry for scalable multiagent learning. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1. pp. 731–738. International Foundation for Autonomous Agents and Multiagent Systems (2010)
28. Gauci, J., Stanley, K.O.: Autonomous evolution of topographic regularities in artificial neural networks. Neural computation 22(7), 1860–1898 (2010)
29. Woolley, B.G., Stanley, K.O.: Evolving a single scalable controller for an octopus arm with a variable number of segments. In: International Conference on Parallel Problem Solving from Nature. pp. 270–279. Springer (2010)