Toward an Estimation of Distribution Algorithm for the Evolution of Artificial Neural Networks

Graham Holker
Ryerson University
350 Victoria St.
Toronto, ON, Canada
gholker@ryerson.ca

Marcus Vinicius dos Santos
Ryerson University
350 Victoria St.
Toronto, ON, Canada
m3santos@ryerson.ca

ABSTRACT

This paper presents the preliminary results of a unique method of neuroevolution called Probabilistic Developmental Neuroevolution (PDNE). PDNE builds upon Gene Expression Programming (GEP) and Probabilistic Incremental Program Evolution (PIPE). Instead of building a Probabilistic Prototype Tree, as in PIPE, a Probabilistic Prototype Chromosome is built. The chromosome has a similar structure to a GEP chromosome (head, tail, and weight domain) and contains probabilities for each element of the gene. With this methodology, neural networks can be expressed in a similar manner to GEP, and solutions can be evolved via an Estimation of Distribution Algorithm. Preliminary results show promise, but further work is required to match the results of GEP.

1. INTRODUCTION

In this paper we put forward the hypothesis that the evolvability of the topology and weights of an artificial neural network (ANN) can be improved by an Estimation of Distribution Algorithm (EDA) that maintains a probability distribution of encodings for the topologies and weights of candidate ANN solutions.

Neural networks exist in artificial and biological systems. ANNs were inspired by the biological neural networks found in brains [4]. ANNs contain any number of neurons connecting the inputs and the outputs. The neuron performs a weighted sum of its inputs and generates an output based on the sum. The specifics of neuron function and network architecture are application dependent. Artificial neural networks have many uses such as function approximation, regression analysis, classification, data processing, game playing, and robot control.

Neurons are defined by the number of inputs, the weights of those inputs, and their activation function. The inputs can come as inputs to the network or can be the output of neurons 'down-stream' in the network. The activation function determines whether the neuron's output is 1 or 0.

Evolutionary computation (EC) uses methods inspired by biological evolution: reproduction, mutation, recombination, and selection. Candidate solutions to a user-defined problem play the role of individuals in a population, and the fitness function defines the quality measure for the solutions. Evolution of the population takes place by repeatedly selecting the highest quality solutions from the population.

Neuroevolution is the generation of Artificial Neural Networks (ANN) through evolutionary computation. The type of problems where neuroevolution is often applied is that which a set of input-output pairs is unavailable, meaning supervised learning would not be possible. neuroevolution is best suited to applications such as game playing and robot control.

In order to evolve neural networks, an EC approach requires a method of representing a neural network, a method of generating new representations, and a method of testing fitness. Early neuroevolution approaches focused on evolving the weights of neural networks on a fixed topology as opposed to evolving the complete network, *i.e.*, weights and topology. For example Symbiotic Adaptive Neuro-Evolution (SANE) [5].

The method used in a SANE keeps a fixed topology with an input layer, an output layer, and a hidden layer of n neurons. Neurons from the hidden layer are evolved. The evolved details of the neuron are its connections with the input layer and the output layer including the weight of the connection. At the same time, network blueprints are evolved that detail which neurons (when combined in a network) perform well at the given task.

Alternatively, newer approaches, collectively called Topology and Weight Evolving Artificial Neural Networks (*TWEANN*), have focused on the evolution of the full neural network design. Examples include: Koza's Genetic Programming (GP) [3], Gene Expression Programming applied to Neural Networks (GEP-NN) [1], and Neuro-Evolution of Augmenting Technologies (NEAT) [7].

The GP approach to neuroevolution follows from canonical GP. In canonical GP a tree is built from a set of functions and terminals and new trees are developed by mutating or combining trees. GP for neural networks changes the function set and terminal set to represent ANN. Specifically the function set includes a neuron processing function which takes a defined number of inputs, a weighting function that performs multiplication but requires one argument to be a number (representing the weight) and the other to be a signal (either an input to the network or an output from another neuron). The next generation of trees is developed in a similar manner, but considerations are made such that the syntax remains correct.

GEP encodes trees in a gene such that the gene contains a head region (which contains functions or terminals) and the tail region (which contains only terminals). GEP-NN makes the function set neurons with a certain number of inputs and the terminal set the inputs into the network. The weights are encoded in a region called the weight domain. The genes are used to generate neural network trees which can be evaluated. GEP-NN is an example of an indirect encoding scheme since an encoding is evolved as opposed to in GP where the tree structure itself is evolved.

In NEAT, the networks evolved start as the simplest possible network and grow in complexity through structural mutations which add a connection or add a neuron. Markings are given to the neurons and connections such that ANN can be separated by species (*i.e.* networks of similar structure). Speciation is done in order to preserve the diversity of evolved networks.

In this paper we propose a new method of neuroevolution, here called Probabilistic Developmental Neuroevolution (PDNE), which uses an indirect encoding scheme inspired by Gene Expression Programming (GEP) [1]. The gene is a string whose constituents represent neurons, inputs, and the weights of the connections.

In order to generate populations, PDNE uses an Estimation of Distribution Algorithm (EDA). Instead of applying crossover and mutation to the previous generation, an EDA uses a list of probabilities to generate individuals. For example, a gene's tail region encodes for inputs; if there are two inputs then for each portion of the tail there would be two probabilities, one for the first input and one for the second. These probabilities are normalized (sum to zero) and are initialized with a normal distribution (e.g., first input probability is 0.5 and second input probability is 0.5). Probabilistic Incremental Program Evolution (PIPE) [6] demonstrates a Genetic Programming method using an EDA [6]; our algorithm uses a similar approach except it generates GEP genes not program trees.

We compare our results with those of GEP for evolving binary, feed-forward neural networks. We test fitness against the XOR and 6-bit multiplexer benchmarks used by Ferreira [1]. Preliminary results have shown promise. The challenge is in attempting to evolve the structure and weights simultaneously.

This paper is organized with a methods section, results section, and a discussion section. The methods section gives an overview of the artifacts being generated (ANN), the encoding scheme we used (GEP), and the generative process (EDA). The results sections describes the tests completed so far and the discussion compares the results to that of GEP.

2. METHODS

PDNE is designed as an EDA for neuroevolution. EDAs have been shown to be more efficient in Genetic Programming examples [6, 2] and it is our hope to make PDNE an efficient method of neuroevolution.

In this section we discuss the GEP chromosome and how it is capable of representing ANN. Then we discuss the specifics of the ANN and finally, we describe the representation and the algorithm developed for PDNE.

PDNE uses a probability distribution that is structurally similar to the GEP-NN chromosome. Instead of storing a single representation of a ANN, the structure represents a distribution of ANN representations. Each element of the chromosome stores probabilities for each function or terminal. From this representation a population of GEP-NN chromosome can be generated where the elements of each gene would be in accordance with the distribution. The distribution is updated with an algorithm similar to that developed in PIPE [6]; such that the distribution becomes incrementally closer to a solution.

2.1 GEP Chromosome

Gene Expression Programming for program trees contained a gene that listed functions and terminals. The gene was separated into two sections known as the head and tail regions (the former appears first in the gene). The tail consists of only terminals whereas the head contains functions and terminals. The tail region's length is defined by the length of the head region and the maximum 'arity' (number of parameters) of the functions. By defining the gene in this way any gene will represent a functionally correct program tree. The tail is long enough so in the case that the head consists only of functions of maximum arity there will be terminals to place as parameters to those functions. The opposing extreme is when the first portion of the head is a terminal. In that situation the output is equivalent to that terminal and the remaining elements of the gene are unused. Usually a gene will be somewhere between these two extremes.

In order to evolve neural networks with a similar gene the functions and terminals are substituted for neurons and inputs and a third region of the gene is defined. Neurons can be imagined to be functions which perform a summation on a given number of inputs. Neurons with 2, 3, and 4 inputs are called D, T, Q respectively. The terminals are replaced by inputs. The third region added to the GEP gene is known as the weight domain and encodes the weights. GEP-NN used 10 weights randomly generated in the range $-2, 2$. The elements of the weight domain are pointers to these 10 weights.

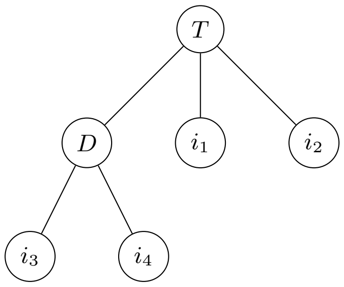
Before an ANN can be evaluated it must be generated from its gene encoding. The generative process begins at the leftmost element in the gene and constructs a tree in a breadth-first manner. The weights are then added in a breadth-first manner as well.

An example gene could be:

$$TDi_1i_2i_3i_4i_1i_2i_3W_1W_2W_3W_4W_5W_8W_6W_9$$

Where D, T, and Q are neurons with 2, 3, or 4 inputs (respectively). i_1 through i_4 are inputs to the network. Finally, W_1 through W_{10} represent the weights.

Which encodes the following ANN:



This example gene has a head length of 2. The tail size was calculated as: $T_l = (MA - 1) \cdot H_l + 1 = 7$. Where T_l is the tail length, MA is the maximum arity (or max number of inputs), and H_l is the head length. The length of the weight domain is $W_l = H_l \cdot MA = 8$.

Multiple genes can be used to generate a neural network, this is called a multigenic system. To combine multiple genes the networks are generated and those outputs are combined with an OR function. The OR function represented by a binary ANN has any number of inputs with a weight of 1 on each. Therefore, if one or more outputs 1 the output is 1.

In order to evolve these genes GEP uses a method involving crossover and mutation. This work did not use crossover and mutation, but replaced them with a probabilistic model-building approach that maintains a probability distribution of the most promising solutions and samples this distribution to create new individuals.

Another difference between PDNE and GEP is that the weights are not applied in a breadth-first manner. Instead the weights were grouped with respect to the neuron they would be applied to when used. The sections were as long as the maximum number of inputs. It was hoped that this would help when the structure changed such that the evolved weights would not be disrupted. For example if the T neuron in the above example became a Q neuron then i_3 would now be attached to Q and the weight would be the same as was between D and itself. This would disrupt functionality significantly. A similar change has been considered for building the structure such that small changes in the structure do not disrupt other areas. This will be attempted in future work.

2.2 Neural Networks

In this work, the evolved neural networks are functionally equivalent to those generated by GEP-NN [1] since the gene representation is the same. The limitation of this approach is that the networks are evaluated like a program tree. In a program tree there are functions and terminals; functions could be addition, subtraction and multiplication, and the terminals could be any floating point or integer number. The neural networks evolved by GEP and PDNE are limited to having only one output in the same way a program tree will only have one output. This could be mediated by evolving an ANN for each output, but this would be inefficient when the same or similar functionality is required by multiple outputs. The neurons are substituted in the place of the functions (addition, subtraction) and are limited to having only one output. Again, this limitation is mediated by evolving the same funcitonality multiple times. These are inefficiencies of the representation, but it is still possible to achieve functionally equivelant networks. ANN in theory can have multiple outputs and the neuron's output can be connected to multiple neurons 'up-stream' (similar to having multiple outputs). The terminals are substituted with inputs to the network. Since an input could be connected to multiple neurons the input, at least functionally, appears to have multiple outputs.

Another limitation of the system is that the ANN generated are feed-forward, meaning they contain no loops. The ANN evolved here are also binary in that their inputs and outputs must be either 1 or 0. These limitations could possibly be avoided with a different encoding scheme and are only limiting in applications where the evolved neural network requires such capabilities.

A network may consist of any number of neurons. In this study the simplest neural network possible is that where an input is directly connected to the output. This is the case when the first element of the gene represents an input.

A neuron computes its output in the following two steps:

1. It calculates a weighted sum of its inputs, as follows:

$$sum = \sum_{i=0}^{\text{number of inputs}} input_i \cdot weight_i$$

where $weight_i$ is a real number.

2. It 'fires' (outputs the value 1) if the weighted sum exceeds a threshold (we used a threshold of 1), as follows:

```
if sum >= threshold:
        output = 1
else:
        output = 0
```

Given two inputs and one output there are 2^2 possible input-output pairs. More generally, given a number of inputs n_i there will be 2^{n_i} input-output pairs for the network. Fitness was evaluated as $Fitness = 2^{n_i} - C$ where C is the number of inputs the ANN correctly evaluated. When $C = 2^{n_i}$ then the fitness is zero and we have found a solution to the given problem. Henceour goal is to minimize fitness.

2.3 Probabilistic Prototype Chromosome (PPC)

The Probabilistic Prototype Chromosome (PPC) created in this work is similar to the structure of the GEP gene shown above except a list of probabilities for each spot in the head,

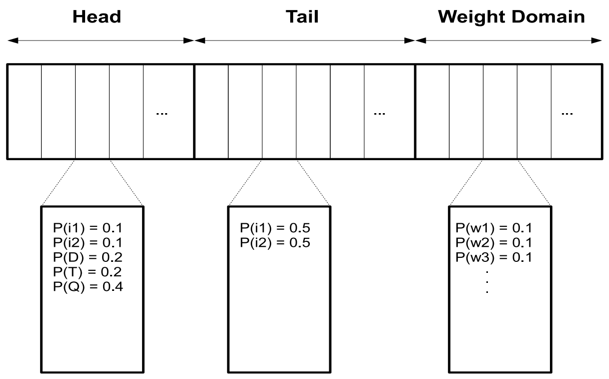


Figure 1: Probabilistic Prototype Chromosome

tail, and weight regions. It is similar to the PPC developed in Probabilistic Developmental Program Evolution (PDPE) [2] and the Probabilistic Probability Tree (PPT) from PIPE [6]. The structure is demonstrated in Figure 1.

The PPC is initialized with uniform probabilities for each element giving every GEP-NN gene equal probability. The algorithm produces a set of networks (a generation) and evaluates their fitness. The member with the best fitness is used as a model to adapt the PPC toward. Essentially, the probability of creating the best of generation neural network is calculated given the current PPC and then the PPC elements are incremented such that the probability of creating the generation's best ANN is increased. Hence, in the next generation the probability of creating a good gene has increased.

The probability of a particular gene being created is given as the product of each element's probability that was used in its generation:

$$P_{\text{gene}} = \prod_{i=\text{first element of head}}^{\text{last element used}} P_{\text{element}}$$

Then we define probability we'd like the gene to have:

$$P_{\text{target}} = P_{\text{gene}} + (1 - P_{\text{gene}}) \cdot LR \cdot ((\epsilon + f_E)/(\epsilon + f_B))$$

Where LR is the learning rate, ϵ is the fitness constant, f_E is the fitness of the best individual of all generations, and f_B is the fitness of the best individual of the current generation.

We then increment the probability of each element that contributed to the best gene, until the probability of the gene is greater than or equal to the target probability:

```
while (P(gene) < P(target)):
    for element in gene:
        P(element) += 0.1 * LR * (1 - P(element))
```

There is also a small probability of performing elitist learning, wherein the best of all generations (the elite individual) is used as a guide for adapting the PPC.

i_1	i_2	o_1
0	0	0
1	0	1
0	1	1
1	1	0

Table 1: XOR Truth Table

2.4 PPC Weight Domain

Evolving the weights at the same time as evolving the structure is a challenge because the function of a particular weight will change as the structure changes. NEAT [7] addresses this issue by using only two types of structural mutations. 1) adding a connection between two neurons and 2) adding a neuron. When changes occur within a GEP gene the changes can be much more disruptive, i.e. a small change in a GEP gene can have a bigger impact on the resulting ANN than the changes that occur in NEAT.

A list of 10 weights are initialized at the beginning and the system finds the probability of using a particular weight for each element in the gene. With only this, the weights would remain the same for an entire run, hence we followed the PIPE method of evolving the ephemeral constant (a real valued number used in a program tree). In PIPE, when an ephemeral constant is used it is modified by a small random value.

The current implementation of PDNE extended this idea for the evolution of weights. Each location in the weight domain contains its own array of the weights and modifies them on each use. There is also a check to see whether the modification improved fitness and if not the value is not changed. The change is also much larger than the changes to the ephemeral constant in PIPE. The change is a normally distributed random value centred on zero with a range of 2.

2.5 PPC Mutation

For every generation, this method of mutating the PPC is performed. First we find the probability of mutating a single element in the PPC, defined as:

$$P_{\text{mutate element}} = \frac{P_{\text{mutation}}}{(F + I + W) \cdot \sqrt{size}}$$

Where F, T, and W are the number of functions, the number of terminals, and the number of weights respectively.

We modify each element in the PPC with the probability of mutating an element ($P_{\text{mutate element}}$), but only those that were used in creating the generation's best individual. We add to the probability of the element selected a small amount:

$$P_{\text{element}} = M_R \cdot (1 - P_{\text{element}});$$

Where M_R is the mutation rate.

3. RESULTS

The algorithm was tested with two benchmarks, *viz.* the XOR problem and a 6-bit multiplexer. XOR takes two inputs and has the truth table as shown in Table 1.

It is a good benchmark because it is simple. There are 4 possible input combinations and therefore the fitness ranges

Number of runs	100
Number of generations	50
Population size	30
Function set	D T Q
Terminal set	a b
Weights array length	10
Weights range	[-2, 2]
Head length	4
Number of genes	1
Chromosome length	33
Probability Elitist Learning	0.01
Learning Rate	0.01
Fitness Constant (ϵ)	0.000001
Probability of Mutation	0.4
Mutation Rate	0.4

Table 2: Experimental setup for XOR benchmark

from 0 (every fitness case is satisfied) to 4 (none of the fitness cases are satisfied). It is also a good test because it is only possible in neural networks with at least two layers.

The 6-bit multiplexer takes 6 inputs where the first two represent the index and the other four represent the inputs. The index represents which of the inputs should be given as the output. E.g., for inputs (i_0, i_1, i_2, i_3) an index of 00 would output i_0, 01 would output i_1, 10 would output i_2, and 11 would output i_3. There are 64 input cases (2^6) and so our fitness ranges from 0 to 64.

The benchmarks were run using a combination of parameters as defined in GEP and PIPE.

3.1 XOR

For the XOR benchmark we used parameters listed in Table 2.

GEP success rate for such a test was 77%. Since the fitness case is either correct or not, we define success as finding a solution with fitness of zero. In our trial, the model achieved a 32% success rate.

Further testing showed that larger populations aided PDNE in discovering the correct solution. So that the results remained comparable to those of GEP we ensured that the total number of evaluations remained the same. For example, GEP used 50 generations and a population size of 30; the number of evaluations is the product,1500. We ran the experiment again with 5 generations and a population size of 300 and achieved a success rate of 55%.

3.2 XOR - Compact

The XOR Compact test has the same parameters as the XOR experiment except that the head length of the gene is 2 (as opposed to 4). This results in a gene length of 17. Our results in this test were 9%, whereas GEP achieve 30%. Using 5 generations and a population size of 300, the success rate rose to 32%.

Number of runs	100
Number of generations	2000
Population size	50
Function set	D T Q
Terminal set	a b c d e f
Weights array length	10
Weights range	[-2, 2]
Head length	5
Number of genes	4
Chromosome length	124
Probability Elitist Learning	0.01
Learning Rate	0.01
Fitness Constant (ϵ)	0.000001
Probability of Mutation	0.4
Mutation Rate	0.4

Table 3: Experimental Setup for 6-bit multiplexer benchmark

3.3 6-Bit Multiplexer

For the 6-bit multiplexer benchmark the parameters used are shown in Table 3.

The GEP success rate for this benchmark was 6%. Our results for this test have shown a 0% success rate. The average fitness of the best of run individuals for our test was 16. Increasing the population size showed similar results.

4. DISCUSSION

The results are not at the level of GEP except in the compact XOR experiment. This shows the promise of the PDNE algorithm. In the 6-bit multiplexer experiment the average fitness was found around 16. Since the networks are binary for each fitness case there is a 50% chance that the system will get the correct answer randomly. An average fitness of 16 shows that our system finds an output that is correct about three-fourths (75%) of the time. This shows that our system is learning and converging on a solution, just not before running out of evaluations.

Using a larger population and less generations proved to be effective at improving the results. This can be attributed to the nature of incrementing the probabilities. With a smaller population the probabilities may be incremented toward a bad result or could undo adaptations toward a good result. Therefore, it is in our best interest to use a large population and find a significantly good individual. The variance is larger in EDA systems compared to an evolutionary system with crossover and mutation because the individuals created are highly random. This variance gives more credence to the need for a high population.

The weight domain is a challenge to evolve. PDNE attempts to evolve the structure and the weights simultaneously. The difficulty is that a change in the structure will affect how the weight affects the output. The opposite is also true, in that a change in the weights will affect the functionality of two networks with the same structure. We have made an attempt at mitigating this problem by grouping the weights such that they are not moved from neuron to neuron when the structure changes.

Another problem with disruptive changes occurs within the structure alone. When an element early on in the gene changes, it can disrupt the functionality of the elements further along in the gene. Hence, a method to group the neurons and inputs by layer and by the neuron they connect to could mitigate this issue. If the maximum inputs is 4 then after the first element of the gene the next four elements would be part of a group (the second layer). Then the next 16 would be grouped as the third layer and those 16 would be broken into 4 groups of 4 by the neuron they connect to. This way learned functionality is not lost and does not need to be unlearned.

When two viable solutions are discovered they are likely to compete. For example, if an element in the head could create a viable solution with a Q or a T then in the model at that element the $p(T) \approx p(Q) \approx 0.5$. The result of this is that the system has slowed or even halted its convergence on a single solution. This is because there are multiple solutions that are correct.

5. CONCLUSION AND FUTURE WORK

In this paper we have presented a method of performing neuroevolution with an EDA. The results show that the method is viable, but we have seen that it is not yet performing well enough.

The approach presented here has the following limitations. Firstly, the proposed probabilistic model is unable to account for dependencies between genetic elements. *E.g.* weight elements are highly dependent on the structural elements. This could be addressed through a model able to capture jointly distributed distributions. *I.e.,* $p(x, y)$ which will return the probability of the current element being y given the previous element is x.

Secondly, we have shown how two solutions can compete in the probabilistic model and how convergence on a solution is slowed or even halted as a result. This could be addressed through speciation as in NEAT or could also be alleviated by dependent probabilities as described above. Finally, real-valued weight evolution could be improved by searching for the mean and the standard deviation as opposed to the value itself.

6. REFERENCES

[1] Candida Ferreira. Designing neural networks using gene expression programming. *Applied Soft Computing Technologies: The Challenge of Complexity*, 34, 2006.

[2] Elmira Ghoulbeigi and Marcus Vinicius dos Santos. Probabilistic developmental program evolution. 2009.

[3] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[4] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[5] D.E. Moriarty and R. Miikkulainen. Hierarchical evolution of neural networks. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 428 –433, may 1998.

[6] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary computation*, 5(2):123–141, 1997.

[7] Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 569–577, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.