Implicit Policies for Deformable Object Manipulation with Arbitrary Start and End States: A Novel Evolutionary Approach*

Solvi Arnold and Kimitoshi Yamazaki, *Member, IEEE*

***Abstract*— We present a novel method for training (evolving) fully convolutional neural networks (CNNs) for deformable object manipulation. Instead of using a weight update rule, we evolve an ensemble of compositional pattern generating networks (CPPNs) by means of a genetic algorithm (GA). These ensembles generate the convolutional kernels that comprise the CNN. This allows the GA to search for fit kernels in the space of spatial 2D patterns, which allows for evolution of larger networks than is feasible with direct evolution of the connection weight vector. We apply this method to a thread manipulation task, in an attempt to let the CNN grasp the deformation dynamics of the thread. We report results both on single-manipulation tasks, and on tasks requiring a sequence of two manipulations, establishing the viability of this approach.**

I. Introduction

In the area of object manipulation, deformable objects pose a particularly hard challenge. The common approach to object manipulation is to first construct a 3D model of the object, and perform manipulation planning routines with respect to this model. When working with deformable objects, this default approach breaks down, as every manipulation of the object will change the object's shape and thereby obsolete the model. To incorporate deformation in the planning process, we can introduce object models that incorporate the deformation dynamics of the object, along with accurate physical simulation (see [1] for a survey). However, unlike the initial shape of the object, its deformation dynamics cannot easily be obtained by visual inspection alone.

The difficulties encountered in robotic manipulation of soft objects raise the question how humans manage deformable object manipulation. When we manipulate a soft object, do we really have accurate representations of the state of the object at every point during the manipulation? Do we actively simulate object deformation in our heads? We think the answer to these questions is 'no'. We seem to manipulate deformable objects in an iterative and intuitive way that does not involve explicit simulation of transitional object shapes. We acquire the relevant intuitions through ample physical interaction with soft objects, some of it in goal-directed settings, but most of it in non-goal-directed settings.

* This work was partly funded by NEDO.

Both authors are with Shinshu University, Nagano, Japan.
Phone: +81-26-269-5155; e-mail: (s_arnold, kyamazaki)@shinshu-u.ac.jp.

We aim to realise a similar intuitive and implicit grasp of soft object dynamics in neural networks (NNs), as NNs are known to be capable of capturing various other hard-to-formalise aspects of cognition (such e.g. as diffuse category delineations). NNs trained to navigate the state spaces of soft objects could serve as the core element of a generalised soft object manipulation system. Section 2 explains the central concepts of our approach, and the motivation behind it.

II. Concepts & Motivation

Our approach features a novel combination of two connectionist technologies, namely convolutional neural networks (CNNs) and compositional pattern producing networks (CPPNs). We briefly introduce both technologies in the following sections.

A. Convolutional Neural Networks

In the field of computer vision, CNNs are presently the technology of choice for a wide variety of image recognition tasks. The convolutional layers found in CNNs provide a natural way of exploiting the spatial structure of input data, which is highly effective for processing image data [2, 3]. In recent years, CNNs have also been receiving increasing attention in the field of robotics, as many control problems allow for similar exploitation of spatial structure. Examples of application in object manipulation can be found in [4, 5]. The main obstacle to wide-spread use in object manipulation is the difficulty of training CNNs without relying on large, annotated datasets. This issue becomes especially pressing in the context of deformable objects, as these objects themselves have such large state spaces. We explore a novel approach here.

B. Compositional Pattern Producing Networks

In the field of neuroevolution (NE), much research has focused on the search for indirect encodings of neural networks that improve evolvability. Evolving sets of connection weights for NNs using Genetic Algorithms (GAs) is effective for small networks, but quickly becomes infeasible as network size increases. The use of smart indirect encodings allows the GA to search in a space of connectivity patterns rather than having to find every connection weight individually, making evolution of far larger networks feasible. The HyperNEAT method developed by Stanley et al. [6] in particular has garnered substantial attention. The central idea in HyperNEAT is

to introduce a spatial substrate (so that the neurons in the NN have spatial positions) and let network architecture be generated by a Compositional Pattern Producing Network (CPPN) that maps pairs of neuron positions to connection weights. This allows the GA to search for network architectures at a higher level of abstraction, exploiting repeating motifs and the spatial structure of a task (both characteristic for biological brains, but absent in conventional NNs). Here a point of contact can be seen between HyperNEAT and CNNs: CNNs architectures are hardcoded to exploit spatial structure by means of repeating motifs (in the form of convolution kernels that are applied uniformly over the input).

Conceptually, NE offers substantial advantages over conventional NN training techniques, but in terms of scale and performance it is not yet competitive with present day CNNs (on those tasks where CNNs are applicable), and examples of practical application of NE are scarce still (but see [7] for a recent example of application to a computer vision task (recognition of handwritten digits)). CPPNs have also received attention in other domains, such as soft robotics [8]. In the present paper we explore how CPPNs can be imported into the CNN framework to solve planning problems. As far as we are aware this is the first attempt to evolve CNNs by means of CPPNs (though see [9] for GAs and CPPNs being used to "trick" CNNs into producing strange image recognition results), and the first application of CPPNs to a deformable object manipulation task.

C. Motivation

Our main motivation for pursuing a combination of CNN and CPPN technology is to provide a more effective way of "training" (evolving, in this case) CNNs in cases where we want to let a system learn to solve a task without prescribing *how* to solve that task (i.e. when we want to let the system invent its own policy). The most common training style for CNNs, supervised learning, is unfit for this purpose. Reinforcement learning approaches are viable for CNNs, but (see e.g. [10]), but delegate the CNN to the role of value estimator in a compound system with a separate action selection routine, incurring computational overhead in the process. Neither approach can produce neural networks that directly map observations to actions. We pursue an integrated fully neural architecture, for both theoretical and practical reasons. Theoretically speaking, this pursuit is of interest as biological brains are such systems. Connectionism is built on biological inspiration, and more may be gained by pushing further in this direction. Practically speaking, it is by no means evident that explicit computation of state or action values as seen in reinforcement learning is necessary or advantageous. Bypassing such overheads may reduce computation cost and improve performance. The present work explores this prospect.

III. Generating CNNs with CPPNs

CNNs are an example of hand-designed NNs exploiting spatial structure of their input by means of repeating motifs, with the repeating motifs in this case being the convolution kernels (the kernels are applied uniformly across the input, effectively functioning as an array of identical kernels each). Whereas HyperNEAT generates the complete NN architecture by means of a CPPN, we introduce the CPPN concept at the level of the convolution kernels of a fixed-architecture CNN. This approach is clearly less ambitious in scope than full-fledged HyperNEAT, but hopefully allows our system to tap into the strength of the tried-and-tested convolutional architecture as well as the adaptive flexibility of GAs.

The convolution kernels of CNNs used in computer vision are essentially 2D matrices of connection weights. As such they are easily visualised by mapping the connection weights to pixel colour values. This results in images showing the feature detected by a given kernel. Visual inspection of the kernels of trained CNNs often reveals simple line and edge detectors in the lower layers, and detectors for compound shapes composed thereof in the higher layers. Presence of such features invites the drawing of parallels with findings from neuroscience

Given that training (the convolutional layers of) a CNN appears to boil down to a search for the right set of 2D patterns, it can seem odd that we perform this search at the granularity of individual pixels/weights. Essentially we search for patterned images in the space of pixel value arrays. If we know we are looking for patterns, why not search in a space of patterns instead? This is the approach we explore here. In concrete, we search the space of patterns by evolving CPPNs, and let the CPPNs generate convolutional kernels for a CNN. In the next section we explain the experimental setup we used to assess the viability of this approach.

IV. Experimental Setup

A. Deformable Object Manipulation Task

As noted above, we aim to create a control system for manipulating deformable objects that does not rely on explicit modelling or simulation. We do, however, use simulation to generate tasks and evaluate performance to run the evolution process that produces the control system. In this initial work we target a simple thread on a flat surface. Thread simulation is basic, but sufficient to capture the characteristics that make deformable object manipulation so challenging. Note that our research goal is not to devise a control system for this particular object, but rather to devise a methodology that can produce control systems for manipulation of a wide variety of deformable objects.

The thread is modelled as a list of nodes, and laid out on the $[-1,+1]^2$ plane. Maximum distance between nodes is chosen so as to produce a granularity that is finer than the resolution of the control system's perception (input maps). The shape and movement of the thread are restricted by imposing a maximum on the distance between adjacent nodes and the angle between adjacent thread segments. Manipulations are performed by picking the thread up at one node and moving this node to a different location. The gripper is not modelled explicitly. We assume that gripper's rotation re-

mains fixed during manipulation, so that the moved node will retain its angle. The positions of the other nodes are then updated so as to satisfy the constraints on distance and angle between adjacent nodes and thread segments.

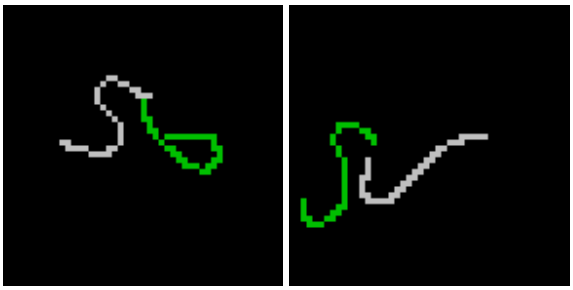

Figure 1. Example tasks for N=1 (left) and N=2 (right). White lines show initial states, green lines goal states. Note how the goal state in the two-step task cannot be reached from the initial state in a single manipulation.

Tasks are generated as follows. We start with a thread laid out in an arbitrary shape, and take a snapshot of its state (initial state). Then we select one node in the thread, and move it to a randomly selected position in the field. We move randomly selected nodes to randomly selected position for N steps, updating all other nodes in accordance with the thread properties (in this paper, we present experiments run for N=1 and N=2). Then we take a snapshot of the resulting thread state (goal state). The two snapshots together constitute one task. We compute a distance measure between the two snapshots as the average movement of the node between the snapshots (ignoring intermediate frames). A task is discarded if the distance measurement falls below a set threshold for task viability (as minimising the distance with the goal state becomes the target for the GA). At any given time, the control system sees the current state of the thread (initialised to the first snapshot, and updated with every manipulation performed by the system) and the goal state (the second snapshot).

Note that the start and goal states have no particular meaning or salience. We could set up our system to learn to lay the thread out in a straight line or neat circle in every task, but our intentions lie elsewhere. Our goal is to breed an intuitive grasp of deformable objects' dynamics into a CNN, essentially training it to map (sequences of) manipulation instructions onto (sequences of) transitions of the object through its state space. Training on fixed shapes would expose only a subset of the object's possible transitions, and cannot be expected to produce a general affinity with the object.

As the control system is a CNN, we must translate the thread states into a format that the CNN can process. Also, we need the format to be such that we could reasonably obtain it from camera images of real objects. We opt for a simple binary rasterization of the thread. The $[-1,+1]^2$ plane is discretised into a grid of 48×48 cells, and every cell containing at least one thread node is set to 1 (the maximum distance between nodes is set to equal the size of one grid cell, so that no gaps can occur in the rasterization of the thread). All other cells are set to 0. This format is suitable as CNN input, and very similar input could be obtained by simple pre-processing of camera images of a real thread.

In order to run a GA we need a *fitness function*. We measure performance by computing the distance between the final thread state (the result of the CNN's manipulations) and the goal state, in the same way we computed the distance between the initial and goal state. This distance is normalised by dividing by the distance between the initial and goal state. This gives us a fitness measure where 0 means *perfect goal alignment* and 1 means *no improvement w.r.t. the initial state*. This fitness evaluation is performed at the end of every task. Note that this means that for two-step tasks, the result if the first manipulation is not explicitly evaluated, as we do not want to enforce any particular outcome here. It is up to evolution to invent the policy. The good first move is a move that puts the thread into a state from which the CNN itself knows how to reach the goal state.

B. *Control System*

Figure 2 shows the global structure of the system. Since we use a GA as our mechanism of adaptation, we have a *population* (of fixed size) of *individuals*. An individual in our system is an *ensemble of CPPNs*. Note that there are two level of grouping levels here: the population and the ensemble. The fitness of a CPPN ensemble is determined by the performance of the (fixed-architecture) CNN composed of the set of kernels it generates. Let us emphasise that the GA is not involved in determining the manipulations for individual tasks. Tasks are solved by the CNNs, while the GA optimises the CPPN ensembles that generate the CNNs. Once the GA has run its course, we can store the best CNN for actual use and discard the rest of the population.

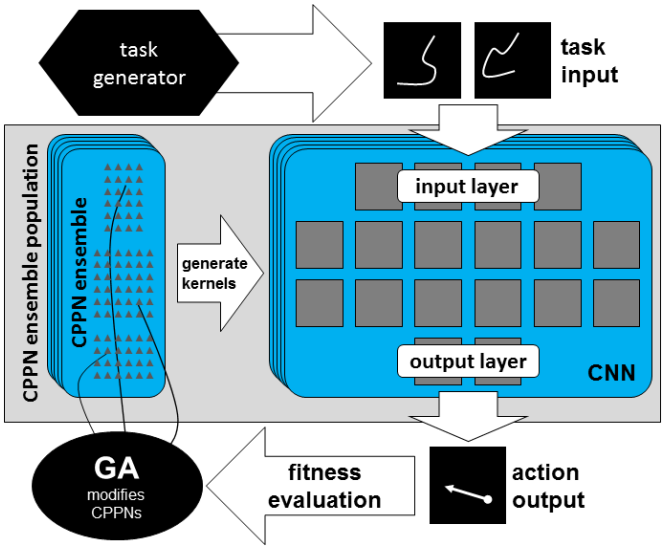

Figure 2. Global system structure.

C. *CNN*

Readers familiar with the CNN literature will know that a CNN usually contains some number of fully connected layers before the output layer. It is in principle possible to construct fully connected layers out of kernels (one can simply use a

kernels of the same resolution as the maps in a given layer and apply no padding), but it is unlikely that CPPN-generated kernels would be effective in such usage. We opt for a fully convolutional architecture (no fully connected layers at all). We are not the first to do so (see [11]). We do not include any pooling layers either. Pooling layers would integrate easily with CPPN-generated kernels, but for this particular task we want our output to have the same resolution as our input, so we have no use for pooling. The CNN architecture is shown in Table I. Our CNNs have relatively small numbers of maps, and relatively large convolutional kernels. Preliminary experimentation showed these settings to be most effective. The CNN is implemented in TensorFlow [12] and run on a Titan X GPU.

Due to the absence of pooling layers (and the use of zero-padding), all maps have the same resolution, equal to the resolution of the task field (48×48). Input consists of rasterisations (at the CNN's input resolution) of the current thread state and goal state (as provided by the task generator). Due to the fully convolutional architecture, spatial coherence is maintained through all layers of the net, and the output is in the same 48×48 format as the input. Actions take the form of a pair of coordinates, one indicating where to pick the thread up and one indicating where to put it down. We obtain these coordinates from the CNN's output by identifying the coordinates of the cell with the highest activation value in each of the output maps. The coordinates for picking the thread up can be invalid, as only a relatively small number of cells actually contain a node of the thread. When there is no node in the cell indicated by the CNN, we look for the nearest node. If this node is within 1/20 the size of the field, we perform the manipulation on that node. If the node is further away, we perform no manipulation. Failures are not explicitly penalised; the fitness evaluation as explained above naturally assigns them a score of 1.

Each layer beyond the first performs a simple per-map normalisation. We found that the softmax function performs well here. The idea is that this should help prevent maps from overshadowing one another, as it forces all maps to sum to one. The softmax function additionally has a "sharpening" effect, accentuating peaks and ridges in the activation pattern of a map.

TABLE I. CNN ARCHITECTURE.

Layer	Number of maps	Number of kernels	Kernel Resolution
1	2 (input)	16 (2×8)	17×17
2	8 (hidden)	64 (8×8)	17×17
3	8 (hidden)	64 (8×8)	17×17
4	8 (hidden)	16 (8×2)	17×17
5	2 (output)		

D. CPPNs

CPPNs are essentially mathematical expressions for computing pixel values from pixel coordinates. Consider for example the expression $v = x \cdot y$, where x and y are coordinates of a point in the $[0,1]^2$ plane and v is the value for that point (e.g. 0.0 for black and 1.0 for white). If we evaluate this expression over (a discretisation of) the $[0,1]^2$ plane, the resulting image will show a compound gradient spanning white in the (1.0, 1.0) corner to black in the other corners. More complex expressions can produce more complex images. CPPNs are often visualised as networks that show the structure of the expression. In network representation, the input nodes take pixel coordinates and the output node (or nodes, when using colour) produce the pixel values. Hidden nodes apply mathematical operations from a given repertoire to one or more input values. We chose a fairly simple style of CPPN, where each hidden node has two weighted incoming connections, and applies one of the following functions: sine, cosine, Gaussian, hyperbolic tangent.

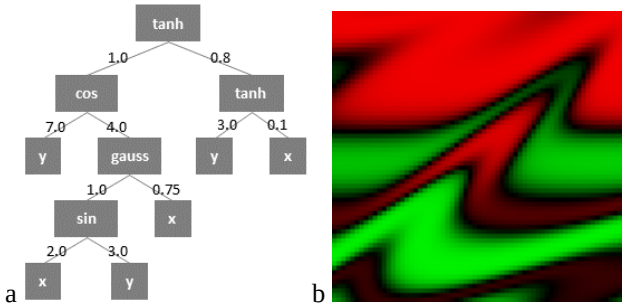

Figure 3. Example CPPN and the image it generates on the $[-1,+1]^2$ plane (at a resolution of 64×64). The value range for the pixels is [-1, +1]. Red and green represent negative and positive values, respectively.

Input to CPPNs is usually given in the form of Cartesian coordinates. When using CPPNs to generate convolution kernels, this is not necessarily the best choice. In the propagation process of a CNN, each application of a kernel computes one input value for a cell in the map the kernel outputs to. This value expresses the extent to which the pattern the kernel detects is present in the area around the coordinates of that cell in the map the kernel takes its input from. As such a kernels detect patterns relative to their central point (kernels are almost invariably given odd resolutions so that they have an unambiguous centre). In consideration of this fact, we decided to provide not just Cartesian coordinates, but also polar coordinates as input to the CPPN. While this does not actually provide the CPPN with any novel information (polar coordinates are easily computed from Cartesian coordinates), it does facilitate the evolution of patterns that are oriented specifically with respect to the kernel centre. We further introduce scale and shift parameters for each of the four coordinate values. Modification of these parameters causes global scaling and shifting of the pattern produced by the CPPN (these parameters were added to facilitate evolution, although we have not experimentally verified their necessity yet). Figure 3 shows an example of a CPPN and the image it produces on the $[-1,+1]^2$ plane. We let each CPPN in an ensemble generate such an image-like 2D array of values. These arrays are used as the convolutional kernels of the CNN (so individual values in the array function as connection weights in the CNN). While we generate the kernels at a resolution of

17×17 (see Table I), the same CPPN ensemble can just as well generate the kernel set at any other resolution. While we have not explored this feature here, it makes it possible to rescale the whole CNN to a different resolution without a need to rerun the evolution process. In this sense, CPPN-generated CNNs are *resolution-free*.

E. Genetic Algorithm

GA trials are structured as follows. We randomly generate a population of 64 individuals (recall that individuals here are CPPN ensembles). Every generation of the evolution process, we randomly generate a batch of either 64 tasks (one-step experiment) or 32 tasks (two-step experiment). Each individual is tested on all tasks. The aggregate score over all tasks is used as the individual's fitness. The population is then sorted by fitness. The bottom 7/8 of the population is discarded and replaced by new individuals, which are produced by crossover between two randomly selected individuals from the top 1/8, followed by mutation.

Crossover is implemented at the level of CPPN ensembles: when crossing ensembles A and B, for each CPPN slot we randomly pick either the CPPN A has in that slot or the CPPN B has in that slot (note that beyond the first few generations, A and B are bound to contain mostly identical or similar CPPNs, hence crossover will usually produce ensembles of similar quality as the parent ensembles). Crossover could also be implemented to operate on pairs of CPPNs (as is done in HyperNEAT), but we have not pursued this option here.

When mutating a CPPN ensemble, each CPPN has a small probability (5%) of being mutated. 50% of mutations modify the CPPN, 25% discard the whole CPPN, replacing it with a randomly generated new one, and the remaining 25% copies a CPPN from a different slot in the ensemble. Modifying mutations apply any of the following modifications, at random locations in the CPPN:

- Changing the input coordinate of a leaf node.
- Changing the operator in a hidden node.
- Inserting a hidden node at a random position in the graph. The node previously at that position becomes one of the child nodes of the new node. A new random leaf is generated for the other input.
- Removal of a random hidden node. The node is replaced with one of its child nodes.
- Changing any of the CPPN's global scale or shift values.

After letting the GA run for 200 generations, we measure the performance of the best individual of the last generation on a batch of 512 tasks.

V. Results

Table II shows scores obtained on one-step and two-step tasks, along with scored examples of manipulation outcomes in Figure 4. The scores shown for the example manipulation outcomes give an impression of what the numbers in Table II mean in terms of performance.

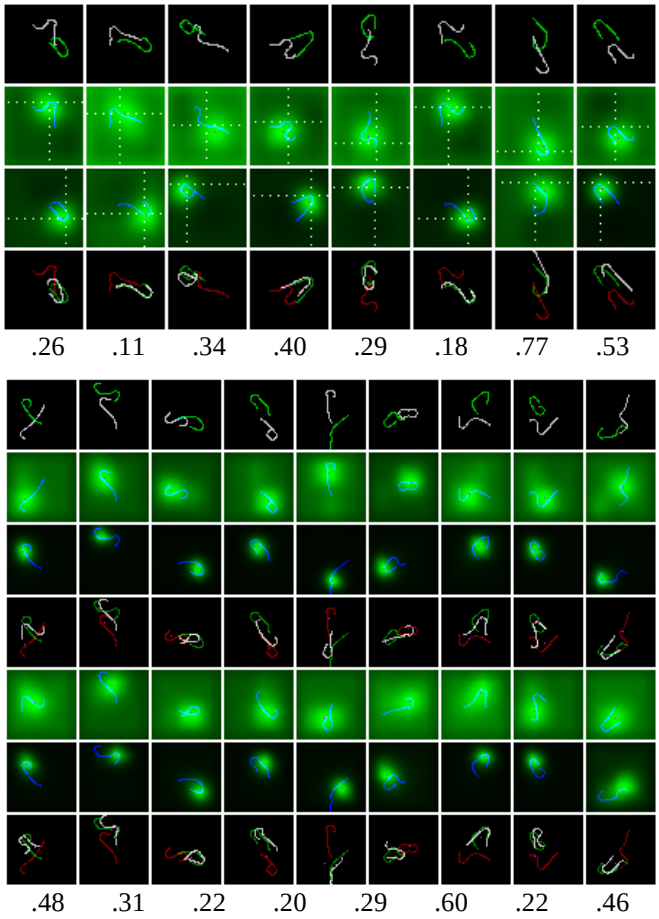

Figure 4. Representative manipulations by the best individuals of the final populations for the one-step task (top) and the two-step task (bottom). Each column shows on task / manipulation sequence. White lines show thread states, green lines goal states, and red lines (in frames beyond the first) echo the initial state. Between thread-state displays, we show the output of the CNN, with lighter shades of green indicating higher activation values. The location of the peak activation value (marked in pink) is used to determine the location to pick up (rows 2 & 5) and put down (rows 3 and 6) the thread. Numbers below the images show the fitness scores for these manipulation outcomes. For the one-step task, dotted crosshairs indicate the optimal solution. For the two-step task, row 4 shows the intermediate state in between the two manipulations.

We can observe that alignment between goal state and manipulation outcome is not exact, but usually a decent approximation of the goal state is obtained. The evolved CNNs show a mixture of patterned kernels and near-uniform kernels. Figure 5 shows a selection of kernels from the best individual from both experiments. The kernels in Figure 5 show intricate patterns, but we note that roughly half the kernels in the evolved nets turn out uniform or near-uniform. Such kernels likely contribute little to the CNN's performance. The presence of such kernels marks a difference with the error back-propagation approach. In standard forms of error back-propagation learning, there is no way for the algorithm *not* to use a given connection. With an evolutionary approach, it is very well possible for part of the computational substrate to remain simply unused. The patterned kernels show a variety of lines as well as high and low frequency striped patterns at various angles and curvatures, as would seem useful given the nature of the task. Analysis of how the superimposition of

these kernels' outputs computes appropriate manipulations is not trivial, but we intend to explore this topic in future work.

TABLE II. PERFORMANCE.

Task	Score average	Standard deviation
One-step	0.330	0.16
Two-step	0.387	0.13

In observing the evolution process we noticed that the largest performance gains are made in the first 10 or so generations. Over such a small number of generations, it is unlikely that sufficient mutations occur for substantial tuning of the CPPNs themselves. This suggests that quite a lot can be achieved by recombination of the randomly generated initial CPPN ensembles (via crossover operations and the copy mutation). This echoes a finding from [7]: The main system evaluated there is a classifier built on top of feature detectors generated by evolved CPPNs. In a control experiment, the authors found that a classifier built on top of a set of feature detectors generated by random CPPNs already performed surprisingly well (though it was outperformed by the classifier with evolved CPPNs).

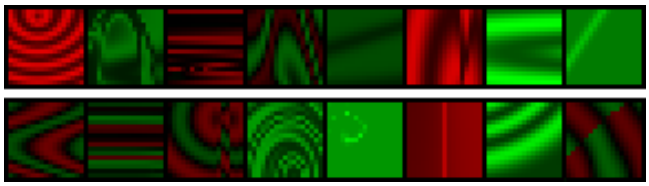

Figure 5. Example kernels from the best individuals of the one-step task (top) and two-step task (bottom).

Recall that one of our central goals is to circumvent the restrictions of supervised learning. The one-step task can be solved using supervised learning (there is an unambiguous correct solution for every one-step task). Multi-step task cannot be solved using supervised learning, as it involves policy-invention (we would have to predetermine how we want the CNN to solve the task, in order to supervise the non-final steps in each task). While our system would likely be outperformed by a supervised learner on the one-step task, it obtains similar performance across the one-step and two-step tasks. This empirically demonstrates our approach' flexibility. Furthermore, once a fit CNN is obtained, it computes a suitable manipulation with a single forward propagation, a feat RL systems cannot match.

VI. CONCLUSION

We have presented a novel approach to training (evolving) CNNs for deformable object manipulation with arbitrary start and end states, using CPPNs in combination with a GA instead of error back-propagation methods, and tested this approach on a thread manipulation task. We found that this approach can successfully solve tasks that require a single manipulation. On tasks requiring two subsequent manipulations we observe partial success. These early results suggest that our approach is viable. Future development will include an input format that avoids ambiguity in the rasterisations of crossed threads, search for a simplified type of CPPN (as the present CPPN format allows for unnecessarily complex patterns), and a restructuring of the GA to assign fitness scores to individual CPPNs (so as to improve efficiency of the evolution process).

Whereas we pursued deformable object manipulation here, the core concept of this work should be equally applicable in other contexts. Optimised CNNs consist of kernels that express spatial patterns, but traditional training methods fail to exploit this fact. By explicitly searching in the space of spatial patterns, it may become quite feasible to produce CNNs by means of free-form search algorithms (such as GAs), and thereby circumvent the restrictions imposed by supervised learning and reinforcement learning algorithms.

REFERENCES

[1] P. Jiménez, "Survey on model-based manipulation planning of deformable objects," *Robotics and Computer-Integrated Manufacturing*, vol. 28, no 2, pp. 154–163, Apr. 2012.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097-1105, 2012.

[3] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning Hierarchical Features for Scene Labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915-1929, 2013.

[4] S. Levine, C. Finn, T. Darrell, P. Abbeel, "End-to-End Training of Deep Visuomotor Policies," arXiv:1504.00702, 2015.

[5] L. Pinto and A. Gupta, "Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours," arXiv:1509.06825, 2015.

[6] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212. Cambridge, MA: MIT Press, 2009.

[7] P. A. Szerlip, G. Morse, J. K. Pugh, and K. O. Stanley, "Unsupervised Feature Learning through Divergent Discriminative Feature Accumulation," arXiv:1406.1833, 2014

[8] N. Cheney, J. Clune, and H. Lipson, "Evolved Electrophysiological Soft Robots," *Proceedings of Artifical Life 14: The Fourteenth International Conference on the Simulation and Synthesis of Living Systems (ALife14)*, MIT Press, 2014.

[9] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "*Intriguing properties of neural networks*," arXiv:1312.6199, 2013.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, Amir Sadik, Ioannis Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature* 518, 529–533 (26 February 2015) doi:10.1038/nature14236

[11] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," arXiv:1412.6806, 2014.

[12] J. Dean, R. Monga, et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," TensorFlow.org, Nov. 2015.