# Deep HyperNEAT: Evolving the Size and Depth of the Substrate

**Evolutionary Complexity Research Group Undergraduate Research Report**

**Felix A. Sosa and Kenneth O. Stanley**

University of Central Florida, Department of Computer Science
Emails: felixanthonysosa@gmail.com, kstanley@cs.ucf.edu

## ABSTRACT

This report describes DeepHyperNEAT, an extension of HyperNEAT to allow it to alter the topology of its indirectly-encoded neural network (called the *substrate*) so that it can continue to grow and increase in complexity over evolution. This aim is accomplished by augmenting HyperNEAT's *compositional pattern producing networks* (CPPNs) with new information that allows them to represent substrate topology, and by adding three novel mutations to HyperNEAT that exploit this new information. The purpose of this report is to detail the method and validate its ability to evolve, which is shown through a simple XOR-based test. Significantly more work will be needed to analyze the algorithmic and scientific implications of these new capabilities, but by releasing this report it becomes possible for others also to explore the opportunities that such an extension opens up. A link to associated source code is also provided at the end of this document.

## Introduction

HyperNEAT [1; 2; 3] (Hypercube-based NeuroEvolution of Augmenting Topologies) is a method for evolving neural networks (NNs) through an indirect encoding based on the NEAT method [4] that came before it. The main idea in HyperNEAT is that a special genetic representation called a *compositional pattern producing network* (CPPN) [5] that is evolved by NEAT generates a pattern of weights that connect a fixed set of neurons positioned within a two-dimensional or three-dimensional space called the *substrate*. A longstanding limitation of the conventional HyperNEAT algorithm is that while the CPPN can increase in size, the number of neurons and layers in the substrate remains fixed from the beginning of evolution. While there exists one method called *evolvable substrate HyperNEAT* (ES-HyperNEAT) [6] that allows HyperNEAT to increase the complexity of its substrate that indeed offers opportunities not available in the original HyperNEAT, the new method proposed here, called *Deep HyperNEAT* is designed explicitly to evolve substrates reminiscent of architectures seen in deep learning today, with increasing numbers of layers and separate parallel pathways through the network.

Deep HyperNEAT achieves its new ability by fitting more information into the standard CPPN of HyperNEAT, which now can mutate to describe new layers and new neurons in the substrate. This paper is a preliminary technical report on the method, released to allow others to experiment with its ideas quickly. Thus many intriguing scientific questions about the opportunities created by this new capability will not be answered here; instead the focus is on introducing the technical components of the method and showing that they work correctly (i.e. perform as expected) in a simple validation experiment on the XOR problem. Rigorous scientific investigation is left for the future.

The paper is organized as follows: HyperNEAT and the CPPN indirect encoding are reviewed first. The main extensions to HyperNEAT and CPPNs that constitute Deep HyperNEAT are then described, and demonstrated running in a simple XOR test. The paper concludes with thoughts on implications and future work. A public repository containing all of the code for the method is linked at the end of the paper.

## Background: HyperNEAT

HyperNEAT [1; 2; 3] is a neuroevolution method that utilizes indirect encoding to evolve connectivity patterns in neural networks (NN). HyperNEAT historically has found applications in a wide breadth of domains, including multiagent control [7], checkers [2; 3], quadruped locomotion [8; 9], Robocup Keepaway [10], and simulated driving in traffic [11].

The motivation for HyperNEAT was to capture the ability of natural genetic encoding (DNA) to reuse information to encode massive neural structures with orders of magnitude fewer genes. The key principle in HyperNEAT that enabled this kind of meaningful reuse is that neural geometry should be able to exploit problem structure. (This principle is similar to the inspiration behind incorporating convolution in deep learning, but its realization and implications in HyperNEAT are different.) For example, two hidden neurons that process adjacent parts of an input field (e.g. a visual field) should also be roughly adjacent within the neural geometry. That way, locality within an NN becomes a heuristic for relatedness.

To situate neurons in an NN within a geometric context, each neuron in HyperNEAT is assigned a Cartesian coordinate. For example, if the NN is configured as a stack of two-dimensional layers of neurons, then the position of each neuron within a layer is simply $(x,y)$. In HyperNEAT it is conventional that the coordinate system of each layer centers at $(0,0)$ and ranges between [-1, 1] in both $x$ and $y$, thereby forming a square filter or "sheet" of neurons. That way, each neuron $n$ within a layer $l$ can be identified by its position $(x,y)$.

Specifying neurons in the NN as a function of the neural geometry allows for an elegant formalism for encoding the connectivity patterns between layers. The connectivity pattern between one layer and another (e.g. all possible connections and weights between neurons in some layer $i$ and another layer $j$) can be encoded by a function `ConnectLayers` that takes the positions of the source and target neurons as input.

Instead of evolving the weights of every connection individually as in past approaches to neuroevolution like NEAT [12], HyperNEAT evolves a graphical representation of a function, `ConnectLayers`, that describes how the neurons are connected. In this way, `ConnectLayers` acts as a lower-dimensional encoding for how to generate connectivity patterns. For example, we might want to query `ConnectLayers` for the weight of a connection between a neuron $a$ in layer 1 with coordinates $(x_1,y_1)$ and a neuron $b$ in layer 2 with coordinates $(x_2,y_2)$. `ConnectLayers` then returns the weight $w_{x_1,y_1,x_2,y_2}$:

$$w_{x_1,y_1,x_2,y_2} = \texttt{ConnectLayers}(x_1,y_1,x_2,y_2). \tag{1}$$

The main idea in HyperNEAT is to evolve `ConnectLayers`. In HyperNEAT, `ConnectLayers` is represented by a compositional pattern producing network (CPPN) [13] (figure 1). A CPPN is an evolvable directed graph representation for functions. Much like NNs, nodes are functions and connections have weights that are applied to the outputs of nodes before passing into other nodes. Importantly, CPPN node functions are heterogeneous in any given CPPN (they are sampled from a set of possible functions usually containing sigmoid, linear, Gaussian, and cosine) and there is no notion of layers. A thorough introduction or review of CPPNs is in [5].

The CPPN begins with a simple topology (usually just its input nodes and its output node with a sparse number of connections between them) and complexifies via evolution in HyperNEAT. Evolving the CPPN this way, through complexification, allows the CPPN to be biased towards compressed representations of connectivity patterns. This approach leads to increasingly intricate yet organized connectivity patterns. In practice CPPNs have proven effective and popular in part because they are designed to evolve increasing complexity with the original NEAT algorithm, which underlies evolution in HyperNEAT. The predicate "hyper" in HyperNEAT signifies that the CPPN is in effect encoding a pattern within a four-dimensional or six-dimensional hypercube (because `ConnectLayers` typically takes four or six inputs), which is then painted onto the connectivity of a network.

In effect, HyperNEAT provides the unique ability to evolve a compressed function that can encode the connectivity of an arbitrarily large network as a function of its geometry. That way, it is possible to produce NNs whose connections display regular and compositional structure through the reuse of motifs encoded in the CPPN. The

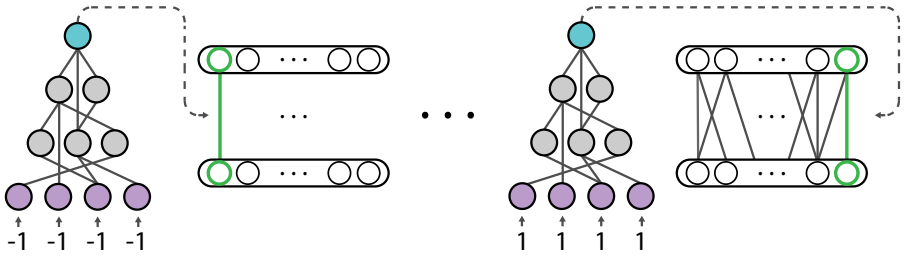


**Figure 1.** Querying the CPPN (left) for weights within a substrate (right) in HyperNEAT. Substrates begin with no connections. Going pair by pair (highlighted), the CPPN will take as input the coordinates of both nodes $(x_1, y_1), (x_2, y_2)$ for the given pair and outputs a weight. If the weight is above a threshold, a connection is added to the substrate with that weight, otherwise no connection is made for that pair. The CPPN is queried for all possible pairs to form a connectivity pattern between two layers.

HyperNEAT algorithm is summarized in Algorithm 1.

```
Input: Substrate
Output: CPPN
Init population with minimal CPPNs;
while Stopping criteria is not met do
    foreach CPPN in population do
        foreach Possible connection in Substrate do
            Query CPPN for weight w of connection;
            if Abs(w) > Threshold then
                Add connection with weight w to Substrate;
            end
        end
        Ascertain fitness of Substrate in task domain;
    end
    Reproduce CPPNs according to NEAT to create next generation;
end
Output solution CPPN;
```

**Algorithm 1:** HyperNEAT

### HyperNEAT Limitations

While HyperNEAT advanced beyond NEAT by adding indirect encoding, there is one sense in which it is a step backwards: users must decide the NN topology a priori. HyperNEAT does not have a method for evolving representations of topology along with connectivity. Specifically, the user must decide a priori how many neurons are in each layer, how many layers there are, and which layers connect to which. Once these decisions are made, HyperNEAT can evolve the connectivity patterns among the various layers, but it cannot on its own evolve, for example, the number of layers. In this way HyperNEAT lacks NEAT's appealing ability to evolve the overall topology of the NN and it is this limitation that we address in our proposed method below.

## DeepHyperNEAT

HyperNEAT's indirect encoding allowed arbitrary connectivity patterns to evolve in NNs and significantly increased the size of neural networks amenable to neuroevolution. However, as noted above, the configuration of the substrate has to be set by the user a priori. To remove this constraint, we add more information to the CPPN's output nodes and introduce three novel mutations to HyperNEAT. These enhancements allow the CPPN to represent both the

configuration of the substrate (a new capability) and the connectivity pattern of the NN as before. Specifically, these mutations enable the addition of depth (`IncrementDepth`), breadth (`IncrementBreadth`), and new connectivity between previously unconnected layers (`AddMapping`) in a substrate, all with minimal disruption throughout the course of evolution.

### Revisiting the CPPN

In HyperNEAT, the CPPN output node is read for a scalar value, $w$ (the output of `ConnectLayers`), representing a connection weight between two neurons in a substrate (Figure 1). The CPPN is queried for each pair of neurons in the substrate to generate the substrate's connectivity pattern. One method to enable the CPPN to represent connectivity patterns between different layers in substrates with hidden layers is to give the nodes in a substrate three-dimensional coordinates $(x,y,z)$, where $z$ corresponds with the height or layer in the substrate. The CPPN is adjusted accordingly to take three-dimensional coordinates as input. Intuitively, giving each node a third dimension to represent its layer allows the CPPN to differentiate between nodes within different layers and encode connectivity patterns conditioned on which layers the nodes being queried connect. This approach can be expressed as

$$w_{x_1,y_1,z_1,x_2,y_2,z_2} = \texttt{ConnectLayers}(x_1,y_1,z_1,x_2,y_2,z_2). \tag{2}$$

The downside of this formalism is that it implies that the connectivity pattern of one layer should be strongly related to the pattern of nearby layers, which is not necessarily well supported.

A potentially better way to represent connectivity patterns between different layers through a CPPN is by simply adding another output node. This approach then requires that we index the layers of the substrate and indicate which layer of connections each output node in a CPPN is responsible for encoding. Importantly, in this way, the CPPN not only encodes multiple layers of patterns but it also implicitly encodes the topology of the substrate. In effect, the indices of the output nodes specify the layers they are responsible for connecting, which means it is possible to infer the (bare) topology of a substrate. With this capability, by incorporating new mutations into HyperNEAT that add *output nodes* with the necessary information, evolving CPPNs can now change both the connectivity patterns and topologies within substrates.

The information we embed in the output nodes of the CPPN is called a `MappingTuple`, which is a tuple of tuples, $((a,b),(c,d))$, that details the source $(a,b)$ and target $(c,d)$ of a connectivity pattern. As we discuss below, to allow the CPPN to increase both the depth and breadth of a substrate, new filters or "sheets" can be added. A sheet constitutes an entire layer if it is the only sheet in the layer, or an individual filter within a layer if it is not the only sheet (Figure 2). For each `MappingTuple`, $a$ and $c$ represent the layer of the source and target and $b$ and $d$ represent the sheet within the source and target layers, respectively.

With this information included in the CPPN, decoding the CPPN into a substrate follows the same process as HyperNEAT but with the added step of reading the `MappingTuples` from of the CPPN output nodes to first construct the topology (and positioning) within the substrate. This process is here called `Decoding` and is presented in Algorithm 2.

### Increasing Depth

To add depth a new output node is added to the CPPN whose `MappingTuple` depends on the current set of `MappingTuples` in the CPPN; those previous `MappingTuples` are then updated as well. The main idea is to *insert* the new sheet of nodes between two preexisting such layers. A key challenge in this process is minimizing the disruption of this mutation on the substrate. In principle, adding a layer to the substrate *en masse* can be highly disruptive by fundamentally altering the function computed by the overall substrate.

To minimize such disruption, the CPPN initially encodes an *identity pattern* between the newly added layer of nodes and the preexisting layer to which it now connects such that the activations of the original source or incoming nodes (above which the new layer is inserted) are passed without change to the target or outgoing nodes. This identity pattern is encoded through a function similar to convolution wherein each new node position projects a single connection to the node in the next layer up at the same coordinate (see Figure 3). Specifically, for any two nodes $i$ and $j$, a connection is made by the CPPN if and only if their coordinates are equivalent, $x_i = x_j$ and $y_i = y_j$.

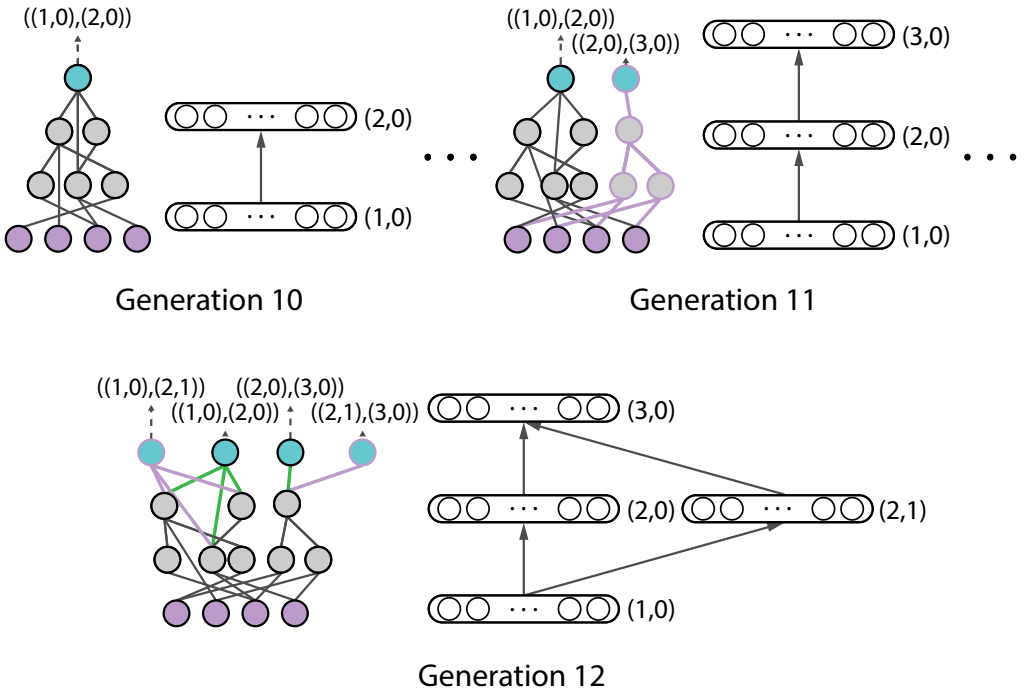


**Figure 2.** Evolving topology and connectivity patterns in Deep HyperNEAT. Each output node encodes a specific connectivity pattern between two layers in a substrate. In Generation 10, the CPPN only has one output node, encoding a substrate with no hidden layers. In Generation 11, the CPPN undergoes a mutation encoding a new layer with a connectivity mapping that encodes identity, adding the new output node, three Gaussian nodes, and their corresponding connections (highlighted in pink). In Generation 12, the CPPN undergoes a mutation adding breadth to the new hidden layer, copying the two output nodes and connections responsible for encoding the incoming and outgoing connectivity pattern associated with the original layer (highlighted in pink and green).

This connectivity pattern can be forced through a composite function of Gaussians (added to the CPPN), expressed formally as

$$\texttt{ConnectLayers}(x_1, y_1, x_2, y_2) = e^{(e^{-(x_1-x_2-\mu)^2} + e^{-(y_1-y_2-\mu)^2} + \mu)^2}. \quad (3)$$

This identity-based `ConnectLayers` function can be encoded in a CPPN a s shown in Figure 3. Importantly, through evolution, which can add and remove new nodes and connections to a CPPN, this initial representation can gradually differentiate into a more interesting filter or sheet (e.g. more complex). In this way, we gain the ability to add a whole new layer of depth to a substrate with minimal functional impact and then gradually differentiate its connectivity pattern in future generations into something useful.

### Increasing Breadth

To increase breadth with minimal disruption an existing sheet in a substrate is simply duplicated. All CPPN output nodes associated with the preexisting sheet (including the incoming connections to those nodes in the CPPN) that represent the incoming and outgoing connections to and from the preexisting sheet are copied. The weights incoming to the original and copied output nodes are then halved. This process means that the old and new sheets specified by those output nodes together will now in aggregate produce the same functionality as the original sheets did alone (see Figure 4). As with `IncrememntDepth`, this newly duplicated set of connectivity patterns can then begin to differentiate throughout evolution into something more interesting and useful.

```
Input: CPPN
Output: Substrate
/* First decode topology                                                    */
Substrate ← {∅} ;                                  // Substrate begins as empty set
foreach OutputNode in CPPN do
    Mapping ← Mapping Tuple of OutputNode;
    Append Mapping to Substrate;
end
/* Query CPPN for Substrate connectivity                                    */
foreach Mapping in Substrate do
    Source ← Mapping[0];
    Target ← Mapping[1];
    foreach PossibleConnection between Source and Target do
        Weight ← CPPN(PossibleConnection) ;                  // Query CPPN for Weight
        if Weight > ε then
            Append PossibleConnection to Substrate;
        end
    end
end
Output Substrate;
```

**Algorithm 2:** Decoder

### Adding Connectivity

As the substrate complexifies through evolution, some sheets inevitably remain unconnected after incrementing depth or breadth as proposed above. To allow arbitrary connections between sheets to evolve, it is necessary to implement a mutation that simply adds an output node representing a connectivity pattern between sheets not currently connected in the substrate. Such an addition makes possible the evolution of bypass connections and recurrent connection groups.

To connect currently unconnected preexisting sheets with minimal disruption, an output node is added whose output is 0 for all possible connections. This approach in practice means adding an output node to the CPPN with no incoming connections. This null connectivity pattern can then gradually evolve into something more interesting and useful for the problem domain (Figure 5).

## Experiment: XOR

This preliminary report aims to introduce the method and provide a functional implementation. The real potential of the method is left for future work, but the functionality and correctness of the implementation is validated here through XOR. XOR is not linearly separable so the NN solution requires hidden units. This structural requirement makes XOR suitable minimal benchmark for testing DHN's ability to evolve substrate topology. For example, DHN's method for adding new layers might be too destructive to allow new layers to enter the population. Or, it could find a local optimum with a poor connectivity pattern that dominates the population, causing the system to fail to evolve the optimal connectivity pattern. Third, changing the NN topolgy could render past connection weight values obsolete. If that happens, the algorithm would have trouble enlarging topologies that are already specialized. While this task is trivial scientifically, it serves as a minimal demonstration that DHN is not impeded by such potential obstacles and can effectively evolve topology and connectivity reliably and efficiently when needed.

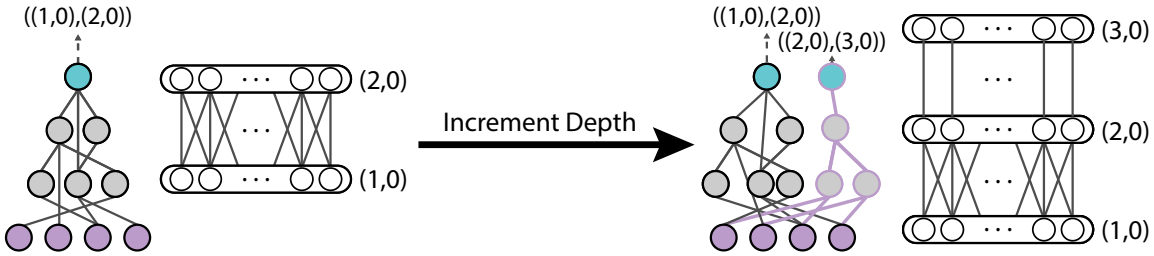


**Figure 3.** Adding layers to a substrate. To increment the depth of a substrate with minimal disruption, three Gaussian nodes are added (highlighted in pink). One takes as input the *x*-coordinates and another takes as input the *y*-coordinates. Both then feed into the third Gaussian which feeds into the output node. This augmentation successfully encodes an identity connectivity pattern that simply passes the activations from the previous layer to the next layer but can gradually differentiate over evolution into something useful and interesting.

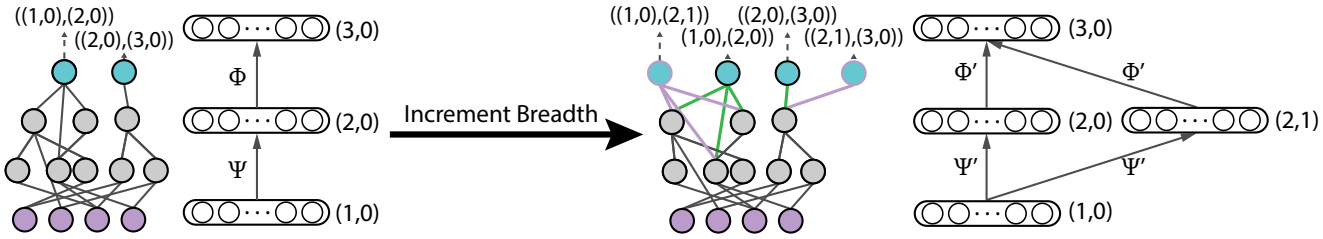


**Figure 4.** Increasing the breadth of a layer (adding filters). To increase breadth with minimal disruption, a sheet within a layer is randomly selected to be copied. The output nodes and connections in the CPPN that encode the sheet's associated connectivity patterns are then copied in the CPPN, creating new output nodes (highlighted in pink). To minimize disruption, the weights feeding into the new (highlighted in pink) and copied (highlighted in green) output nodes of the CPPN are halved. This process halves the functions encoded by the original connectivity patterns $\phi$ and $\psi$ into $2\phi$' and $2\psi$', which are equivalent.

### Parameter Settings

There are a handful of parameters available to the user that derive from HyperNEAT and NEAT before it. The most relevant are the population size, mutation rates, substrate input and (initial) output layer dimensions, and elitism. Additionally, unique to DeepHyperNEAT, one other parameter is the number and positioning of nodes within the sheets; sheets default to the input layer specification but can be arbitrarily specified for different configurations by the user. All parameters with the exception of sheet size were held constant across all evolutionary runs. As discussed below, we altered the sheet dimensions across runs to demonstrate DHN's functionality.

All evolutionary runs had a population size of 150 and ran 100 times. The mutation rates for each mutation to a CPPN were as follows: adding a hidden node to the CPPN occurred with probability of 0.2, deleting a hidden node occurred with probability of 0.2, adding a connection occurred with probability of 0.5, deleting a connection occurred with probability of 0.5, increasing depth occurred with probability of 0.1, increasing breadth occurred with probability of 0.1, and adding connectivity occurred with probability of 0.1.

Sheet sizes in DeepHyperNEAT define the geometry of a sheet. In particular, a sheet size of (1, 1) means a given sheet within the substrate consists of only one node with coordinates (0, 0). A sheet size of (1, 3) means a given sheet within the substrate consists of of three nodes with coordinates, (0, -1), (0, 0), and (0, 1). DeepHyperNEAT was run with sheet sizes (1,1) and (1,3).

Fitness was judged using sum of squared error (SSE). Deep HyperNEAT found a solution when the output substrate obtained a SSE $< 0.01$. That is, a substrate was considered a solution to XOR when it obtained 99% accuracy.

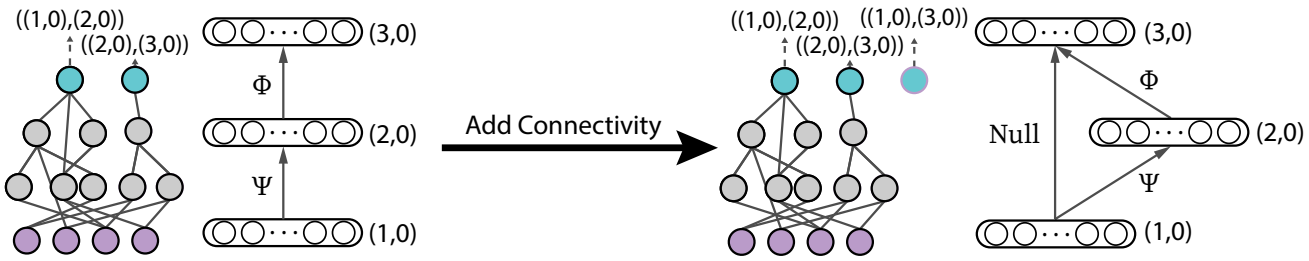


**Figure 5.** Adding connectivity. To add connectivity between two previously unconnected sheets or layers, an output node is added to the CPPN with no incoming connections. This simply creates a null connectivity pattern between two layers or sheets. The connectivity can differentiate over the course of evolution by having new connections being added to the new output node in the CPPN. Importantly, this mutation allows the emergence of both recurrent and bypass connections in neural networks at the scale found in deep learning.

### Results

Using sheet sizes of (1,1) (i.e. each sheet consists of a single node with coordinates (0, 0)), Deep HyperNEAT solved XOR within a mean of 7.07 generations with a standard deviation of 1.63. The mean number of `IncrementDepth` mutations found in the champions was 1.07 with a standard deviation of 0.26. The mean number of `IncrementBreadth` mutations found in the champions was 1.21 with a standard deviation of 0.45. Both `IncrementDepth` and `IncrementBreadth` mutations were found in the champions of all (100 out of 100) runs.

Using sheet sizes of (1,3) (i.e. each sheet consists of three nodes with coordinates (0, -1), (0, 0), and (0, 1), respectively), Deep HyperNEAT solved XOR within a mean of 25.82 generations with a standard deviation of 19.79. The mean number of `IncrementDepth` mutations found in the champions was 1.11 with a standard deviation of 0.34, The mean number of `IncrementBreadth` mutations found in the champions was 1.85 with a standard deviation of 1.06. Both `IncrementDepth` and `IncrementBreadth` mutations were found in the champions of all (100 out of 100) runs.

These number of generations to a solution are in line with the performance of e.g. conventional NEAT on XOR [4], hinting that Deep HyperNEAT can serve potentially as a variant of HyperNEAT wherein the substrate complexifies without a major performance hit versus a direct encoding. Of course, this domain is meant only as a validation of the system's operation and does not provide sufficient information for scientific insight, but it does help to establish that Deep HyperNEAT and its source code release are suitable for further investigation.

## Discussion and Concluding Remarks

A major bottleneck in neuroevolution through indirect encoding has been the evolution of connectivity and topology at the scale of common deep neural networks. This report proposes a novel method that has the ability to do just that: DeepHyperNEAT. It achieves this aim by adding three new mutations to HyperNEAT and augmenting the CPPN with more information. The functionality of the code was demonstrated on a minimal benchmark, XOR, where it successfully evolved solutions.

Our hope in releasing this project is to inspire further research into Deep HyperNEAT. It is the first HyperNEAT variant that allows depth and breadth in networks of two-dimensional layers to increase indefinitely and arbitrarily in such a way that mutations in architecture yield minimal initial disruption in functionality. We expect it to be particularly useful when the proper depth and complexity of the substrate is unknown and needs to be discovered by evolution itself. In this way, Deep HyperNEAT offers a NEAT-like opportunity for unbounded complexification within the substrate itself. Conceivable extensions might include plasticity (like in adaptive HyperNEAT [14]) or combinations with gradient descent.

## Source code

We are continuing to expand and refine the system but have released source code for DeepHyperNEAT here: https://github.com/flxsosa/DeepHyperNEAT.

## References


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

2. Gauci, J. & Stanley, K. O. Autonomous evolution of topographic regularities in artificial neural networks. *Neural Comput.* **22**, 1860–1898 (2010).

3. Gauci, J. & Stanley, K. O. A case study on the critical role of geometric regularity in machine learning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008)* (AAAI Press, Menlo Park, CA, 2008).

4. Stanley, K. O. & Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. Comput.* **10**, 99–127 (2002).

5. Stanley, K. O. Compositional pattern producing networks: A novel abstraction of development. *Genet. Program. Evolvable Mach. Special Issue on Dev. Syst.* **8**, 131–162 (2007).

6. Risi, S. & Stanley, K. O. An enhanced hypercube-based encoding for evolving the placement, density and connectivity of neurons. *Artif. Life journal* **18**, 331–363 (2012).

7. D'Ambrosio, D., Lehman, J., Risi, S. & Stanley, K. O. Evolving policy geometry for scalable multiagent learning. In *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS-2010)*, 731–738 (International Foundation for Autonomous Agents and Multiagent Systems, 2010).

8. 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).

9. Clune, J., Pennock, R. T. & Ofria, C. The sensitivity of HyperNEAT to different geometric representations of a problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2009)* (ACM Press, New York, NY, USA, 2009).

10. Verbancsics, P. & Stanley, K. O. Evolving static representations for task transfer. *J. Mach. Learn. Res. (JMLR)* **11**, 1737–1769 (2010).

11. Drchal, J., Koutník, J. & Snorek, M. HyperNEAT controlled robots learn to drive on roads in simulated environment. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC-2009)* (IEEE Press, Piscataway, NJ, USA, 2009).

12. Stanley, K. O. & Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. computation* **10**, 99–127 (2002).

13. Stanley, K. O. Compositional pattern producing networks: A novel abstraction of development. *Genet. programming evolvable machines* **8**, 131–162 (2007).

14. Risi, S. & Stanley, K. O. Indirectly encoding neural plasticity as a pattern of local rules. In *Proceedings of the 11th International Conference on Simulation of Adaptive Behavior (SAB2010)* (Springer, Berlin, 2010).