Interval methods for seeking fixed points of recurrent neural networks

Bartłomiej Jacek Kubica, Paweł Hoser, and Artur Wiliński

Institute of Information Technology, Warsaw University of Life Sciences – SGGW,
ul. Nowoursynowska 159, 02-776 Warsaw, Poland
{bartlomiej_kubica,pawel_hoser,artur_wilinski}@sggw.pl

Abstract. The paper describes an application of interval methods to train recurrent neural networks and investigate their behavior. The HIBA_USNE multithreaded interval solver for nonlinear systems and algorithmic differentiation using ADHC are used. Using interval methods, we can not only train the network, but precisely localize all stationary points of the network. Preliminary numerical results for continuous Hopfield-like networks are presented.

Keywords: interval computations, nonlinear systems, HIBA_USNE, recurrent neural network, Hopfield network

1 Introduction

Artificial neural networks (ANN) have been used in many branches of science and technology, for the purposes of classification, modeling, approximation, etc. Several training algorithms have been proposed for this tool. In particular, several authors have applied interval algorithms for this purpose (cf., e.g., [6], [19], [5]). Most of these efforts (all known to the authors) have been devoted to feedforward neural networks.

Nevertheless, in some applications (like prediction of a time series or other issues related to dynamical systems, but also, e.g., in some implementations of the associative memory), we need the neural network to remember its previous states – and this can be achieved by using the feedback connections. In this paper, we apply interval methods to train this sort of networks.

2 Hopfield-like network

Let us focus on a simple model, similar to popular Hopfield networks, described, i.a., in [18]. There is only a single layer and each neuron is connected to all other ones. Fig. 1 illustrates this architecture.

Let us present the mathematical formulae. Following [10], we denote vectors by small letters and matrices – by capital ones.

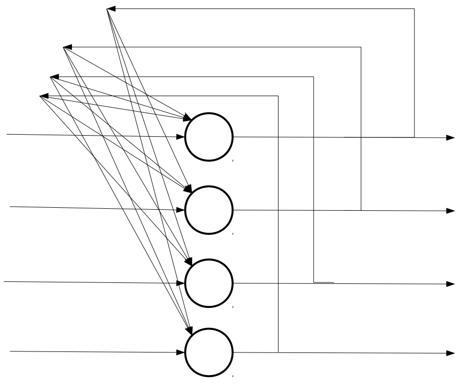

Fig. 1. A Hopfield-type neural network

The output of the network is the vector of responses of each neuron, which, for the i'th one, is:

$$y_i = \sigma\Big(\sum_{j=1}^{n} w_{ij} x_j\Big), \quad \text{for } i = 1, \ldots, n \ , \tag{1}$$

where $\sigma(\cdot)$ is the activation function, described below.

The weights can have both positive and negative values, i.e., neurons can both attract or repel each other. Also, typically, it is assumed that $w_{ii} = 0$, i.e., neurons do not influence themselves directly, but only by means of influencing other neurons. Unlike most papers on Hopfield networks, we assume that the states of neurons are not discrete, but continuous: $x_i \in [-1, 1]$. As the activation function, the step function has been used originally:

$$\sigma(t) = \mathcal{H}(t) = \begin{cases} 1 \text{ for } t \geq 0, \\ -1 \text{ for } t < 0. \end{cases} \ , \tag{2}$$

but sigmoid functions can be used, as well; for instance:

$$\sigma(t) = \frac{2}{1 + \exp(-\beta \cdot t)} - 1 \ , \tag{3}$$

the hyperbolic tangent or the arctan. Please mind that in both above functions, (2) and (3), the value of the activation function ranges from -1 to 1 and not from 0 to 1, like we would have for some other types of ANNs.

In our experiments, we stick to using activation functions of type (3) with $\beta = 1$, but other values $\beta > 0$ would make sense, as well.

What is the purpose of such a network? It is an associative memory that can store some patterns. These patterns are fixed points of this network: when we feed the network with a vector, being one of the patters, the network results in the same vector on the output.

What if we give another input, not being one of the remembered vectors? Then, the network will find the closest one of the patterns and return it. This may take a few iterations, before the output stabilizes.

Networks presented in Fig. 1, very popular in previous decades, has become less commonly used in practice, nowadays. Nevertheless, it is still an interesting object for investigations and results obtained for Hopfield-like networks should be easily extended to other ANNs.

3 Training Hopfield-like networks

How to train a Hopfield network? There are various approaches and heuristics.

Usually, we assume that the network is supposed to remember a given number of patterns (vectors) that should become its stationary points.

An example is the Hebb rule, used when patterns are vectors of values $+1$ and -1 only, and the discrete activation function (2) is used. Its essence is to use the following weights:

$$w_{ij} = \begin{cases} \sum_{k=1}^{N} x_i^k \cdot x_j^k & \text{for } i \neq j, \\ 0 & \text{for } i = j. \end{cases} , \tag{4}$$

which results in the weights matrix of the form:

$$W = \sum_{k=1}^{N} x^k (x^k)^T - \operatorname{diag}\big((x_i^k)^2\big) .$$

Neither the above rules, nor most other training heuristics take into account problems that may arise while training the network:

– several “spurious patterns” will, in fact, be stationary points of the network, as well as actual patterns,
– capacity of the network is limited and there may exist no weight matrix, responding to all training vectors properly.

Let us try to develop a more general approach.

4 Problems under solution

In general, there are two problems we may want to solve with respect to a recurrent ANN, described in Section 2:

1. We know the weights of the network and we want to find all stationary points.
2. We know all stationary points the network should have and we want to determine the weights, so that this condition was satisfied.

In both cases, the system under consideration is similar:

$$x_i - \sigma\Big(\sum_{j=1}^{n} w_{ij} x_j\Big) = 0, \quad \text{for } i = 1, \ldots, n \ , \tag{5}$$

but different quantities are unknowns under the search or the given parameters. In the first case, we know the matrix of weights: $W = [w_{ij}]$ and we seek x_i's and in the second case – vice versa.

Also, the number of equations differs in both cases. The first problem is always well-determined: the number of unknowns and of equations is equal to the number of neurons n. The second problem is not necessarily well-determined: we have $n \cdot (n-1)$ unknowns and the number of equations is equal to $n \cdot N$, where N is the number of vectors to remember.

To be more explicit: in the first case, we obtain the following problem:

$$\begin{gathered}\text{Find } x_i,\ i = 1, \ldots, n, \text{ such that:} \\ x_i - \sigma\Big(\sum_{j=1}^{n} w_{ij} x_j\Big) = 0, \quad \text{for } i = 1, \ldots, n \ . \end{gathered} \tag{6}$$

In the second case, it is:

$$\begin{gathered}\text{Find } w_{ij},\ i, j = 1, \ldots, n, \text{ such that:} \\ x_i^k - \sigma\Big(\sum_{j=1}^{n} w_{ij} x_j^k\Big) = 0, \quad \text{for } k = 1, \ldots, N \ . \end{gathered} \tag{7}$$

But in both cases, it is a system of nonlinear equations. What tools shall we apply to solve it?

5 Interval tools

Interval analysis is well-known to be a tractable approach to finding a solution – or all solutions of a nonlinear equations system, like the above ones.

There are several interval solvers of nonlinear systems (GlobSol, Ibex, Realpaver and SONIC are representative examples). In our research, we are using HIBA_USNE [4], developed by the first author. The name HIBA_USNE stands for Heuristical Interval Branch-and-prune Algorithm for Underdetermined and well-determined Systems of Nonlinear Equations and it has been described in a series of papers (including [11], [12], [14], [15] and [16]; cf. Chap. 5 of [17] and the references therein).

As the name states, the solver is based on interval methods (see, e.g., [8], [9], [20]), that operate on intervals instead of real numbers (so that result of an operation on numbers always belongs to the result of operation on intervals that contain the numerical inputs). Such methods are robust, guaranteed to enclose *all* solutions, even if they are computationally intensive and memory demanding. Their important advantage is allowing not only to locate solutions of well-determined and underdetermined systems, but also to *verify* them, i.e., prove that in a given box there is a solution point (or a segment of the solution manifold).

Details can be found in several textbooks, i.a., in these quoted above.

5.1 HIBA_USNE

Let us present the main algorithm (the standard interval notation, described in [10], will be used). The solver is based on the branch-and-prune (B&P) schema that can be expressed by pseudocode presented in Algorithm 1.

Algorithm 1 Interval branch-and-prune algorithm

```
Require: $L, \mathsf{f}, \varepsilon$
 1: {$L$ – the list of initial boxes, often containing a single box $\mathbf{x}^{(0)}$}
 2: {$L_{ver}$ – verified solution boxes, $L_{pos}$ – possible solution boxes}
 3: $L_{ver} = L_{pos} = \emptyset$
 4: $\mathbf{x}$ = pop ($L$)
 5: loop
 6:   process the box $\mathbf{x}$, using the rejection/reduction tests
 7:   if ($\mathbf{x}$ does not contain solutions) then
 8:     discard $\mathbf{x}$
 9:   else if ($\mathbf{x}$ is verified to contain a segment of the solution manifold) then
10:     push ($L_{ver}$, $\mathbf{x}$)
11:   else if (the tests resulted in two subboxes of $\mathbf{x}$: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) then
12:     $\mathbf{x} = \mathbf{x}^{(1)}$
13:     push ($L$, $\mathbf{x}^{(2)}$)
14:     cycle loop
15:   else if (wid $\mathbf{x} < \varepsilon$) then
16:     push ($L_{pos}$, $\mathbf{x}$) {The box $\mathbf{x}$ is too small for bisection}
17:   if ($\mathbf{x}$ was discarded or $\mathbf{x}$ was stored) then
18:     if ($L == \emptyset$) then
19:       return $L_{ver}, L_{pos}$ {All boxes have been considered}
20:     $\mathbf{x}$ = pop ($L$)
21:   else
22:     bisect ($\mathbf{x}$), obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
23:     $\mathbf{x} = \mathbf{x}^{(1)}$
24:     push ($L$, $\mathbf{x}^{(2)}$)
```

The "rejection/reduction tests", mentioned in the algorithm are described in previous papers (cf., e.g., [14], [15] and [16] and references therein):

– switching between the componentwise Newton operator (for larger boxes) and Gauss-Seidel with inverse-midpoint preconditioner, for smaller ones,
– a heuristic to choose whether to use or not the BC3 algorithm,
– a heuristic to choose when to use bound-consistency,

– a heuristic to choose when to use hull-consistency,
– sophisticated heuristics to choose the bisected component,
– an additional second-order approximation procedure,
– an initial exclusion phase of the algorithm (deleting some regions, not containing solutions) – based on Sobol sequences.

It is also worth mentioning that as Algorithm 1, as some of the tests performed on subsequent boxes are implemented in a multithreaded manner. Papers [11–16] discuss several details of this implementation and a summary can be found in Chap. 5 of [17].

5.2 ADHC

The HIBA_USNE solver collaborates with a library for algorithmic differentiation, also written by the first author. The library is called ADHC (Algorithmic Differentiation and Hull Consistency enforcing) [3]. Version 1.0 has been used in our experiments. This version has all necessary operations, including the exp function, used in (3), and division (that was not implemented in earlier versions of the package).

6 Numerical experiments

Numerical experiments have been performed on a machine with two Intel Xeon E5-2695 v2 processors (2.4 GHz). Each of them has 12 cores and on each core two hyper-threads (HT) can run. So, $2 \times 12 \times 2 = 48$ HT can be executed in parallel. The machine runs under control of a 64-bit GNU/Linux operating system, with the kernel 3.10.0-123.e17.x86_64 and glibc 2.17. They have non-uniform turbo frequencies from range 2.9–3.2 GHz. As there have been other users performing their computations also, we limited ourselves to using 24 threads only.

The Intel C++ compiler ICC 15.0.2 has been used. The solver has been written in C++, using the C++11 standard. The C-XSC library (version 2.5.4) [1] was used for interval computations. The parallelization was done with the packaged version of TBB 4.3 [2].

The author's HIBA_USNE solver has been used in version Beta2.5 and ADHC library, version 1.0.

We consider the network with n neurons ($n = 4$ or $n = 8$) and storing 1 or 3 vectors. The first vector to remember is always $(1, 1, \ldots, 1)$. The second one consists of $\frac{n}{2}$ values $+1$ and $\frac{n}{2}$ values -1. The third one consists of $n - 2$ values $+1$ and 2 values -1.

The following notation is used in the tables:

– fun.evals, grad.evals, Hesse evals – numbers of functions evaluations, functions' gradients and Hesse matrices evaluations (in the interval automatic differentiation arithmetic),
– bisecs – the number of boxes bisections,

Table 1. Computational results for Problem (6)

problem	$n = 4, N = 1$	$n = 8, N = 1$	$n = 4, N = 3$	$n = 8, N = 3$	$n = 12, N = 3$
fun. evals	1133	365,826	2,010	432,551	4,048,010,515
grad.evals	1629	483,302	2,484	689,774	6,298,617,714
Hesse evals	4	3,568	124	6,974	398,970,087
bisections	37	29,210	117	41,899	245,816,596
preconds	68	32,970	149	45,813	252,657,916
Sobol excl.	14	62	15	63	143
Sobol resul.	321	1,541	346	1,548	3,869
pos.boxes	1	2	4	0	3
verif.boxes	2	1	0	5	6
Leb.poss.	3e-36	4e-70	3e-27	0.0	4e-77
Leb.verif.	6e-30	2e-92	0.0	5e-71	1e-129
time (sec.)	< 1	1	<1	2	12,107

Table 2. Computational results for Problem (7)

problem	$n = 4, N = 1$	$n = 8, N = 1$	$n = 4, N = 3$	$n = 8, N = 3$	$n = 12, N = 3$
fun. evals	7376	369100	8870	420,484	3,303,590
grad.evals	16	64	48	192	432
Hesse evals	4	8	12	24	36
bisections	0	0	0	0	0
preconds	0	0	0	0	0
Sobol excl.	144	3136	144	3,136	17,424
Sobol resul.	0	0	0	0	0
pos.boxes	0	0	0	0	0
verif.boxes	0	0	0	0	0
Leb.poss.	0.0	0.0	0.0	0.0	0.0
Leb.verif.	0.0	0.0	0.0	0.0	0.0
time (sec.)	<1	< 1	<1	< 1	2

- preconds – the number of preconditioning matrix computations (i.e., performed Gauss-Seidel steps),
- Sobol excl. – the number of boxes to be excluded generated by the initial exclusion phase,
- Sobol resul. – the number of boxes resulting from the exclusion phase (cf. [13], [14]),
- pos.boxes, verif.boxes – number of elements in the computed lists of boxes containing possible and verified solutions,
- Leb.pos., Leb.verif. – total Lebesgue measures of both sets,
- time – computation time in seconds.

7 Analysis of the results

The HIBA_USNE solver can find solutions of Problem (6) pretty efficiently.

The solutions get found correctly. For instance, in the case of four neurons and a single stored pattern, three solutions are quickly found; two of the solutions are guaranteed:

```
x = [-1.030683E-008,1.031144E-008]
[-2.702852E-010,2.703469E-010]
[-3.596875E-009,3.598297E-009]
[-1.189070E-009,1.188578E-009]

x = [  0.858559,  0.858560]
[  0.858559,  0.858560]
[  0.858559,  0.858560]
[  0.858559,  0.858560]
```

and one is a possible solution:

```
x = [ -0.858560, -0.858559]
[ -0.858560, -0.858559]
[ -0.858560, -0.858559]
[ -0.858560, -0.858559]
```

Because of the properties of the sigmoid function (3), that nowhere reaches the values ± 1, the actual pattern $(1, 1, 1, 1)$ cannot be the solution of the equations. Yet, the solution is a (relatively crude), approximation of the pattern. Another solutions are the point $(0, 0, 0, 0)$, and minus the first solution. The number of solutions that get verified or found as possible solutions only, varies (cf. Table 6), but all of them get bounded correctly.

For problems of small dimensionality, all solutions get found immediately. Unfortunately, the time increases quickly with the number of neurons (but not with the number of stored patterns!) in the network. This is partially because Hopfield networks are 'dense': each neuron is connected to all other ones. Multilayer networks have a more 'sparse' structure, that may improve the scalability of the branch-and-prune method.

For Problem (7) of computing the weights matrix, the HIBA_USNE solver was less successful. This is not surprising: Problem (7) is underdetermined, and can have uncountably many solutions.

Actually, the solver has been successful on (7) when there had been no solutions: this can be verified easily, in many cases. As the sigmoid function (3) does not reach values ± 1 for finite arguments, there are no weights for which sequences of ± 1's are stationary points of the network, and the solver verifies it easily.

Unfortunately, seeking weights for a feasible solution is not that efficient. For instance, seeking weights for a network with a single stationary point at $(0.858, 0.858, 0.858, 0.858)$, had to be interrupted after three hours, without obtaining the results!

Possibly, it would make sense to seek solutions of Problem (7) with some additional constraints, but this has not been determined yet. In such case, it might be beneficial to transform the equations to the form:

$$\sum_{j=1}^{n} w_{ij} x_j^k = -\frac{1}{\beta} \cdot \ln\left(\frac{2}{1+x_i^k} - 1\right), \quad i = 1, \ldots, n, \;\; k = 1, \ldots, N \ .$$

Now, the equations are linear with respect to w_{ij}.

Also, interval methods can naturally be applied to seek *approximate* fixed points, instead of precise ones, but such experiments have not been performed yet.

8 Conclusions

The paper presents a promising application of interval methods and the HIBA_USNE solver. It can be used both to train and investigate the behavior of a recurrent neural network. The interval solver of nonlinear systems can potentially be applied to determining the weights matrix of the network, but more importantly: to localizing all stationary points of the network.

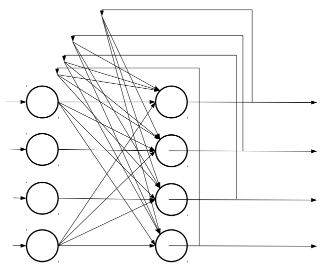

Fig. 2. A Hamming-type neural network

We have considered single-layer continuous Hopfield-like networks, but generalization to Hamming networks (Fig. 2) or convolutional multilayer ANNs (e.g., [7]) seems straightforward. This will be the subject of our further research, as well as further studies about Hopfield network: seeking for periodic states, seeking for approximate stationary points, and more sophisticated interval algorithms to train the network.

References

1. C++ eXtended Scientific Computing library. `http://www.xsc.de`, 2017.
2. Intel TBB. `http://www.threadingbuildingblocks.org`, 2017.
3. ADHC, C++ library. `https://www.researchgate.net/publication/316610415_ADHC_Algorithmic_Differentiation_and_Hull_Consistency_Alfa-05`, 2018.

4. HIBA_USNE, C++ library. `https://www.researchgate.net/publication/316687827_HIBA_USNE_Heuristical_Interval_Branch-and-prune_Algorithm_for_Underdetermined_and_well-determined_Systems_of_Nonlinear_Equations_-_Beta_25`, 2018.
5. S. P. Adam, D. A. Karras, G. D. Magoulas, and M. N. Vrahatis. Solving the linear interval tolerance problem for weight initialization of neural networks. *Neural Networks*, 54:17–37, 2014.
6. M. Beheshti, A. Berrached, A. de Korvin, C. Hu, and O. Sirisaengtaksin. On interval weighted three-layer neural networks. In *Simulation Symposium, 1998. Proceedings. 31st Annual*, pages 188–194. IEEE, 1998.
7. I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.
8. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, London, 2001.
9. R. B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer, Dordrecht, 1996.
10. R. B. Kearfott, M. T. Nakao, A. Neumaier, S. M. Rump, S. P. Shary, and P. van Hentenryck. Standardized notation in interval analysis. *Vychislennyie Tiehnologii (Computational Technologies)*, 15(1):7–13, 2010.
11. B. J. Kubica. Interval methods for solving underdetermined nonlinear equations systems. *Reliable Computing*, 15:207–217, 2011. Proceedings of SCAN 2008.
12. B. J. Kubica. Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. *Lecture Notes in Computer Science*, 7204:467–476, 2012. Proceedings of PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics).
13. B. J. Kubica. Excluding regions using Sobol sequences in an interval branch-and-prune method for nonlinear systems. *Reliable Computing*, 19(4):385–397, 2014. Proceedings of SCAN 2012 (15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics).
14. B. J. Kubica. Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. *Numerical Algorithms*, 70(4):929–963, 2015.
15. B. J. Kubica. Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. *Journal of Parallel and Distributed Computing*, 107:57–66, 2017.
16. B. J. Kubica. Role of hull-consistency in the HIBA_USNE multithreaded solver for nonlinear systems. *Lecture Notes in Computer Science*, 10778:381–390, 2018. Proceedings of PPAM 2017.
17. B. J. Kubica. *Interval methods for solving nonlinear constraint satisfaction, optimization and similar problems: From inequalities systems to game solutions*, volume 805 of *Studies in Computational Intelligence*. Springer, 2019.
18. J. Mańdziuk. *Hopfield-type neural networks. Theory and applications*. Akademicka Oficyna Wydawnicza EXIT, 2000. (in Polish).
19. P. V. Saraev. Numerical methods of interval analysis in learning neural network. *Automation and Remote Control*, 73(11):1865–1876, 2012.
20. S. P. Shary. *Finite-dimensional Interval Analysis*. Institute of Computational Technologies, Sibirian Branch of Russian Academy of Science, Novosibirsk, 2013.