NEAT in HyperNEAT Substituted with Genetic Programming

Zdeněk Buk, Jan koutník, and Miroslav Šnorek

Computational Intelligence Group
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
`bukz1|koutnij|snorek@fel.cvut.cz`,
WWW home page: `http://cig.felk.cvut.cz`

Abstract. In this paper we present application of genetic programming (GP) [1] to evolution of indirect encoding of neural network weights. We compare usage of original HyperNEAT algorithm with our implementation, in which we replaced the underlying NEAT with genetic programming. The algorithm was named HyperGP. The evolved neural networks were used as controllers of autonomous mobile agents (robots) in simulation. The agents were trained to drive with maximum average speed. This forces them to learn how to drive on roads and avoid collisions. The genetic programming lacking the NEAT complexification property shows better exploration ability and tends to generate more complex solutions in fewer generations. On the other hand, the basic genetic programming generates quite complex functions for weights generation. Both approaches generate neural controllers with similar abilities.

1 Introduction

In training of artificial neural networks using evolutionary algorithms a method of encoding the neural networks into individuals in the evolution is needed. Basically, there are two types of encodings. Direct encoding of either connection weights or a network structure causes individuals and therefore the search space complexity growth. Indirect encoding that utilizes a system, which develops the neural network from information encoded by the individual can overcome such drawback of direct encoding.

One of the most perspective algorithm of neural network weights and structure encoding is the hypercube encoding invented by Ken Stanley [2, 3] as HyperNEAT algorithm. The HyperNEAT algorithm consists of a function or a set of functions, which generates weights for neural network. The neural network consist of neurons placed in a rectangular mesh called substrate. The substrate coordinates serve as inputs for the function. The function output is the weight of connection between two neurons. In the original HyperNEAT algorithm, the function is called CPPN (Compositional Pattern Production Network) and is

constructed from a set of nodes with scalar product on their inputs and non-linear function at their outputs. The network structure reminds a neural network (therefore, it is named network rather than a function). The CPPN in the HyperNEAT is generated using the NEAT (NeuroEvolution of Augmenting Topologies) [3] algorithm. NEAT is a type of evolutionary algorithm, which evolves the network from a simple form featuring complexification and niching.

The NEAT algorithm is the component that we decided to replace with a different style of weights encoding and generation in our approach. Rather, we use genetic programming, which generates functions that compute weights for connections among neurons in the substrate.

The application domain is a control of autonomous agents in simulated environment. The agents are equipped with sensors with scalable resolution. Encoding of neurons weights allows the resolution of the sensory input to be changed independently of the size of the individual that contains the weight generating function. The previous experiments presented in [4] show that the NEAT can produce recurrent neural network, which can control the agent to move through the simulated environment with maximum average speed. The fitness of the evolved neural network is the average speed of the controlled agent. Our goal is to replace the NEAT algorithm with genetic programming and compare it to the original HyperNEAT algorithm.

1.1 Related Work

Many techniques for evolution of either weights or structure of neural networks were already developed such as Analog Genetic Encoding [5–7], Continual Evolution Algorithm [8], GNARL [9], Evolino [10] and NeuroEvolution of Augmenting Topologies (NEAT) [11]. The NEAT algorithm became a part of the HyperNEAT algorithm as a tool for evolution of CPPNs.

HyperNEAT algorithm was already applied to control artificial agents in food gathering problem [2]. It was shown that HyperNEAT is capable of large scale networks evolution ($> 8 \cdot 10^6$ connections). The simulated agent was equipped with concentric sensors for food in particular directions linked with effector, which drives the agent to the direction. In our approach, the sensors are organized in polar rays with particular angular and distance resolution. The sensors are sensitive to the surface color.

The agents can share portion of one substrate together [12]. The substrate splits to local but linked areas. The agents can exploit cooperative behavior afterwards.

Agents can complete common goals also with a minimum information from the sensors with evolutionary trained feed forward networks as well [13]. In the case, the agents exhibit reactive behavior.

This paper is organized as follows. Section 2 describes the HyperGP algorithm. Section 3 describes the simulation environment and the agent setup. Section 4 describes the experimental results, performance of HyperGP is compared with HyperNEAT. Final section concludes the paper.
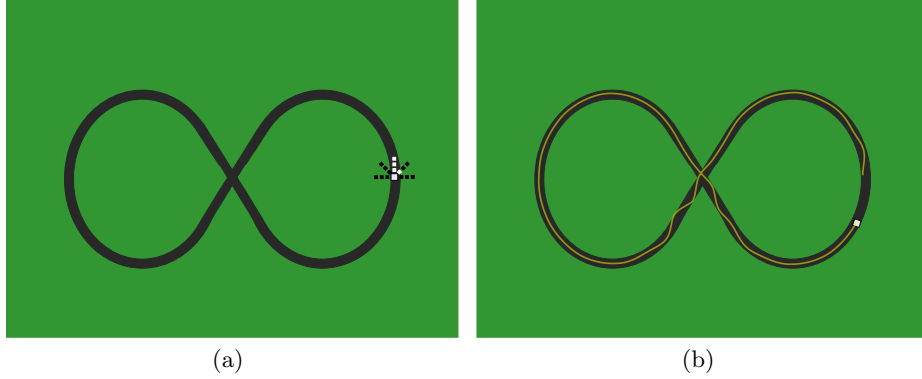
(a)   (b)

Fig. 1. Experimental scenario with an agent placed in starting positions (a). The agent has 3x5 sensors array. The color of the sensor represents a surface friction mapped to 0 and 1 for the neural network input. Track of the trained agent is depicted in (b).

2 HyperGP Algorithm

2.1 Genetic Programming

Our implementation of GP uses recombination and mutation operators and direct representation of mathematical expressions. The final expression is a function of up to 4 variables (x_1, y_1, x_1, y_2), which was named CPPF (Compositional Pattern Production Function). During the evolution the expressions are generated from the sets of elementary functions and atoms (variables and constants). The operators are:

- **Random expression generator** generates expressions with defined depth. The functions list (patterns) defines the set of basic function that the final expressions will be composed of. The atoms list contains the variables (input variables for the final expressions/functions) and constants. At the beginning the first function is randomly chosen from the list and then the generator is applied recursively on this expression - in this case not only functions but also the atoms are used in random selection. The maximal depth of the expression can be specified, so this value is decremented whenever the generator is recursively run on some subexpression.
- **Mutation** selects random place/subexpression in the given expression and replaces the subexpression by some randomly generated one (with respect to the maximum depth of the expression). It splits the expression into particular subexpressions, then randomly chooses one of them, and using the random expression generator replaces it by some random expression. The depth parameter is also used, so the final (mutated) expression fulfills the maximal depth condition.
- **Recombination** combines two expressions. Crossover position is selected randomly in both expressions. It generates the positions of all subexpressions

in two given parent expressions (individuals). Then it replaces the subexpression in the first individual by the subexpression from the second individual. So it keeps the positions and swaps the subexpressions. The maximal depth condition is still fulfilled.

2.2 Evolution

We are working with the fixed size population of individuals (expressions). The GP algorithm generates in each iteration the set of offsprings (the number of the offsprings does not depend on the size of the population) using the mutation and recombination operators. The parent population and the offspring population is then joined into one set (pool) and using the selection function (based on the fitness value of the individuals) the new population is created (all the unused individuals are deleted).

3 Experimental Setup

3.1 Simulation Environment

Experiments with the agents were performed in simulation environment called ViVAE (Visual Vector Agent Environment) featuring easy design of simulation scenarios in SVG vector format [4]. There are two types of surfaces in the simulation (road and grass) with different frictions. The grass has friction 5 times higher than the road.

ViVAE supports number of different agents equipped with various sensors for surfaces and other objects in the scenario. In the current experiment, scenario with one agent was user, see Figure 1.

3.2 Agent Setup

The agent in the simulation is controlled by neural network controller constructed by the HyperGP or HyperNEAT. The agent is driven by two simulated wheels and is equipped with a number of sensors. The controlling neural network is organized in a single layer of possibly fully interconnected perceptron (global) type neurons (neurons compute biased scalar product, which is transformed by bipolar logistic sigmoidal function). Steering angle is proportional to inverse actual speed of the robot.

The sensors as well as the neural network are spread in a substrate. Neurons and sensors are addressed with polar coordinates, see Figure 2. Two of the neurons in the output substrate are dedicated to control acceleration of the wheels. The neurons are marked with the red color in Figure 5.

Each individual in the evolution contains three different CPPFs, see Figure 2. Function f_i generates weights between input sensors an the neurons, function f_b computes biases of the neurons and function f_o expresses connection weights among the neurons in the upper layer. Since the bias is a property of a neuron in the output substrate, inputs x_2 and y_2 are setup to 0 for f_b computation.
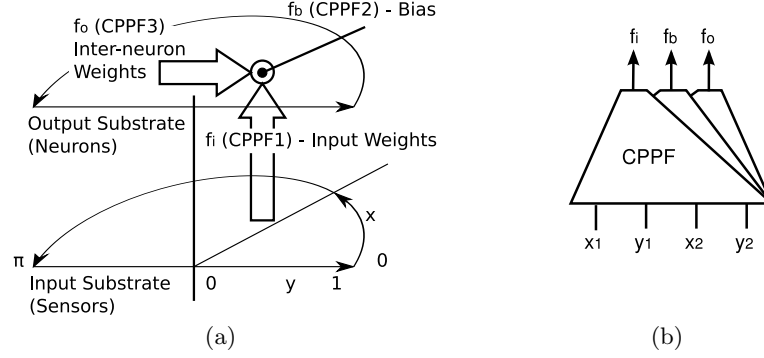
(a)　　　　　　　　　　　　　　(b)

Fig. 2. Organization of the HyperGP substrate. There were two distinct substrates used (a) and three different GP function trees (CPPF) were evolved. Function f_i is weight between input substrate (sensors) and substrate with the neurons. Second function f_b generates bias for neurons in the upper substrate. For bias calculation third and fourth CPPF inputs are set to 0. Last function f_o represents connection weights among neurons in the upper substrate.

3.3 HyperGP Setup

The HyperGP algorithm described in Section 2 was executed with the following set of functions, from which the functions are selected.

$$x + y, \quad x \cdot y, \quad \sin x, \quad \cos x, \quad \tan^{-1} x, \quad \sqrt{|x|}, \quad |x|, \quad e^{-x^2}, \quad e^{-(x-y)^2} \quad (1)$$

The list of atoms used in the functions is the following one:

$$x_1, \quad x_2, \quad x_3, \quad x_4, \quad -1, \quad \text{RandomReal(-5,5)} \quad (2)$$

Besides the CPPF inputs, there is extra negative multiplier and random constant between -5 and 5. Depth of the expression was set up to 3.

3.4 HyperNEAT Setup

We have used our own implementation of the HyperNEAT algorithm. The NEAT part resembles Stanley's original implementation. The HyperNEAT extension is inspired mainly by the David D'Ambrosio's HyperSharpNEAT[1]. Following function have been used as output function of the NEAT nodes:

$$\frac{2}{1 + e^{-4.9\,x}} - 1, \quad x, \quad e^{-2.5\,x^2}, \quad |x|, \quad sin(x), \quad cos(x) \quad (3)$$

---

[1] Both Stanley's original NEAT implementation and D'Ambrosio's HyperSharpNEAT can be found on http://www.cs.ucf.edu/ kstanley.

The parameter settings are summarized in Table 1. Note, that we have extended the original set of constants which determine the genotype distance between two individuals (C_1, C_2 and C_3) by the new constant C_{ACT}. The constant C_{ACT} was added due to the fact that, unlike in classic NEAT, we evolve networks (CPPNs) with heterogeneous nodes. C_{ACT} multiplies the number of not matching output nodes of aligned link genes. The CPPN output nodes were limited to bipolar sigmoidal functions in order to constrain the output.

Table 1. HyperNEAT parameters

Parameter	Value
population size	100
CPPN weights amplitude	3.0
CPPN output amplitude	1.0
controller network weights amplitude	3.0
distance threshold	15.0
distance C_1	2.0
distance C_2	2.0
distance C_3	0.5
distance C_{ACT}	1.0
mating probability	0.75
add link mutation probability	0.3
add node mutation probability	0.1
elitism per species	5%

4   Experimental Results

4.1   HyperGP with Mutation Only

All experimental results are collected from 10 runs of each algorithm. The HyperGP was executed 2×10 times. In the first set, the mutation only was used as the genetic operator. The convergence of the HyperGP algorithm is plotted in Figures 3. Sub-figure (a) contains convergence plots for the 10 experiment runs. Sub-figure (b) contains plot of the whole population in one experiment run (50 iterations). The individuals are sorted according to their fitness (from left to right). We consider the fitness of 0.83 to be enough for the robot to follow the road. HyperGP with mutation only reaches that fitness in 20 generations (median). Mean fitness reached in the 10 runs is 0.875.

Following set of functions is the one generated in one of the experiment run:

$$f_b = e^{-\left(e^{-\frac{1}{2}(x_2-x_3)^2}-x_2\right)^2} \tag{4}$$

$$f_o = \sin x_1 (x_3 + \sin x_4) \tag{5}$$

$$f_i = \exp\left(\left(\left(e^{-\left(e^{-e^{-2\left(x_2 - \tan^{-1}(x_4)\right)^2} + x_3 + x_4}\right)^2} - |x_1|\right)\right)^2\right) \quad (6)$$

4.2 HyperGP with Crossover

In the second set of runs, the crossover operator was added (see Figure 4). Probability of the crossover is 0.75, probability of the mutation is 0.25. We can see that the performance of the algorithm has decreased. The target fitness (0.83) was reached in 7 out of 10 runs only. Algorithm convergence is slowed down. Average fitness reached after 50 evolution generations is of 0.82. Additional increasing of the crossover probability decreased the performance of the algorithm.
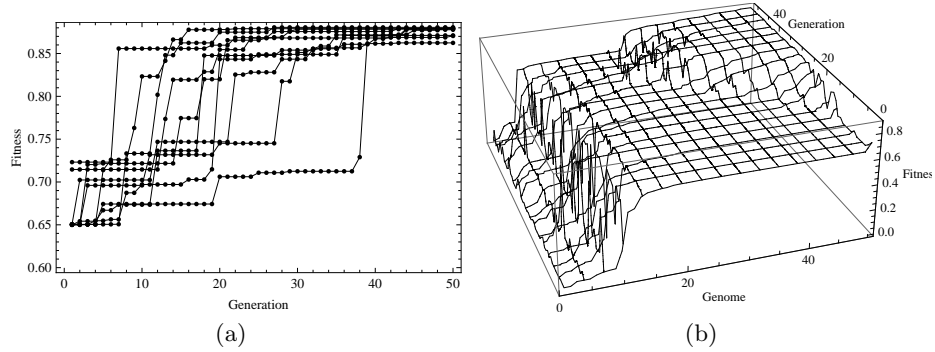

(a)   (b)

Fig. 3. Convergence of HyperGP algorithm with mutations only. Plot (a) contains linearly interpolated convergence of 10 independent experimental runs. Figure (b) displays how the genomes with the fitness approaching the local optimum are growing in the population. All 50 individuals are sorted by their fitness. Solution with fitness of 0.65 is very common in the initial population. In the populations after 30 generations solution with fitness close to 0.88 spreads in the population.

4.3 HyperNEAT

The HyperNEAT was setup according to Table 1. The algorithm was executed 10 times. Convergence of the algorithm is plotted in Figure 6. The red line in the figure appears in 50th generation, in which the HyperGP was stopped. The target fitness of 0.83 was reached in 92th generation (median). We can observe that the HyperGP algorithm outperforms the HyperNEAT in the speed of the convergence.
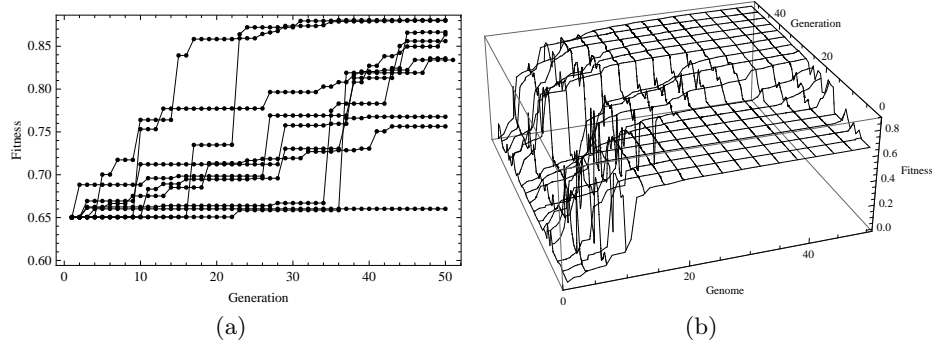
Fig. 4. Convergence of HyperGP algorithm with crossover probability of 0.75 and mutation probability of 0.25. Plot (a) contains linearly interpolated convergence of 10 independent experimental runs. Figure (b) displays how the genomes with the fitness approaching the local optimum are growing in the population. All 50 individuals are sorted by their fitness.
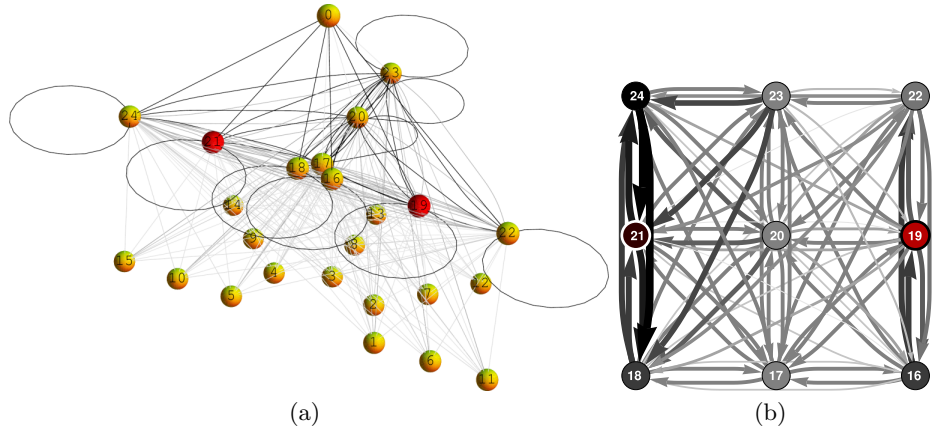

Fig. 5. Visualization of the evolved neural network controlling the agents. Figure (a) displays the complete network including inputs. Layer of 3×3 neurons is depicted in figure (b). The network consists of two layers. The bottom layer represents input sensors in a grid of 3x5 inputs. The upper layer contains 9 (3 × 3) neurons. The neurons are mapped into a substrate in polar coordinates to match shape of the input substrate. Red spheres represent neurons (numbers 19 and 21) that steer the agent wheels. The most upper sphere represents bias for the neurons. Connections are displayed with lines. The most visible lines represent connections with stronger synaptic values. The neurons in the neuron layer are emphasized in figure (b). The weights of connections between the neurons are represented by a thickness of arrows. Recurrent weight of a neuron is represented by a color of the disk representing the neuron. Darker neuron has a stronger recurrent self-connection.
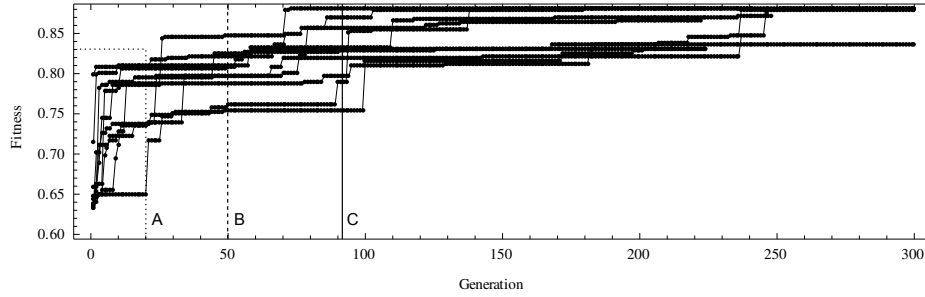
Fig. 6. Convergence of HyperNEAT algorithm in 300 generations performed. The HyperGP algorithm was stopped after 50 generations (dashed line - B) and reached target fitness (0.83) in after 20 generations (median, dotted line - A). The HyperNEAT algorithm reached the fitness of 0.83 after 92 generations (median, solid line - C).

5 Conclusion and Future Work

In this paper we present results of experiment with generation of recurrent neural network weights using hyper-cubic encoding and multidimensional functions generated by genetic programming. We have replaced the authentic NEAT algorithm in the original HyperNEAT by genetic programming, the algorithm was named HyperGP. The neural networks were used as controllers for mobile agents in simulated environment. Fitness function of the particular agents is the average speed of the agent, which forces it to drive on road.

Both HyperNEAT and HyperGP generate suitable solution with agents moving with possibly maximum speed through the scenario, following the roads. HyperNEAT algorithm utilizes it's complexification feature and the resulting CPPNs are simple but the evolution reaches the desired fitness after 92 generations. HyperGP algorithm with same size of population reaches the same fitness in 20 generations.

The HyperGP algorithm has a better explorative property and generates more complex functions in early state of the evolution. The HyperNEAT algorithm sets up the weights in the CPPN in the early evolution phase. More CPPN nodes are added in the latter evolution phase. Both algorithms require more testing and some tuning is required as well.

Both mutation and crossover operators were tested. The best results were reached with mutation operator only. The crossover operation causes major changes in the function. The mutation operator is more tender to the function structure than the crossover. Besides, the problem is sensitive to the CPPF structure and combination of two functions together can completely change the function output within the desired range.

The future work will involve experiments with different presets of both approaches to generation of the weights functions as well testing on different scenarios and goals in the mobile agents.

Acknowledgment

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992)
2. D'Ambrosio, D.B., Stanley, K.O.: A novel generative encoding for exploiting neural network sensor and output geometry. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, New York, NY, USA, ACM (2007) 974–981
3. Gauci, J., Stanley, K.: Generating large-scale neural networks through discovering geometric regularities. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, New York, NY, USA, ACM (2007) 997–1004
4. Drchal, J., Koutník, J., Šnorek, M.: Hyperneat controlled robots learn to drive on roads in simulated environment. In: Submitted to IEEE Congress on Evolutionary Computation (CEC 2009). (2009)
5. Mattiussi, C.: Evolutionary synthesis of analog networks. PhD thesis, EPFL, Lausanne (2005)
6. Dürr, P., Mattiussi, C., Floreano, D.: Neuroevolution with Analog Genetic Encoding. In: Parallel Problem Solving from Nature - PPSN iX. Volume 9 of Lecture Notes in Computer Science. (2006) 671–680
7. Dürr, P., Mattiussi, C., Soltoggio, A., Floreano, D.: Evolvability of Neuromodulated Learning for Robots. In: The 2008 ECSIS Symposium on Learning and Adaptive Behavior in Robotic Systems, Los Alamitos, CA, IEEE Computer Society (2008) 41–46
8. Buk, Z., Šnorek, M.: Hybrid evolution of heterogeneous neural networks. In: Artificial Neural Networks - ICANN 2008. Volume 5163., Springer Berlin / Heidelberg (2008) 426–434
9. Angeline, P.J., Saunders, G.M., Pollack, J.B.: An evolutionary algorithm that constructs recurrent neural networks. IEEE Transactions on Neural Networks **5** (1993) 54–65
10. Schmidhuber, J., Wierstra, D., Gagliolo, M., Gomez, F.: Training recurrent networks by evolino. Neural computation **19**(3) (March 2007) 757–779
11. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10** (2002) 99–127
12. D'Ambrosio, D.B., Stanley, K.O.: Generative encoding for multiagent learning. In: GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation, New York, NY, USA, ACM (2008) 819–826
13. Waibel, M.: Evolution of Cooperation in Artificial Ants. PhD thesis, EPFL (2007)