Evolution of Layer Based Neural Networks: Preliminary Report

Edward R Pantridge
Hampshire College
Amherst, MA 01002
erp12@hampshire.edu

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

ABSTRACT

Modern applications of Artificial Neural Networks (ANNs) largely feature networks organized into layers of nodes. Each layer contains an arbitrary number of nodes, and these nodes only share edges with nodes in certain other layers, as determined by the network's topology. Topologies of ANNs are frequently designed by human intuition, due to the lack of a versatile method of determining the best topology for any given problem. Previous attempts at creating a system to automate the discovery of network topologies have utilized evolutionary computing [15]. The evolution in these systems built networks on a node-by-node basis, limiting the probability of larger, layered topologies. This paper provides on overview of Growth from Embryo of Layered Neural Networks (GELNN), which attempts to evolve topologies of neural networks in terms of layers, and inter-layer connections, instead of individual nodes and edges.

1. INTRODUCTION

Artificial neural networks are a method of machine learning that can be applied effectively to a wide variety of problems. One of the biggest determiners of a networks effectiveness at solving a problem is its topology. A neural network topology is defined by the number of nodes, and how the nodes are organized and connected.

This paper briefly discusses the progress that has been made in making them more versatile. Previous attempts at automating the discovery of effective topologies with evolution are considered, and a new method of searching for effective topologies using evolution will be presented.

To fully understand how evolution was used to automate the discovery of neural network topologies, a brief explanation of Genetic Programming is provided. The Growth from Embryo of Layered Neural Networks (GELNN) system detailed in this paper specifically uses a Push genetic programming system for a variety of reasons explained in later sections.

An encoding of neural network topologies was created for the system described in this paper. This encoding was designed to be easily generated by a Push genetic programming system, and is distinct from many other neural network encodings in that it represents the network's topology in terms of layers, each containing multiple nodes. This contrasts with many other network encodings which represent networks in terms of either sets of nodes, sets of edges, or a combination of both [15] [8].

The network topologies generated by the genetic programming system are trained on classification problems using backpropagation. Their training and generalization errors after training are recorded and used during their evolutionary fitness evaluation.

Together, a neural network framework and a Push genetic programming system are used to test the effectiveness of Growth from Embryo of Layered Neural Networks (GELNN) system, which aims to evolve network topologies on a layer-by-layer basis.

GECCO'16 Companion, July 20 - 24, 2016, Denver, CO, USA

ISBN 978-1-4503-4323-7/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2908961.2931664

2. BACKGROUND

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) performs computations by exchanging signals between artificial neurons in a way that was largely inspired by biological nervous systems [3]. Despite many improvements in the training of artificial neural networks, [4] less progress has been made on how best to choose an effective network topology.

One current common practice is to attempt to design smaller sub-networks that are trained to perform specific parts of an overall computation. These trained sub-networks are then included in a much larger neural network, and the larger network is trained as a whole [5].

The question of how best to design a topology of a neural network is still largely unanswered. The method described in this paper, Growth from Embryo of Layered Neural Networks (GELNN) aims to address this issue.

2.2 Genetic Programming and PushGP

Genetic Programming (GP) is a method of generating computer programs using an evolutionary algorithm, as diagramed by Figure 1. GP begins with an initial population of random programs. These programs are evaluated based on a fitness function which determines how effective at completing the desired task each program is [2]. Programs with the best fitness are then processed through various genetic operators, such as mutation and crossover, to produce a new generation of the population. Mutation operators take a

single program from the population and randomly change its genome. Crossover operations take two programs from the population and merge their genomes in a way that is inspired by gene crossover in biological DNA [11]. After creating a new generation, the GP algorithm returns to the fitness evaluations and repeats the cycle until a program in the population can perform the desired task [2].

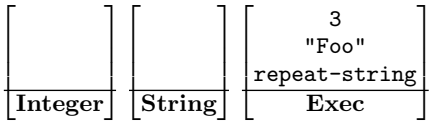
PushGP is a genetic programming system that evolves programs in the Push language. Push is a Turing complete, stack based language that features a separate stack for each data type, including code. Programs in Push systems are expressed as lists of instructions and literals. Literals are values that get placed on their corresponding stacks when processed. Instructions pop values off the stacks, modify them, and push them back on the appropriate stacks. The list of instructions is run sequentially through an interpreter, modifying the stacks, and the final state of the stacks is the output of the program [13].

To illustrate the execution of a Push program, the following Push program will be used as an example.

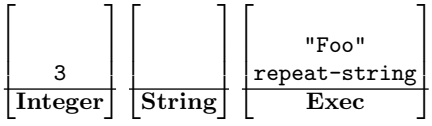
```
(3 "Foo" repeat-string)
```

In this Push program, there are two literals (`3` and `"Foo"`) and one instruction (`repeat-string`). The `repeat-string` instruction is defined as part of the Push interpreter as a function that takes one string argument, s, and one integer argument, i. The `repeat-string` pushes one string containing the string s repeated i times onto the string stack.

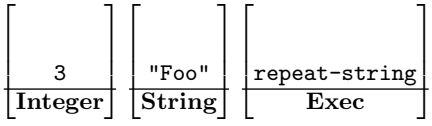
Below is a figure depicting the state of the push stacks before the above program is run.



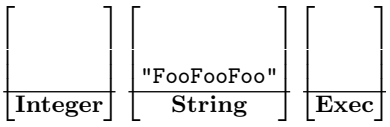
The `Exec` stack holds the list of instructions and literals that have yet to be executed. The top value on the `Exec` stack is popped off and processed. In this example, it is the terminal `3`. The terminal is pushed to its corresponding stack. In this case, the integer stack.



The next value is then popped from `Exec` stack. The value `"Foo"` is a string terminal, and is pushed to the string stack.



The next value on the `Exec` stack is the `repeat-string` instruction. Upon processing the `repeat-string` instruction, the Push stacks would look as follows:



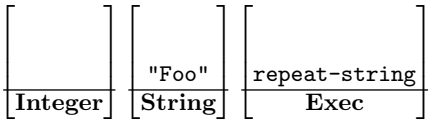
The `Exec` stack is now empty, and the state of Push stacks is the output of the program.

Representing programs this way in a GP system has proven very effective. Almost any combination of instructions is a valid program. When an instruction takes inputs of a certain type, and the corresponding stack is empty, the instruction is ignored and the program can continue to be processed by the interpreter.

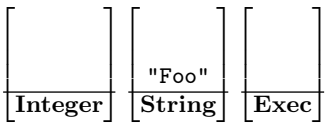
For example, if the example Push program from above is modified to be:

```
("Foo" repeat-string)
```

The Push stacks would reach the following state:



The `repeat-string` instruction will then be processed by the interpreter, but will have insufficient arguments due to the fact the `integer` stack is empty. The `repeat-string` instruction will be disregarded and the resulting state of the Push stacks would be:



This makes the implementation of a variety of genetic operators trivial, as mutations and crossovers will never produce programs that cannot be run by the interpreter [13]. This project relies heavily on this property of PushGP to ensure only valid ANN topologies are produced by evolution.

Push genetic programming systems can be easily tailored to tackle specific tasks through the addition of new push instructions and stacks. To control the evolution of valid neural network encodings, additional Push instructions were written to incrementally build an encoding on a designated `auxiliary` stack. These Push instructions are detailed in a later section.

2.3 NEAT

One well known previous attempt at using evolutionary computing to build neural network topologies is NeuralEvolution of Augmenting Topologies (NEAT) [15]. NEAT is a method of using a genetic algorithm to evolve a neural network topology and the weights of the network's edges.

Similar to the GELNN system proposed in this paper, NEAT systems use evolutionary computing to produce a network encoding. The NEAT method uses genetic encodings to represent the evolved neural network. Genetic Encodings consist of a list of *connection genes* and a separate list of *node genes*. Each *connection gene* refers to two *node genes* and stores the weight value of the connection between

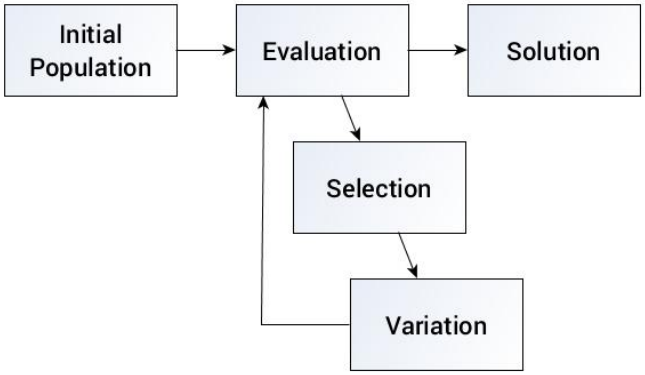


Figure 1: The evolutionary computing cycle used by genetic programming. The population consists of programs. The evaluation stage runs the programs and examining their output to determine fitness.

the two nodes. When a genetic encoding is undergoing variation (ie. mutation or crossover) the operation must ensure that each *node gene* referred to by the *connection genes* is present in the resulting encoding. This restricts the possible changes to the genetic encoding through mutations and crossovers.

As noted, NEAT attempts to evolve both the topology and weights of the ANN. Methods of evolving both ANN weights and topologies are called *Topology and Weight Evolving Artificial Neural Networks* (TWEANNs). In particular, NEAT evolves weights using a specialized mutation operation, which randomly "perturbs" the weights in a random subset of *connection genes*. In contrast with NEAT, GELNN systems are designed to evolve networks made up of a larger number of nodes and edges. For this reason, the network's weights are not optimized by evolution. Instead GELNN network encodings represent topology only, and are translated into neural networks with random weights and later trained using Backpropagation [12].

NEAT systems start with a population of genetic encodings that are as small as possible. Each generation of a NEAT system incrementally grows the genetic encodings by adding connections, adding nodes, or changes weights. To evaluate the population at each generation, these genetic encodings are directly translated into networks and their performance is tested. In other words, the NEAT treats the encodings produced by evolution as the genomes in the evolutionary population as well.

GELNN systems take a distinctly different approach to evolving neural networks, by using a genetic programming (GP) system. GELNN systems use a GP system, such as PushGP, to evolve programs. When executed, these programs produce a valid artificial neural network encoding. These encodings are then decoded into a network, trained, and tested.

GELNN system were designed to use GP system, as opposed to a genetic encoding to allow due to the increased flexibility provided by GP. One way in which the use of a GP system allows for more flexibility is that it is possible for the programs evolved by GP to utilize loops and other forms of modularity that could result in repeated topology within the network.

Another benefit of using a GP system is the wider variety of genetic operators that still produce valid network encodings. NEAT systems had separate mutation operators to add nodes, add/remove connections, modify weights. Nodes would only be removed from a genetic encoding if no edge pointed to them. These specialized mutation operators were created to ensure the genetic encoding always expressed a valid network. In GELNN systems, the genetic programming system can utilize any genetic operator which produces a valid program, and the result of executing the program will be a valid network encoding.

3. GROWTH FROM EMBRYO OF LAYERED NEURAL NETWORKS (GELNN)

3.1 Network Encoding

Artificial Neural Networks are often thought of as graphs of nodes and edges. In larger neural networks, it is common to organize nodes into layers. ANNs tend to include an input layer, output layer, and an arbitrary number of hidden layers. Figure 2 depicts an ANN with 3 layers: 1 input layer, 1 hidden layer, and 1 output layer. As shown by the arrows in Figure 2, the input layer is "fully connecte" to the hidden layer. This means that every node in the input layer has an edge leading to every node in the hidden layer. The hidden layer is also "fully connected" to the output layer. In larger ANNs it is common for some of the many layers to be connected in ways other than "fully connected." This project does not explore the possibilities of determining the best method of connecting layers through evolution, but this is likely a valuable area of future research.

To encode the topology of the network in Figure 2, there are a few properties that must be known: How many layers there will be, how many nodes will be in each layer, which layers are connected, and finally, how should the nodes in connected layers share edges.

The number of layers in an encoded network is given by a simple list of layer ids. The ids of I and O are required, as every network requires an input and output layer. Every other layer is given the id of Hx where H denotes a hidden layer, and x is a unique integer value.

To encode the number of neurons in each layer, each id described above is used as a key in a key-value data struc-

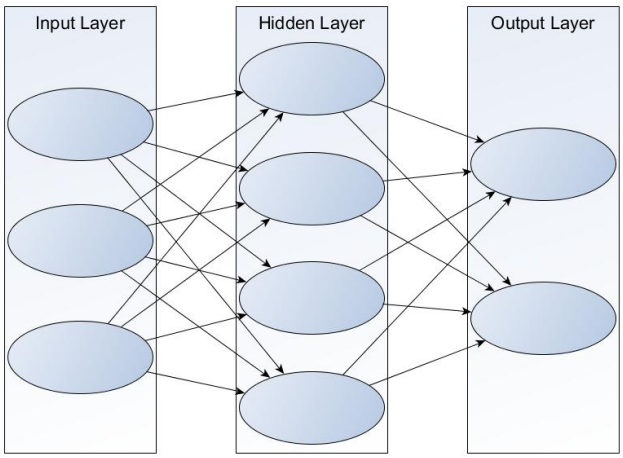


Figure 2: A neural network consisting of an input layer with 3 nodes, and output layer with 2 nodes, and a single hidden layer with 4 nodes.

ture[1]. The corresponding values in the data structure for each given id is another key-value data structure containing all information pertaining to the relevant layer. For the scope of this project, there will be a single value: the number of nodes in the layer. This value can be any positive integer.

Encoding which layers are connected to each other is done through a simple list of connections. A connection is denoted using a two element list containing the id of the layer the connection is coming from and the id of the layer the connection is leading to. The order of these ids is crucial, as inter-layer connections are one directional.

Although it is necessary to know which nodes share edges between connected layer, as mentioned earlier, we are making the assumption that all connected layers are fully connected.

The GELNN encoding for the artificial neural network in Figure 2 would be as follows:

Figure 2 GELNN Encoding
Layers I → {num-nodes = 3}, H1 → {num-nodes = 4}, O → {num-nodes = 2}
Layer Connections $[I \Rightarrow H1]$, $[H1 \Rightarrow O]$

The above technique of encoding ANNs will always create a valid representation of a neural network given that the number of nodes in each layer is greater than zero, and there is exactly one input layer labeled `I` and exactly one output layer labeled `O`.

One benefit of encoding the neural network this way is that any additional information about a layer can easily be added as an entry to the layers corresponding id in the `layers` data structure. For example, additional features that could be implemented include different layer connection types other than fully connected, and different transfer functions for each layer.

This encoding of a neural network is also extremely easy to simplify, or remove layers that will have no effect. For example, if inside the `layer-connections` there is a connection from the output layer to a hidden layer, say `[O, H3]`, yet there is no connection `[H3, L]` where `L` is any layer id, then it is clear that `H3` will not affect the output or training of the network, and can be eliminated to save on computing time and memory during training. This functionality could prove to be very beneficial when the network encoding is the result of the evolutionary cycle, because it is very likely that at many points through an evolutionary run, individuals in the population will contain useless topology that can be removed.[2]

3.2 Building Network Encodings Using Push Instructions

This project requires the use of a push interpreter with specially made instructions, as well as the use of an `auxiliary` stack to hold the current state of the network encoding.

When a program of push instructions is processed by the push interpreter, the `auxiliary` stack is first loaded with an "embryo encoding." This is the smallest valid network applicable to the data-set or problem an effective topology is being found for. To determine an optimal topology for the same problem the network in Figure 2 is applied to, the "embryo encoding" would be as follows:

Figure 2 GELNN "Embryo Encoding"
Layers I → {num-nodes = 3}, O → {num-nodes = 2}
Layer Connections $[I \Rightarrow O]$

Note that the number of nodes in the input layer is three, and the number of nodes in the output layer is two. This restricts the neural network encoded here to problems which provide three features as inputs, and attempt to predict two output values.

The "embryo encoding" is the initial state of the `auxiliary` stack, and is modified incrementally by all other neural network push instructions in the program to produce a full network encoding.

All neural network push instructions made for this project were designed with the assumption that there would be a valid neural network encoding on the `auxiliary` stack. All neural network push instructions modify the encoding on the auxiliary in a way that always produces a valid neural network encoding, which replaces the previous network encoding on the `auxiliary` stack.

The instructions made for this project were heavily inspired by the topology operators described by Sean Luke and Lee Spector in *Evolving Graphs and Networks with Edge Encoding: Preliminary Report* [8]. The reason GELNN systems

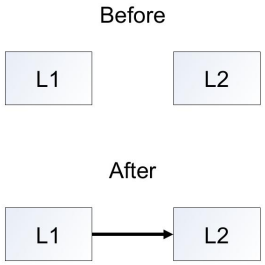
[1]Such as a Python dictionary. In the experiments done for this paper, the data structure was a Clojure map

[2]It is important during evolution for these pieces of "useless" topology to not be removed from the genome, but rather just from the phenome which in this case is the network topology encoding. Otherwise, important genetic information that could result in better performance in future generations will be lost.

use these push instructions is to ensure any arrangement of these instructions will produce a valid network. This contrasts with NEAT's Genetic Encodings, which could only undergo specialized mutation and crossover operations to ensure that a valid network topology was being represented.

3.2.1 Connect Layers

The `connect-layers` instruction pops two integers, i and j, off the integer stack. The current state of the network encoding is popped off the `auxiliary` stack.

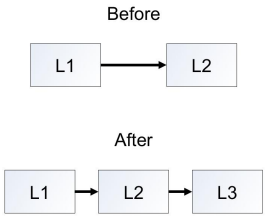


Both i and j are brought between zero and the number of layers using a modulus operation. The layers at the indices i and j are selected from the `layers` collection. These layers can be denoted as L_1 and L_2.

An inter-layer connection from L_1 to L_2 is created. It can be denoted as $C(L_1, L_2)$. This inter-layer connection is added to the `layer-connections` collection in the network encoding. The resulting network encoding is pushed onto the `auxiliary` stack.

3.2.2 Bud

The `bud` instruction pops one integer, i, off the integer stack. The current state of the network encoding is popped off the `auxiliary` stack.



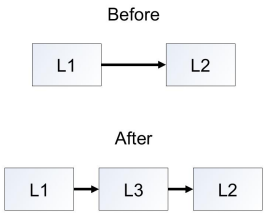
The inter-layer connection, in the list of `layer-connections` at index i, denoted $C(L_1, L_2)$, is selected.

A new layer, L_3, is added to the collection of `layers`, and given the id `:Hx` where `x` is the next sequential natural number not yet included in the id of a hidden layer. The number of nodes in layer L_3 is set to be equal to the number of nodes in L_2.

An inter-layer connection $C(L_2, L_3)$ is added to the collection of `layer-connections`. The resulting network encoding is pushed onto the `auxiliary` stack.

3.2.3 Split

The `split` instruction pops one integer, i, off the integer stack. The current state of the network encoding is popped off the `auxiliary` stack.



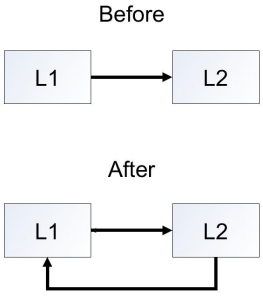
The inter-layer connection in the list of `layer-connections` at index i, denoted $C(L_1, L_2)$, is selected.

A new layer, L_3, is added to the collection of `layers`, and given the id `:Hx` where `x` is the next sequential natural number not yet included in the id of a hidden layer. The number of nodes in layer L_3 is set to be equal to the number of nodes in L_1.

The inter-layer connection $C(L_1, L_2)$ is removed from the collection of `layer-connections`. Two inter-layer connections, $C(L_1, L_3)$ and $C(L_3, L_2)$, are added to the collection of `layer-connections`. The resulting network encoding is pushed onto the `auxiliary` stack.

3.2.4 Loop

The `loop` instruction pops one integer, i, off the integer stack.

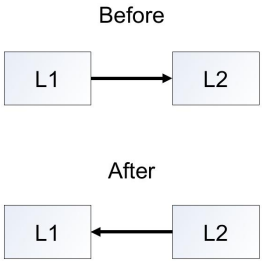


The inter-layer connection, $C(L_1, L_2)$, in the list of `layer-connections` at index i is selected.

A new inter-layer connection, $C(L_2, L_1)$, is added to the collection of `layer-connections`. The resulting network encoding is pushed onto the `auxiliary` stack.

3.2.5 Reverse

The `reverse` instruction pops one integer, i, off the integer stack.



The inter-layer connection in the list of `layer-connections` at index i, denoted $C(L_1, L_2)$, is selected.

The inter-layer connection $C(L_1, L_2)$ is removed from the collection of `layer-connections`. A new inter-layer connection, $C(L_2, L_1)$, is added to the collection of `layer-connections`. The resulting network encoding is pushed onto the `auxiliary` stack.

3.2.6 Set Number of Nodes In Layer

The `set-number-nodes-layer` instruction pops two integers, i and j, off the integer stack. The current state of the network encoding is popped off the `auxiliary` stack.

The layer at index i is selected. If this layer is the input layer given by the id `I`, or the output node given by the id `O`, the instruction is skipped and has no effect. This is to prevent the resulting ANN encoding from becoming incompatible with the applied dataset or problem.h

If the selected layer is any hidden layer, the `num-nodes` value is set to j. The resulting network encoding is pushed onto the `auxiliary` stack..

3.3 Full GELNN Process

A Growth from Embryo of Layered Neural Networks (GELNN) system consists of a push genetic programming framework and an artificial neural network framework.

The push GP framework must be modified to have a dedicated `auxiliary` stack to store the network encoding that is being "grown," and the previous detailed push instructions must be added.

The neural network framework must be capable of decoding the network encodings described earlier in this paper, and contain an implementation of the backpropagation algorithm capable of training networks with an arbitrary

number of hidden layers, each with any positive number of nodes. The neural network framework must also be capable of testing how well the network was trained.

As diagrammed in Figure 3, the process of discovering effective neural network topologies with GELNN starts with the Push GP system generating a population of random Push programs. These programs consist of the instructions detailed in this paper.

The next stage of evolution is the fitness evaluations of every individual in the population. In a GELNN system, the first step to evaluating the fitness of the population is to process all programs in the population through the push interpreter to produce valid network encodings. These encodings are then passed to the ANN framework capable of constructing connectivity matrix implementations from them. These decoded neural networks are trained by the ANN framework using backpropagation, and their errors on training, hybrid, and generalization data-sets are given back to the PushGP framework to be used as their fitness values.

The PushGP system continues on with the evolutionary cycle by selecting parents and modifying the programs in the population to produce the next generation. The genetic operators used to select parents and perform mutations and crossovers for the experiments performed for this project are lexicase selection, alternation, and uniform mutation [14].

If at any point during evolution the generalization error of one of the artificial neural networks is reported to be below a given stopping threshold, evolution is interrupted and the topology of that network is considered effective.

4. PRELIMINARY RESULTS

4.1 XOR

As an preliminary test of GELNN, an effective topology for the XOR problem was evolved. This problem was chosen due to the fact that the embryo encoding has been proven to be incapable of learning a solution [9].

After only 30 generations of evolution, a solution to the XOR problem was evolved. The program producing the best topology for the XOR problem is as follows:

```
(integer-max exec-pop () integer-dec integer-mult
integer-pop nn-bud exec-rot (integer-mod integer-
stackdepth nn-split exec-yankdup exec-dorange
(integer-stackdepth) exec-dorange (exec-dotimes
(exec-yankdup exec-flush exec-dotimes (integer-
yankdup exec-dorange (integer-swap integer-dup exec-
dup (exec-yankdup) 2 integer-rot exec-yankdup) exec-
flush integer-shove exec-y (integer-rot) integer-
swap)))) () ())
```

The GELNN push instructions in the above figure are underlined. All other instructions in this programs are instructions implemented in Clojush that manipulate the `integer` and `exec` stacks. These instructions allow for loops and other modular code in the programs evolved by Push [13].

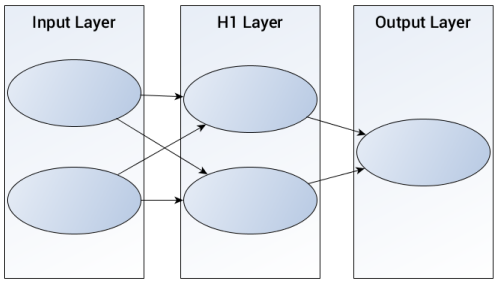
Many instructions in the above programs have no effect on the output of the overall program. This is due to a lack of arguments to these instructions on the stacks. By removing the Push instructions that do not have an effect on the output of the program, a more simplified program can be produced. After simplification of the above program, the same push program can be expressed using the following instructions:

```
(integer-stackdepth nn-split)
```

The `integer-stackdepth` instruction pushes the the number of elements in the `integer` stack onto the `integer` stack. In this case it pushes a 0. Then `nn-split` is processed as described earlier in this paper. After processing this program through a push interpreter, the following GELNN encoding is left `auxiliary` stack.

XOR GELNN Encoding
Layers I → {num-nodes = 2}, O → {num-nodes = 1}, H1 → {num-nodes = 2}
Layer Connections $[I \Rightarrow H1]$, $[H1 \Rightarrow O]$

After decoding this topology, the resulting network topology can be diagrammed as such:



This network topology was trained using backpropagation and the following hyper-parameters:

Hyper-Parameter	**Value**
Transfer Function	Sigmoid
Transfer Function Derivative	Derivative of Sigmoid
Learning Rate	0.5
Max Initial Weight	0.1
Max Epoch	1000
Validation Stop Threshold	0.08

Below is a plot of this network's validation error every epoch of training on the XOR problem.

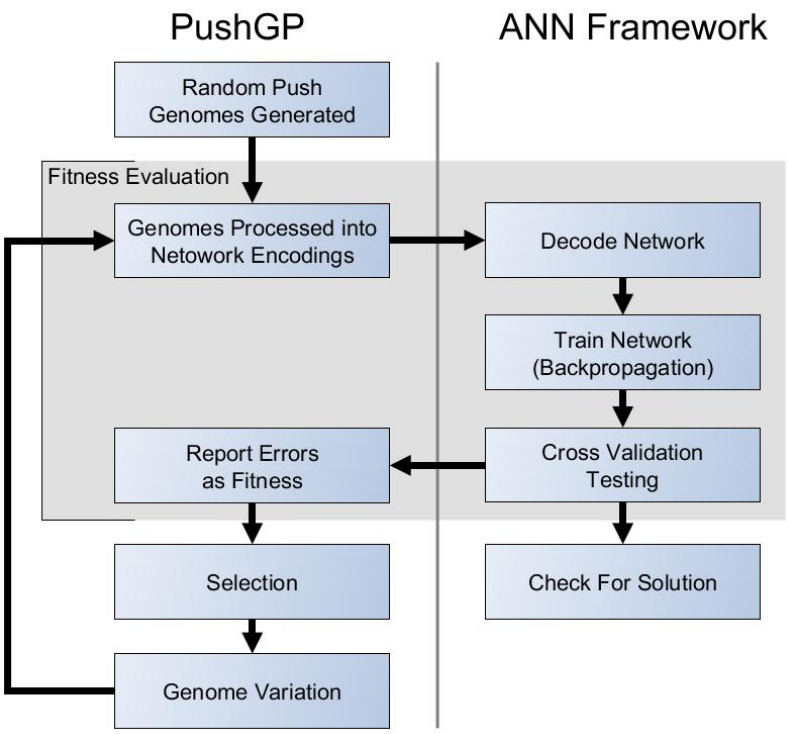
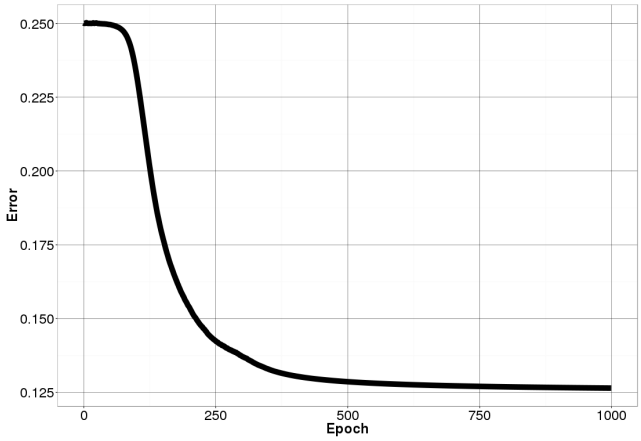


Figure 3: A diagram of the process within a GELNN system. For the preliminary experiments done for this project the PushGP system chosen was "Clojush" and an ANN framework called "Cloann" was developed. Both frameworks are implemented in the programming language Clojure. Source code for Clojush can be found at `https://github.com/lspector/Clojush` and the source code for Cloann can be found at `https://github.com/erp12/Cloann`



It is clear that this topology is effective at learning a solution to the XOR problem because it was able to begin performing far better than random chance in less than 300 epochs.

4.2 Other Data-Sets

The GELNN system was tested on a variety of other data-sets with less success than XOR. The data-sets the GELNN system was tested with are: Iris [7], Wine [1], and Student Alcohol Consumption [10]. So far, only a few experiments on these data-sets have been run, due to the computationally intensive nature of a GELNN system. For these data-sets, every experiment results in the system being unable to produce topologies that were more effective the topology described by the embryo encoding.

These experiments were not a strong indication of GELNN's effectiveness, as they were restricted to a less than 50 evolutionary generations with a maximum of 100 individuals in the population. These restrictions were imposed to keep the run time of the experiments reasonable, and thus allow for an initial glimpse into the nature of GELNN systems.

The results of these small experiments could speak to a weakness of GELNN systems, or the simplicity of the three chosen data-sets. Embryo encodings are simply a neural network with no hidden nodes. It is possible that any network with hidden nodes is either training less effectively on these data-sets, or over-fitting the training data and reporting a poor generalization error during fitness evaluation.

Further investigation into a wider variety of data-sets, and less restricted evolutionary runs, is needed to obtain a full understanding of GELNN's capabilities.

5. FUTURE RESEARCH

5.1 Auxiliary Layer Attributes

The GELNN system described in this paper uses evolution to search for effective ANN topologies by optimizing the number of layers, how many nodes are in each layer, and how those layers are connected. There are a number

of other attributes of neural networks that evolution could optimize. One such attribute is the transfer function used by the network. It is not uncommon for networks to use different transfer functions for the nodes in different layers. There are a variety of transfer functions that have been found useful by human designed networks. Evolution could either search for the most appropriate of these functions, or could be used to evolve the actual function.

The latter option is likely a difficult task due to the restrictions of what is an acceptable transfer function. It is generally considered useful for transfer functions to be monotonically increasing, continuous, and differentiable. If evolution were to evolve the transfer functions using a GP system, it would introduce the question of how to deal with transfer functions that were not monotonically increasing, continuous, and differentiable. For this reason, it is likely a more logical next step to allow evolution to pick the optimal transfer function for each layer from a list of known effective functions.

Another attribute of ANN layers that evolution could search over is the way that layers are connected. The current implementation of GELNN assumes that if an inter-layer connection is present between two layers, those layers are fully connected. Recent advances in artificial neural networks have shown the effectiveness of convolutional topologies that contain layers connected in a variety of ways other than fully connected [6].

Evolution could find the optimal value for another attribute which controls the layer's type (ie. Fully Connected, Convolutional, Max Pooling, Soft Max, etc). This would complicate the decoding of the network, as some layer types would need to be followed by layers containing a certain number of nodes.

5.2 Versatile Training

Two type of ANNs that are not currently supported by GELNN are Convolutional Neural Networks and Recurrent Neural Networks due to the fact that changes to the backpropagation algorithm would need to be made to support each of these types of ANNs.

These networks have both proven to be successful at different types of problems. If a versatile training algorithm could be designed that was capable of training both Convolutional Neural Networks and Recurrent Neural Networks based on auxiliary layer attributes in a GELNN encoding, it would expand greatly expand the problems that a GELNN system would be able to address.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 1129139 and 1331283. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

6. REFERENCES

[1] M. F. et al. An extendible package for data exploration, classification and correlation. *Institute of Pharmaceutical and Food Analysis and Technologies*.

[2] W. B. et al. *Genetic programming: an introduction.*, volume 270. 1998.

[3] B. Farley and W. Clark. Simulation of self-organizing systems by digital computer. *Transactions of the IRE Professional Group on Information Theory Trans. IRE Prof. Group Inf. Theory*, 4(4):76–84, 1954.

[4] G. E. Hinton. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–07, 2006.

[5] G. E. Hinton. Learning multiple layers of representation. *Trends in Cognitive Sciences*, 11(10):428–34, 2007.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*.

[7] M. Lichman. Uci machine learning repository. *University of California, School of Information and Computer Science*, 2013.

[8] S. Luke and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. *Late-Breaking Papers at the Genetic Programming 1996 Conference*, pages 428–34, 2007.

[9] M. Minsky and S. Papert. Perceptrons; an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19(88), 1969.

[10] F. Pagnotta and H. M. Amran. Using data mining to predict secondary school student alcohol consumption. *Department of Computer Science,University of Camerino*.

[11] R. Poli, W. B. Langdon, and N. F. McPhee. *A ĭňĄeld guide to genetic programming*, volume 270. 2008.

[12] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. *Readings in Cognitive Science*, pages 399–421, 1988.

[13] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3, 2002.

[14] T. H. L. Spector and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Xplore*, 313(5786):504–07, Oct 2006.

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