Farhad M. Kazemi

Reference Article: Khan, Maryam Mahsal, et al. "Fast learning neural networks using Cartesian genetic .programming " *Neurocomputing* 121 (2013): 274-289

a) Cartesian Genetic Programming of Artificial Neural Network (CGPANN) algorithms (Forward CGPANN and Recurrent CGPANN) investigated to check their capabilities as follows.

One of the advantages of NeuroEvolution (NE) is that it can be applied to reinforcement learning type tasks. These are tasks where there is no list of desirable input-output pairs, only a single figure of merit of the overall performance of a solution. As standard training methods for Artificial Neural Network (ANN)s, such as back propagation, cannot be applied to reinforcement learning tasks, only supervised learning, control benchmarks serve to showcase one of the advantages of NeuroEvolution methods. That is to say, they can apply ANNs to a whole new domain of tasks.

Pole Balancing (dataset)

The pole balancing benchmarks have been used for many years in the Artificial Intelligence (AI) and control theory literatures and also, represent a range of difficult multiple-input single-output control tasks. The task is to balance one or more poles on a moveable cart by applying a horizontal force. The most widely used pole balancing benchmarks are single pole balancing and double pole balancing; Figures 1.a and 1.b respectively. Single pole balancing problem consists of a pole while double pole balancing requires balancing two poles that attached by a hinged to a wheel cart. The track of the cart is limited to $-2.4 < x < 2.4$. The objective is to apply force F to the cart where the angle of the pole doesn't exceed ($-12^\circ < \vartheta < +12^\circ$) for the single pole and ($-36^\circ < \vartheta < +36^\circ$) for the double poles that the cart doesn't leave the track. The controller has to balance the pole(s) for approximately 30 minutes which corresponds to 100,000 time steps. Thus the neural network must apply force to the cart to keep the pole(s) balanced for as long as possible. If the output of the CGPANN is greater than or equal to zero then a force F equal to +10N will be applied on the cart and if it is less than zero then a force 'F' equal to -10N is applied. The system inputs are the pole-angles (ϑ_i), angular velocity of pole ($\dot{\theta}_t$), position of cart (x) and velocity of cart ($\dot{x}$). The equations which describe the dynamics of the pole-cart system are given in Equations 1-4 with the symbol definitions and commonly used constants given in Tables 1 and 2. These equations are used to compute the effective mass of the poles, acceleration of the poles, acceleration of cart, and effective force on each pole.

$$\hat{m}_i = m_i\left(1 - \frac{3}{4}\cos^2\theta_i\right) \tag{1}$$

$$\ddot{\theta}_i = -\frac{3}{4l_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i}\right) \tag{2}$$

$$\ddot{x} = \frac{F - \mu\,\mathrm{sgn}(\dot{x}) + \sum_{i=0}^{N}\hat{F}_i}{M + \sum_{i=0}^{N}\hat{m}_i} \tag{3}$$

$$\hat{F}_i = m_i l_i \theta_i^2 \sin\theta_i + \frac{3}{4}m_i\cos\theta_i\left(\frac{\mu_{pi}\theta}{m_i l_i} + g\sin\theta_i\right) \tag{4}$$

===

b) NeuroEvolution

NeuroEvolution (NE) is a sub field of Machine Learning (ML) which combines both Evolutionary Algorithm (EA)s and Artificial Neural Network (ANN)s. NE is the application of EAs to the training of ANNs. This section describes artificial evolution on ANNs known as Neuroevolution (NE). The term NE refers to evolution of different attributes of a neural network. It is achieved through combination of an ANN with a genetic algorithm, with the network working as the phenotype and the genetic algorithm acting on the corresponding genotype. The genotype can include connection weights, connection type, node function or even the topology of the network. The genotype is evolved until the desired phenotypic behaviour is obtained. Since the choice of encoding affects the search space of solutions, it is an important part of the design of any NE system. Some methods evolve only weights of the network, some topology and some evolve both. Actually, there are a number of possible advantages of using NeuroEvolution to train ANNs. These advantages are summarised as follows: -Suited to supervised and reinforcement learning applications -Capable of training feed-forward and recurrent ANNs -Capable of manipulating ANN topology -No transfer function limitations -Capable of creating homogeneous and heterogeneous ANNs -Always returns a solution.

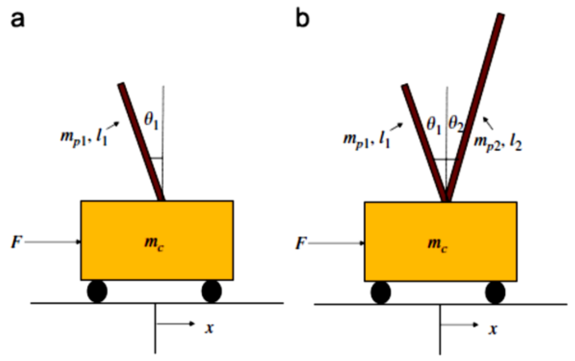


Fig. 1. (a) Single pole balancing, (b)double pole balancing

Table 1
Parameters for single pole balancing task.

Parameters	Value
Mass of cart (m_c)	1 Kg
Mass of Pole (m_{p1})	10 Kg
Length of Pole (l_1)	0.5 m
Width of the Track (h)	4.8 m

Table 2
Parameters for double pole balancing task.

Parameters	Value
Mass of cart (m_c)	1 Kg
Mass of Poles (m_{p1},m_{p2})	0.01,10 Kg
Length of Poles (l_1,l_2)	0.05,0.5 m
Friction of the Poles (μ_p)	0.000002
Friction of the cart (μ_c)	0.0005
Width of the Track (h)	4.8 m

Cartesian Genetic Programming (for details refer to reference)

Although CGP has not been adopted to the same extent as the more popular tree-based GP, it has a number of advantageous properties over tree-based GP which often make it a suitable alternative:

-CGP does not suffer from program bloat. -CGP greatly benefits neutral genetic drift. -CGP is naturally suited to MIMO tasks. -CGP allows internally calculated values to be reused.

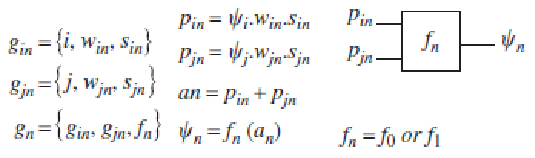
Cartesian Genetic Programming of Artificial Neural Networks

Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) is a NeuroEvolution (NE) method based on the application of Cartesian Genetic Programming (CGP) to the training of Artificial Neural Network (ANN)s. The idea of CGPANN method is to encode neural network parameters i.e. topology, weight and function as CGP genotype.

1) Feed forward Cartesian genetic programming evolved ANN (FCGPANN)

The genotype consists of nodes, where each node corresponds to a neuron of ANN. The node includes inputs, weights, connection and function. Inputs can be program inputs or outputs of the previous nodes. An input is said to be connected if the connection is 1 and unconnected if the connection is 0. Weights are randomly generated between -1 to +1. The output(s) of the genotype can be output(s) from any node or program input(s). Input and Weight are multiplied for all the connected inputs and is summed up. It is then forwarded to a non-linear function such as sigmoid or tangent hyperbolic to produce an output at each node. This output can either be the input to the next node or output of the system. The CGPANN genotype is then evolved from one generation to next (through the process of mutation) until the desired behaviour is achieved. Figure 2 is a block representation of a 2-input CGPANN node with inputs (ψ_i ψ_j), weights (W_{in},W_{jn}) and connection switches (s_{in},s_{jn}) respectively. ψ_i denotes an external input, x_i when $i<n_i$ and a node output otherwise. Fig. 2 represents the parameters of node 'n' (For this example we assume two inputs). W_{in} corresponds to a weight assigned to ψ_i and W_{jn} represents weight of ψ_j for node 'n' respectively. Figure 2 displays the inside view of CGPANN node. The two node inputs (ψ_i ψ_j) are multiplied with the corresponding weights (W_{in},W_{jn}). In general, the result after summation of the connected inputs is given to the node function which generates an output value for the node 'n'. In this study node functions are either hyperbolic tangent or sigmoidal function. Also, figure 3(a) shows a CGPANN genotype of a 2*2 network assuming 2 program inputs (x_0, x_1), 2 functions (f_0 *and* f_1) and 1 output. Figure 3(b) shows the inside view of the network in Figure 3(a). The node '5' includes input x_2, x_1 where x_2 is the output of node 2 and x_1 is one of the inputs to the system. Thus the network first computes the output of node '2' and then passes the result to node '5'. In this example the remaining nodes 4 and 3 are not used (non-coding). Figure 3 (c) represents the neural architecture of the genotype of Figure 3(a). The network derived shows that neurons are not fully connected because they may be non-coding and may or may not use all the inputs supplied to them (because of binary connection switch genes). This differs from standard ANN architecture. An evolutionary algorithm

operating on this representation has the possibility to select topologies and weights of the networks simultaneously using this indirect presentation.



$g_{in} = \{i, w_{in}, s_{in}\}$
$g_{jn} = \{j, w_{jn}, s_{jn}\}$
$g_n = \{g_{in}, g_{jn}, f_n\}$
$p_{in} = \psi_i . w_{in} . s_{in}$
$p_{jn} = \psi_j . w_{jn} . s_{jn}$
$an = p_{in} + p_{jn}$
$\psi_n = f_n(a_n)$
$f_n = f_0$ or f_1

Fig 2. Inside Process of CGPANN for node 'n'

(CGPANN node with two inputs)

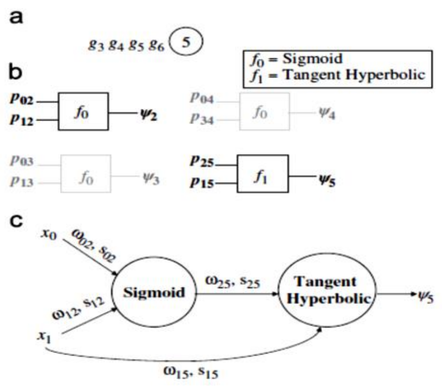


Fig 3. FCGPANN based genotype for a problem with two inputs $x_0 and x_1$.

2) Recurrent Cartesian genetic programming evolved ANNs (RCGPANN)

In order to capture a broader domain of systems that are dynamic and non-linear, the need for recurrent networks becomes essential. This section describes a NeuroEvolutionary algorithm that exploits the powerful representation of CGP in generating recurrent artificial neural architecture. RCGPANN follows the direct encoding strategy where it encodes topology, weight and functions in one genotype and then evolves it for augmented topology, best possible weights and functions. In general, TWEANN- (Topology and Weight Evolution) algorithms are either constructive or destructive. RCGPANN is both constructive and destructive algorithm. It begins with random topological features and incrementally removes and adds new features. It does so by mutating functions, inputs, weights, connection types and outputs. When a connection is disabled through mutation it is not fully removed. It has a probability to become re-enabled in the later generation. The network derived constitutes neurons that are not fully connected, and that the program input(s) are not supplied to every neuron in the input layer which is different than the traditional ANN architecture. Thus such an evolutionary algorithm has the possibility to produce topologies that are efficient in terms of hardware implementation and time. The recurrent network produced is a representation of 'Jordan Network'. The RCGPANN genotype consists of nodes that represents neurons of ANN. The node includes inputs, weights, connection and function. Inputs can be program inputs or inputs from the previous nodes or a feedback. The first layer of the RCGPANN genotype consists of inputs that are recurrent. The layers following that might have recurrent connections based on whether the feedback input is randomly selected as an input to the node or not. An input is said to be connected if the Connection is 1 otherwise the Connection is 0. Weights are randomly generated between -1 to +1 but for the feedback input the weight is always assigned a value 1. Input and Weight are multiplied for all the connected inputs and is summed up. It is then forwarded to a linear or a non-linear function such as sigmoid, tangent hyperbolic, step or linear to produce an output at each node. This output can either be the input to the next node or output of the system. The output(s) of the genotype can be output(s) from any node or program input(s). The output of the genotype is feed back to the nodes if the recurrent input is enabled or connected. The RCGPANN genotype is then evolved from one generation to next (through the process of mutation) until the desired fitness is achieved. No mutation is applied on the connection or weight of the state units (feedback units). The resultant genotype is then transformed to an artificial neural architecture. Fig.4 (a) is a block representation of a 3-input RCGPANN node with inputs (x_0, x_1, x_2), weights (W_{03},W_{13}, W_{23}) and connections (s_{03},s_{13}, s_{23}=1). Fig.4(c) represents the parameters of node '3'. W_{03} corresponds to a weight assigned to x_0 , W_{13} represents weight of x_1 and W_{23} represents a recurrent input which is the output of the system feedback as an input having a weight of W_{23}=1 for node '3' respectively. The initial value of the recurrent input x_2 is taken as 0. The three inputs (x_0, x_1, x_2) are multiplied with the corresponding weights (W_{03},W_{13}, W_{23}). The result after summation is then forwarded to a sigmoid function that generates an output value for the node '3'. Fig.4(b) is a RCGPANN genotype of a 2*2 network with 3 inputs (x_0, x_1, x_2), 2 functions (f_0 and f_1) and one output node '6'.Fig.2(b) shows the inside view of the network in Fig.2(a). The node '6' takes input x_3, x_1 and its feedback value which for initial step is 0. x_3 is the output of node 3 having x_0, x_1 and a feedback of the system (in this case x_2 represents output of node '6') as inputs with function f_1. Thus the network first computes the output of node

‘3’ and then processes the result further. The resultant genotype is transformed to the neural architecture shown in Fig.4 (c).

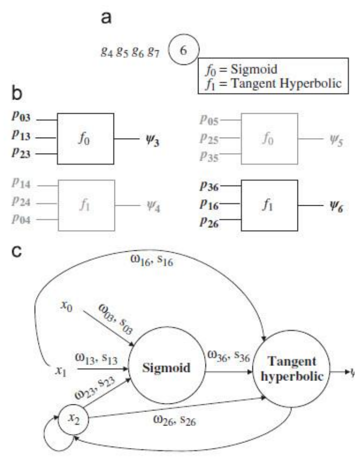


Fig 4. RCGPANN based genotype for a problem with two inputs x_0 and x_1.

======================

c) This neuroevolutionary algorithm is tested in the following cases:

1-Single pole: Markovian and Non-Markovian

For single pole Markovian case, the network has access to all four inputs $(x, \dot{x}, \theta_1, \dot{\theta_1})$while for the non-Markovian case the number of inputs is reduced to two (x, θ_1).

2- Double poles: Markovian and Non-Markovian

In double pole balancing, for the Markovian case the network has access to all six inputs $(x, \dot{x}, \theta_{1,2}, \dot{\theta_{1,2}})$ while for the non-Markovian case the number of inputs is reduced to three (x, θ_1, θ_2) . Double pole balancing is a non-linear problem.

Comparison of feedforward and recurrent CGPANN with other neuroevolutionary techniques for Markovian and non-Markovian cases tested on single pole balancing task. The FCGPANN network has generated solutions at an average of 21 evaluations and the RCGPANN at an average of 17 evaluations for the Markovian case. FCGPANN was unable to find any solution at non-Markovian case while RCGPANN produced solutions at an average of 55 evaluations. These algorithms clearly outperformed and produce solutions in fewer evaluations.

In this study a fast learning NE (neuroevolutionary) algorithm based on Cartesian genetic programming (CGPANN) of both feedforward (FCGPANN) and recurrent (RCGPANN) architecture was presented. The algorithm was tested on the pole-balancing benchmark. The results demonstrated that CGPANN extends the powerful and flexible representation of CGP to the evolution of neural networks. It was realized that FCGPANN and RCGPANN generated solutions with fewer evaluations on an average for pole balancing tasks as compared to other published techniques. Adding recurrence to the CGPANN network improved the performance of the network by finding solutions to non-linear and non-Markovian states. Experimental results demonstrate that the FCGPANN and RCGPANN produce robust controllers with better generalization ability than other methods reported in the paper. Clearly CGPANN is highly competitive with established neuroevolutionary techniques such as NEAT, SANE, ESP and CoSyNE. ==

d) In this study, a form of CGP has been used which implemented a Jordan type architecture for allowing feedback. Although using Jordan type architectures represents a simple method for allowing recurrent connections, it does so in a very restricted form. For instance, the user must decide in advance how many and what type of recurrent connections will be used. Also, I think we can test tree based GP instead of CGP for the mentioned problems in this paper.

From my point of view, it is important to consider that the performance of both the FCGPANN and RCGPANN is based on a number of parameters namely mutation rate, network size, number of inputs to each node, averaging the number of outputs and the functions tested. I think a more investigation of these parameters are maybe to produce even better results. Also, I think we have to consider time and memory complexities for this algorithm (cpu time) because I think the time complexity of this method is high.