Shortcomings with Tree-structured Edge Encodings for Neural Networks

Gregory S. Hornby

QSS Group Inc., NASA Ames Research Center
Mail Stop 269-3, Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov

Abstract. In evolutionary algorithms a common method for encoding neural networks is to use a tree-structured assembly procedure for constructing them. Since node operators have difficulties in specifying edge weights and these operators are execution-order dependent, an alternative is to use edge operators. Here we identify three problems with edge operators: in the initialization phase most randomly created genotypes produce an incorrect number of inputs and outputs; variation operators can easily change the number of input/output (I/O) units; and units have a connectivity bias based on their order of creation. Instead of creating I/O nodes as part of the construction process we propose using parameterized operators to connect to pre-existing I/O units. Results from experiments show that these parameterized operators greatly improve the probability of creating and maintaining networks with the correct number of I/O units, remove the connectivity bias with I/O units and produce better controllers for a goal-scoring task.

1 Introduction

Neural networks are one of the more common types of controllers used for artificial creatures and evolutionary robotics [1]. Since representations that directly encode the weights and connections of a network have scalability problems indirect representations must be used for larger networks – although to achieve better scalability the indirect representation must allow for reuse of the genotype in creating the phenotype [2]. One type of indirect representation that is becoming increasingly popular for encoding neural networks is to use a tree-structured genotype which specifies how to construct them. Advantages of indirect, tree-structured representations are that they better allow for variable sized networks than directly using a weight matrix, and Genetic Programming style recombination between two trees is easier and more meaningful than trying to swap sub-networks with a graph-structured representation.

One of the original systems for encoding neural networks in tree-structured assembly procedures is cellular encoding [3]. Yet cellular encoding has been found to have shortcomings due to its use of node operators: subtrees swapped through recombination do not produce the same subgraphs because node operators are execution-order dependent and specifying connection weights is problematic since node operators can create an arbitrary number of edges [4]. Consequently, of growing interest is the use of edge-encoding commands in which operators act on edges instead of nodes [5–7].

In this paper we point out three different shortcomings of edge-encoding languages. First, regardless of whether the first N nodes are taken as input/output (I/O) units or if special node-construction commands are used for creating I/O units, when creating an initial population it is difficult to ensure that randomly created genotypes have the correct number of them. A second problem is that as evolution proceeds the variation operators have a high probability of changing the genotype so that it produces networks with incorrect numbers of I/O units. Finally, a more serious problem with tree-structured assembly procedures is the node creation-order connectivity bias (NCOCB). The NCOCB problem is that nodes created from edge operators at the bottom of the genotype will have only a single input and output, whereas nodes created from operators higher up in the genotype will have a connectivity proportional to 2^{height}.

One way to address the problems of producing the correct number of I/O nodes and the NCOCB with I/O nodes is by changing the construction language. Rather than having commands in the language for creating a new I/O unit, or assigning the Nth created unit as the ith I/O unit, we propose starting network construction with the desired number of I/O units and then using parameterized-connection operators for adding edges to these units. Problems in creating and maintaining networks with the correct number of I/O units are reduced since all networks start with the desired number and no commands exist for creating/removing them. Also, parameterized connection commands mean that the expected number of connections for all I/O units is equal for randomly created genotypes and does not suffer from the NCOCB.

In the following sections we first describe a canonical method for using edge encoding operators to represent neural networks as well as our parameterized operators for connecting to I/O units. Next we present our experiments which show the different biases with standard edge-encoding operators and demonstrate that evolution with the parameterized operators for connecting to I/O units produces better controllers on a goal-scoring task. Finally we close with a discussion on the underlying problem with edge operators and tree-structured representations and a conclusion in which we restate our findings.

2 Encoding Neural-Networks

In this section after describing the type of neural networks that we want to evolve we then describe a tree-structured representation for encoding them, followed by two different methods for handling input and output (I/O) units. The first method for handling I/O units uses a standard edge-encoding language (SEEL) and has special commands for creating I/O nodes. Since this method has problems in creating the correct number of I/O nodes and also has a node creation-order connectivity bias (NCOCB) we then describe a second method for handling I/O units. In this second method the initial network starts with the desired number of I/O units and operators in the language connect to them using operator parameters to specify which unit to connect to (PEEL, for parameterized edge-encoding language).

2.1 Neural Network Architecture

The neural networks used in these experiments are continuous-time, recurrent networks similar to those of Beer and Gallagher [8], and of our previous work [9, 10]. Each non-input neuron has an input bias, θ, and a time constant, τ. The activation value of a non-input neuron a_i at time t is:

$$a_{i,t} = \tau_i a_{i,t-1} + (1-\tau) tanh\Big(\sum_j W_{ji} a_{j,t-1} + \theta_i\Big) \quad (1)$$

For input neurons, their activation value is the value of the corresponding sensor.

2.2 Creating a Network from a Tree

The different methods for encoding neural networks or graphs with a tree-structured assembly procedure all start with a single node and edge and then new nodes/edges are added by executing the operators in the assembly procedure. Using an edge-encoding language in which graph-construction operators act on the edge connecting from unit A to unit B, a typical set of commands are as follows.

- `add_reverse` creates a link from B to A.
- `add_split`(n) creates a new neuron, C, with a bias of $\theta = n$, and adds a new link from A to C and creates a new edge connecting from neuron C to neuron B. The bias of this node is set to $\theta = n$, and the time constant is set to zero.
- `add_split_cont`(m, n) acts the same as `add_split()`, only it creates a continuous time neuron with a bias of $\theta = m$ and a time constant of $\tau = n$.
- `connect` creates a new link from neuron A to neuron B.
- `dest_to_next` changes the to-neuron in the current link to its next sibling.
- `loop` creates a new link from neuron B to itself. The `no-op` command performs no operation.
- `set_weight`(n) sets the weight of the current link to n.
- `source_to_next` changes the from-neuron in the current link to its next sibling.
- `source_to_parent` changes the from-neuron in the current link to the input-neuron of the current from-neuron.

Of these commands `add_split`(n) and `add_split_cont`(m, n) have exactly three children commands since after their execution the edge they act on becomes three edges. The `set_weight`(n) command has no children, consequently it is always a leaf node and the `no-op` has either zero or one children so it can be either a leaf node and halt the development of the graph on the current edge, or it can be used to delay execution on the current edge for a round allowing time for the rest of the graph to develop more. The rest of the commands result in the addition of a single new edge to the graph so they have exactly two children commands: one to continue graph construction along the original edge one command to construction along the new edge.

Using the commands described above the sequence of graphs from figure 1b-i shows the construction of a network from the genotype in figure 1.a. Graphs are constructed from this genotype by starting with a single neuron linked to itself, figure 1.b, and executing the commands in the assembly procedure in breadth-first order. First, executing `split(0.01)` adds node `b` with a bias of 0.01 and pair of links, figure 1.c. The command `set-weight(0.1)` sets the weight of the link $\overrightarrow{aa}$ to 0.1, `no-op` performs no operation, and then `split(0.02)` results in the creation of neuron `c` with a bias of 0.02 and two more links, figure 1.d. `Source-to-parent` creates a second link $\overrightarrow{ab}$, and `set-weight(0.2)` sets the weight of the link $\overrightarrow{ba}$ to 0.2, figure 1.e. The second `source-to-parent` command creates the link $\overrightarrow{ac}$, executing `set-weight(0.3)` sets the weight of the link $\overrightarrow{ca}$ to 0.3 and `set-weight(-0.2)` results in a weight of -0.2 assigned to the link $\overrightarrow{ab}$, figure 1.f. The `source-to-next` command results in the link $\overrightarrow{bb}$ being created, figure 1.g. The command `set-weight(0.4)` sets the weight of link $\overrightarrow{bc}$ to 0.4 and then executing `connect` creates an additional link $\overrightarrow{ac}$, figure 1.h. Executing `set-weight(0.5)` sets the weight of link $\overrightarrow{ab}$ to 0.5, `set-weight(0.6)` sets the weight of link $\overrightarrow{bb}$ to 0.6, `no-op` sets the weight of link $\overrightarrow{ab}$ to 0.0, and `set-weight(0.7)` sets the weight of link $\overrightarrow{ac}$ to 0.7, figure 1.i. In addition, after all tree-construction operators have been executed, there is a pruning phase that consolidates the weights of links with the same source and destination neurons, figure 1.j, and removes hidden neurons that are not on a directed path to an output neuron.

2.3 Standard Edge Encoding Language

The graph-construction language of the previous sub-section can be used to create neural networks either by assigning the first n units as I/O units or by adding commands specifically for creating I/O units. Assigning arbitrary units to be I/O units has the drawback that changes in the genotype can add/delete units in the network so that units shift position and what was a the ith input unit in the parent becomes the $i+1$ input unit in the child. To avoid this disruption the SEEL we use has specialized commands for creating I/O units.

I/O units are created through the use of the `add_input` and `output_split`(n) commands. Since these are edge operators, we label the edge they are associated with to be from the vertex `A` to the vertex `B`. Executing the `add_input` command creates a new input unit and an edge connecting from this unit to `A`. Output units are created with the `output_split`(n) command, which performs a `split` on the existing edge and the newly created neuron is set as an output unit with a bias of $\theta = n$.

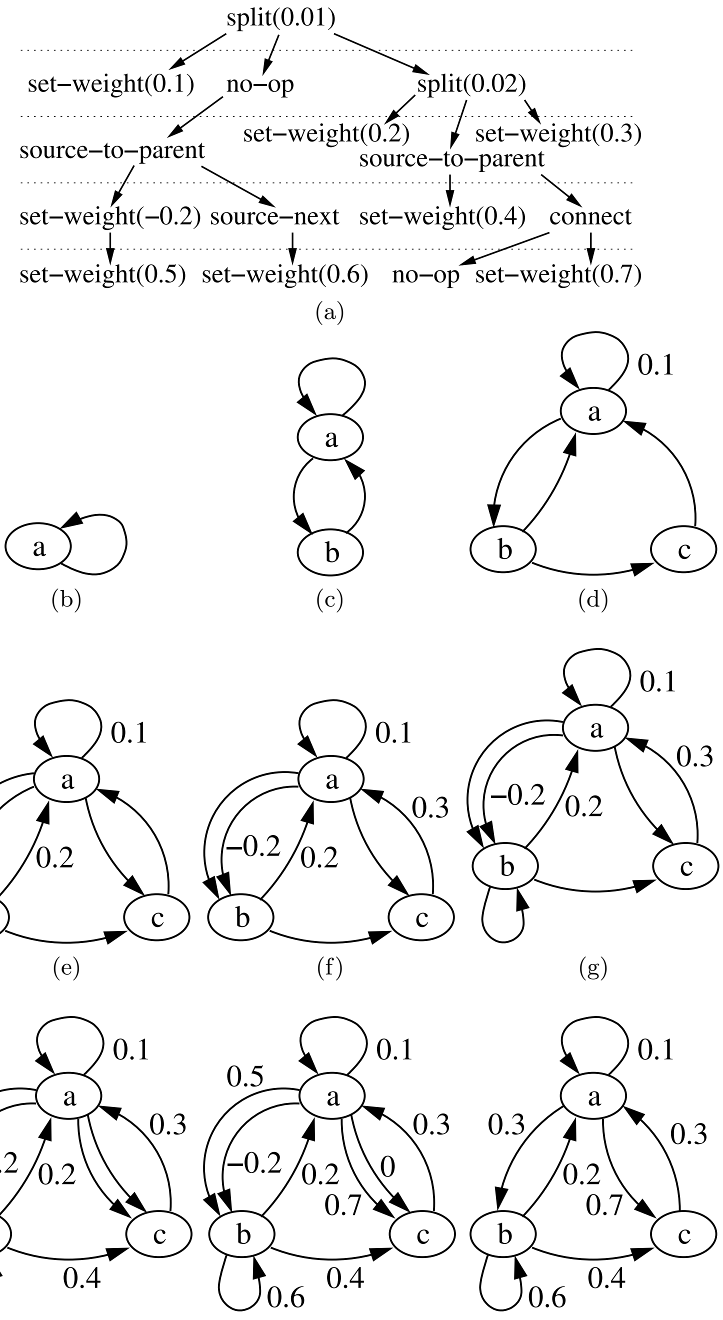


Fig. 1. A tree-structured encoding of a network (a), with dashed-lines to separate the layers, and (b-j) construction of the network it encodes.

2.4 Parametric Edge Encoding Language

A method to remove the connectivity bias with I/O nodes is by having these nodes exist in the initial graph and then adding connections to them, such as with the commands `connect_input(`i`)` and `connect_output(`i`)`. Labeling the current edge as connecting from unit A to unit B, `connect_input(`i`)` creates a link from the ith input neuron to B and `connect_output(`i`)` creates a link from B to the ith output neuron. Since each of these commands creates a new edge, both commands have exactly two children operators: one to continue network construction along the original edge and one to construct along the new edge.

3 Experiments

In this section we present our experiments comparing SEEL with PEEL. First we show that randomly created genotypes using SEEL have problems producing networks with the desired number of I/O neurons whereas this problem is greatly reduced when using PEEL. Next we show that networks encoded with PEEL are more robust to maintaining the correct number of I/O units under mutation and recombination than are networks encoded with SEEL. In our third set of experiments we demonstrate the existence of the node creation-order connectivity bias. Finally, we demonstrate that using PEEL results in better neural-controllers for the evolution of a goal-scoring behavior.

3.1 Initialization Comparison

One benefit of starting with the desired number of I/O neurons is that randomly created, network-constructing, assembly procedures are more likely to have the correct number of I/O neurons. This can be shown by comparing the number of valid networks created using both network construction languages. A network is considered valid if it has four input neurons and four output neurons (arbitrary values selected for this experiment) and for each input neuron there is a path to at least one output neuron and each output neuron is on a path from at least one input neuron. Table 1 shows the number of valid networks created from ten thousand randomly created assembly procedures for various tree depths. From this table it can be seen that valid networks are significantly more likely to be created with PEEL than with SEEL. The reason PEEL does not score 100% even though it starts with the correct number of I/O neurons is because some input neurons may not be on a path to an output neuron.

3.2 Variation Comparison

In addition to the problem of creating initial individuals with the correct number of I/O units, SEELs have difficulty maintaining these numbers under mutation and recombination. To show that PEELs better maintain valid networks we compare the number of networks that still have four inputs and four outputs after mutation and recombination from valid parents.

Depth	≤4	5	6	7	8	9	10	11	12	13
SEEL	0	3	103	183	93	34	13	6	2	0
PEEL	0	0	12	314	1973	4657	6733	8072	8643	8848

Table 1. Number of valid networks generated out of ten thousand randomly created tree-structured assembly procedures.

For this comparison the mutation operator modifies an individual by changing one symbol with another, perturbing the parameter value of a symbol, adding/deleting some symbols, or recombining an individual with itself. Two types of recombination are used, with equal probability of using one or the other. The first recombination operator is the standard GP recombination that swaps random subtrees between parents [11]. The second recombination operator is similar to one-point crossover [12] and we call it matched recombination (MR). MR works by lining up two trees and, starting at the root, matches up the children nodes by type and argument values, finds the locations at which subtrees differ and then picks one of these places at random to swap.

Since random trees of depth seven produced the most valid networks with SEEL, we compared ten thousand mutations and recombinations between SEEL and PEEL on valid, randomly created individuals. With SEEL mutation had a success rate of 84.8% and recombination had a success rate of 79.2%. In comparison, with PEEL mutation produced valid children 93.5% of the time and recombination did so 89.5% of them. These results show that networks encoded with a PEEL are more robust to variation operators.

3.3 Node Creation Order Connectivity Bias

A more serious problem with tree-structured assembly procedures is the node creation-order connectivity bias (NCOCB). Nodes created from commands early in the construction process tend to have a greater number of edges into and out of them then nodes created later in the the process. One consequence of this bias is that I/O neurons that are created early in the construction process will have a significantly higher number of outputs/inputs than those I/O neurons created at the end of the construction process.

The graph in figure 2.a shows the average connectivity (sum of inputs and outputs) of a node plotted against its creation order. From this graph it can be seen that nodes created earlier in the construction process have more connections than those created later and most nodes only have two connections: one input and one output link. Thus if I/O nodes are created by the tree-structured assembly procedure, the first I/O nodes will have significantly more inputs/outputs from/to them than those created later in the construction process. For input neurons, this suggests that the first inputs are likely to have a greater influence on the behavior of the network than the latter inputs and for output neurons this suggests that more processing of inputs is happening for the activation values of the first output neurons than for the latter output neurons.

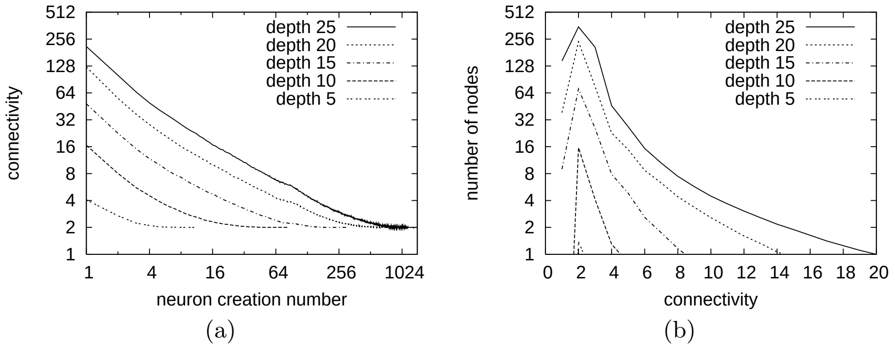


Fig. 2. Graphs of (a) the average node connectivity by order of creation, and (b) the number of nodes with a given connectivity for randomly created individuals.

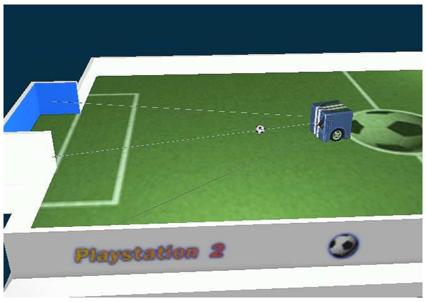
Because the connectivity of a node is strongly biased by its height in the tree-structured assembly procedure and since most commands are near the leaves in the tree this results in a bias in the distribution of the number of nodes with a given connectivity. Most nodes in the network will have a connectivity of two – one input and one output – and the number of nodes with a given connectivity decreases exponentially (figure 2.b).

3.4 Comparison on Evolving a Goal-Scoring Behavior

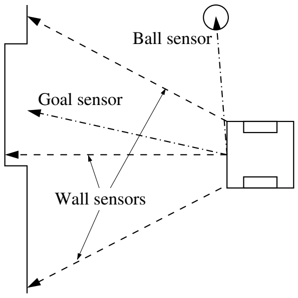
While PEEL has been shown to be better than SEEL for removing various biases, of practical importance is whether evolution with PEEL produces better controllers than evolution with SEEL. To test this we evolve neural-controllers for a goal-scoring task.

Goal-scoring takes place in a computer-simulated, 275x152.5 walled, soccer field with goals at each end (figure 3.a). Inside the soccer field is a two-wheeled, soccer player which has seven sensor inputs (three sensors to detect distance to the wall (one pointing directly in front and the other two at 30° to the left and right), and four sensors that return angle to the ball, distance to the ball, angle to the goal and distance to the goal) and two outputs (desired wheel-speed for the left and right wheels) (figure 3.b).

Evaluating an individual consists of summing the score from eight trials, two each with the ball initially placed in each of the four corners of the field, and the soccer-player placed in the middle of the field. Initial locations for both the player and ball are perturbed by a small random amount and then the player is given 60 seconds (at 30fps this results in 1800 network updates) to score as many goals as it can. For each goal scored the distance from the vehicle's starting position to the ball plus the distance from the ball's initial location to the goal is added to the network's score. After a goal is scored the ball is randomly located at the center of the field (±30, ±30), the minimum distances to the ball and to



(a)



(b)

Fig. 3. (a) The soccer field and (b) the soccer player and its sensors.

the goal are reset, and the network is allowed to try to score another goal. Once time runs out, a network's score is increased by how much closer it moved the player to the ball and how much closer it moved the ball to the goal. In addition, if the player scores an own-goal, its score is reduced by the distance it moved the ball from its starting position to the goal.

To perform these experiments the EA was run on a Linux-PC with evaluations farmed out to five PlayStation® 2[1] development systems. Each experiment consisted of evolving fifty individuals for fifty generations. A generational EA was used and new individuals were created with equal probability of either mutation or recombination and an elitism of three. Evaluating one generation of fifty individuals took approximately four minutes. The results of experiments are shown in figure 4 and show that evolution with PEEL produces soccer players almost twice as fit as with SEEL.

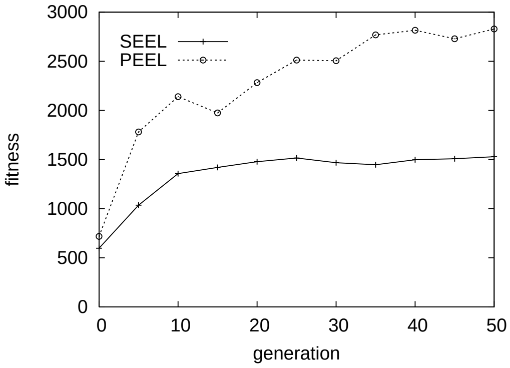


Fig. 4. Fitness of the best evolved goal-scores averaged over four trials.

[1] PlayStation is a registered trademark of Sony Computer Entertainment Inc.

The higher fitness of networks encoded with PEEL is reflected in the behaviors produced by these networks. Networks encoded with SEEL produced soccer players that tended to spin in place and move awkwardly or in a looping pattern. These networks only moved toward the ball somewhat haphazardly and generally did not appear to be aiming their shots. In contrast, networks encoded with PEEL would move to position themselves on the other side of the ball from the goal and then either push the ball toward the goal or spin to kick it toward the goal. The best of these networks seldom missed in its shots and an example of its behavior is shown in the sequence of images in figure 5.

4 Discussion

While the results of the experiments section show that various biases hold for the edge-encoding languages presented here, of interest is the degree to which these biases exist in other edge-encoding languages. The edge-encoding language of section 2 differs from Luke's [4] in that edges are not explicitly deleted, rather they disappear if they are not assigned a weight, and the split command does not delete the link $\overrightarrow{ab}$ when it creates the new neuron c and links $\overrightarrow{ac}$ and $\overrightarrow{cb}$. A command for explicitly deleting links would not necessarily change the biases in resulting networks since the `no-op` command with no children has the same effect. In contrast, since the split operator used here adds links to existing neurons without removing any, it should produce a larger bias than Luke's split operator. Although the differences in operators between different edge encoding languages affect the degree of connectivity bias that can be expected, the main cause of the biases is the tree-structure of the representation. When a neuron is created it has a single input and output edge. Since edge operators can add one input or output to an existing neuron (except for the `loop` command, which adds both an input and an output) the expected connectivity of a neuron is on the order of 2^{height}.

Since PEEL only addresses the NCOCB for I/O units and does not scale for large networks the direction to go for addressing the various shortcomings of edge encodings is not clear. One way to remove the NCOCB is to change from tree-structured to graph-structured genotypes, but then there are difficulties in creating meaningful recombination operators. Another way is to switch to operators in which the connectivity of a new node is not dependent on its depth in the genotype; but these would be node operators which have their own shortcomings [4].

5 Conclusion

In this paper we identified three shortcomings with typical edge encoding operators for representing neural networks: individuals created at random in the initialization phase do not usually have the correct number of inputs/outputs; variation operators can easily change the number input/output neurons; and the node creation-order connectivity bias (NCOCB). To address these problems we

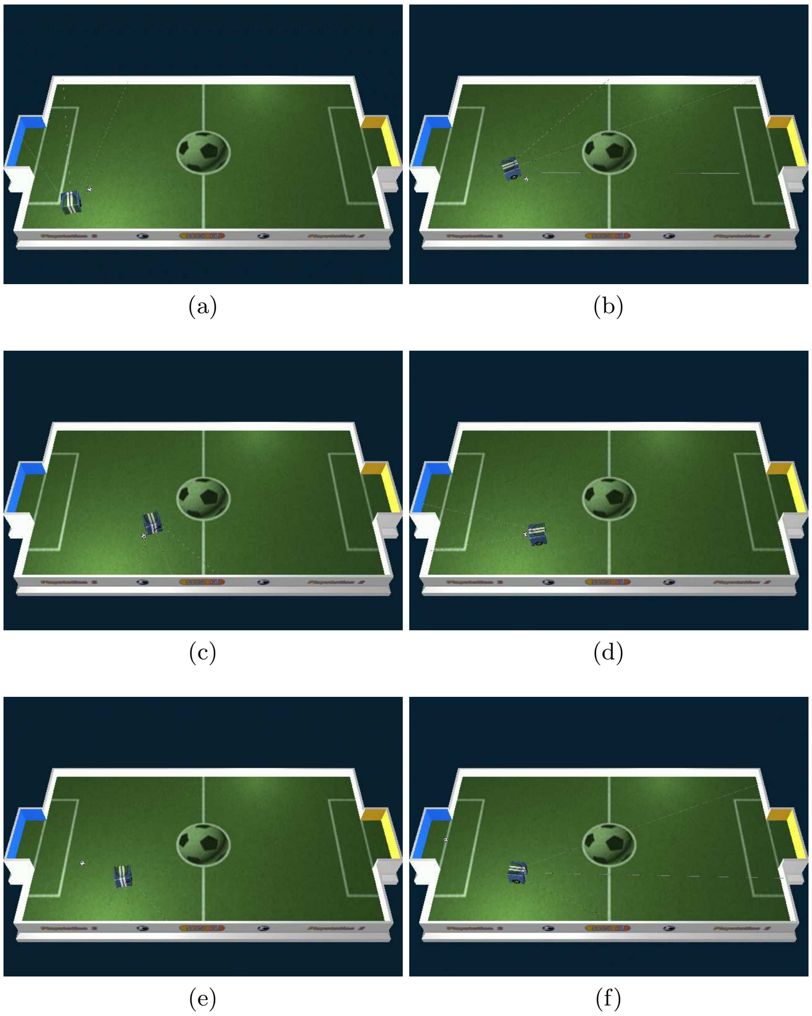


Fig. 5. An evolved goal-scorer in action: (a)-(c) the soccer-player circles around the ball; (d) it pushes the ball toward the goal; (e)-(f), while the ball is going into the goal the player moves to the center of the field where the ball will re-appear after the goal is scored.

proposed using parameterized operators for connecting to input/output units and demonstrated that evolution with these operators produces better neural networks on a goal-scoring task. While these parameterized operators greatly improve the probability of creating and maintaining networks with the correct number of input/output units it does not address the NCOCB problem for hidden units. Consequently the contribution of this paper is more an observation that these shortcomings exist. Future work with edge encoding operators will need to address more general solutions to these problems that scale with the size of the network and work for hidden units.

Acknowledgements

Most of this research was conducted while the author was at Sony Computer Entertainment America, Research and Development and then at Brandeis University. The soccer game and simulator was developed by Eric Larsen at SCEA R&D.

References

1. Nolfi, S., Floreano, D., eds.: Evolutionary Robotics. MIT Press, Cambridge, MA (2000)
2. Hornby, G.S., Pollack, J.B.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life **8** (2002) 223–246
3. Gruau, F.: Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. PhD thesis, Ecole Normale Supérieure de Lyon (1994)
4. Luke, S., Spector, L.: Evolving graphs and networks with edge encoding: Preliminary report. In Koza, J., ed.: Late-breaking Papers of Genetic Programming 96, Stanford Bookstore (1996) 117–124
5. Brave, S.: Evolving deterministic finite automata using cellular encoding. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) 39–44
6. Hornby, G.S., Pollack, J.B.: Body-brain coevolution using L-systems as a generative encoding. In: Genetic and Evolutionary Computation Conference. (2001) 868–875
7. Koza, J., Bennett, F., Andre, D., Keane, M.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann (1999)
8. Beer, R.D., Gallagher, J.G.: Evolving dynamical neural networks for adaptive behavior. Adaptive Behavior **1** (1992) 91–122
9. Hornby, G.S., Mirtich, B.: Diffuse versus true coevolution in a physics-based world. In Banzhaf, W., et al., eds.: Proc. of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann (1999) 1305–1312
10. Hornby, G.S., Takamura, S., Hanagata, O., Fujita, M., Pollack, J.: Evolution of controllers from a high-level simulator to a high dof robot. In Miller, J., ed.: Evolvable Systems: from biology to hardware; Proc. of the Third Intl. Conf. Lecture Notes in Computer Science; Vol. 1801, Springer (2000) 80–89
11. Koza, J.R.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, Mass. (1992)
12. Poli, R., Langdon, W.B.: Schema theory for genetic programming with one-point crossover and point mutation. Evolutionary Computation **6** (1998) 231–252