A Comparison of Architectural Constraints for Feedforward Neural Diversity Machines

Obrien Sim, Tomas Maul, and Chris Roadknight
The University of Nottingham Malaysia Campus
Jalan Broga, 43500 Selangor, Malaysia
obrien_sim@live.co.uk, tomas.maul@nottingham.edu.my, chris.roadknight@nottingham.edu.my

ABSTRACT

This paper is a follow-up study on an earlier introductory paper on Neural Diversity Machines (NDM). NDMs are a subclass of Hybrid Artificial Neural Networks (HANNs), which are digital representations of biological neural networks present in the human brain. As opposed to traditional artificial neural networks (ANNs) which tend to be focused around uniform neurons, HANNs and NDMs tend to adopt heterogeneous types of neurons, partly with the aim of exploring the potential benefits of neural diversity in ANNs. This paper demonstrates and analyzes the performance of three architectural variants of a subclass of NDM (coined Mini-NDMs) in solving classification problems when tested on real life data sets.

KEYWORDS

Hybrid artificial neural networks; neural diversity machines; machine learning; pattern classification; genetic algorithms;

1. INTRODUCTION

One of the major limitations of digital computers lies in their limited capability to learn, resulting in large static knowledge databases which need to be constantly updated to ensure optimal performance. Artificial neural networks (ANNs), digital equivalents of biological neural networks present in the human brain, seek to help rectify this problem by mimicking the dynamic and parallel information processing methods that humans use unconsciously. Although traditional neural networks such as the canonical multi-layer perceptron have been known to perform at a satisfactory level, their ability to converge fast and generalize well when scaled-up to more complex datasets can be somewhat unreliable. This has prompted researchers within the field to actively explore other alternatives, such as hybrid artificial neural networks (HANNs).

Diversity is what tends to differentiate a traditional ANN from a HANN. A recent review on ANNs classified diversity into three categories: model, algorithms, and data [1]. From the model perspective, diversity is measured by the number of activation and weight functions used within the network, also known as functional diversity, as used in [2] and [3]. From the algorithms' perspective, diversity is measured by the different kinds of learning paradigms which are utilized throughout the learning process of the network. A simple example would be how a deep belief network which uses a semi-supervised learning paradigm is more diverse than an ANN which utilizes either supervised or unsupervised learning approaches [4] and [5]. This type of diversity is coined by the term learning diversity. The third interpretation of diversity is dependent on the data that the neural networks are trained and tested on. It is measured by the different approaches with which the dataset is handled such as weighting and its pre-processing, an approach commonly used in [5] and [6].

The diversity that the experiments conducted with NDMs seek to explore consists of functional diversity. One of the biggest challenges in the field of neural networks is the decision of an optimal ANN architecture. There are countless possibilities with which a neural network can be tweaked and to go through the entire search space would be inefficient and time consuming to carry out. The goal of using NDMs is to identify what types of architectures with respect to functional diversity are most effective in the domain of classification. This discovery could potentially eliminate portions of the search space and allow for more efficient utilization of computational time and resources.

2. MINI NEURAL DIVERSITY MACHINES

The NDMs are a HANN model proposed by Maul [7] which adopts a multilayer topology but is not restricted in connectivity as well as functional diversity. The basic model of an NDM is a HANN with the following characteristics:

1. At least 3 activation functions (referred to as transfer functions in [7]) and 3 input functions (referred to as weight functions in [7]) are available to choose from for each node. This leads to a pool of at least nine possible node functions. The term node function used throughout this paper refers to a combination of an input and activation function.
2. Nodes may exhibit any combination of activation and output functions.
3. Networks can simultaneously use any number of the available node types.
4. Unrestrained connectivity between nodes.

The Mini-NDMs reported in this paper are slightly different from the original NDMs that they are based on. A Mini-NDM is an NDM with a looser requirement for the number of transfer functions available to the network as well as a topological restriction in terms of feedforward connectivity. Mini-NDMs possess the following characteristics:

1. Six node functions to choose from. This is less than the NDM requirement which stipulates a minimal pool of nine node functions to choose from. The six possible nodes functions to choose from are as follows:
    a. Dot-Product/Sigmoid
    b. Dot-Product/Gaussian
    c. Dot-Product/Identity
    d. Euclidean/Sigmoid
    e. Euclidean/Gaussian
    f. Euclidean/Identity
2. Neuroevolution as its learning or optimization approach.
3. Fully connected feed-forward connectivity.
4. A multilayer topology. A three layer topology (three layers of nodes and therefore two layers of connections) was implemented for the neural networks trained and tested throughout this paper.

Three types of Mini-NDMs were experimented with, each utilizing the same global stochastic optimization algorithm although a limitation in functional diversity was placed on each type.

Table 1. Different kinds of Mini-NDMs.

Architecture	Constraint
Free Mini-NDM	None
Uniform Mini-NDM	Nodes in the same layer must have the same node function
Simple Mini- NDM	Only two node functions are allowed to be used in the whole neural network

The main aim of the experiments was to test the impact of the above architectural constraints on the convergence and generalization properties of mini NDMs viv-a-vis different real-world data-sets, particularly when associated with different numbers of hidden nodes.

3. OPTIMIZATION APPROACH

The neuroevolutionary approach adopted in this work consists of a global stochastic optimization algorithm loosely incorporating elements of both differential evolution and genetic algorithms (through the utilization of operators such as mutation and crossover). The optimization process involves the tuning of not only connection weights but also the activation and input functions adopted by the networks themselves.

The following is the pseudocode of the global stochastic optimization algorithm utilized.

```
Procedure
globalStochasticOptimization(dataset,
parameters)
  dontStop := true
  sols :=
initializeSolutionsRandomly(dataset,
parameters)
  sols := evalSol(sols, dataset)
  sols := sortSols(sols, dataset)
  sols := diffEvolution(sols,
parameters)
  while dontStop do
   sols := trimSols(sols, parameters)
   sols := geneticEvolution(sols,
parameters)
   sols := sortSols(sol, costs)
   costs := evalSols(sols, dataset)
```

```
   sols := diffEvolution(sols,
parameters)
   costs := evalSols(sols, dataset)
   sol := sortSols (sol, costs)
   dontStop := checkStop(…)
  end while
end procedure.
```

The Global Stochastic Optimization (GSO) algorithm begins by generating genotypes randomly. The size of the genotype must be known beforehand and is calculated based on the dataset that the Mini-NDM aims to learn along with its optimization parameters. Once the genotypes have been created, they are evaluated and sorted before the differential evolution algorithm is applied onto the set of solutions.

The genotypes were programmed in such a manner that each neural network could be represented by a uniform size vector. A specific number of hidden layers and hidden nodes were specified manually for each experiment as these traits are not encoded in the genotype; this was to avoid several optimization issues which would be experienced if the genotypes were of a non-uniform size. Based on this basic information the number of connection weights can be calculated and the appropriate size of the solution vector can be determined. The solution vectors represent only the activation and input functions of the hidden and output nodes along with each individual connection weight present in the network.

The following is the pseudocode of the differential evolution algorithm utilized.

```
Procedure
differentialEvolution(sols,parameters)
  velocities := []
  numSols := size(sols)
  leaderIndex := 1
  while leaderIndex < numSols do
    leader := sols[leaderIndex]
    followerIndex :=
firstDiffSol(leader,sols,parameters)
    if followerIndex != null then
        velocity :=
parameters.alphaRate *
(sols[leaderIndex] -
sols[followerIndex])
      newSol := sol[leaderIndex] +
velocity
      newSol := normalize (newSol,
parameters)
      sols := addSol(sols, newSol)
      velocities := store(velocities,
velocity)
      leaderIndex := nextLeader(leader,
parameters)
    end if
end while
grandVelocity := parameters.gvAmplify *
mean(velocities)
  firstLeader := sols[1]
  newSol := firstLeader + grandVelocity
  newSol := normalize(newSol,
parameters)
  sols := addSol(sols,newSol)
end procedure.
```

Differential Evolution (DE) was originally proposed by Price and Storn as an evolutionary algorithm which generates new solutions based on the difference between existing solutions [8]. Generally speaking, the differential evolution approach generates new solutions based on the difference between two randomly chosen individuals in the population. This results in a search for an area instead of a single point in the state space. This improves the search's capability and reduces the tendency of searching for solutions only in areas of local optimality.

The variant of the original DE algorithm implemented throughout this project modifies some aspects of the original DE algorithm. Instead of generating solutions for each individual, this is instead done for each *"Leader"*. For each leader, a sufficiently different solution (termed the *"Follower"*) is then searched for in the population and once found, a new solution is generated from these two solutions. The new solution is the leader solution added with the difference between the leader and the follower (termed the *velocity vector*) multiplied by the *alpha rate*, a multiplier that is set in the optimization parameters. Once a new solution has been generated, a new leader then will be found and the process is repeated until all leaders have been found in the initial population. A final solution is then created from the first leader and the mean of all the velocity vectors (termed the *grand or global velocity*) multiplied by the *gvAmplify* parameter, a multiplier whose function is similar to that of the alpha rate but only utilised during the generation of the final solution.

After the first differential evolution, the GSO algorithm iterates through a cycle of trimming, (deleting solutions to a certain portion of the solution set, leaving only solutions which are sufficiently distinct from each other) genetic evolution and differential evolution until the termination criteria is met. The termination criterion in this implementation is set to true when one of the following conditions is fulfilled:

1. The GSO has iterated for the maximum number of iterations
2. A solution that has a cost less than or equal to the minimum cost specified in the parameters is found

The following is the pseudocode of the genetic algorithm being utilized.

```
procedure
geneticEvolution(sols,parameters)
numSols := size(sols)
numSolsToGenerate :=
parameters.generationSize – numSols
  n := 1
  while n <= numsolsToGenerate
     method := ceil(random(3))
      if method == 1 then
        newSol :=
mutate(sols,parameters)
      else if method == 2 then
        newSol :=
crossover(sols,parameters)
      else
        newSol  :=
createRandomSol(parameters)
      end if
        sols := addSol(sols,newSol)
     n++
  end while
end procedure.
```

In sum, the genetic evolution operator randomly picks an operator between *mutation*, *crossover*, and *random generation*.

Mutation

The mutation operator modifies the values in the genotype in a random-like manner to increase the diversity of the search space for evolutionary algorithms which tend to become biased in optimizing solutions which are already proven to be good (locally optimal solutions). The mutation operator is dependent on two parameters which determines how often and how much to mutate a genotype: *mutation probability* and *mutation range*.

Each gene in the genotype has a probability of mutating within the mutation range. Once a genotype is determined to be mutated, a random number generated within the boundaries of the mutation range will be added to the gene. The gene is then normalized in the case that the new values are out of the boundaries of the genotype's numerical range.

Crossover

Crossover is the creation of a new genotype from two existing genotypes Crossover represents the reproduction of living things where offspring inherit characteristics from both parents. Crossover is implemented in such a way that a portion of each parent's genome is combined to form the offspring's genome. However, the magnitude of each parent's contribution to the resulting genome is not set in stone. A random number generator is used to select the *cutoff point* of the crossover operator each time it is called upon.

Random Generation

The random generation of new solutions is carried out in the same way as solutions are initialized at the start of the global stochastic optimization process.

4. RESULTS & DISCUSSION

The experiments were run on 3 different datasets; all retrieved from the UCI Machine Learning Repository which are genuine real-life data. Each experiment on each dataset was run 30 times and the results averaged to reduce random bias. A 60:40 ratio was used in dividing the dataset into training and testing sets respectively. A summary of the datasets that were tested on can be found in Table 2

Table 2. Summary of the utilized datasets

Name	Attributes	Classes	Instances	Balanced
Teaching	5	3	151	Yes
Hayes-Roth	4	3	160	Yes
Iris	4	3	150	Yes

The main objective of this research was to reveal the architecture with the best generalization properties for each data set, and through this to assess whether any one of the architectures is consistently the best generalizer. The experimental results are summarized in Table 3 as follows:

Table 3. Classification rate results

Name	Hidden Nodes	Free Architecture	Simple Architecture	Uniform Architecture
Teaching	1	34.61%	34.50%	35.00%
	2	32.61%	33.17%	33.56%
	3	33.45%	34.67%	33.45%
	4	33.78%	35.11%	34.78%
	5	35.39%	34.72%	34.45%
	6	35.78%	35.22%	35.89%
	7	36.67%	34.61%	35.78%
Hayes-Roth	1	59.40%	56.55%	60.48%
	2	57.14%	52.74%	55.12%
	3	52.74%	54.4%	57.86%
	4	49.05%	53.1%	59.17%
	5	50.71%	50.72%	60.00%
	6	49.05%	52.02%	60.71%
	7	48.33%	49.64%	61.43%
Iris	1	91.56%	82.28%	90.83%
	2	88.28%	78.72%	81.61%
	3	79.28%	72.61%	81.00%
	4	68.44%	63.22%	79.56%
	5	55.33%	55.17%	81.89%
	6	45.72%	51.22%	82.78%

The results show that out of the three datasets on which the experiments were carried out, two were in favor of the free architecture with one in favor of the uniform architecture. These results alone are still inconclusive but they do suggest that although in general the free architecture seems to perform better than the uniform architecture which represents traditional neural networks, in certain cases traditional neural networks can still outperform hybrid models. Generally, the simple architecture tends to be outperformed by the other two. One might have expected there to be an optimal level of functional diversity lying somewhere between completely uniform hidden layers and completely free hidden layers, however, as the experimental results show, what lies in between the two extremes actually performs the worst.

Possible explanations for this behavior might be due to the incompatibility of the node functions provided with each other when only two types were allowed within the hidden layer. Further experiments need to be carried out with a larger node function pool as well as a larger minimum number of node function types per layer. When only using 1 hidden node, as one would expect, the simple architecture performs similarly to the other two architectures but as more hidden nodes are added, performance degrades at a faster rate compared to the other two. More research is needed to clarify this phenomenon.

Another hypothesis that was tested in this work, states that the optimal number of hidden nodes within an artificial neural network should be $\log_2$(numTrainingPatterns). This was inspired by [10] who confirmed that this theory worked in his study of traditional multilayer perceptrons. Throughout this study however, the opposite was actually observed, with the performance of neural networks usually depreciating as more hidden nodes were added. This was most likely due to overfitting which happened at a faster rate due to the existence of multiple transfer functions. This phenomenon can be clearly seen for example in the Iris data-set as depicted in Figure 1.

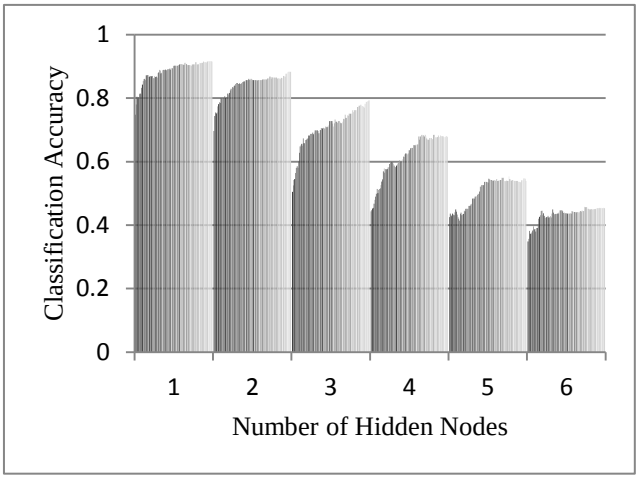

Fig. 1. Results for the classification rate of the Iris dataset using a Free Mini-NDM

5. FUTURE WORK

Throughout this paper, it has been demonstrated that Mini-NDMs are indeed capable of learning in the domain of pattern recognition. This does not however, imply that no further improvements can be made. Based on the results of the experiments, it can be seen that some future work could be appended. First of all is the pool of the node functions which the mini-NDMs are able to select. This could be expanded by adding

more input functions such as: higher-order products, standard deviation, weighted minimum, and weighted maximum. This would result in a total number of 18 node functions, increasing the functional diversity even further. That aside, other kinds of multilayer architectures could also be tested instead of a basic three-layered topology with unrestrained connectivity as found in the current implementation. Other topological models which could be tested include "fan in" and "fan out" models with more than one hidden layer.

There could also be improvements made to the global stochastic optimization algorithm which is heavily reliant on its parameter set. Unsuitable parameters can lead to problems such as entrapment within the local minima. This could be improved by encoding the GSO algorithm parameters into the genotypes as well, subjecting it to the optimization process as well, possibly ending up with more optimal parameters for the optimization itself. G-Prop is an example of how this could be implemented [10]. An improvement to the differential evolution algorithm is also proposed by Das et al. [11] by changing the alpha rate into a randomly generated number instead of a constant. The proposed improvement is expected to result in a higher number of stochastic variations in the velocity vectors which would help retain diversity and combat selective pressure. To reduce the training bias even further, a cross validation approach could also be undertaken.

6. REFERENCES

1. P. Gutierrez, C. Hervas-Martinez.: Hybrid Artificial Neural Networks: Models, Algorithms, and Data. In: Advances in Computational Intelligence 6692, pp. 177-184. (2011)

2. S. Cohen, N. Intrator.: A hybrid projection-based and radial basis function architecture: Initials values and global optimization. In: Pattern Analysis & Applications 5, pp. 113-120. (2002)

3. N. Jankowski, W. Duch.: Optimal transfer function neural networks. In: 9th European Symposium on Artificial Neural Network, pp. 101-106. (2001)

4. G.E. Hinton, S. Osindero, Y. The.: A fast learning algorithm for deep belief nets. In: Neural Computation 18, pp. 1527-1554. (2006)

5. Y. Bengio.: Learning deep architectures for AI. In: Foundations and Trends in Machine Learning 2, pp. 1-127. (2002)

6. K. McGarry, S. Wemter, J. MacIntyre .: Hybrid neural systems: from simple coupling to fully integrated neural networks. In: Neural Computing Surveys 2, pp. 62-93. (1999)

7. T. H. Maul.: Early Experiments with Neural Diversity Machines. In: Neurocomputing. (2012)

8. R. Storn, K. Price.: Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. In: Journal of Global Optimization 11, pp. 341-359. (1997)

9. G. Mirchandani, W. Cao.: On Hidden Nodes for Neural Nets. In: IEEE Transactions on Circuits and Systems 36, pp. 309-312. (1989)

10. P. A. Castillo, V. Rivas, J.J. Merelo, J. Gonzalez, A. Prieto, G. Romero.: G-Prop-III: Global optimization of multilayer perceptron using an evolutionary algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference 1, pp. 13-17. (1999)

11. S. Das, A. Konar, U.K. Chakraborty.: Two Improved Differential Evolution Schemes for Faster Global Search. In: GECCO'05. (2005).