

An Evolutionary Algorithm for Inversion of ANNs

Henrik Jacobsson and Björn Olsson
Dept. of Computer Science, University of Skövde
Box 408, 541 28, Skövde, Sweden
henrikj@ida.his.se, bjorne@ida.his.se

-Abstract

Before using a trained artificial neural network (ANN) in an application it is important to identify inputs which cause incorrect behaviours. We therefore propose the use of an evolutionary algorithm (EA) to invert the mappings of ANNs. The EA is used to search for input patterns which produce strong (distinct) classifications into one of the classes. Since the input space is typically very large, multimodal, and poorly understood, EAs are likely to be more robust than gradient methods, with a lower probability of getting stuck on local optima. Analysis of our results supports this hypothesis. Our evolutionary algorithm also involves the use of *niching*, which allows it to simultaneously explore multiple regions of the search space. The resulting population of input patterns therefore typically represents a set of distinctly different instances. This property is important, since the aim is to identify inputs which are erroneously classified. We show how analysis of the set of inputs found by inversion can lead to detection of flaws in the ANN, and we discuss the possibilities of using this inversion method as a tool for validation and re-training.

1 Introduction

Artificial Neural Networks (ANNs) have become general tools for solving problems which are difficult to formalize. One of the main advantages of ANNs is their capacity for generalizing to unseen data. This is taken advantage of during training, where typically a small subset of the domain is used as a training set. Without such reduction in the amount of data, training would become intractable, since the domain is usually very large or infinite. However, the reduction of the training set also leads to some problems. Given that the training set is a small sample from the domain, how do we know that it contains all the information necessary for successful training? To some extent, this problem is alleviated by using a test set to verify that the network generalizes to unseen data. However, even the test set is (in most cases) just a small sample from the whole domain, and is often sampled using the same method as for the training set. If this method contains flaws, erroneous ANN behaviour could go undetected. A validation method which does not depend on sampling of the domain is therefore desirable.

ANNs can indeed be perceived as “unreliable” since they are not explicitly programmed and often have complex architectures of connected nodes

where the function is distributed in the whole system. The behaviour of the network emerges from many simple components and this makes it hard to understand. This also has implications for the application of ANNs: an engineer who needs a computing component in an application wants to know as much as possible about its function and degree of reliability. As expressed in [3, p. 41]:

Don’t be too quick to assume that a network, even an apparently successful one, has actually fixed on the features on which you wanted it to fix. An up-and-running network is an opaque beast which requires further analysis if we are to understand what it is actually doing and why.

Rather than a detailed analysis of network behaviour, we are seeking in this paper a method for rapid and reliable identification of critical flaws in a fully trained ANN. Given a network which has undergone training and testing, we want a procedure for identification of situations in the domain which would be misclassified by the network. Such a procedure is a candidate method for verification (or quality-assurance) of ANNs. Analysis of the result of the inversion could also potentially give insight into ANN behaviour, which could serve as a guide for improvement, e.g. of the training set, the architecture or the training algorithm.

The problem of locating samples leading to critical misclassifications is here seen as a search problem, i.e. given a target output vector t , we search in the space of possible inputs for those which produce t . Due to the size and possible multimodality of this search space, we use an EA to avoid getting stuck on local optima. Further, since we want to identify the cases where the ANN produces t erroneously, it is important that the search algorithm returns a set I of t -producing inputs which covers as much as possible of the complete set \mathcal{I} of all t -producing inputs (given the obvious constraint of the finite population size). Therefore, we extend the EA with niching, which promotes diversity among the solutions and increases the probability of locating a number of sufficiently good solutions, rather than concentrating the entire population on the same optimum.

2 Previous Work and Related Approaches

There are several methods available for analysis of ANNs. The analysis is often very problem-dependent, and therefore it is hard to find a general method which can be applied to any kind of network. A common approach for analysing the internal behaviour is to analyse the activation of the

hidden nodes when the network is triggered by input examples (for an overview of such methods see [2]). This activation can then be visualized by Hierarchical Cluster Analysis (HCA) or other techniques for visualizing multidimensional spaces. An HCA was, for example, used to analyze the famous NET-talk [8] which learned to pronounce English words. Another approach is to extract symbolic rules from the network and analyse these rules [1] [9]. A third method is to invert the mapping of the ANN. In contrast to the other methods, which are based on analysing the internal structure of the network, inversion-based approaches treat the network as a “black box”, since no consideration is given to the internal behaviour of the network - the only aim is to search for input patterns which give rise to a given output. Inversion methods are therefore more focused on validating the correctness of the network, rather than providing detailed knowledge about its internal structure.

An inversion algorithm for neural networks was proposed already in [10]. The proposed method was based on the backpropagation algorithm [7], but instead of adjusting the network weights, the algorithm was used for adjusting the input vector to minimize the deviation of the actual output o from the target output t (while the network weights were fixed). This form of gradient-based inversion was developed further by Kindermann [6] by introducing other error measures than the standard sum-squared error.

There are some drawbacks of gradient-based inversion. An inherent limitation of this method is that it only returns a single input pattern. This makes the method unreliable for detecting faulty patterns, since the returned pattern might be a true member of the target class. Such a result does not detect any faults in the network.

Another problem is that the search spaces (for multi-layered ANNs) are non-linear and probably contain many local optima where gradient methods are likely to get stuck. If this happens, the method will not return any t -producing input at all. Gradient methods are also limited if we try to acquire multiple answers by running the algorithm multiple times with different starting points. There is no guarantee of diversity of the resulting patterns since most of the runs may find the same local optimum.

3 An Evolutionary Algorithm for Inversion

Our proposed method for inversion of ANN mappings is to use an EA to conduct the search, since EAs are believed to be good at dealing with local optima and nonlinear, multimodal, search spaces [4], and since EAs return a population of candidate solutions, rather than a single one. Several considerations have to be addressed in the choice of EA, and in its implementation. These issues will be discussed in the following description of our method.

3.1 Evolutionary Algorithm

Given n input nodes, we have a search space of input vectors \mathfrak{R}^n . We use the EA to find a subset

$I \subset \mathfrak{R}^n$ where each element $i \in I$ generates an output o similar to the target output t . The EA should be designed so that it meets two criteria simultaneously:

- to maximize the similarity of all o to t
- to ensure a high level of diversity in I .

An EA meeting both these criteria can be used to search for a diverse set of inputs which are all classified by the ANN as belonging to the target class. Analysis of these inputs can then potentially expose crucial errors in the ANN’s behaviour (or indicate that it is correct).

Fitness values in EAs should reflect the quality of the candidate solutions, and provide the selection mechanism with useful fitness differences. In our case, fitness is based on the deviation of the actual output from the desired output, and assigns higher fitness to candidates with smaller deviations. To determine the fitness $f(x_i)$ of individual x_i , we present the ANN with the input vector which is encoded on the chromosome of x_i , and calculate the fitness according to

$$f(x_i) = \frac{1}{\sum_j (o_j - t_j)^2 + \epsilon} \quad (1)$$

where o is the output vector, t is the desired target vector, and j is an index over the output units. This fitness function promotes the reproduction of individuals encoding inputs which make the ANN produce patterns resembling the target output.

Sharing was proposed in [5] as a method of promoting niching and diversify the population to find more than just small variations of the same answer. The method is based on penalizing individuals which are too similar to other individuals by decreasing their fitness. Given that individual x_i has fitness $f(x_i)$ according to Eq. 1, we calculate its shared fitness $f_s(x_i)$ according to

$$f_s(x_i) = \frac{f(x_i)}{\sum_j s(d(x_i, x_j))} \quad (2)$$

where j is an index over a random sample of the individuals in the population. The distance function $d(x_i, x_j)$ returns the Euclidean distance between the genotypes of x_i and x_j , and the sharing function $s(d(x_i, x_j))$ determines how much x_i should be penalized given its distance to x_j . The sharing function $s(d)$ is defined as

$$s(d) = \frac{d_{max} - d}{d_{max}} \quad (3)$$

where d_{max} is the maximum Euclidean distance between a chromosome pair, given the current problem and chromosome representation. This gives a simple triangular sharing function [5], resulting in $0.0 \leq s(d) \leq 1.0$. Selection of parents is based on the shared fitness values, using standard fitness-proportional selection.

3.2 Experiments

In [6] two example problems were used to test their proposed method. The second, and more interesting, problem is recognition of hand-written digits from 0 to 9, coded as bitmaps of size 11×8 . We will use the same problem and data set for evaluation of our method. The training set includes 49 examples of each digit. An additional “garbage class” of 1,000 random bitmaps is used, since it was found in [6] that this improves the results of the inversion algorithm. A network with a single hidden layer of 20 nodes is trained to classify the digits into ten classes.

Our first experiment uses the digit recognition problem for evaluation of solution coverage. We evolve a population of 1,000 candidate input patterns using the digit ‘3’ as the target class (as in [6]). Since the input to the network is a 11×8 bitmap, the chromosomes are matrices of the same dimensions.

The second experiment uses the same task to evaluate the EA’s ability to avoid becoming stuck on local optima. We reduce the population size to 100, since a smaller population should run a higher risk of becoming stuck. We make 100 runs for each of the 10 different digits and let the EA run until the population contains at least one pattern giving higher fitness than all input vectors in the training set. By recording the maximum number of generations used for each digit, we can detect any cases where the progress of the search is unusually slow.

4 Results and Analysis

We here compare two versions of our method - one using the “raw” fitness according to equation 1, and the second using the “shared” fitness according to equation 2. These will be referred to as EA-R (EA using “raw” fitness), and EA-S (EA using “shared” fitness). We hypothesize that the proposed advantages (better solution coverage and higher robustness against local optima) of EA-based inversion over gradient methods is dependent on the use of a mechanism for promoting diversity. To test this hypothesis we make sure that the only difference between EA-S and EA-R is the choice of fitness function, while all other properties (such as selection mechanism, genetic operators, etc) are identical for EA-S and EA-R.

4.1 Solution Coverage

Fig. 1 shows how the diversity of the population develops over the runs, when we used the EA to search for inputs generating the output corresponding to the digit ‘3’ (the ANN had been successfully trained on all digits). The maximum and average Euclidean distances between chromosome pairs drop very quickly in the EA-R runs, whereas the EA-S runs show increasing levels of population diversity through the whole runs. Fig. 1 shows that the similarity of the input patterns to the target pattern is almost the same for the two versions. Since population diversity is much higher with sharing, and the high average fitness is preserved, we hypothesize that the populations evolved by EA-S contain many different (and correct) input patterns, whereas the populations evolved by EA-R only contain very few such patterns.

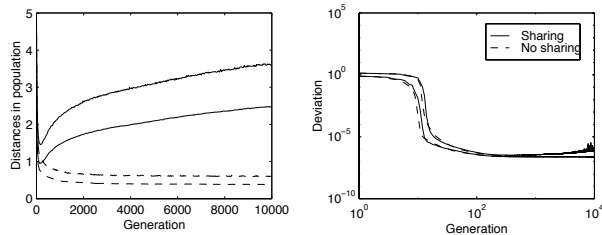


Figure 1: Average results from 10 runs on the digit recognition problem. Population statistics for standard EA and EA with sharing. The population’s average and maximum pairwise chromosome distance $d(x_i, x_j)$ (left). The population’s average and maximum deviation from the target output (i.e. the inversion of the fitness function in equation 1) (right).

Nr	0	1	2	3	4	5	6	7	8	9
Min	6	44	4	15	67	20	4	23	1	3
Avg	30	126	10	38	199	68	20	65	2	7
Max	88	392	40	177	374	181	117	152	4	36

Table 1: Number of generations needed to find an input pattern of “satisfactory” quality (i.e. with higher fitness than every pattern in the training set).

4.2 Local Optima Avoidance

One proposed advantage of EA-based inversion is that EAs are believed to be robust in the presence of local optima. Table 1 shows an evaluation of our method’s ability to avoid getting stuck on local optima. The table shows the number of generations which were needed by EA-S until the population contained some pattern with higher fitness than every pattern in the training set. Such a pattern is defined as a “satisfactory” solution, since it produces an output more similar to the target output than what is produced by any of the patterns in the training set.

The highest number of generations was 392 in a run with the digit ‘1’ as target class. This corresponds to 39,200 evaluated input patterns. Since this is not excessively high, we interpret it as showing that no run got stuck on any (low-quality) local optimum. Most of the runs found satisfactory solutions much earlier, and the average number of generations is only approximately 57.

It is notable that the average number of generations varies considerably for the different digits, and this may also provide some insight into the trained network’s behaviour. It takes on average only approximately two generations to find an input pattern which is classified strongly as a digit ‘8’ by the network. This indicates that the network actually treats ‘8’ as a default class.

4.3 Analysis of Evolved Patterns

While scanning through some of the input patterns found by the algorithm we noted the repeated occurrence of incorrect inputs. Many of the patterns found for the digit ‘3’ network had a single black pixel along the left edge of the pattern, as in the example in fig. 2. This is an incorrect feature of these inputs, since a digit ‘3’ can never contain a

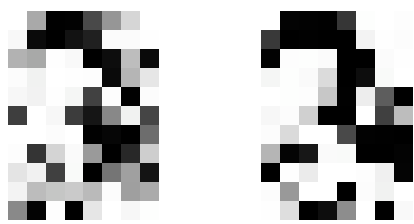


Figure 2: Left pattern is an example “incorrect” digit three found by EA-S before retraining. Right pattern is a typical pattern found by EA-S after retraining.

black pixel at this location. Since these incorrect patterns were assigned high fitness values, the inversion exposes a flaw in the network’s behaviour, i.e. that it uses the faulty pixel as an indication that the input should be classified as a ’3’. The exact same behaviour was observed in [6], and it may indicate a problem with the training set. We used the exact same training set as in [6], which contains a “garbage class” of 1,000 random patterns (as described in section 3.2). However, the same garbage set is used throughout the training, which means that it is not truly random. In an attempt to improve the training, we instead trained a new network by, in each epoch, generating a new set of random garbage patterns. The solutions found by inversion of the new network did not have the incorrect pixel. Fig. 2 shows an example of a pattern found by inverting the improved network. We view this result as an example that inversion is a potential tool for finding erroneous behaviour in ANNs.

5 Conclusions

We proposed an EA-based method for inversion of neural network mappings. Our aim was to show that using EA-based search in this domain improves on the results of gradient-based inversion methods. Our method avoids getting stuck on local optima, and finds a diverse set of multiple solutions, which increases the usefulness of the inversion. We have also provided an example of faulty network behaviour being detected by our inversion method.

To make the method more generally useful as a tool for network improvement, it is desirable to find ways of formalizing the analysis of inversion results. The question is whether there is a method by which the inversion results can be traced back to the training procedure, training set, or network architecture, in such a way that the cause of the faulty behaviour is detected. This might be necessary to make it possible to find the actions which should be taken to improve the network after faulty behaviour has been detected. How this can be formalized, or even automated, may be the key question to investigate in future work.

One route forward may be to use the inverted patterns as counter-examples during retraining of the faulty network. Our inversion method finds a diverse set of inputs which are distinctly classified as class members by the network - despite not being true members of the class - which shows that the method finds the most crucial counter-examples.

Adding these to the training set should improve the network’s ability to discriminate true negatives.

6 Acknowledgments

This research was funded by a grant to the authors from *The Foundation for Knowledge and Competence Development (1507/97)*, Sweden.

The authors also wish to thank Fredrik Linåker for providing helpful comments and suggestions during this project.

References

- [1] R. Andrews, J. Diedrich, and A.B. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge Based Systems*, 8(6):373–389, 1995.
- [2] J.A. Bullinaria. Analyzing the internal representations of trained artificial neural networks. In A. Browne, editor, *Neural Network Analysis, Architectures and Applications*, pages 3–26. TOP Publishing, 1997.
- [3] A. Clark. *Associative engines, connectionism, and representational change*. MIT Press, 1993.
- [4] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [5] D.E. Goldberg and John Richardsson. Genetic algorithms with sharing for multimodal function optimization. In J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Morgan Kaufmann Publishers, 1987.
- [6] J. Kindermann and A. Linden. Inversion of neural networks by gradient descent. *Parallel Computing*, 14:270–86, 1990.
- [7] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323:533–536, 1986.
- [8] T. Sejnowski and C. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168, 1987.
- [9] A.B. Tickle, R. Andrews, M. Golea, and J. Diedrich. *Rule Extraction from Artificial Neural Networks*. TOP Publishing, 1997.
- [10] R.J. Williams. Inverting a connectionist network mapping by backpropagation of error. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society*, 1986.