Developmental Plasticity in Cartesian Genetic Programming Artificial Neural Networks

Maryam Mahsal Khan, Gul Muhammad Khan
Department of Computer System Engineering, NWFP University of Engineering and Technology, Peshawar,Pakistan
maryam@nwfpuet.edu.pk, gk502@nwfpuet.edu.p

Julian F. Miller
Department of Electronics, University of York, York, UK
jfm7@ohm.york.ac.uk

Keywords: Generative and Developmental Approaches, NeuroEvolutionary algorithms, Pole Balancing

Abstract: This work presents a method for exploiting developmental plasticity in Artificial Neural Networks using Cartesian Genetic Programming. This is inspired by developmental plasticity that exists in the biological brain allowing it to adapt to a changing environment. The network architecture used is that of a static Cartesian Genetic Programming ANN, which has recently been introduced. The network is plastic in terms of its dynamic architecture, connectivity, weights and functionality that can change in response to the environmental signals. The dynamic capabilities of the algorithm are tested on a standard benchmark linear/non-linear control problems (i.e. pole-balancing).

1 Introduction

Natural neural systems are not static in nature. They interact and respond to environmental factors. Such dynamic interactions lead to modification or development of the system with time. A static system can be trained to give a specific response to a set of environmental stimuli however it cannot learn or adapt to a changing environment. To do this one requires a network that develops at runtime. Evolutionary computation based algorithms have proved to be effective at finding a particular solution to a computational problem, however, if the problem domain is slightly changed the evolved network is unable to solve the problem. Catastrophic forgetting (CF) is the phenomenon in which a trained artificial neural network loses its accuracy when the network is trained again on a different computational problem (McCloskey and Cohen, 1989; Ratcliff, 1990; Sharkey and Sharkey, 1995; Ans et al., 2002; French, 1994). Forgetting is inevitable if the learning resides purely in the weights of fixed connections.

Our aim is to find a set of computational functions that encode neural network architecture with an ability to adapt to the changing environment as a result of development. Such plastic neural networks are different from conventional ANN models as they are constantly adjusting *both* topology and weights in response to external environmental signals. This means, that they can grow new networks of connections when the problem domain requires it. The main motivation for such a new model is to arrive at a plastic neural network model that can produce a single neural network that can solve multiple linear/non-linear problems (i.e. not suffer from a phenomenon called catastrophic interference). We think that this can be acheived by arriving at a neural network model in which new neural sub-structures automatically come into existence when the network is exposed to new learning scenarios.

In this paper we introduce a plastic neural network based on the representation of Cartesian Genetic Programming (Miller and Thomson, 2000) (PCGPANN). We have tested the performance of this algorithm on the standard benchmark problem of pole-balancing for both linear and non-linear conditions.

In Section 2 we will briefly review the current developmental algorithms presented to date. Section 3 give an overview of Cartesian Genetic Programming (CGP). In section 4, we will provide an overview of the standard CGPANN algorithm. Section 5 describes how self-modification or developmental plasticity could be embedded in the standard CGPANN algorithm. Section 6 - 8 is the application of the algo-

rithm on a standard benchmark problem - single and double pole-balancing along with results and discussion. Section 9 concludes with the findings and future work.

2 Related Work

GP has been used in a number of ways to produce ANNs. The simplest of all is using direct encoding schemes in which GP is used to encode the weights of ANN or architecture or both. In direct encoding schemes each and every unit in phenotype is specified in the genotype. Specifying each connection of ANN in genotype is commonly referred to as 'structural encoding' (Hussain and Browse, 2000).

Another important form of encoding that is used to develop ANNs is Grammar encoding. There are two types of grammar encoding schemes: developmental grammatical encoding and derivation grammatical encoding. Kitano used developmental grammatical encoding scheme for development of ANN (Kitano, 1990). In this, genes describe the grammatical rule that is used for development of the ANN. In derivation grammatical encoding, the gene contains a derivation sequence that specify the whole network (Gruau, 1994; Jacob and Rehder, 1993).

Cellular Encoding (CE) devised by Gruau et.al (Gruau et al., 1996) is a developmental neuroevolutionary algorithm. Weights, connection, graph rewriting rules are evolved based on the evolutionary algorithm. CE transforms graph in a manner that controls cell division which grows into an ANN. CE was applied on a control problem of balancing single and double poles. This technique was found to scale better and was effective at optimizing both architecture and weights together.

Nolfi et al presented a model in which the genotype-phenotype mapping takes place during the individual's lifetime and is influenced both by the genotype and by the external environment (Nolfi et al., 1994). The 2D neural networks adapt during their lifetime as environmental conditions affect the growth of the axons. The neurons had only upward growing axons with no dendrites.

Cangelosi proposed a related neural development model that starts with a single cell. This undergoes a process of cell division and migration until a collection of neurons arranged in 2D space is developed (Cangelosi et al., 1994). The rules for cell division and migration are specified in genotype; for a related approach see (Dalaert and Beer, 1994; Gruau, 1994).

Rust and Adams devised a developmental model coupled with a genetic algorithm to evolve *parameters* that grow into artificial neurons with biologically-realistic morphologies (Rust et al., 2000). They also investigated activity dependent mechanisms so that neural activity would influence growing morphologies.

Jakobi created an artificial genomic regulatory network (Jacobi, 1995). He used proteins to define neurons with excitatory or inhibitory dendrites. The individual cell divides and moves due to protein interactions with an artificial genome, causing a complete multicellular network to develop. After differentiation each cell grows dendrites following chemical sensitive growth cones to form connections between cells. This develops into a complete recurrent ANN, that is used to control a simulated Khepera robot for obstacle avoidance and corridor following.

Karl Sims used a graph based GP approach to evolve virtual robotic creatures. The morphology of these creatures and the neural systems for controlling the forces acting between body parts were both genetically determined (Sims, 1994). The genotypes were structured as directed graphs of nodes and connections. When a creature is synthesized from its genetic description, the neural components described within each part are generated along with the morphological structure.

Roggen et al. devised a hardware cellular model of developmental spiking ANNs (Roggen et al., 2007). Each cell can hold one of the two types of fixed input weight neurons, excitatory or inhibitory. The neuron integrates the weighted input signals and fires an action potential when it exceeds a certain membrane threshold. This is followed by a short refractory period. They have a leakage mechanism which decrements membrane potentials over time.

Rivero, evolved a specially designed tree-representation which could produced an ANN graph (Rivero et al., 2007). He tested his network on a number of data mining applications.

Harding et al. adapted the graph-based genetic programming approach, known as Cartesian Genetic Programming (CGP) by introducing a number of innovations. Foremost of which was the introduction of self-modifying functions. The method is called self-modifying CGP (SMCGP) (Harding et al., 2010). Self-modifying functions allow an arbitrary number of phenotypes to be encoded in a single genotype. SMCGP was evaluated on a range of problems e.g. digital circuits, pattern and sequence generation and, notably, was found to produce general solutions to some classes of problem (Harding et al., 2010).

Khan et. al presented a neuro-inspired developmental model using indirect encoding scheme to develop learning neural architectures (Khan et al.,

2007). They used cartesian genetic programming to encode a seven chromosome neuron genotype. The programs encoded controlled the processing of signals and development of neural structures with morphology similar in certain ways to biological neurons. The approach has been tested on a range of AI problems including: Wumpus world, Solving Mazes and the Game of Checkers (Khan et al., 2007).

McPhee et.al (Nicholas F.McPhee and Poli, 2009) explored a particular method for introducing developmental plasticity into GP. It introduced 'Incremental Fitness based Development (IFD)' which extends the N-gram GP system to allow developmental plasticity in the generation of linear-GP like programs. N-gram GP is an Estimation of Distribution Algorithm (EDA) for the evolution of linear computer programs which has been shown to perform well on a number of symbolic regression problems. The IFD approach was found to be frequently better than the standard N-gram system, and never worse.

HyperNEAT is one of the indirect encoding strategies of developmental ANNs (Stanley et al., 2009). HyperNEAT uses generative encoding and the basic evolutionary principles based on the NEAT (Neuro Evolution of Augmented Topology) algorithm (Stanley and Miikkulainen, 2002). In HyperNEAT the weights(inputs) are presented to an evolved program called a Compositional Pattern Producing Networks (CPPNs). The CPPN takes coordinates of neurons in a grid and outputs the corresponding weight for that connection. The algorithm has been applied to Robot locomotion. Regular quadruped gaits in the legged locomotion problem have been successfully generated using this algorithm (Clune et al., 2008).

Synaptic plasticity in Artificial neural network can be introduced by using local learning rules that modify the weights of the network at runtime (Baxter, 1992). Floreano and Urzelai have demonstrated that evolving network with synaptic plasticity can solve complex problems better than recurrent networks with fixed-weights (Floreano and Urzelai, 2000). A number of researchers have investigated and compared the performance of plastic and recurrent networks obtaining mixed results with either of them performing better in various problem domains (for a review see (Risi and Stanley, 2010)).

One of the interesting features of the approach we propose is that, unlike previous approaches investigated, it has both synaptic and developmental plasticity.

3 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is an evolutionary technique for evolving graphs (Miller and Thomson, 2000). The graphs are arranged in the form of rectangular grids like multi-layer perceptrons. The graph topology is defined using three parameters: number of columns, number of rows and levels-back. The number of nodes in the graph encoded in the genotype is the product of the number of columns and the number of rows. Usually feedforward graphs are evolved in which inputs to a node are the primary inputs or outputs from the previous nodes. The levels-back parameter constrains connections between nodes. For instance, if levels-back is one, all nodes can only be connected to nodes immediately preceeding them, while if levels-back is equal to the number of columns of nodes (primary inputs are allowed to disobey this constraint) the encoded graphs could be an arbitrary feed-forward graph. Genotypes consist of a fixed length string of integers (referred to as genes). These genes represents nodes with inputs and functions. Functions can be any linear/non linear functions. The output of the genotype can be from any node or the primary inputs. CGP generally uses a 1+4 evolutionary strategy to evolve the genotype. In this the parent genotype is unchanged and 4 offspring are produced by mutating the parent genotype. It is important to note that when genotypes are decoded nodes may be ignored as they are not connected in the path from inputs to outputs. The genes associated with such nodes are non-coding. This means that the *active* final graph encoded could consist of a phenotype containing any subset of number of nodes defined in the genotype. Mutating a genotype can thus change the phenotype either not at all or drastically.

We give a simple example of CGP in Figure.1 where 1(a) shows a 1x2 (rows x columns) genotype for a problem with 2 binary inputs '$\psi_{0,1}$' and 1 binary output. The inputs to each node is represented as i_{mn} where 'm' represents the input from and 'n' is the input to the node. The functions 'f_n' used are logical 'OR' and logical 'AND'. Figure.1(b) shows the graphical representation of the genotype in Figure.1(a). The resultant phenotype shown in Figure.1(c) can be represented as a logical equation in Eq.(1) where '+' represent OR and '.' represents AND operation.

$$\psi_3 = \psi_1 . \psi_0 + \psi_1 \quad (1)$$

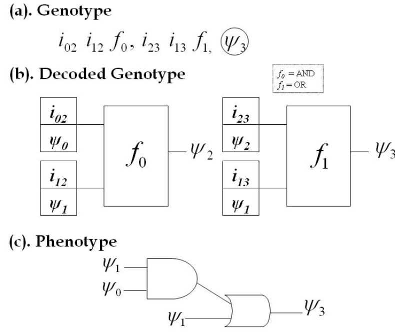

Figure 1: (a) CGP based Genotype for a problem with 2 binary inputs ψ_0 and ψ_1. Single program output is taken from the last gene (circled). (b) Decoding of the genotype. (c). Phenotype represented in terms of a digital circuit

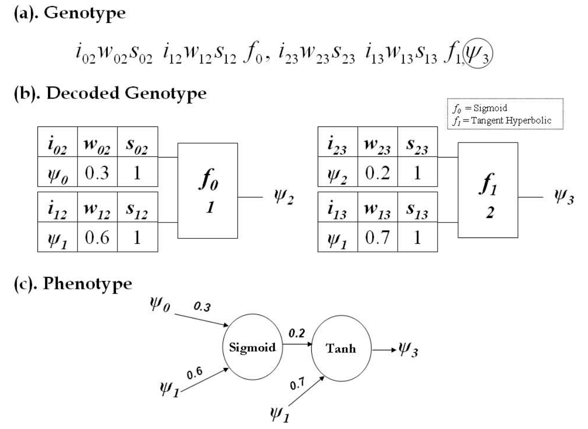

Figure 2: (a). Genotype of a 1x2 architecutre with two inputs (b). Decoded Genotype (c).Phenotype represented as ANN of the genotype

4 CGP Evolved ANN

The representation of CGP has been exploited in generating static neural architectures. Both feedforward and recurrent (FCGPANN , RCGPANN) based networks have been designed and tested on various applications (Khan et al., 2010b; Khan et al., 2010a).

In CGPANN, along with standard CGP genes (inputs, functions) weight and switch genes are also introduced. The weight values are randomly assigned to each input with values from $(-1,+1)$. Switch genes are either '1' or '0' where '1' is a connected and '0' a disconnected input. The genotype is then evolved to obtain best topology, optimal weights and combination of functions.

Figure.2 is an example of a CGPANN genotype with two primary inputs $\psi_{0,1}$ and one output. The activation function f_n used are sigmoid and tangent hyperbolic. The inputs to each node is represented as i_{mn}. Similarly weights w_{mn} are assigned to each input. Whether an input is connected or not is encoded as a switching gene s_{mn}. Figure.2(a) is the genotype of a 1x2 architecture with the output taken from node ψ_3. Figure.2(b) is the graphical view of the genotype with values assigned e.g. the weight assigned to the input i_{12} supplied to node 2 is $w_{12} = 0.6$. The phenotype of the genotype is shown in Figure.2(c) which is mathematically expressed in Eq.(2).

$$\psi_3 = Tanh(0.7\psi_1 + 0.2sig(0.3\psi_0 + 0.6\psi_1)) \quad (2)$$

5 Plastic CGP Neural Network

A plastic CGPANN (PCGPANN) is a developmental form of Cartesian Genetic Programming ANN algorithm. The basic structure and representation of the PCGPANN is similar to the CGPANN structure as shown in Figure. 2. In the PCGPANN method, an additional output gene is added to the genotype. It is used for making developmental decisions. According to whether the output value is less than zero or above, a 'mutation' of the genotype is introduced to create a new phenotype. In this way, a single genotype can evoke an unlimited series of phenotypes depending on the real-time output of the program encoded in the genotype. The modification of the genotype results in different phenotypic structures. The mutation produced is randomly chosen from the types listed below. It should be noted that some gene changes may result in no phenotypic change as in CGP many genes are redundant. The phenotypic mutation of neural network can be any of the following: inputs, weights, switches,outputs and activation functions. Genes are mutated and assigned a valid value based on sets of constraints.

- If a gene represents a function then it is replaced by any random function selected from 1 to total number of functions (n_f) available to the system.
- If the gene represents a weight is replaced by a weight value between (-1, +1).
- It the gene is a switching gene then it is complemented.
- Node input genes are mutated by assigning an input value such that the levels-back parameter is still obeyed.
- An output can be assigned any random value from 1 to total number of nodes plus inputs presented to the system.

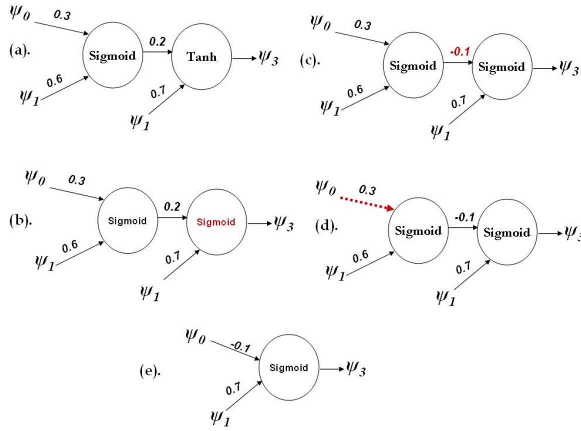

Figure 3: PCGPANN - Plastic Development of Neural Network Parameters (a).Initial Phenotype (b)Mutation of Function (c)Mutation of Weight (d)Mutation of Switch (e)Mutation of Input

Figure.3 shows a step by step process of mutation on a phenotype. Figure.3(a) is the initial phenotype of the genotype given in Figure.2(c). As in the described example the activation functions used were two - sigmoid and tangent hyperbolic. Thus the genotype has only two functions to randomly choose from. In Figure.3(b) the function is mutated from tangent hyperbolic to sigmoid . The mathematical expression of the updated genotype is thus changed as shown in Eq.(3).

$$\psi_3 = sig(0.7\psi_1 + 0.2sig(0.3\psi_0 + 0.6\psi_1)) \quad (3)$$

Figure.3(c) represents a mutated phenotype where the value of weight is changed from 0.2 to -0.1 expressed in Eq.(4).

$$\psi_3 = sig(0.7\psi_1 - 0.1sig(0.3\psi_0 + 0.6\psi_1)) \quad (4)$$

Similarly Figure.3(d) displays the mutated phenotype where the switching gene is toggled and switched off (shown as dotted line). Thus the input is no longer connected to the node. Mathematically expressed by Eq.(5).

$$\psi_3 = sig(0.7\psi_1 - 0.1sig(0.6\psi_1)) \quad (5)$$

Mutation of inputs is one of the major tunning parameter during the developmental process of ANN. From the phenotype displayed in Figure.3(d), the input to the second node is ψ_1 and an output from the first node. After mutating the input, the input to the second node is modified to two primary inputs of the system $\psi_{0,1}$. The resultant phenotype is displayed in Figure.3(e) with its function in Eq.(6). Thus during runtime different functional equations can be produced generating desired response.

$$\psi_3 = sig(-0.1\psi_0 + 0.7\psi_1) \quad (6)$$

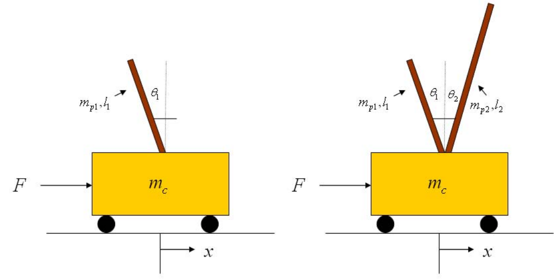

Figure 4: (a).Single Pole Balancing (b).Double Pole Balancing

Similarly output of any genotype can also be mutated and changed to another node. In this way inactive genes present in the genotype can be activated or vice versa. This in turn effects the phenotypic structure of the genotype.

It is worth mentioning here that mutation of genotype at runtime affects only a single gene where as mutation from generation to generation affect 10% (mutation rate) genes of the genotype thus changing the starting genotype for each development.

6 Case Study: Balancing Pole(s)

Pole balancing also known as inverted pendulum is a standard benchmark problem in the field of control theory. The performance of various neuroevolutionary algorithms is tested on this problem. Single pole balancing task consists of a pole attached by a hinged to a wheel cart while double pole balancing constitutes two poles. In both the cases the track of the cart is limited to $-2.4 < x < +2.4$ as shown in Figure.4. The objective is to apply force 'F' to the cart such that the cart doesn't leave the track and the angle of the pole doesn't exceed (-12°$<\theta_1<$ +12°) for single pole and (-36°$<\theta_{1,2}<$ +36°) for the double pole balancing task. 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. The input to the system are the angle of pole(s) ($\theta_{1,2}$), angular velocity of pole ($\dot{\theta}_{1,2}$), position of cart (x) and velocity of cart ($\dot{x}$). Equations that are used to compute the effective mass of poles, acceleration of poles, acceleration of cart, effective force on each pole, the next state of angle of poles, velocity of poles, position of cart and velocity of cart can be found in (Khan et al., 2010a).

Table 1: Parameters for Single Pole Balancing Task

Parameters	Value
Mass of cart (m_c)	1 Kg
Mass of Pole (m_{p1})	0.1Kg
Length of Pole (l_1)	0.5m
Width of the Track (h)	4.8m

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)	4.8m

7 Experimental Setup

PCGPANN genotypes are generated for various network sizes for both single and double pole balancing task. A mutation rate of $\mu_r = 10\%$ is used for all the networks. The activation functions f_n used are sigmoid and tangent hyperbolic. Each genotype is initialized with random weights 'w' from -1 to +1, switching 's' values of 0 or 1 and random inputs 'ψ_i'. The number of outputs 'O_i' from the system are 2 i.e. O_1 and O_2 . Both the outputs are normalized between -1 to +1. Output O_1 represents the 'Force (F)' that controls the force applied to the cart and the output O_2 is associated with the developmental plasticity of the network. Using one of the outputs of the genotype for making developmental decision acts as a feedback to the system for generating plastic changes.

We have simulated the network for two developmental strategies Dev_1 and Dev_2 as shown in Eq.(7,8). Both the strategies are tested on the pole balancing task for single and double pole(s) with zero and random initial states. The intention is to identify faster and robust strategies for the problem investigated.

$$Dev_1 = \begin{cases} No-Change & -1 \leq O_2 \leq 0 \\ Mutation-of-Parameters & 0 < O_2 \leq 1 \end{cases} \quad (7)$$

$$Dev_2 = \begin{cases} Mutation-of-Parameters & -1 \leq O_2 \leq 0 \\ No-Change & 0 < O_2 \leq 1 \end{cases} \quad (8)$$

7.1 Single Pole:Zero and Random Initial States

In single pole balancing the input to the networks are position of cart x, velocity of cart $\dot{x}$, angle of pole θ_1

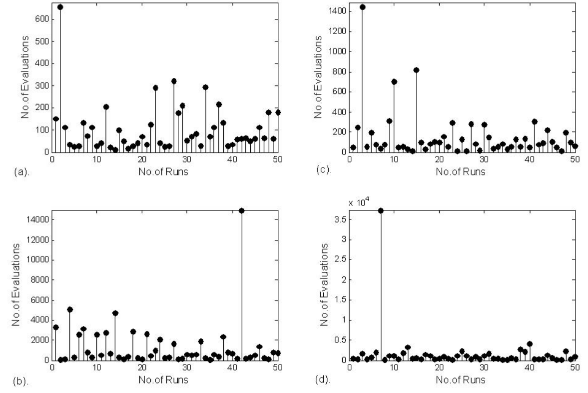

Figure 5: Variation in the number of evaluations of the best evolved network's genotypes for single(a,c) and double poles(b,d) with zero and random initial states

and velocity of pole $\dot{\theta}_1$. The number of inputs to each node is taken as 4, which is equivalent to the number of inputs to the system.

7.2 Double Pole:Zero and Random Initial States

Double pole balancing involves balancing an additional pole. The number of input to the networks are position of cart x, velocity of cart $\dot{x}$, angle of poles $\theta_{1,2}$ and angular velocity of poles $\dot{\theta}_{1,2}$. The number of input to each node is taken as 6, which is equivalent to the number of inputs to the system.

In both the cases (single and double poles), the networks are generated with initial input values of zero or with random values of -0.25rad $<\theta_1<$0.25rad and -2.4 $<x<$+2.4 for single pole and -0.6rad $<\theta_{1,2}$ $<$0.6rad and -2.4 $<x<$+2.4 for double poles. The performance of the PCGPANN algorithm is based on the average number of balancing attempts (evaluations) for fifty independent evolutionary runs.

8 Results and Discussions

Table 3 represents average number of balancing attempts for inputs starting from initial states of zero and random states of network with varying network sizes for single and double pole(s) balancing tasks under both the developmental strategies. The results are the average of 50 independent evolutionary runs. Figure.5 shows the variation in the number of evaluations of the best evolved network's genotypes for single/double pole(s) with zero and random initial states. From the figure it is evident that out of fifty genotypes

Table 3: Performance of PCGPANN on Single and Double Poles

Network Representation		SinglePole Evaluations		DoublePole Evaluations	
Initial State	Network Arch.	Dev_1	Dev_2	Dev_1	Dev_2
Zero	4x4	221	659	1716	6310
	5x5	186	570	**1169**	**5494**
	10x10	**104**	**519**	1337	8315
	15x15	168	643	1934	7334
Random	4x4	390	608	1738	3310
	5x5	303	387	1981	4033
	10x10	**157**	**243**	**1612**	**3308**
	15x15	160	260	1916	4362

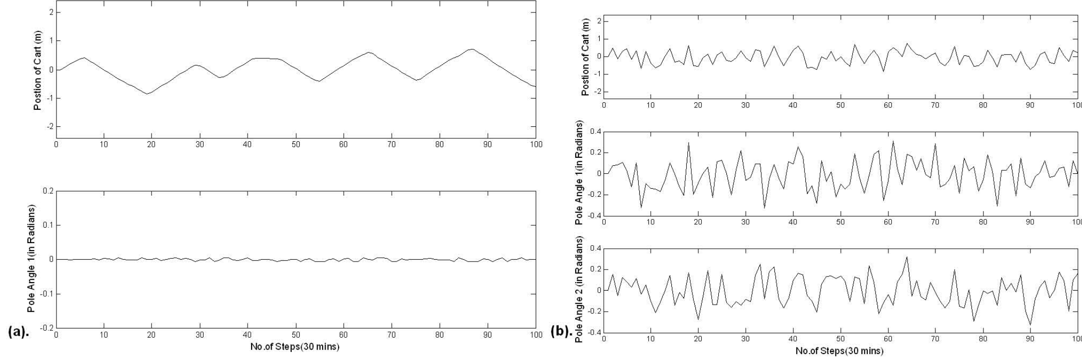

Figure 6: Pole Angle(s) and Position of Cart simulated for 100,000 steps for (a).Single Pole Task (b).Double Pole Task

only one of them took longer than expected number of evaluations thus increasing the overall average. It is observed in all the four cases of Figure.5 that learning speed is much faster in 98% of the evolutionary runs. It is observed in Table 3 that the developmental strategy Dev_1 has performed better than the Dev_2 strategy for both the single and double pole balancing tasks. From Table 3, the minimum average evaluations of 50 independent runs for the single pole and double pole was 104 and 1169 respectively. Random initial states took longer time to develop into a stable structure as compared to zero initial states for obvious reasons. For both single and double pole balancing task, fast learning in terms of minimum average evaluations has been found in the 10x10 network architecture.

Figure.6 represents the pole angles and position of cart simulated for 30 minutes respectively (100,000 steps are down-sampled to produce 100 steps for demonstration purpose only) for the single pole and double pole balancing task. In Figure.6(a) the cart is moving back and forth to make the pole balanced while minute changes in the angle of pole is observed. Similarly in case of Figure.6(b) random behaviour of movement of both the poles and the cart is observed for the double pole balancing task. Pole balancing task is used as a benchmark

Table 4: Comparison of PCGPANN with other neuroevolutionary algorithms Applied on Single and Double Pole Balancing Task: average number of network evaluations

Method	SinglePole	DoublePole
CNE	352	22100
SANE	302	12600
ESP	289	3800
NEAT	743	3600
CoSyNE	98	954
FCGPANN	21	77
PCGPANN	**104**	**1169**

by researchers to investigate the performance of algorithms including: Conventional Neuro Evolution (CNE) (Wieland, 1991), Symbiotic, Adaptive Neural Evolution (SANE)(Moriarty, 1997), Enforced Sub-Population (ESP) (Gomez and Miikkulainen, 1999), Neuro Evolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) and Cooperative synapse neuroevolution (CoSyNE) (Gomez et al., 2008). All these neuroevolutionary algorithms produces static ANNs detail of which can be found in (Khan et al., 2010a). Table 4 represents the comparison of performance of the Plastic CGPANN with all these neuroevolutionary algorithms for both the single

and double pole balancing tasks. Developmental CGPANN has outperformed most of these neuroevolutionary algorithms by quite a large margin. As pointed out by Risi et al. learning a specific task is easy but learning the strategy to learn and adapting its architecture to the changing task takes time (Risi and Stanley, 2010). This is the reason that PCGPANN took more evaluations to arrive at a solution as compared to its static counterpart - FeedForward CGPANN (FCGPANN)(Khan et al., 2010a).

8.1 Developmental Plasticity

It is observed in the CGP literature that only upto 5% nodes are usually active at runtime, thus having 95% of garbage space (inactive nodes) that may be activated during evolution (Miller and Smith, 2006). As the overall architecture of the system in PCGPANN is fixed (5x5, 10x10, etc networks), only the interconnectivity of the neurons changes at runtime thus changing the active morphology of the ultimate phenotype. This phenomenon is similar to what is observed in biological brain with different parts of nervous system responding to different sets of input at runtime (Kandel et al., 2002). In Plastic CGPANN algorithm neurons are presented in terms of network sizes where a 5x5 network corresponds to 25 neurons. The target is then to use these neurons like resources in a system in an effective and efficient manner. The structure produced connects and disconnects during the developmental process generating different sets of equations at run time. This corresponds to the creation of different phenotypes during a generation, similar to the work done by Nolfi (Nolfi et al., 1994). Such plasticity embeds efficient utilization of resources and reduces hardware cost. Till date no developmental algorithm other than PCGPANN have been proposed that has excessive neuron structures that activate and deactivate with time rather fix architectures are mostly proposed. Figure.7 shows the logarithmic graphs for developmental outputs of six different evolved genotypes for both the developmental strategies while balancing double poles for 30mins (100,000 time steps). It is evident from these graphs that after some early development the ultimate output of the network comes causes the network to stay same (no change decision). It can be seen in Figure.7(a,c,e) that genotypes produces developmental output with oscillating behaviour at the beginning resulting in change of phenotypic structure and weights more often. Similar behaviour is also observed in biological brain where most of the learning and development happens in the immature brain as compared to the adult brain. Hence changing or mutating parameters at the early stage has greater probability of convergence to the solution.

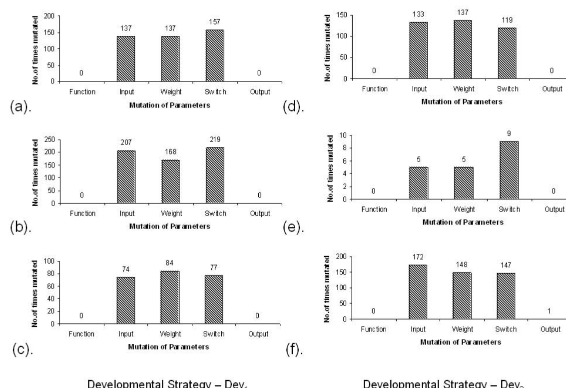

Figure 8: Statistics of the changes in network parameters for the six cases in Figure.7

Figure.8 shows the statistics of various parameters that are mutated for the six cases in Figure.7 during 100,000 time steps while the pole(s) are being balanced. A similar trend is observed in all the cases with switches, inputs and weights updated with almost same ratio. On average the network is changed by 0.3 to 0.4% during runtime with most (95%) development taking place at the early stage as evident from Figure.7.

As learning to learn is a complex dynamic nonlinear process even a biological brain does not process inputs linearly. Remaining dormant for a number of steps reduces the chance of quick learning while it is observed that changing and adapting (to remain dormant) to the change is an effective strategy of learning.

8.2 Robustness of PCGPANN

The robustness of the genotypes is inferred by subjecting the genotype to 625 random initial states and checking it for 1000 steps only. If the genotype is able to balance then it is termed as robust. Figure. 9 and 10 represents the performance of 50 genotypes (solutions of the single and double pole task) on 625 random initial states. For the single pole, 48% of the genotype were able to balance at 90-100% (which corresponds to 563-625) of the random states which can be inferred from the graph presented in Figure.11, thus attaining an average of 456. While for the double pole 44% genotypes were able to balance at 60-100% of the random states achieving an average of 349. Thus the genotypes undergoing developmental plasticity during the 1000 steps exhibit robustness.

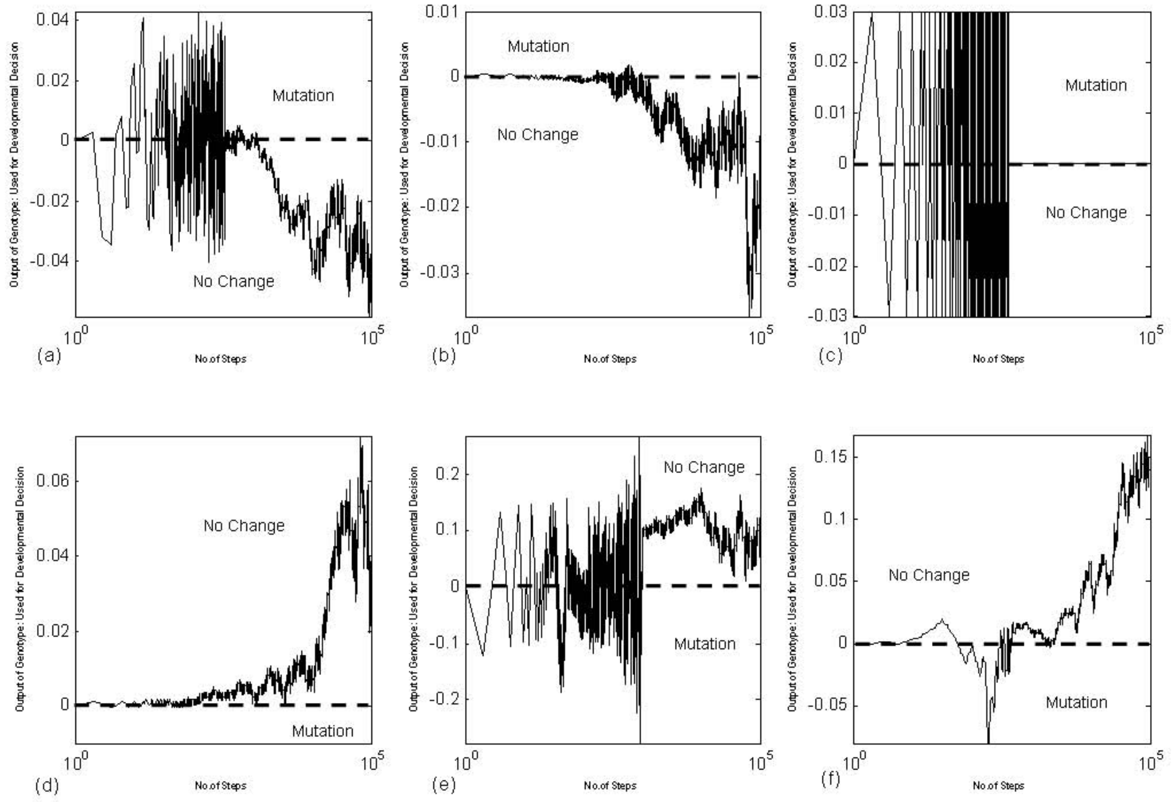

Figure 7: Developmental Behaviour of the network - (a,b,c) Dev_1 Strategy and (d,e,f) Dev_2 strategy

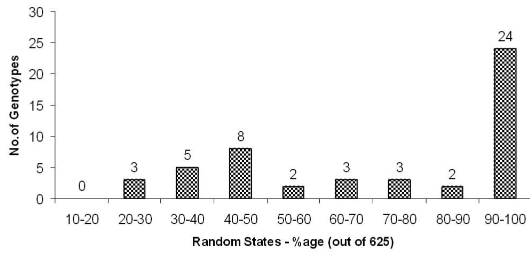

Figure 9: SinglePole Task-Number of Genotypes that are successful in balancing the pole in 625 random initial states

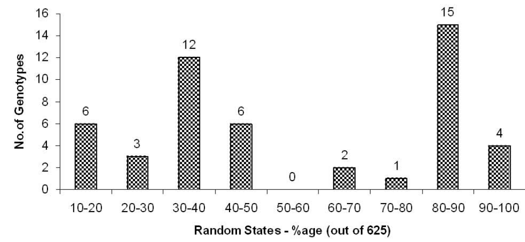

Figure 10: DoublePole Task-Number of Genotypes that are successful in balancing the pole in 625 random initial states

Figure.11(a,b) represents the performance of 50 independent evolved genotypes on 625 random initial states for the single and double pole tasks. The generalization score is plotted clearly demonstrating that most of the genotypes exhibit robust behaviour. Table.5 shows the generalization of the PCGPANN genotypes in comparison to feedforward CGPANN for both the single and double pole balancing scenarios. The PCGPANN strategy proves more robust in the double pole balancing task (which is a non-linear problem) where on average a genotype at 349 out of 625 different initial states is developed into successful and feasible solutions.

Table 5: Average number of random cart-pole initializations (out of 625) that can be solved

Type	SinglePole	DoublePole
FCGPANN	590	277.38
PCGPANN	456	349

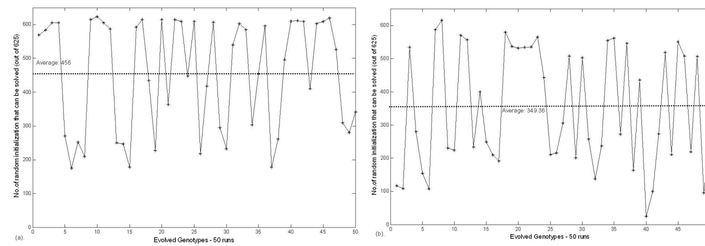

Figure 11: (a) SinglePole, (b) Double pole Task - Performance for Random initialization on 50 evolved genotypes

9 Conclusion

In this paper, plastic feedforward neural network based on the representation of Cartesian Genetic Programming (PCGPANN) is proposed and applied on a standard benchmark control problems. Plasticity involves mutating the neural network parameters namely weight, switches, inputs, functions and output at run-time. Plastic/Developmental decisions are made by using a developmental output from the network which induces plastic changes at runtime. The neurocontrollers produced by PCGPANN algorithm were found to be more robust than its FCGPANN counterpart. The network phenotype changes during run time, thus modifying its architecture in response to changing input patterns. The algorithm was also found to outperform most of the static neuroevolutionary algorithms in terms of speed of learning. It should be noted that the phenotypic mutations happen each time the new output is obtained from the evolved program in the genotype. However, in principle, this could be done at a slower rate (for instance, after a certain number of evolved program executions, rather than at every timestep). This remains to be investigated.

In future plasticity will be investigated for recurrent networks and tested for the non-markovian control problems. The goal is to investigate whether induction of both signal based feedback (recurrent) and structural feedback (plasticity) can enhance the performance. Also we intend to examine whether the PCGPANN is able to solve problems faster and more accurately by obtaining repeated experience of its task environment (post evolution). Eventually the aim is to investigate whether PCGPANNs can be trained to solve a sequence of problems without forgetting how to solve earlier problems.

REFERENCES

Ans, B., Rousset, S., French, R. M., and Musca, S. (2002). Preventing catastrophic interference in multiple-sequence learning using coupled reverberating elman networks. In *Proc. 24th, Annual conf. of cognitive science society*, pages 71–76. Erlbaum.

Baxter, J. (1992). The evolution of learning algorithms for artificial neural networks. *D.Green & T.Bossomaier(Eds.), Complex Systems*, page 313 326.

Cangelosi, A., Nolfi, S., and Parisi, D. (1994). Cell division and migration in a 'genotype' for neural networks. *Network-Computation in Neural Systems*, 5:497–515.

Clune, J., Beckmann, B. E., Ofria, C., and Pennock, R. T. (2008). Evolving coordinated quadruped gaits with the hyperneat generative encoding. *Proc. IEEE CEC'2008*, pages 2764–2771.

Dalaert, F. and Beer, R. (1994). Towards an evolvable model of development for autonomous agent synthesis. In *Brooks, R. and Maes, P. eds. Proceedings of the Fourth Conference on Artificial Life*. MIT Press.

Floreano, D. and Urzelai, J. (2000). Evolutionary robots with online self-organization and behavioral fitness. *Neural Networks*, 13:431 – 443.

French, R. M. (1994). Catastrophic forgetting in connectionist networks: Causes, consequences and solutions. In *Trends in Cognitive Sciences*, pages 128–135.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *J. Mach. Learn. Res.*, 9:937–965.

Gomez, F. J. and Miikkulainen, R. (1999). Solving non-markovian control tasks with neuroevolution. In *Proc. Int. joint Conf. on Artificial intelligence*, pages 1356–1361. Morgan Kaufmann Publishers Inc.

Gruau, F. (1994). Automatic definition of modular neural networks. *Adaptive Behaviour*, 3:151–183.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural network. In *Genetic Programming 1996:Proceeding of the First Annual conference*, pages 81–89 MIT Press.

Harding, S., Miller, J. F., and Benzhaf, W. (2010). Developments in cartesian genetic programming:selfmodifying cgp. *GPEM*, 11(2):397–439.

Hussain, T. and Browse, R. (2000). Evolving neural networks using attribute grammars. *Combinations of Evolutionary Computation and Neural Networks, 2000 IEEE Symposium on*, pages 37 – 42.

Jacob, C. and Rehder, J. (1993). Evolution of neural net architectures by a hierarchical grammar-based genetic system. In *Proc. ICANNGA93*, pages 72–79. Springer-Verlag.

Jacobi, N. (1995). *Harnessing Morphogenesis, Cognitive Science Research Paper 423, COGS*. University of Sussex.

Kandel, E. R., Schwartz, J. H., and Jessell (2002). *Principles of Neural Science, 4rth Edition*. McGraw-Hill.

Khan, G., Miller, J., and Halliday, D. (2007). Coevolution of intelligent agents using cartesian genetic programming. In *Proc. GECCO'2007*, pages 269 – 276.

Khan, M., Khan, G., and F. Miller, J. (2010a). Evolution of optimal anns for non-linear control problems using cartesian genetic programming. In *Proc. IEEE. ICAI'2010*.

Khan, M., Khan, G., and Miller, J. (2010b). Efficient representation of recurrent neural networks for markovian/non-markovian non-linear control problems. In *Proc. ISDA'2010*, pages 615–620.

Kitano, H. (1990). Designing neural networks using genetic algorithm with graph generation system. *Complex Systems*, 4:461–476.

McCloskey, M. and Cohen, N. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24:109–165.

Miller, J. and Smith, S. (2006). Redundancy and computation efficiency in cartesian genetic programming. *IEEE Trans. Evol. Comp.*, 10:167–174.

Miller, J. F. and Thomson, P. (2000). Cartesian genetic programming. In *Proc. EuroGP'2000*, volume 1802, pages 121–132.

Moriarty, D. (1997). *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, University of Texas at Austin.

Nicholas F.McPhee, Ellery Crane, S. E. and Poli, R. (2009). Developmental plasticity in linear genetic programming. *Proc. GECCO'2009*, pages 1019–1026.

Nolfi, S., Miglino, O., and Parisi, D. (1994). Phenotypic plasticity in evolving neural networks. In *Proc. Int. Conf. from perception to action*. IEEE Press.

Ratcliff, R. (1990). Connectionist models of recognition and memory:constraints imposed by learning and forgetting functions. *Psychological Review*, 97:205–308.

Risi, Sebastian., H. C. and Stanley, K. (2010). Evolving plastic neural networks with novelty search. *Adaptive Behavior*.

Rivero, D., Rabual, J., Dorado, J., and Pazos, A. (2007). Automatic design of anns by means of gp for data mining tasks: Iris flower classification problem. *Adaptive and Natural Computing Algorithms*, 4431:276–285.

Roggen, D., Federici, D., and Floreano, D. (2007). Evolutionary morphogenesis for multi-cellular systems. *Journal of Genetic Programming and Evolvable Machines*, 8:61–96.

Rust, A., Adams, R., and H., B. (2000). Evolutionary neural topiary: Growing and sculpting artificial neurons to order. In *Proc. ALife VII*, pages 146–150. MIT Press.

Sharkey, N. and Sharkey, A. (1995). An analysis of catastrophic interference. *Connection Science*, 7(3-4):301–330(30).

Sims, K. (1994). Evolving 3d morphology and behavior by competition. In *Artificial life 4 proceedings*, pages 28–39. MIT Press.

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

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural network through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Wieland, A. P. (1991). Evolving neural network controllers for unstable systems. In *Proc. Int. Joint Conf. Neural Networks*, pages 667–673.