Communication Papers of the 18th Conference on Computer Science and Intelligence Systems pp. 173–180
DOI: 10.15439/2023F865
ISSN 2300-5963 ACSIS, Vol. 37

Developing an interval method for training denoising autoencoders by bounding the noise

Bartłomiej Jacek Kubica

0000-0002-5547-3759
Institute of Information Technology,
Warsaw University of Life Sciences – SGGW,
ul. Nowoursynowska 159, 02-776 Warsaw, Poland
E-mail: bartlomiej_kubica@sggw.edu.pl

***Abstract*—This paper discusses prospects of using interval methods to training denoising autoencoders. Advantages and disadvantages of using the interval approach are discussed. It is proposed to formulate the problem of training the proper neural network as a constraint-satisfaction, and not optimization, problem. Pros and cons of this approach are considered. Preliminary numerical experiments are also presented.**

I. Introduction

AUTOENCODERS (AE) are commonly used, nowadays, and they found applications in various branches related to machine learning (ML). They can be used, i.a., for feature extraction, dimensionality reduction, denoising, and even as some kind of generative models (so-called variational autoencoders).

Interval methods, while used by several authors for neural network training (see, e.g., [1], [2], [3], [4], [5], [6], [7]), have rarely been used in conjunction with AE, so far. The only known exception is the paper [8]. This is surprising, because interval methods have – as we shall see – several natural advantages, when applied to training AEs. This paper intends to fill this gap, at least to some extent.

The paper is organized as follows. Section II introduces the idea of autoencoders. It (briefly) discusses various types of the AEs, their features, and applications. Section III introduces the interval calculus, and basics of the algorithms that use it. In Section IV, we discuss the interval approach to training AEs, and discuss the possible solutions.

II. Autoencoders and their types

Autoencoders (also called autoassociotors, at least in the early papers) are a specific kind of unsupervised (or semi-supervised) feed-forward neural networks [9], [10]. Their use dates back to the eighties; cf. [11], [12], [13].

The AE consists of at least three layers: the input layer (x), the hidden layer (h), and the output layer (y). Its essence is to reproduce the input on the output, but not in a trivial manner: $y = x$, but approximately. Depending on the structure and dimensionality of the hidden layer, the input can be reconstructed more or less precisely, and – as we shall see – the reconstruction process will capture various features of the data.

An AE can be logically decomposed into two parts:

- the *encoder*, transforming the input x to the representation of data, in the hidden layer: $h = f(x)$,
- the *decoder*, transforming the representation to the data from the original space: $y = f^*(h)$.

The f^* function in the above description is some sort of an 'approximate inverse' of f.

We train the AE to have:

$$y = f^*(f(x)) \approx x \ , \tag{1}$$

but how close y can get to x depends on the restrictions on the representation h: what is its dimensionality, etc.

Training is usually performed by minimizing some loss function (least-squares or the Kullback-Leibler divergence [14]), possibly plus some regularization term(s) forcing the satisfaction of some additional conditions, e.g., the presence of some features.

The general structure of the AE is presented in Fig. 1.

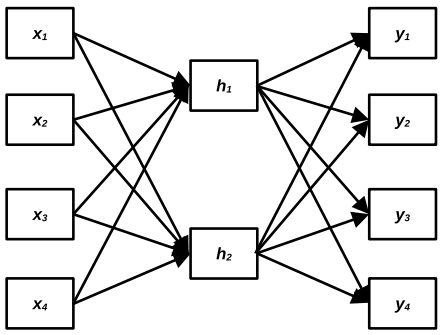

Fig. 1. General structure of an AE

This figure suggests that there are fewer neurons in the hidden layer than there are in the input and output layers. This is often the case: such an AE is called *undercomplete*, and the limitation of the representation in the hidden layer forces the network to learn the most important features only.

Topical area: Advanced Artificial Intelligence in Applications

Yet, another type of an AE is used as well: an *overcomplete autoencoder*, that has more neurons in the hidden layer than in the input or output ones (Fig. 2).

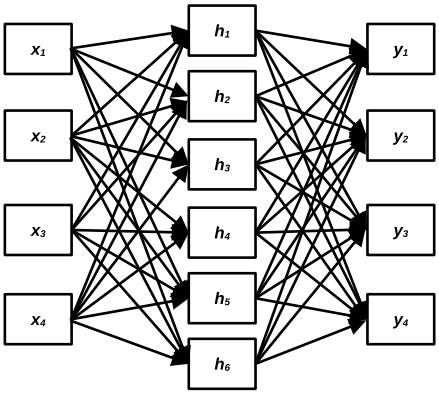

Fig. 2. An overcomplete autoencoder

How does such an AE work? Will it simply learn to approximate the identity function $y = x$, or will it find a useful representation? As it has already been mentioned, the essence is to provide the proper regularization in the learning process: the minimized loss function should contain a regularization term, being the 'penalty' for using too many neurons.

An example of such a regularization is used in the so-called *sparse autoencoder* (SAE). In this case, it is penalized to activate too many neurons in the hidden layer. This can be obtained, in particular, by using as the regularization term the Kullback-Leibler divergence of the hidden layer – see, e.g., [15].

Overcomplete AEs are usually considered to be more difficult to train than undercomplete ones, but more powerful. The source of their power is the ability to omit local optima, during the learning process.

Another kind of the AE, particularly important in this paper is a *denoising autoencoder* (DAE). In this case, the training process is subtly modified to obtain a representation that is robust in the presence of noise. Hence, instead of feeding the input layer with training vectors x, we use perturbed training data (let us denote it by $x + \varepsilon$), yet expecting the output layer to return $y \approx x$, and not $y \approx x + \varepsilon$. How can it be achieved?

Firstly, we have to assume some specific distribution of the noise; usually Gaussian distribution is used, but it it not the only possibility. Secondly, the training set has to be increased, as for each input vector x, we must now have a few its perturbed counterparts. Thirdly, reconstructing the values is usually done basing on the interdependency of various components of x, but this is not the only possibility. Sometimes, the values of x belong to a discrete set, and small perturbations can easilly be corrected.

One of the drawbacks of the traditional approach to training DAEs is the enlargement of the training set. Actually, later in the paper we propose an approach to mittigate it, by using some interval methods for training the DAE. Please compare also the latter discussion in Section VI.

In general, the autoencoder does not have to be limited to three layers. The number of intermediate layers can be increased; such an AE is called a *deep autoencoder* (or a *stacked autoencoder*). An example structure of a deep AE is presented in Fig. 3.

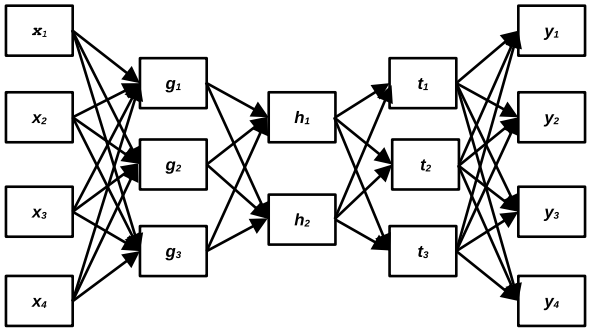

Fig. 3. A deep autoencoder

The virtue of a deep AE is that it can be trained iteratively: first we find the weights for a single hidden layer, such that $g = f_1(x)$, and $y = f_1^*(g) \approx x$, then $h = f_2(g)$, and $t = f_2^*(h) \approx g$, thus obtaining:

$$\begin{aligned} h &= f_2(f_1(x)) \ , \\ y &= f_1^*(f_2^*(h)) \ . \end{aligned}$$

Yet another kind of an AE is a contracting autoencoder (CAE). The essence is to train the AE so that we had $h = f(x)$, and the derivative of f was close to zero at the training points. Usually, it is obtained by adding the the loss function a regularization term, penalizing a norm of the Jacobi matrix of f. An analog for a deep CAE is straightforward.

But why would an AE satisfy such a condition? What do derivatives close to zero imply?

Actually, the idea is pretty similar (but not mathematically equivalent!) to a DAE: when the derivative of f is close to zero, adding a noise to x does not change its representation significantly. There are two key differences between DAE and CAE:

- DAE enforces some conditions on the output layer, i.e., on $y = f^*(f(x))$, while CAE enforces some conditions on the hidden layer, i.e., $h = f(x)$,
- CAE does not make any assumptions about the distribution of the noise, while DAE uses the noise generated using a specific distribution.

There are some approaches to combine DAE and CAE, e.e., the marginalized DAE (mDAE), proposed in [16].

Let us conclude this survey with a *variational autoencoder* (VAE). This kind of an AE tends to learn rather a probabilistic distribution of the data than a representation of a single

element of the training set. The idea has been proposed by [17].

Usually, it is assumed that the distribution is Gaussian, and we have to fit its mean value and standard deviation, but other distributions can be used as well [10]. To find the optimal values of the parameters, the Kullback-Leibler divergence is minimized, so that the distribution was as close to the given one, as possible.

A VAE is a neural network very different from the ones described up to now. In all previous architectures, the most important part of the network was the encoder; the role of the decoder was reduced to validating the encoder. For a VAE, the decoder is the most important part. It is worth noting that the role of such a network is to *generate* the data resembling (but not identical to) the input ones. We shall not use VAEs in this paper, but the topic is worth further consideration.

Activation functions

Let us mention one more detail: what activation functions are used in AEs? Several such functions happen to be used in artificial neural networks [9], [10], but for AEs two of them are the most common. We shall stick to using them, as well.

Two such activation functions – ReLU (Rectified Linear Unit):

$$ReLU(t) = \max(t, 0) \ . \tag{2}$$

and the sigmoid function:

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

III. Interval methods

The interval calculus is the tool of choice for us, in this paper. What is it?

It can be defined a branch of numerical analysis and mathematics that operates on intervals rather than precise numbers. A good introduction can be found in many classical textbooks, including, i.a., [18], [19], [20], [21], [22], [23], or a most recent one [24].

Arithmetic operations (as well as other operations and functions) on intervals are designed, so that the following condition was fulfilled:

$$\odot \in \{+, -, \cdot, /\}, \ a \in \mathbf{a}, \ b \in \mathbf{b} \ \text{ implies } \ a \odot b \in \mathbf{a} \odot \mathbf{b} \ . \tag{4}$$

In other words, the result of an operation on numbers will should contained in the result of an analogous operation on intervals, containing these numbers.

This results in the following formulae for arithmetic operations (cf., e.g., the aforementioned textbooks):

$$\begin{aligned}
[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \ , \\
[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \ , \\
[\underline{a}, \overline{a}] \cdot [\underline{b}, \overline{b}] &= [\min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), \max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})] \ , \\
[\underline{a}, \overline{a}] \ / \ [\underline{b}, \overline{b}] &= [\underline{a}, \overline{a}] \cdot \left[1 \,/\, \overline{b}, 1 \,/\, \underline{b}\right] \ , \qquad 0 \notin [\underline{b}, \overline{b}] \ .
\end{aligned} \tag{5}$$

It is worth noting that the above formulae are not the only possible ones. Alternative (and even more general) formulations are possible as well. Details can be found, i.a., in Chapter 2 of [24]. Also, let us mention that the division by an interval containing zero is also possible. This is done in the so-called extended Kahan-Novoa-Ratz arithmetic; cf., e.g., [20] for details.

Similarly to the arithmetic operations, we can define the power of an interval:

$$[\underline{a}, \overline{a}]^n = \begin{cases} [\underline{a}^n, \overline{a}^n] \text{ for odd } n \\ [\min\{\underline{a}^n, \overline{a}^n\}, \max\{\underline{a}^n, \overline{a}^n\}] \text{ for} \\ \quad \text{even } n \text{ and } 0 \notin [\underline{a}, \overline{a}] \\ [0, \max\{\underline{a}^n, \overline{a}^n\}] \text{ for even } n \text{ and} \\ \quad 0 \in [\underline{a}, \overline{a}] \end{cases} \ , \tag{6}$$

and other transcendental functions, like:

$$\begin{aligned}
\exp\left([\underline{a}, \overline{a}]\right) &= [\exp(\underline{a}), \exp(\overline{a})], \\
\log\left([\underline{a}, \overline{a}]\right) &= [\log(\underline{a}), \log(\overline{a})], \text{ for } \underline{a} > 0. \\
&\dots
\end{aligned}$$

For details, please consult, e.g., Section 2.3 of [24].

A. The problem under solution for interval algorithms

The approach described in the preamble of this section finds several applications. Not to repeat the whole discussion from Chapter 4 of [24], let us state that they can be used to seek the solutions of problems of the following form:

$$\text{Find } \textit{all } x \in X \text{ such that } P(x) \text{ is fulfilled.} \tag{7}$$

Here, $P(x)$ is a formula with a free variable x and $X \subseteq \mathbb{R}^n$. In particular, constraint satisfaction problems (CSP), and optimization problems (unconstrained and constrained ones) are specific instances of Problem (7).

It should be noted that Formula P can contain, in addition to the variable x, also some parameters; let us denote them by $a \in \mathbb{R}^k$. We have two main possibilities here:

$$\text{Find } \textit{all } x \in X \text{ such that } \ (\forall a \in \mathbf{a}) \ \ P(x, a) \text{ is fulfilled.}$$

or:

$$\text{Find } \textit{all } x \in X \text{ such that } \ (\exists a \in \mathbf{a}) \ \ P(x, a) \text{ is fulfilled.}$$

Other variants use various quantifiers for various components of the vector a, e.g., $(\forall a_1 \in \mathbf{a}_1)(\exists a_2 \in \mathbf{a}_2)(\forall a_3 \in \mathbf{a}_3)$, etc.

In the above manner, the intervals give us a natural tool to bound several kinds of uncertainty. While, in the opinion of the author, interval calculus should *not* be understood as a tool of uncertainty description, but rather as a general approach to seek points satisfying a certain logical condition, the uncertainty description remains an important family of its applications.

B. Interval branch-and-bound type methods

How to solve problems of type (7)? The generic algorithm proposed in [24] is called the branch-and-bound type method (B&BT). Instances of the B&BT algorithm are, among others, these popular ones:

- the branch-and-bound methods for optimization problems,

- the branch-and-prune method for CSPs.

The essence in both cases is to subdivide the boxes subsequently (starting from the initial box or the list of initial boxes), discarding the boxes that do not contain solutions, and possibly verifying some boxes to contain the solution(s). Details can be found in [24].

It is a specific instance of the so-called divide-and-conquer strategy (but the author himself dislikes this term).

Let us focus on the problem of solving a CSP, i.e., trying to compute the set:

$$S = \{x \in X \mid g_i(x) \leq 0, i = 1, \ldots, m\} \ ,$$

or succinctly: $S = \{x \in X \mid g(x) \leq 0\}$, where $g = (g_1, \ldots, g_m)$.

Using interval methods, we compute two lists of boxes: *verified* and *possible* solutions.

In case of a system of inequalities, the interior of the solution set S is nonempty and the verified solutions are boxes contained in this interior (boxes that contain solutions only). Possible boxes lie usually on the boundaries of S, and they contain some points both from the set S and from its complement $\mathbb{R}^n \setminus S$. Typically there are several possible boxes, unless S is a box itself (which would be highly unlikely).

The branch-and-prune algorithm for a CSP can be formulated as follows:

The 'rejection/reduction tests', mentioned in the algorithm have been described in the author's previous papers; specifically, please consult [25], [26], [27], [28] and the references therein.

In our version of the solver, the most important tool is hull-consistency (HC).

Definition 3.1: A box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ is hull-consistent with respect to a constraint $c(x_1, \ldots, x_n)$, iff:

$$\begin{aligned} &\forall i \ \mathbf{x}_i = \square\{s \in \mathbf{x}_i \mid \exists x_1 \in \mathbf{x}_1, \cdots \exists x_{i-1} \in \mathbf{x}_{i-1}, \\ &\exists x_{i+1} \in \mathbf{x}_{i+1} \cdots \exists x_n \in \mathbf{x}_n \\ &c(x_1, \ldots, x_{i-1}, s, x_{i+1}, \ldots, x_n)\} \ . \end{aligned}$$

A popular algorithm to enforce HC is called HC4. Details can be found, i.a., in [28] or Subsect. 5.5. of [24].

IV. Interval algorithms for training AEs

A. Interval methods and neural networks

As it has already been mentioned, interval algorithms have been extensively used for training various kinds of neural networks. Two approaches have been formulated; in [3], they are simply called 'type 1' and 'type 2' interval neural network problems. 'Type 1' problems can be solved, by obtaining the solution of an equations system:

$$f(x_i, w) = y_i, \text{ for } i = 1, \ldots, N \ , \tag{8}$$

while 'type 2' problems require solving an optimization problem:

$$\min_w \left(||f(x_i, w) - y_i|| \right) \ . \tag{9}$$

Algorithm 1 Interval branch-and-prune algorithm for a system of inequalities

Require: $\mathbf{x}^{(0)}, \mathrm{g}, \varepsilon$

{$\mathbf{x}^{(0)}$ is the initial box, $\mathrm{g}(\cdot)$ is the interval extension of the function $g\colon \mathbb{R}^n \to \mathbb{R}^m$}
{L_{ver} – verified solution boxes, L_{pos} – possible solution boxes}
$L_{ver} = \emptyset$
$L_{pos} = \emptyset$
$\mathbf{x} = \mathbf{x}^{(0)}$
loop
  compute $\mathbf{y} = \mathrm{g}(\mathbf{x})$
  optionally, process the box $\mathbf{x}$, using additional rejection/reduction tests
  if $(\underline{y} > 0)$ **then**
    discard $\mathbf{x}$
  else if $(\overline{y} \leq 0])$ **then**
    push $(L_{ver}, \mathbf{x})$
  else if $(\mathrm{wid}\,\mathbf{x} < \varepsilon)$ **then**
    push $(L_{pos}, \mathbf{x})$ {The box $\mathbf{x}$ is too small for bisection}
  end if
  if ($\mathbf{x}$ was discarded **or** $\mathbf{x}$ was stored) **then**
    if $(L == \emptyset)$ **then**
      return L_{ver}, L_{pos} {All boxes have been considered}
    end if
    $\mathbf{x}$ = pop (L)
  else
    bisect $(\mathbf{x})$, obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push $(L, \mathbf{x}^{(2)})$
  end if
end loop

In the above formulae, by f we denoted the function, represented by the neural network, (x_i, y_i) were the pairs from the training set, and w – the vector of weights.

It is worth noting that virtually all non-interval methods, use the approach (9) for training neural networks. The objective function from (9) can be modified to contain some regularization terms, e.g., to obtain the sparsity or other features of the solution.

Only the interval calculus allows replacing optimization with the direct search of points satisfying certain constraints. In (8) they are equations, but inequality constraints are even more natural in the interval formulation:

$$\mathrm{f}(\mathbf{x}_i, \mathbf{w}) \subseteq \mathbf{y}_i, \text{ for } i = 1, \ldots, N \ . \tag{8'}$$

Also, instead of using the regularization terms, we can directly check (non)satisfiability of various constraints on subsequent boxes. This is the virtue of the interval calculus, that is a natural approach to seeking points satisfying certain logical conditions. Details can be found in [24].

B. The case of an autoencoder

As it has already been stated, interval methods have many natural advantages, when applied to training AEs.

Firstly, we can use CSP solving approach (8'), instead of the optimization problem (9), which is a more straightforward approach.

Secondly, handling all uncertainties, including the noise in a denoising AE is natural, by the virtues of the interval calculus.

Also, when using interval branch-and-bound type methods, we can restrict ourselves to using undercomplete AEs. The additional information, that would be processed for overcomplete AEs, will still be available for various parameterizations of the network, represented by various boxes in the B&BT procedure.

What is more, as stacked AEs can be trained subsequently, the problem under consideration is of lower dimensionality than for some other neural networks.

V. The solver and other software used

In a series of earlier papers, including [25], [26], [27], [28], the author has introduced his solver HIBA_USNE (Heuristical Interval Branch-and-prune Algorithm for Underdetermined and well-determined Systems of Nonlinear Equations) [29]. It was also used in [4] to analyze a Hopfield-like neural network.

HIBA_USNE in its original form could also be used to train a DAE, but it would not be the optimal tool for this purpose. Please note that now the CSP under consideration has a specific form. It contains both equations and inequalities, but:

- the equations refer to hidden layer(s), and they always have the form: $h_i = \sigma\left(\sum_{j=1}^{n} w_{ij}^1 x_j\right)$,
- the inequalities refer to the output layer; they come from Formula (8'), and they have the form: $\sigma\left(\sum_{j=1}^{n} w_{ij}^2 h_j\right) \subseteq [\underline{y}_i, \overline{y}_i]$.

Obviously, if we had more hidden layers, there would be analogous equations for each of them.

What is important is to note that the role of equations is mostly to establish the relations between the input x and the output y. The specific, 'explicit' form of these equations:

$$h_i = \sigma\left(\sum_{j=1}^{n} w_{ij}^1 x_j\right) \ ,$$

makes the use of the Newton operator on them almost useless. The most important tool is the hull-consistency (HC) enforcing operator, which propagates the information forward and backward through the layers of the neural network.

There are also other tools that might improve the performance of computing the weights of the neural network, but they have not been implemented yet. They will be briefly mentioned in Section VII.

Nevertheless, due to the specific structure of the problem under consideration, the HIBA_USNE solver has been forked by the author. In this paper, the fork, called HIBA_TANN [30] is used. The name stands for HIBA_USNE applied to Training Artificial Neural Networks. It has been adapted for this specific applications. All irrelevant tools have been removed from it (precisely: all but the interval Newton operator and the HC enforcing procedure).

Also, the following additions and amendments have been made to it:

- Only variables representing the weights of the network, and not values propagated by the neurons, get bisected.
- A (primitive) procedure to construct a feasible box (i.e., satisfying all constraints) has been implemented.
- Irrelevant examples have been removed, and new ones have been added.

VI. Experiments

A. Environment

All experiments have been performed on the author's laptop computer, with AMD Ryzen 5-4600H CPU (6 cores, 12 hardware threads; 3GHz). The machine ran under control of a 64-bit Manjaro GNU/Linux operating system with glibc 2.37-2 and the Linux kernel 5.15.93-1-MANJARO (with SMP and PREEMPT options).

The software was written in C++ and compiled using the GCC compiler (GCC 12.2.1). The parallelization (8 threads) was done with TBB 2021.5.0-2 [31]. OpenBLAS 0.3.17 [32] was linked for BLAS operations.

As for the the author's libraries, the following versions have been used:

- ADHC 2.2.1,
- survive-CXSC 2.6.1,
- HIBA_TANN 0.9.1, the fork of HIBA_USNE.

B. The test data set

The experiments have been performed using one of the classical datasets used to test ML tools: *the Iris dataset* [33]. This popular dataset contains descriptions of 150 individuals of some Iris plants, belonging to three species: Iris-setosa, Iris-versicolor, and Iris-virginica. Each of the individuals is described by four numerical attributes: sepal length, sepal width, petal length, and petal width – all of them in centimeters.

Usually, this dataset is used for verifying classification tools; in our experiments, we shall attempt to train an AE to reproduce the attribute values. Adhesion to a specific class will be ignored.

C. Uncertainty

The Iris dataset in its original form contains no uncertainty. In the experiments, the author has induced it by adding to all attributes a random noise. It had a normal distribution with the mean-value zero, and a few values of the standard deviation σ.

Two versions of the uncertainty representation have been considered:

- *Probabilistic uncertainty*: the random values are generated, using the C++11 `std::normal_distribution` class. The size of the dataset is increased four times, to have four

instances of each individual, with various noise values added to its attributes.
- *Interval uncertainty*: the noise value is bounded by the interval $[-3\sigma, +3\sigma]$.

The first version results in significant increasing of the size of the problem under consideration. Instead of $150 \times 4 = 600$ (150 individuals times 4 attributes), we now have $150 \times 4 \times 4 = 2400$ inequality constraints plus the related equations.

The second version does not require increasing the problem dimension. Thanks to the virtues of the interval calculus, all possible values are bound by a single interval.

Remark 6.1: It is worth noting that formally, the interval $[-3\sigma, +3\sigma]$ does not bound the whole support set of the normal distribution $N(0, \sigma)$. Indeed, theoretically, this support is the whole set $\mathbb{R}$. Nevertheless, virtually all practical generators of the normally distributed points generate the points from this range, only.

D. The structure of the neural network

The AE we are training in the presented experiments has the structure precisely described by Fig. 1: there is a single hidden layer, and the numbers of neurons in the input, hidden, and output layers are: 4, 2, 4, respectively.

Each layer is dense, i.e., each neuron of the hidden layer is connected to all neurons of both the input and output layers.

E. Numerical results

We consider the following versions of the problem:

- The ReLU activation function for the neurons, and the noise with $\sigma = 1.0$.
- The ReLU activation function, and the noise with $\sigma = 0.1$.
- The sigmoid activation function, and the noise with $\sigma = 1.0$.
- The sigmoid activation function, and the noise with $\sigma = 0.1$.

All four problems are solved by two versions of the program – using the interval or probabilistic uncertainty description.

The following notation is used in all of the tables:

- eq.evals, grad.eq.evals – numbers of equations evaluations, and their functions' gradients (in the interval algorithmic differentiation arithmetic),
- ineq.evals, grad.ineq.evals – numbers of inequalities evaluations, and their functions' gradients (in the interval algorithmic differentiation arithmetic),
- bisecs – the number of boxes bisections,
- HC evals – numbers of times hull-consistency has been enforced on a box,
- 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.

F. Analysis of the results

For the sigmoid function, it is not possible to obtain the result satisfying all constraints. Both versions of the program are able to determine it immediately (without any bisections, in a single HC enforcing step!). And it is worth noting that a non-interval algorithm would not be able to tell it for sure – even after a longer search.

For the ReLU function, the version bounding all uncertainty with a single interval, finds a solution quickly, yet it is not able to verify it. The version using a probabilistic representation of the uncertainty was not able to solve the problem in a reasonable time. This fact was surprising to the author – even provided the severely increased number of constraints.

Further studies should improve this version of the algorithm, as well.

VII. Conclusions and future work

This paper describes an attempt to use interval-based constraint-satisfaction algorithms for training denoising autoencoders. The results are interesting, yet only partially successful.

Even for the relatively simple and small problem, considered in the paper, only one version of the algorithm was able to deliver the solution in a reasonable time, and the solution has not been verified with certainty.

Still the results show several important prospects of the proposed approach, in particular the possibility of determining the non-existence of the AE with the given structure that would represent the given data with the assumed precision.

There are several tools that can be used to enhance the considered algorithms. In particular:

- Zonotopes [34] or the Taylor arithmetic [7] could be used to reduce the dependency problem in interval formulae.
- Also, they can be used to represent the covariance matrix of the noise; in the current implementation all attributes are assumed to have independent perturbances.
- A more sophisticated procedure for constructing feasible boxes should be delivered; in particular, it could use Kaucher arithmetic and related theorems, proven by Shary [23], [35].
- For the probabilistic representation, it might be worthwhile to consider only a random subset of the constraints. It is a technique analogous to using the stochastic gradient in optimization problems.
- Finally, processing various equations and inequalities can be parallelized. In the current version of the solver, processing different boxes is done in parallel, but the HC4 algorithm is serial. Cf. the discussion in Sect. 9 of [24].

Also, a similar study is planned for a CAE.

References

[1] 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*, vol. 54, pp. 17–37, 2014.

[2] S. Huang, Z. Ma, S. Yu, and Y. Han, "New discrete-time zeroing neural network for solving time-variant underdetermined nonlinear systems under bound constraint," *Neurocomputing*, vol. 487, pp. 214–227, 2022.

TABLE I
COMPUTATIONAL RESULTS FOR INTERVAL UNCERTAINTY)

	ReLU, $\sigma = 1$	ReLU, $\sigma = 0.1$	sigmoid, $\sigma = 1$	sigmoid, $\sigma = 0.1$
eq. evals	0	0	0	0
grad.eq.evals	300	300	300	300
ineq. evals	325,283	2,006,727	0	0
grad.ineq.evals	0	0	0	0
bisections	268	1,731	0	0
HC evals	340	3,194	0	0
pos. boxes	1	1	0	0
verif. boxes	0	0	0	0
Leb. pos.	3e-249	3e-249	0.0	0.0
Leb. verif.	0.0	0.0	0.0	0.0
time (sec.)	1	3	<1	<1

TABLE II
COMPUTATIONAL RESULTS FOR PROBABILISTIC UNCERTAINTY)

	ReLU, $\sigma = 1$	ReLU, $\sigma = 0.1$	sigmoid, $\sigma = 1$	sigmoid, $\sigma = 0.1$
eq. evals	n/a	n/a	0	0
grad.eq.evals	n/a	n/a	300	300
ineq. evals	n/a	n/a	0	0
grad.ineq.evals	n/a	n/a	0	0
bisections	n/a	n/a	0	0
HC evals	n/a	n/a	0	0
pos. boxes	n/a	n/a	0	0
verif. boxes	n/a	n/a	0	0
Leb. pos.	n/a	n/a	0.0	0.0
Leb. verif.	n/a	n/a	0.0	0.0
time (sec.)	>3600	>3600	<1	<1

[3] 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*. IEEE, 1998, pp. 188–194.

[4] B. J. Kubica, P. Hoser, and A. Wiliński, "Interval methods for seeking fixed points of recurrent neural networks," in *International Conference on Computational Science*. Springer, 2020. doi: 10.1007/978-3-030-50420-5_30 pp. 414–423.

[5] A. Rauh and E. Auer, "Interval extension of neural network models for the electrochemical behavior of high-temperature fuel cells," *Frontiers in Control Engineering*, vol. 3, 2022. doi: 10.3389/fcteg.2022.785123

[6] P. V. Saraev, "Numerical methods of interval analysis in learning neural network," *Automation and Remote Control*, vol. 73, no. 11, pp. 1865–1876, 2012.

[7] E. de Weerdt, Q. Chu, and J. Mulder, "Neural network output optimization using interval analysis," *IEEE Transactions on Neural Networks*, vol. 20, no. 4, pp. 638–653, 2009.

[8] L. V. Utkin, A. V. Podolskaja, and V. S. Zaborovsky, "A robust interval autoencoder," in *2017 International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO)*. IEEE, 2017, pp. 115–120.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[10] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," Institute for Cognitive Science, University of California, San Diego, Tech. Rep. 8506, 1985.

[12] ——, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[13] G. E. Hinton, "Connectionist learning procedures," in *Machine learning*, 1990, pp. 555–610.

[14] O. Kosheleva and V. Kreinovich, "Why deep learning methods use KL divergence instead of least squares: A possible pedagogical explanation," University of Texas at El Paso, Tech. Rep. UTEP-CS-17-95, 2017.

[15] X. Li, S. Feng, N. Hou, R. Wang, H. Li, M. ZGao, and S. Li, "Surface microseismic data denoising based on sparse autoencoder and Kalman filter," *Systems Science & Control Engineering*, vol. 10, no. 1, pp. 616–628, 2022.

[16] M. Chen, K. Weinberger, F. Sha, and YoshuaBengio, "Marginalized denoising auto-encoders for nonlinear representations," in *International conference on machine learning*. PMLR, 2014, pp. 1476–1484.

[17] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[18] E. Hansen and W. Walster, *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 2004.

[19] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*. Springer, London, 2001.

[20] R. B. Kearfott, *Rigorous Global Search: Continuous Problems*. Kluwer, Dordrecht, 1996.

[21] U. Kulisch, *Computer Arithmetic and Validity – Theory, Implementation and Applications*. De Gruyter, Berlin, New York, 2008.

[22] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. SIAM, Philadelphia, 2009.

[23] S. P. Shary, *Finite-dimensional Interval Analysis*. Institute of Computational Technologies, Sibirian Branch of Russian Academy of Science, Novosibirsk, 2013.

[24] B. J. Kubica, *Interval methods for solving nonlinear constraint satisfaction, optimization and similar problems: From inequalities systems to game solutions*, ser. Studies in Computational Intelligence. Springer, 2019, vol. 805.

[25] ——, "Interval methods for solving underdetermined nonlinear equations systems," *Reliable Computing*, vol. 15, pp. 207–217, 2011, proceedings of SCAN 2008.

[26] ——, "Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems," *Numerical Algorithms*, vol. 70, no. 4, pp. 929–963, 2015. doi: 10.1007/s11075-015-9980-y

[27] ——, "Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 57–66, 2017. doi: 10.1016/j.jpdc.2017.03.009

[28] ——, "Role of hull-consistency in the HIBA_USNE multithreaded solver for nonlinear systems," *Lecture Notes in Computer Science*, vol. 10778, pp. 381–390, 2018. doi: 10.1007/978-3-319-78054-2_36 Proceedings of PPAM 2017.

[29] "HIBA_USNE, C++ library," https://gitlab.com/bkubica/hiba_usne, 2023.

[30] "HIBA_TANN, C++ library," https://gitlab.com/bkubica/hiba_tann, 2023.
[31] "Intel TBB," https://github.com/oneapi-src/oneTBB, 2023.
[32] "OpenBLAS library," https://www.openblas.net, 2023.
[33] "Iris Species dataset," https://www.kaggle.com/datasets/uciml/iris?resource=download, 2023.
[34] B. J. Kubica, "Preliminary experiments with an interval Model-Predictive-Control solver," *Lecture Notes in Computer Science*, vol. 9574, pp. 464–473, 2016, PPAM 2015 Proceedings.
[35] S. P. Shary, "Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of Kaucher arithmetic," *Reliable Computing*, vol. 2, no. 1, pp. 3–33, 1996.