The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-06117-6

On the computation of the gradient in implicit neural networks

Béla J. Szekeres[1] · **Ferenc Izsák**[2,3]

Accepted: 29 March 2024

Abstract
Implicit neural networks and the related deep equilibrium models are investigated. To train these networks, the gradient of the corresponding loss function should be computed. Bypassing the implicit function theorem, we develop an explicit representation of this quantity, which leads to an easily accessible computational algorithm. The theoretical findings are also supported by numerical simulations.

Keywords Implicit neural network · Deep equilibrium model · Backpropagation · Directed cyclic graph

1 Introduction

The conventional neural networks have a feedforward structure: several layers are stacked after each other and their output can be computed explicitly. To generalize this structure, the so called implicit neural networks were introduced and analyzed in [1–5]. Also, a related approach called the deep equilibrium models was developed in the works [6–8]. Shortly, this model can be described as a feedforward deep neural network with identic layers. Practically, by increasing the number of layers, the existence and the computation of an equilibrium state is investigated. More precisely, Bai et al. [6] formulated an L-layer feedforward network as

Béla J. Szekeres and Ferenc Izsák have contributed equally to this work.

✉ Béla J. Szekeres
szekeres@inf.elte.hu

Ferenc Izsák
ferenc.izsak@ttk.elte.hu

[1] Department of Numerical Analysis, Faculty of Informatics, Eötvös Loránd University, Pázmány P. stny. 1C, Budapest 1117, Hungary

[2] Alfréd Rényi Institute of Mathematics, Reáltanoda u. 13-15., Budapest 1053, Hungary

[3] MTA ELTE NumNet Research Group, Eötvös Loránd University, Pázmány P. stny. 1C, Budapest 1117, Hungary

Published online: 25 April 2024

$$z^{[k+1]} = f_\theta(z^{[k]}; x), \quad k = 0, \dots, L-1,$$

, where $z^{[k]}$ are the hidden states in the k-th layer and the input x was utilized in each transition. In case of certain stability conditions, the limit $L \to \infty$ corresponds to a fixed-point iteration and hence leads to an equilibrium solution z of the equation

$$z = f_\theta(z; x). \tag{1}$$

However, the main task is to minimize a given loss function $\mathcal{E}(z;t)$ among the solutions of Eq. (1) by optimizing the parameters θ of the transformation f_θ. Here, for a gradient-based optimization technique, the gradient $\frac{\partial \mathcal{E}}{\partial \theta}$ of the loss function has to be calculated. The corresponding theory is based on the implicit function theorem, see [6, 9], and [1]. An efficient numerical approximation of z in Eq. (1) is also rather complex, for further details see [7, 10, 11].

Our contributions in this paper, being theoretical and empirical, are as follows. We introduce implicit neural networks similarly to deep equilibrium models, and propose a novel theory bypassing the traditional reliance on the implicit function theorem. This advancement leads to an easily accessible algorithm for computing the gradient. Our theoretical results are also confirmed with numerical simulations providing empirical evidence for our theoretical contributions.

2 Preliminaries

2.1 Construction of the network

In general, a neural network is represented by a directed graph [12], the computational graph of the network. A network is called feedforward or acyclic if the corresponding graph is acyclic. Similarly, cyclic directed graphs correspond to the so-called implicit neural networks.

We use K for the total number of vertices, which are also called the neurons. Let a_j denote the activation value of the j-th neuron with $f_j : \mathbb{R} \to \mathbb{R}$ being the corresponding activation function. Common examples are, e.g., $f_j(z) = \tanh(z)$ or $f_j(z) = \max\{0, z\}$. Let the lift operator $\hat{\cdot} : \mathbb{R}^L \to \mathbb{R}^K$ be defined by the formula $\hat{x} = \big(x_1, \dots, x_L, 0 \dots, 0\big)^T$. That is, we assume that the input data is copied to the first L neurons of the network for simplicity.

For an input vector $x \in \mathbb{R}^L$, the network is evaluated as follows. If a neuron with index j receives a stimulus of magnitude a_l from its ancestor of index l along the edge with the weight $w_{j,l}$ and a constant stimulus b_j (also called the bias) applied to it, then the cumulated input z_j of this neuron is

$$z_j = \sum_l w_{j,l} a_l + b_j + \hat{x}_j. \tag{2}$$

Accordingly, the activation value of the neuron with index j is

$$a_j = f_j(z_j). \tag{3}$$

Assume for simplicity that the indexing of the neurons is such that the last N neurons are the output ones.

2.2 Problem statement

Indeed, the formulas in (2), (3) define the following system of K equations:

$$\begin{cases} z(x) = Wa(x) + \hat{x} + b \\ a(x) = f(z(x)). \end{cases} \tag{4}$$

Here, summarized, $z(x), a(x), b \in \mathbb{R}^K$, $W \in \mathbb{R}^{K\times K}$ and $x \in \mathbb{R}^L$. Also, the functions $f_j : \mathbb{R} \to \mathbb{R}$ are given for $j = 1, \dots, K$, which define $f(z) = \big(f_1(z_1), \dots, f_K(z_K)\big)^T \in \mathbb{R}^K$ with $z = \big(z_1, \dots, z_K\big)^T$. Sometimes, we simplify the notation and omit the x-dependence of the terms in (4).

We also have M pairs of training samples (x, y) with $x \in \mathbb{R}^L$ the input and $y \in \mathbb{R}^N$ the target vectors assuming that $1 \le L, N \le K$. To compute with vectors of size K, we introduce the operator $\tilde{\cdot} : \mathbb{R}^N \to \mathbb{R}^K$ defined by the formula $\tilde{y} = \big(0, \dots, 0, y_1, \dots, y_N\big)^T$.

At the m-th pair of samples, i.e., at the input $x^{(m)}$, the error is defined as

$$\mathcal{E}^{(m)} = \frac{1}{2} \sum_{j=K-N+1}^{K} \big(\tilde{y}_j^{(m)} - a_j(x^{(m)})\big)^2, \tag{5}$$

where $\tilde{y}_j^{(m)}$ denotes the corresponding component of the m-th training sample $y^{(m)} = (y_1^{(m)}, \dots, y_N^{(m)}) \in \mathbb{R}^N$ of the target vector to be compared with the value of $a_j(x^{(m)})$.

The average error $\mathcal{E}$ over all pairs of training samples is given by

$$\mathcal{E} = \frac{1}{M} \sum_{m=1}^{M} \mathcal{E}^{(m)}, \tag{6}$$

which will be minimized with respect W and b.

Solving Eq. (4) by a fixed-point iteration yields the vector $z = \big(z_1, \dots, z_K\big)^T$ of neuron input values and the vector $a = \big(a_1, \dots, a_K\big)^T$ of activation values. A single step of this has the form

$$z^{(l)} = Wa^{(l-1)} + b + \hat{x}, \quad a^{(l)} = f(z^{(l)}), \quad l \ge 2 \quad \text{and } z^{(1)} = \hat{x} \in \mathbb{R}^K. \tag{7}$$

An important observation is that the iteration in (7) delivers a feedforward neural network of infinite number of layers with K neurons in each layer, so we get a deep

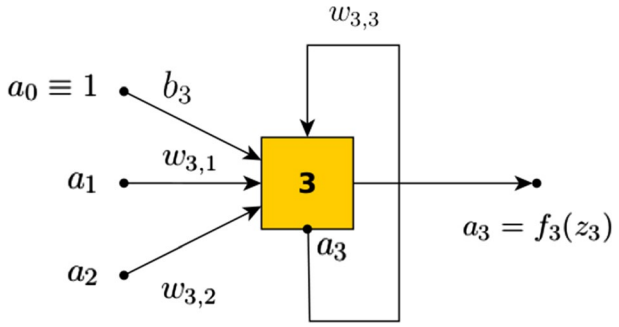

Fig. 1 Example of computing the value of a neuron in an implicit neural network. The neuron of index 3 has three conventional inputs, plus a loop edge leading back to itself and the output of the neuron is also its input, therefore $z_3 = w_{3,1}a_1 + w_{3,2}a_2 + b_3$

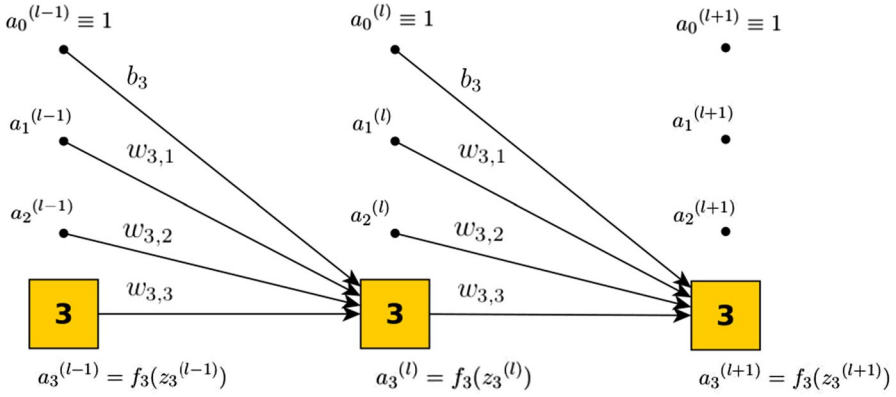

Fig. 2 Unrolling of the implicit neural network shown in Fig. 1 focusing on the third neuron in the $l-1$-th, l-th and $l+1$-th layers, $z_3^{(l)} = w_{3,1}a_1^{(l-1)} + w_{3,2}a_2^{(l-1)} + w_{3,3}a_3^{(l-1)} + b_3$, and $z_3^{(l+1)} = w_{3,1}a_1^{(l)} + w_{3,2}a_2^{(l)} + w_{3,3}a_3^{(l)} + b_3$

equilibrium model. In this framework, the fixed point iteration corresponds to the layer-wise computation with the original input. The weights of the edges passing between each two adjacent layers are given by the matrix $W \in \mathbb{R}^{K\times K}$ and the bias vector $b \in \mathbb{R}^K$. A small network is shown in Fig. 1 while Fig. 2 illustrates the unrolling of this network, focusing on the third neuron.

The pseudocode for computing the network is given in Algorithm 1.

Algorithm 1 Calculation of the network

procedure CALC_NETWORK($x, \mathrm{T}, \mathrm{tol}, W, b, K, L$)
  error $\leftarrow \infty$; $z_i \leftarrow 0, a_i \leftarrow 0, f_i' \leftarrow 0, \quad i = 1, \ldots, K$ ▷ Initialization
  iter $\leftarrow 0$
  while error $>$ tol and iter $<$ T **do** ▷ The fixed-point iteration
    iter $\leftarrow$ iter $+ 1$; $a_{old} \leftarrow a$; $z_{old} \leftarrow z$; $z_i \leftarrow 0; \quad i = 1, \ldots, K$
    for $j = 1 : K$ **do** ▷ $z = Wa + b + \hat{x}$
      for $k = 1 : K$ **do**
        if $\exists j \to k$ edge **then**
          $z_k \leftarrow z_k + w_{k,j} a_j$
        end if
      end for
      $z_j \leftarrow z_j + b_j + \hat{x}_j$
    end for
    for $j = 1 : K$ **do**
      $a_j \leftarrow f_j(z_j)$; $f_j{}' \leftarrow f_j{}'(z_j)$;
    end for
    error $\leftarrow \|z_{old} - z\|_\infty$
  end while
  return z, a, f' ▷ Inputs, activations, derivatives
end procedure

2.3 Further notations

Summarized, we use the following notations in the infinitely deep network:

- The initial vector of the iteration is $z^{(1)} = \hat{x} \in \mathbb{R}^K$. Let $z_i^{(l)}(x)$ denote the input value of the i-th neuron in the l-th layer and use $z_i(x) = \lim_{l \to \infty} z_i^{(l)}(x)$ provided that this exists and is finite. In vector form, we have

$$z^{(l)}(x) = \left(z_1^{(l)}(x), \ldots, z_K^{(l)}(x)\right)^T \quad \text{and} \quad z(x) = \left(z_1(x), \ldots, z_K(x)\right)^T.$$

  Sometimes, for simplicity, we omit the arguments x.
- Let $a_i^{(l)}(x)$ denote the activation value of the i-th neuron in the l-th layer with the input vector x. We use the notation $a_i(x) = \lim_{l \to \infty} a_i^{(l)}(x)$, provided that this exists and is finite. Accordingly, we use

$$f(z^{(l)})(x) = a^{(l)}(x) = \left(a_1^{(l)}(x), \ldots, a_K^{(l)}(x)\right)^T$$

  and $a(x) = \left(a_1(x), \ldots, a_K(x)\right)^T$.
- Parallel with the formula (5), we also introduce $d^{(\infty)} \in \mathbb{R}^K$ as

$$d_j^{(\infty)} = \begin{cases} \left(a_j - \tilde{y}_j^{(m)}\right) f_j'(z_j), & K - N < j \le K \\ 0, & 1 \le j \le K - N. \end{cases}$$

Here, the activation function $f_j : \mathbb{R} \to \mathbb{R}$, which is applied to the j-th neuron.

- We use the notation

$$D_i^{(l)} = f_i'(z_i^{(l)})$$

for the utility value of the i-th neuron in the l-th layer and $D_i = \lim_{l\to\infty} D_i^{(l)}$ provided that it exists and is finite. We also define the diagonal matrix $D \in \mathbb{R}^{K\times K}$ such that $D = \operatorname{diag}\left(D_1, \dots, D_K\right)$ and similarly, the diagonal matrices $D^{(l)} \in \mathbb{R}^{K\times K}$ on the l-th layer.

3 Theoretical results

As discussed previously, we transform the original implicit network into an infinitely deep feedforward one. We apply the gradient backpropagation method in this network. To minimize, the error function (6) using some gradient-based method, we need to determine the partial derivatives $\frac{\partial \mathcal{E}^{(m)}}{\partial w_{j,i}}$ and $\frac{\partial \mathcal{E}^{(m)}}{\partial b_j}$ after calculating the equilibrium in iteration (7). In the following statement, we express these in concrete terms. We make use the gradient backpropagation method [13] by applying it first to a finite network, and then performing a limit transition with respect to the number of the layers. For our main result, we use the following assumptions.

Assumptions:

(i) Equation (4) has a unique solution and the iteration in (7) is convergent such that we also have

$$\frac{\partial a_k}{\partial b_j} = \lim_{R\to\infty} \frac{\partial a_k^{(R),R}}{\partial b_j}$$

for all indices $k = K - N + 1, \dots, K$ and $j = 1, \dots, K$.

(ii) $f_i \in C^1(\mathbb{R})$, $\forall i = 1, \dots, K$ and their derivatives are bounded.

(iii) The linear mapping $DW^T \in \mathbb{R}^{K\times K}$ is a contraction in some induced norm, i.e. $\|DW^T\| < 1$.

Theorem 1 *Using assumptions in* (*i*)–(*iii*), *the system of equations*

$$d = \left(I - DW^T\right)^{-1} d^{(\infty)} \tag{8}$$

has a unique solution. Here, $I \in \mathbb{R}^{K\times K}$ *is the identity matrix. Furthermore*, *the partial derivatives of the error function* $\mathcal{E}^{(m)}$ *can be given as*

$$\frac{\partial \mathcal{E}^{(m)}}{\partial b_j} = d_j \quad \text{and} \quad \frac{\partial \mathcal{E}^{(m)}}{\partial w_{j,i}} = a_i d_j. \tag{9}$$

Proof Consider the finite network that consists of the first $R \geq 2$ layers from the previously constructed infinite forward-connected network. Let $z \in \mathbb{R}^K$ given as the initialization of the fixed point iteration in (7). With these, we have

$$z^{(l),R} = Wa^{(l-1),R} + b + \hat{x}^{(m)} \quad \text{and} \quad z^{(1),R} = z$$

or component wise, $z_j^{(l),R} = \sum_k w_{j,k} a_k^{(l-1),R} + b_j + \hat{x}_j^{(m)}$. Here, the letter R in the superscripts refers to the actual truncated finite network including R layers, where the gradient backpropagation is performed.

Using $(x^{(m)}, y^{(m)})$ as an input–output pair, the error on the R-th layer of this truncated network is given by

$$\mathcal{E}_R^{(m)}(z) = \frac{1}{2} \sum_{j=K-N+1}^{K} \left(a_j^{(R),R}(x^{(m)}) - \tilde{y}_j^{(m)} \right)^2,$$

where, we denote the z-dependence of the error. The partial derivative $d_i^{(l),R} = \frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial z_i^{(l),R}}$ is defined for the output neurons and will be extended to the non-output ones. In the R-th layer, according to the classical algorithm for gradient backpropagation, we have

$$d_j^{(R),R} = \begin{cases} \left(a_j^{(R),R} - \tilde{y}_j^{(m)} \right) f_j'(z_j^{(R),R}), & K-N < j \leq K \\ 0, & 1 \leq j \leq K-N \end{cases} \tag{10}$$

for the output $(K-N < j \leq K)$ and nonoutput $(1 \leq j \leq K-N)$ neurons, respectively.

For $1 \leq l < R$, correspondig to the gradient backpropagation algorithm [13], we have

$$d^{(l),R} = \frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial z^{(l+1),R}} \cdot \frac{\partial z^{(l+1),R}}{\partial z^{(l),R}} = D^{(l)} W^T d^{(l+1),R}. \tag{11}$$

For calculating $\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial b_j}$, we have to sum up the above vectors $d^{(l),R}$. This principle is similar to the one in backpropagation through time [14]. Thus, we get the following identity:

$$\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial b_j} = \sum_{k=0}^{R-1} \frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial z_j^{(R-k),R}} \frac{\partial z_j^{(R-k),R}}{\partial b_j} = \sum_{k=0}^{R-1} d_j^{(R-k),R}. \tag{12}$$

According to the identity in (11), we have

$$d^{(R-k),R} = \left(\prod_{l=0}^{k-1} D^{(R-l)} W^T \right) d^{(R),R}, \tag{13}$$

which can be inserted into (12) to get

$$\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial b_j} = \sum_{k=0}^{R-1} \left[\left(\prod_{l=0}^{k-1} D^{(R-l)} W^T \right) d^{(R),R} \right]_j. \tag{14}$$

Observe that this should be true also for the fixed point $z \in \mathbb{R}^K$ of the iteration in (7). Since in this case, the diagonal matrices $D^{(l)}$ $1 \leq l \leq R$ coincide, denoting their common value with D, Eq. (14) is simplified to

$$\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial b_j} = \sum_{k=0}^{R-1} \left[\left(DW^T \right)^k d^{(R),R} \right]_j. \tag{15}$$

Taking the limit $R \to \infty$ in Eq. (15) and using assumptions (i) and (ii), we get the equation

$$\begin{aligned} \frac{\partial \mathcal{E}^{(m)}(z)}{\partial b_j} &= \lim_{R\to\infty} \frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial b_j} \\ &= \lim_{R\to\infty} \sum_{k=0}^{R-1} \left[\left(DW^T \right)^k d^{(R),R} \right]_j = \left[\left(I - DW^T \right)^{-1} d^{(\infty)} \right]_j, \end{aligned} \tag{16}$$

, where the existence of the inverse follows from the assumption (iii).

We turn now to the statement for $\frac{\mathcal{E}(m,z)}{\partial w_{j,i}}$. Similarly to (12), we have the following equality for arbitrary $z \in \mathbb{R}^K$:

$$\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial w_{j,i}} = \sum_{k=0}^{R-2} \frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial z_j^{(R-k),R}} \frac{\partial z_j^{(R-k),R}}{\partial w_{j,i}} = \sum_{k=0}^{R-2} d_j^{(R-k),R} a_i^{(R-k-1),R}. \tag{17}$$

We can apply formula (13) again in (17) to get

$$\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial w_{j,i}} = \sum_{k=0}^{R-2} \left[\left(\prod_{l=0}^{k-1} D^{(R-l)} W^T \right) d^{(R),R} \right]_j a_i^{(R-k-1),R}. \tag{18}$$

We assume again, that $z \in \mathbb{R}^K$ is the limit in (7). Therefore, $D^{(l)} \equiv D$ holds $\forall \leq l \in \mathbb{N}$. With these, we can rewrite (18) as

$$\frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial w_{j,i}} = \sum_{k=0}^{R-2} \left[\left(DW^T \right)^k d^{(R),R} \right]_j a_i. \tag{19}$$

Performing here limit transition with respect to the number of the layers again, we finally get

$$\begin{aligned}\frac{\partial \mathcal{E}(m,z)}{\partial w_{j,i}} &= \lim_{R\to\infty} \frac{\partial \mathcal{E}_R^{(m)}(z)}{\partial w_{j,i}} \\ &= \lim_{R\to\infty} \sum_{k=0}^{R-2} \left[\left(DW^T\right)^k d^{(R),R} \right]_j a_i = \left[\left(I - DW^T\right)^{-1} d^{(\infty)} \right]_j a_i,\end{aligned} \tag{20}$$

which completes the proof of the theorem. □

The pseudocode of the gradient calculation can be found in Algorithm 2. Note, that here the product $\left(I - DW^T\right)^{-1} d^{(\infty)}$, is approximated using Neumann series.

Algorithm 2 Calculation of the gradient

procedure CALC_GRAD($y, a, f', \mathrm{T}, \mathrm{tol}, W, b, K, L, N$)
  error $\leftarrow \infty$; $d_i \leftarrow 0, dh_i \leftarrow 0, \quad i = 1, \ldots, K$ ▷ Initialization
  iter $\leftarrow 0$
  for $i = K - N + 1 : K$ **do** ▷ $d^{(\infty)}$
    $d_i = \left(a_i - \tilde{y}_i^{(m)}\right) f_i{}'$
  end for
  $dh \leftarrow d$
  while error $>$ tol and iter $<$ T **do** ▷ The fixed-point iteration
    iter $\leftarrow$ iter $+ 1$; $dh_{old} \leftarrow dh$; $dh_i \leftarrow 0$; $i = 1, \ldots, K$
    for $j = 1 : K$ **do** ▷ $d = \sum_{k=0}^{\infty} (DW^T)^k d^{(\infty)}$
      for $k = 1 : K$ **do**
        if $\exists j \to k$ edge **then**
          $dh_j \leftarrow dh_j + dh_{old,k} w_{k,j} f_j{}'$
        end if
      end for
    end for
    $d \leftarrow d + dh$; error $\leftarrow \|dh\|_\infty$
  end while
  for $i = 1 : K$ **do** ▷ Assembling of the gradients
    for $j = 1 : K$ **do**
      if $\exists i \to j$ edge **then**
        $\frac{\mathcal{E}^{(m)}}{\partial w_{j,i}} = d_j a_i$
      end if
    end for
    $\frac{\partial \mathcal{E}^{(m)}}{\partial b_i} = d_i$
  end for
  return $\frac{\partial \mathcal{E}^{(m)}}{\partial b_i}, \forall i = 1, \ldots, K$, and $\frac{\partial \mathcal{E}^{(m)}}{\partial w_{j,i}}$ $\forall i, j = 1, \ldots, K$, if $\exists i \to j$ edge in the network. ▷ Gradients
end procedure

3.1 Computational complexity

If T represents the maximum number of iterations as in Algorithm 2, and K denotes the number of neurons, it is easy to see that the computational complexity of both the network evaluation and gradient calculation is $O(\mathrm{T} \cdot K)$. In the case of a feedforward network, this is considerably more favorable since a calculation is performed exactly once on every neuron for both forward propagation and backpropagation, making the computational complexity for the feedforward case simply $O(K)$. This implies that the training duration for an implicit network could substantially exceed that of a feedforward network.

4 Numerical experiments

We will classify pulsars in the HTRU2 dataset [15] and network intrusions in the NSL-KDD dataset [16]. The methodology involves using autoencoders to compress and reconstruct the multidimensional data, facilitating the identification of normal and anomalous signals. Anomalies are distinguished from typical noise by leveraging the reconstruction error as a metric. This approach demonstrates the effectiveness of autoencoders in detecting patterns within complex datasets and their utility in astronomical data analysis and in the detection of network intrusions.

The structure of this section is as follows. Firstly, the datasets under investigation are described, including the data preprocessing. Then, the applied evaluation metrics and then the training approach based on [17] will be described. Finally, the results of the numerical simulations are shown and discussed.

4.1 The investigated datasets

4.1.1 The HTRU2 dataset

The HTRU2 dataset [15] is a collection of pulsar candidates collected during the high time resolution universe survey. It consists of 17,898 total samples, with 1639 real pulsar examples. These samples are described by eight continuous variables and a class label that distinguishes between the pulsar and nonpulsar candidates. Pulsars are a rare type of neutron stars that emit radiation that can be observed on Earth, making them of significant interest in astrophysics.

We partition the dataset into learning and testing sets. The learning set contains 90% of the original data. Then, the SMOTE algorithm [18] is applied to the learning set. That is, the learning set consists of 29,282 elements in a balanced ratio. This resampled set is further divided into training and validation sets for cross-validation using the above ratio.

Table 1 Sizes of the classes in the training, validation, and testing sets in the HTRU2 dataset

Sample type	Training	Validation	Testing
All	26,353	2929	1790
Non-pulsar	13,164	1477	1618
Pulsar	13,189	1452	172

Then, a simple standardization is used to normalize the data based only on non-pulsar samples. The exact sizes of the sets and the element numbers of the classes are shown in Table 1.

4.1.2 The NSL-KDD dataset

The NSL-KDD dataset [16] is widely used to detect network intrusions. This dataset was created to address certain deficiencies of the KDD Cup 99 dataset [19], such as redundant records in the training and testing sets, which could distort the evaluation of machine learning models. It includes data from normal traffic and various types of attacks.

In the preprocessing phase, numerical representations are derived from non-numerical data to construct inputs suitable for the autoencoder. The features *Protocol*, *Service*, and *Flag* are identified as categorical. The *Protocol* feature, which specifies a network protocol, can assume the values *tcp*, *udp*, or *icmp*. Through the application of one-hot encoding, this feature is transformed into a 3-dimensional vector. The *Service* feature, encompassing 70 distinct categories, is represented by a vector with 70 entries. Similarly, the *Flag* feature, with coded with 11 entries. Consequently, the input data are formulated as vectors of 117 dimensions by combining 33 numerical features with 84 one-hot-encoded categorical features.

Subsequently, the dataset is partitioned into sets for learning and testing. The learning set, comprising 84.83% of the total data, is further partitioned into training and validation subsets to facilitate cross-validation. This arrangement adopts a splitting ratio of 63.62/21.21/15.17% for the training, validation, and test sets. Then, a simple standardization is used to normalize all the features by calculating the two required scalars, the mean and the standard deviation of the training set only for benign samples. Table 2 shows the precise distribution of class elements.

Table 2 Sizes of the classes in the training, validation, and testing sets in the NSL-KDD dataset

Sample type	Training	Validation	Testing
All	94,479	31,494	22,544
Benign	50,528	16,815	12,833
Attack	43,951	14,679	9711

4.2 Evaluation metrics

The predictions made can vary for different values of the threshold c to belonging to the positive class. With this, it can be determined whether a prediction qualifies as true positive (TP_c), false positive (FP_c), true negative (TN_c) or false negative (FN_c). The following evaluation metrics are used based on these values.

- *Accuracy* (A_c) It is the ratio of all correctly identified instances, positive and negative classes, to the total number of instances in the dataset with a given classification threshold c. This quantity is given by the formula

$$A_c = \frac{\mathrm{TP}_c + \mathrm{TN}_c}{\mathrm{TP}_c + \mathrm{TN}_c + \mathrm{FP}_c + \mathrm{FN}_c}.$$

- *Precision* (P_c) It is defined as the ratio of correctly identified positive instances and the total number of instances classified as positive. This metric evaluates the accuracy of positive predictions with a given classification threshold c as follows:

$$P_c = \frac{\mathrm{TP}_c}{\mathrm{TP}_c + \mathrm{FP}_c}.$$

- *Recall* (R_c) *or True Positive Rate* (TPR_c) It quantifies the percentage of positive cases correctly identified with a given c classification threshold. It can be calculated using the following equation:

$$R_c = \mathrm{TPR}_c = \frac{\mathrm{TP}_c}{\mathrm{TP}_c + \mathrm{FN}_c}.$$

- *Matthew's Correlation Coefficient* (MCC_c) We use this as the primary metric to represent the best performance the model can achieve at a fixed threshold c. It is in the range between -1 and 1, where one indicates a perfect prediction and -1 means all predictions are false. In concrete terms, this score is given by

$$\mathrm{MCC}_c = \frac{\mathrm{TP}_c \cdot \mathrm{TN}_c - \mathrm{FP}_c \cdot \mathrm{FN}_c}{\sqrt{\left(\mathrm{TP}_c + \mathrm{FP}_c\right) \cdot \left(\mathrm{TP}_c + \mathrm{FN}_c\right) \cdot \left(\mathrm{TN}_c + \mathrm{FP}_c\right) \cdot \left(\mathrm{TN}_c + \mathrm{FN}_c\right)}}.$$

- *F1-score* ($\mathrm{F1}_c$) It is the harmonic mean of the precision and recall values, it can be calculated as follows.

$$\mathrm{F1}_c = 2 \cdot \frac{P_c \cdot R_c}{P_c + R_c}$$

4.3 Training approach

Here, the steps of the suggested training approach are described, slightly modified from those in article [17].

1. *Preprocessing* Initially, samples of the positive class are excluded from the training dataset. Feature values are scaled via standardization. Specific preprocessing techniques applied to the HTRU2 and NSL-KDD datasets are shown in detail in Sect. 4.1.
2. *Training* The autoencoder is designed to reconstruct inputs so that they closely resemble normal patterns, utilizing the L_2-norm for calculating reconstruction loss, as introduced in the formula (6). Inputs corresponding to anomalies, when processed, are expected to deviate significantly from their original form, facilitating classification based on the disparity between the input and its output. A threshold differentiates anomalies from normal instances. During training, the Stochastic Gradient optimizer and Cosine learning rate scheduler [20] are utilized, incorporating Nesterov momentum and a weight decay of 0.0001. The initial learning rate is set at 0.01, while the final learning rate is adjusted to 0.001. For the HTRU2 dataset, training is conducted over 20 epochs with a minibatch size of 16, whereas for the NSL-KDD dataset, training is conducted for 5 epochs with a minibatch size of 32.
3. *Model Selection* The F1-score is calculated at each epoch multiple times at a particular frequency. The highest F1-score's model weights are utilized upon training completion. It is calculated through the selection of the threshold, as shown in the following step.
4. *Threshold Selection* Selecting the threshold for reconstruction distance between input and output significantly impacts performance. The optimal threshold $\hat{c}$ is determined by evaluating the model on the validation set by maximizing the $\mathrm{F1}_c$ -score in c. Also, this maximal $\mathrm{F1}_c$ value is the F1-score mentioned in the previous step. This process involves standardizing reconstruction distances using mean and variance from negative samples in the validation set to determine the most effective threshold for class separation. This means the Z-score of the validation loss is calculated. The optimal threshold is sought within the $[-4, 4]$ interval with a division of 0.001. According to the properties of the standard normal distribution, 99.994% of the scaled errors falls within the $[-4, 4]$ interval. The Z-score that most effectively separates the anomalies from normal samples in the validation data is the threshold $\hat{c}$.
5. *Evaluation* Performance is assessed using test data, classifying samples based on their Z-score relative to the threshold. Samples identified as anomalies exceed the threshold. Predicted labels are compared to actual labels to compute the model's metrics.

4.4 The proposed implicit autoencoder models

Autoencoder networks for anomaly detection are operating by learning to replicate normal data input. They consist of three main components:

- *Encoder* It compresses the input into a latent-space representation. It learns the most important features of the data, effectively reducing its dimensionality.

- *Decoder* It reconstructs the input data from the latent space representation, aiming to compute output as close to the original input as possible.
- *Latent Layer* This is the core of the autoencoder: a compressed knowledge representation of the input data between the encoder and decoder.

The proposed autoencoder models are shown in Fig. 3. In case of $v1$ models, every neuron in the latent layer is interconnected in a directed manner, with no edges leading back to itself. The activation function chosen for the encoder, decoder, and latent layers is the arctan function. For $v2$ models, the latent layer consists of two blocks of neurons. The pointwise product of the two blocks constitutes the layer's output, which is directed towards the decoder. The input to the first block of the latent layer is sourced from the last encoder layer. The activation function for the first block and the encoders and decoders is the arctan function. In contrast, the activation function for the second block of the latent layer is the sigmoid function. Also, in $v2$ models, no edges lead from a neuron back to itself within the latent layer. With these choices, we examine whether adding extra weights to feedforward networks and, thus, getting implicit networks brings advantages by fixing the number of neurons. Here we made only the smallest, i.e., the latent layer implicit to keep computational costs at a relatively low level.

For a fair comparison, we consider six different feedforward autoencoder configurations, each with 8 and 16 neurons in the latent layer. We refer to these networks as the $v0$ model family. All the configurations are shown in Table 3. In the feedforward

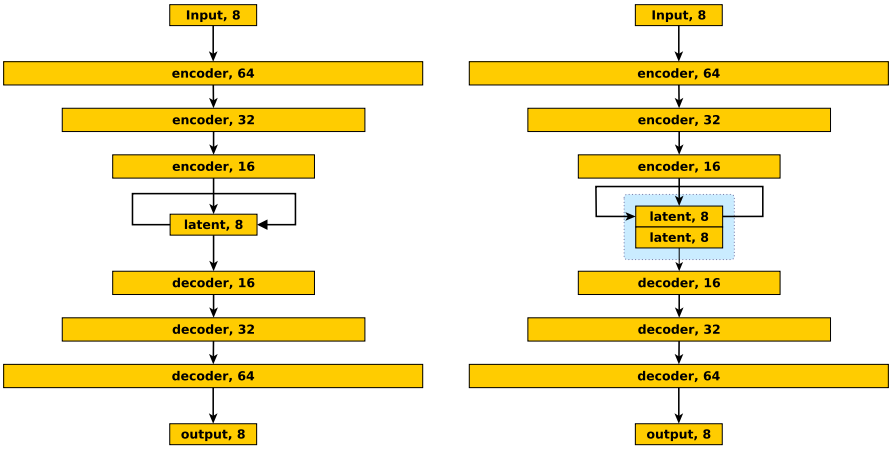

Fig. 3 The proposed implicit autoencoder models. Directed arrows are used to denote fully connected layers. In the left figure, a member of the $v1$ family is shown. The connection of the latent layer to itself indicates that each neuron in this layer is interconnected in a directed way. In the right figure, a member of the $v2$ family is shown. The latent layer consists of two blocks of neurons. The output of this layer is given by the pointwise product of the two blocks. The output of the layer is directed towards the decoder. The input to the first block of the latent layer comes from the last encoder layer. In both cases, no edges are drawn from neurons to themselves in the implicit latent layers. For each layer, it has been indicated to which part of the autoencoder it belongs and the number of neurons contained therein

Table 3 Configuration of the autoencoder models

Model name	Number of neurons for each layer	TP-H	TP-N
(5;32;8) – $v0$	$I-32-16-8-16-32-I$	1904	8989
(5;64;8) – $v0$	$I-64-32-8-32-64-I$	5840	19,901
(7;64;8) – $v0$	$I-64-32-16-8-16-32-64-I$	6640	20,701
(5;32;16) – $v0$	$I-32-16-16-16-32-I$	2168	9253
(5;64;16) – $v0$	$I-64-32-16-32-64-I$	6360	20,421
(7;64;16) – $v0$	$I-64-32-16-16-16-32-64-I$	6904	20,965
(5;32;8) – $v1$	$I-32-16-8-16-32-I$	1960	9045
(5;64;8) – $v1$	$I-64-32-8-32-64-I$	5896	19,957
(7;64;8) – $v1$	$I-64-32-16-8-16-32-64-I$	6696	20,757
(5;32;16) – $v1$	$I-32-16-16-16-32-I$	2408	9493
(5;64;16) – $v1$	$I-64-32-16-32-64-I$	6600	20,661
(7;64;16) – $v1$	$I-64-32-16-16-16-32-64-I$	7144	21,205
(5;32;8) – $v2$	$I-32-16-(8,8)-16-32-I$	1968	9053
(5;64;8) – $v2$	$I-64-32-(8,8)-32-64-I$	5904	19,965
(7;64;8) – $v2$	$I-64-32-16-(8,8)-16-32-64-I$	6704	20,765
(5;32;16) – $v2$	$I-32-16-(16,16)-16-32-I$	2424	9509
(5;64;16) – $v2$	$I-64-32-(16,16)-32-64-I$	6616	20,677
(7;64;16) – $v2$	$I-64-32-16-(16,16)-16-32-64-I$	7160	21,221

TP-H denotes the number of trainable parameters for the HTRU2 dataset, TP-N marks the same quantity for the NSL-KDD dataset, I denotes the number of the input features. The models labeled with $v0$ refer to feedforward networks, while labels $v1$ and $v2$ refer to implicit ones

networks, the ReLu activation function is applied in the encoder, decoder, and latent layers.

Also, for the purpose of ensuring fair comparisons, two classic models, the Random Forest and XGBoost models, are selected.

4.5 Experiment environment

With our implementation, all models were executed on a single NVIDIA Geforce GTX 1080 8GB GPU and a Xeon X5670 CPU, utilizing Python 3.8 and the CUDA C programming language with CUDA version 11.4. We set all possible random seeds during our numerical experiments for reproducibility purposes. To account for the variability of random processes, we repeat each experiment ten times and report the best and average scores in Sect. 5.

Table 4 Evaluation metrics among ten simulations for the HTRU2 dataset with the following abbreviations

Model name	BVF1	CTF1	CTMCC	CTA	CTP	CTR
(5;32;8) – $v0$	**0.9263**	**0.8468**	**0.8307**	**0.9693**	**0.8128**	0.8837
(5;64;8) – $v0$	**0.9368**	0.7959	0.7786	0.9553	0.7091	**0.9070**
(7;64;8) – $v0$	0.9154	0.7568	0.7377	0.9447	0.6553	**0.8953**
(5;32;16) – $v0$	**0.9263**	0.7666	0.7489	0.9469	0.6638	**0.9070**
(5;64;16) – $v0$	0.9253	**0.7990**	**0.7802**	**0.9570**	**0.7251**	0.8895
(7;64;16) – $v0$	0.9163	0.7463	0.7265	0.9419	0.6429	0.8895
(5;32;8) – $v1$	0.9244	0.7711	0.7528	0.9486	0.6739	0.9012
(5;64;8) – $v1$	0.9256	0.6781	0.6638	0.9162	0.5374	**0.9186**
(7;64;8) – $v1$	0.9279	0.7792	0.7622	0.9503	0.6797	**0.9128**
(5;32;16) – $v1$	0.9282	**0.8483**	**0.8322**	**0.9698**	**0.8207**	0.8779
(5;64;16) – $v1$	**0.9293**	**0.8115**	**0.7942**	**0.9598**	**0.7381**	0.9012
(7;64;16) – $v1$	**0.9301**	0.7393	0.7217	0.9385	0.6240	0.9070
(5;32;8) – $v2$	0.9287	0.7482	0.7305	0.9413	0.6367	0.9070
(5;64;8) – $v2$	**0.9288**	**0.7949**	**0.7782**	**0.9547**	**0.7040**	**0.9128**
(7;64;8) – $v2$	0.9226	**0.8257**	**0.8087**	**0.9637**	**0.7662**	0.8953
(5;32;16) – $v2$	**0.9304**	0.7859	0.7684	0.9525	0.6933	0.9070
(5;64;16) – $v2$	**0.9304**	0.7378	0.7227	0.9369	0.6139	**0.9244**
(7;64;16) – $v2$	0.9253	0.7185	0.7021	0.9313	0.5925	**0.9128**
XGBoost	0.9787	0.8674	0.8538	0.9732	0.9085	0.9128
Random Forest	0.9773	0.8674	0.8538	0.9732	0.9085	0.9128

BVF1 the best validation F1-score, *CTF1* test F1-score, *CTMCC* test MCC score, *CTA* test accuracy, *CTP* test precision and *CTR* test recall

In each column, the two largest values are underlined. Additionally, in each column, for each model family, the two largest values are displayed in bold

5 Numerical results

5.1 The HTRU2 dataset

We summarize the results in Table 4. The two largest values for each model family are displayed in bold in each column. Also, the two largest values in each column are underlined. The implicit model, identified as (5;32;16) – $v1$, dominates four out of five test metrics. From the perspective of the dataset under consideration, the most crucial test metric is recall, for which the (5;64;16) – $v2$ model performs the best, achieving the highest score of 0.9244. Recall is crucial for the HTRU2 dataset, because minimizing false negative predictions is essential for pulsar detection. Nevertheless, the XGBoost and Random Forest models exhibit marginally higher performance in the other metrics. The confusion matrices created on the test set by the implicit (5;64;8) – $v1$ and (5;64;16) – $v2$ models can be seen in Fig. 4.

We have also studied the stability of the methods by computing the average test metrics over ten simulations. This can confirm the computations' reliability

Table 5 Average evaluation metrics among ten simulations for the HTRU2 dataset

Model name	AVF1	ATF1	ATMCC	ATA	ATP	ATR
(5;32;8) – $v0$	0.9162	0.7577	0.7396	0.9441	0.6603	0.8959
(5;64;8) – $v0$	**0.9207**	0.7600	0.7416	0.9450	0.6614	**0.8971**
(7;64;8) – $v0$	0.9113	0.7161	0.6973	0.9312	0.6002	0.8930
(5;32;16) – $v0$	0.9163	**0.7650**	**0.7458**	**0.9472**	**0.6694**	0.8930
(5;64;16) – $v0$	**0.9213**	**0.7676**	**0.7497**	**0.9470**	**0.6731**	**0.8988**
(7;64;16) – $v0$	0.9113	0.7341	0.7147	0.9375	0.6276	0.8890
(5;32;8) – $v1$	0.9204	**0.7513**	**0.7334**	**0.9420**	**0.6454**	**0.9023**
(5;64;8) – $v1$	**0.9228**	0.7430	0.7261	0.9387	0.6338	**0.9058**
(7;64;8) – $v1$	0.9198	0.7392	0.7212	0.9382	0.6304	0.8994
(5;32;16) – $v1$	0.9207	0.7317	0.7140	0.9355	0.6219	0.8983
(5;64;16) – $v1$	**0.9233**	**0.7477**	**0.7296**	**0.9409**	**0.6410**	0.9012
(7;64;16) – $v1$	0.9203	0.7471	0.7294	0.9406	0.6401	**0.9023**
(5;32;8) – $v2$	0.9212	0.7459	0.7278	0.9404	0.6391	0.9000
(5;64;8) – $v2$	**0.9225**	0.7628	0.7446	0.9458	0.6651	0.8983
(7;64;8) – $v2$	0.9188	**0.7694**	**0.7517**	**0.9474**	**0.6781**	0.8971
(5;32;16) – $v2$	0.9214	0.7653	0.7474	0.9462	0.6708	0.8977
(5;64;16) – $v2$	**0.9221**	0.7398	0.7218	0.9387	0.6290	**0.9012**
(7;64;16) – $v2$	0.9201	**0.7707**	**0.7529**	**0.9479**	**0.6761**	**0.9006**
XGBoost	0.9732	0.8643	0.8506	0.9725	0.9063	0.9116
Random Forest	0.9711	0.8621	0.8482	0.9796	0.9039	0.9122

AVF1 denotes the average validation F1-score, ATF1 marks the average test F1-score, ATMCC is the average test MCC score, ATA denotes the average test accuracy, ATP means average test precision and ATR is the average test recall

In each column, the two largest values are underlined. Additionally, in each column, for each model family, the two largest values are displayed in bold

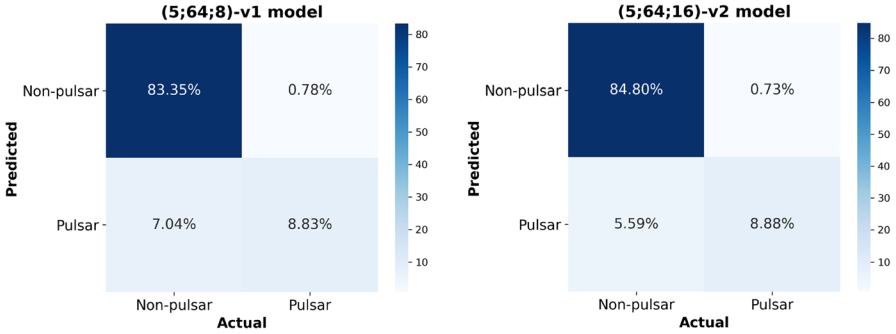

Fig. 4 The confusion matrices generated with the best (5;64;8) – $v1$ and the (5;64;16) – $v2$ implicit models on the testing set for the HTRU2 dataset

in Table 4. The results are shown in Table 5. The results are observed to be very close to one another. The implicit $v1$ family dominates the CTR metric within the

Table 6 Evaluation metrics among 10 simulations for the NSL-KDD dataset with the following abbreviations

Model name	BVF1	CTF1	CTMCC	CTA	CTP	CTR
(5;32;8) – *v*0	**0.9380**	**0.9033**	**0.7798**	**0.8914**	0.9156	**0.8914**
(5;64;8) – *v*0	0.9369	0.8682	0.7459	0.8636	**0.9654**	0.7887
(7;64;8) – *v*0	0.9367	0.8811	0.7652	0.8755	**0.9654**	0.8103
(5;32;16) – *v*0	0.9379	0.8641	0.7399	0.8599	**0.9652**	0.7821
(5;64;16) – *v*0	**0.9402**	**0.9118**	**0.7944**	**0.8993**	0.9091	**0.9144**
(7;64;16) – *v*0	0.9359	0.8739	0.7543	0.8689	**0.9652**	0.7985
(5;32;8) – *v*1	**0.9614**	0.9090	0.7926	0.8977	0.9210	0.8973
(5;64;8) – *v*1	**0.9585**	0.9053	0.7848	0.8937	0.9190	0.8919
(7;64;8) – *v*1	0.9555	0.9185	**0.8121**	**0.9077**	**0.9230**	0.9141
(5;32;16) – *v*1	0.9584	0.9170	0.8098	0.9064	**0.9252**	0.9090
(5;64;16) – *v*1	0.9567	**0.9287**	**0.8323**	**0.9179**	0.9188	**0.9388**
(7;64;16) – *v*1	0.9566	**0.9187**	**0.8121**	**0.9077**	0.9214	**0.9161**
(5;32;8) – *v*2	0.9602	**0.9280**	**0.8311**	**0.9173**	0.9198	**0.9364**
(5;64;8) – *v*2	**0.9645**	0.8912	0.7613	0.8804	0.9245	0.8601
(7;64;8) – *v*2	0.9576	0.8908	0.7605	0.8800	0.9241	0.8598
(5;32;16) – *v*2	0.9598	0.9203	0.8168	0.9099	**0.9271**	0.9136
(5;64;16) – *v*2	**0.9619**	**0.9271**	**0.8310**	**0.9171**	**0.9278**	0.9264
(7;64;16) – *v*2	0.9595	0.9258	0.8275	0.9154	0.9247	**0.9269**
XGBoost	0.9998	0.8227	0.6885	0.8250	0.8456	0.7134
Random Forest	0.9996	0.7870	0.6453	0.7960	0.8278	0.6619

BVF1 the best validation F1-score, *CTF1* test F1-score, *CTMCC* test MCC score, *CTA* test accuracy, *CTP* test precision and *CTR* test recall

In each column, the two largest values are underlined. Additionally, in each column, for each model family, the two largest values are displayed in bold

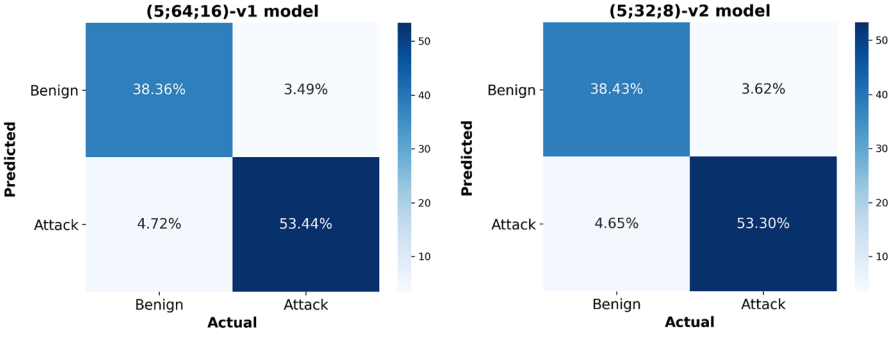

Fig. 5 The confusion matrices generated with the best (5;64;16) – *v*1 and the (5;32;8) – *v*2 implicit models on the testing set for the NSL-KDD dataset

autoencoders, while the $v2$ family excels in the other test metrics. Overall, however, the XGBoost and Random Forest models yield the most consistent values.

5.2 The NSL-KDD dataset

The results are shown in Table 6. The two largest values for each model family are printed in bold in each column. Also, the two largest values in each column are underlined. The (5;64;16) − $v1$ model is observed to dominate in four out of five test metrics, including the CTR metric, which is also this dataset's most crucial test metric. The performance of the best $v2$ model, namely the (5;32;8) − $v2$, is observed to fall behind only a little from this. The XGBoost and Random Forest models are observed to perform substantially weaker than the autoencoders here. The confusion matrices created on the test set by the implicit (5;64;8) − $v1$ and (5;64;16) − $v2$ models can be shown in Fig. 5.

In Table 7, the stability of the models is investigated. Here, the (5;64;16) − $v1$ and (5;32;8) − $v2$ implicit autoencoder are observed to dominate, too.

Table 7 Average evaluation metrics among ten simulations for the NSL-KDD dataset

Model name	AVF1	ATF1	ATMCC	ATA	ATP	ATR
(5;32;8) − $v0$	**0.9332**	0.8934	0.7692	0.8834	0.9299	0.8610
(5;64;8) − $v0$	0.9033	0.8762	0.6952	0.8524	0.8980	0.8720
(7;64;8) − $v0$	0.9315	0.8872	0.7642	0.8788	**0.9408**	0.8410
(5;32;16) − $v0$	**0.9321**	**0.8993**	**0.7786**	**0.8888**	0.9238	**0.8782**
(5;64;16) − $v0$	0.9294	**0.8999**	**0.7764**	**0.8887**	0.9196	**0.8820**
(7;64;16) − $v0$	0.9315	0.8902	0.7677	0.8813	**0.9381**	0.8485
(5;32;8) − $v1$	0.9486	0.9120	0.7995	0.9008	0.9184	0.9066
(5;64;8) − $v1$	0.9508	**0.9224**	**0.8193**	**0.9114**	0.9185	**0.9266**
(7;64;8) − $v1$	0.9489	0.9085	0.7928	0.8974	**0.9200**	0.8979
(5;32;16) − $v1$	**0.9511**	**0.9218**	**0.8182**	**0.9108**	0.9191	**0.9248**
(5;64;16) − $v1$	**0.9521**	0.9176	0.8124	0.9071	**0.9203**	0.9164
(7;64;16) − $v1$	0.9470	0.9161	0.8065	0.9049	0.9187	0.9138
(5;32;8) − $v2$	0.9537	0.9171	0.8096	0.9062	0.9204	0.9143
(5;64;8) − $v2$	**0.9566**	0.9171	0.8090	0.9060	0.9198	0.9147
(7;64;8) − $v2$	0.9515	0.9145	0.8054	0.9037	0.9197	0.9104
(5;32;16) − $v2$	0.9536	**0.9240**	**0.8229**	**0.9132**	0.9208	**0.9273**
(5;64;16) − $v2$	**0.9582**	0.9218	0.8186	0.9110	**0.9217**	0.9220
(7;64;16) − $v2$	0.9551	**0.9236**	**0.8231**	**0.9130**	**0.9222**	**0.9254**
XGBoost	0.9996	0.8320	0.7010	0.8330	0.8510	0.7277
Random Forest	0.9991	0.7701	0.6257	0.7829	0.8199	0.6392

AVF1 denotes the average validation F1-score, ATF1 marks the average test F1-score, ATMCC is the average test MCC score, ATA denotes the average test accuracy, ATP means average test precision and ATR is the average test recall

In each column, the two largest values are underlined. Additionally, in each column, for each model family, the two largest values are displayed in bold

Table 8 Training time TT inference time IT in various models

Model name	HTRU2 dataset		NSL-KDD dataset	
	TT [s]	IT [s]	TT [s]	IT [s]
(5;32;8) – $v0$	28.77	0.17	82.81	2.93
(5;64;8) – $v0$	39.59	0.18	82.40	2.93
(7;64;8) – $v0$	42.13	0.19	78.10	3.00
(5;32;16) – $v0$	34.55	0.18	61.79	2.90
(5;64;16) – $v0$	39.98	0.18	29.08	2.98
(7;64;16) – $v0$	42.24	0.19	78.86	2.98
(5;32;8) – $v1$	149.06	0.65	235.50	7.10
(5;64;8) – $v1$	148.34	0.77	207.95	7.60
(7;64;8) – $v1$	210.14	0.70	267.44	8.39
(5;32;16) – $v1$	159.44	0.65	259.46	9.15
(5;64;16) – $v1$	232.14	0.79	153.71	9.83
(7;64;16) – $v1$	229.34	0.67	89.62	9.82
(5;32;8) – $v2$	112.83	0.40	268.32	5.91
(5;64;8) – $v2$	153.01	0.53	276.47	6.10
(7;64;8) – $v2$	134.90	0.49	311.50	6.69
(5;32;16) – $v2$	101.21	0.41	256.42	5.99
(5;64;16) – $v2$	136.19	0.50	291.86	6.67
(7;64;16) – $v2$	134.83	0.47	317.77	6.82
XGBoost	133.54	0.03	273.00	0.17
Random Forest	187.02	0.37	320.45	2.65

Unfortunately, a significant cost is associated with the implicit models. This can be shown in Table 8. The computational time is about four times as much as their feedforward variants.

The comparison with the XGBoost and Random Forest models in terms of computational time, this is only partially fair, as these were executed without the use of GPUs.

6 Future work

Expanding and deepening our numerical simulations is a priority in the future work, especially in scenarios where current frameworks like PyTorch [21] or TensorFlow [22] encounter limitations. Furthermore, we aim to understand better the impact of activation functions and the sparsity of computational graphs representing network architectures on the speed and efficiency of convergence. Additionally, accelerating computations is a crucial issue, for which we would like to develop further the algorithm proposed in the article [6].

7 Conclusion

In this work, we have shown an illustrative approach to constructing deep equilibrium models, also called implicit neural networks, highlighting that these networks are given by such a computational graph that may even include a directed cycle. We have proved a theorem for calculating the gradient in such a network, enabling the computation of gradients without resorting to the implicit function theorem, but by directly calculating them in an infinitely deep feedforward network associated with the computational graph. Furthermore, numerical experiments confirmed our findings, providing empirical evidence to support the theoretical results. This work lays a foundation for further exploration into the capabilities and applications of Implicit Neural Networks, marrying theoretical insights with practical validation.

Author Contributions The authors contributed equally to this work.

Funding Open access funding provided by Eötvös Loránd University. Béla J. Szekeres was supported by the Project No. 2019-1.3.1-KK-2019-00011 financed by the National Research, Development and Innovation Fund of Hungary under the Establishment of Competence Centers, Development of Research Infrastructure Programme funding scheme. Ferenc Izsák was supported by the National Research, Development and Innovation Office within the framework of the Thematic Excellence Program 2021 - National Research Sub programme: "Artificial intelligence, large networks, data security: mathematical foundation and applications"

Data availability The implemented codes and the dataset used in this article are available on the GitHub repository https://github.com/szbela87/imp_autoencoder. On this page, one can also find the documentation of the developed CUDA C code.

Declarations

Conflict of interest The authors declare no Conflict of interest. We certify that the submission is original work and is not under review at any other publication.

References

1. El Ghaoui L, Gu F, Travacca B, Askari A, Tsai A (2021) Implicit deep learning. SIAM J Math Data Sci 3(3):930–958. https://doi.org/10.1137/20M1358517
2. Bianchini M, Gori M, Sarti L, Scarselli F (2006) Recursive neural networks and graphs: dealing with cycles. In: Apolloni B, Marinaro M, Nicosia G, Tagliaferri R (eds) Neural nets. Springer, Berlin, Heidelberg, pp 38–43

3. Bianchini M, Gori M, Scarselli F (2002) Recursive processing of cyclic graphs. In: Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290), vol 1, pp 154–1591. https://doi.org/10.1109/IJCNN.2002.1005461
4. Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2009) The graph neural network model. IEEE Trans Neural Netw 20(1):61–80. https://doi.org/10.1109/TNN.2008.2005605
5. Kawaguchi K (2021) On the theory of implicit deep learning: global convergence with implicit layers. In: International Conference on Learning Representations. https://openreview.net/forum?id=p-NZIuwqhI4
6. Bai S, Kolter JZ, Koltun V (2019) Deep equilibrium models. In: Wallach H, Larochelle H, Beygelzimer A, d' alché-Buc F, Fox E, Garnett R (eds) Advances in neural information processing systems, vol 32. Curran Associates, Inc., Vancouver, Canada
7. Bai S, Koltun V, Kolter JZ (2020) Multiscale deep equilibrium models, vol 33. Vancouver, Canada, pp 5239–5250
8. Gu F, Chang H, Zhu W, Sojoudi S, El Ghaoui L (2020) Implicit graph neural networks. In: Larochelle H, Ranzato M, Hadsell R, Balcan MF, Lin H (eds) Advances in neural information processing systems, vol 33. Curran Associates Inc., Vancouver, Canada, pp 11984–11995
9. Steven G, Krantz HRP (2013) The implicit function theorem, 1st edn. Modern Birkhäuser classics. Birkhäuser, Boston. https://doi.org/10.1007/978-1-4614-5981-1
10. Winston E, Kolter JZ (2020) Monotone operator equilibrium networks. In: Larochelle H, Ranzato M, Hadsell R, Balcan MF, Lin H (eds) Advances in neural information processing systems, vol 33. Curran Associates Inc., Vancouver, Canada, pp 10718–10728
11. Geng Z, Zhang X-Y, Bai S, Wang Y, Lin Z (2021) On training implicit models. In: Ranzato M, Beygelzimer A, Dauphin Y, Liang PS, Vaughan JW (eds) Advances in neural information processing systems, vol 34. Curran Associates Inc., Vancouver, Canada, pp 24247–24260
12. Bondy JA, Murty USR (2008) Graph theory, 1st edn. Springer, New York
13. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323(6088):533–536. https://doi.org/10.1038/323533a0
14. Werbos PJ (1990) Backpropagation through time: what it does and how to do it. Proc IEEE 78(10):1550–1560. https://doi.org/10.1109/5.58337
15. Lyon R (2017) HTRU2. UCI machine learning repository. https://doi.org/10.24432/C5DK6R
16. Tavallaee M, Bagheri E, Lu W, Ghorbani AA (2009) A detailed analysis of the KDD CUP 99 data set. In: 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications, pp 1–6. https://doi.org/10.1109/CISDA.2009.5356528
17. Song Y, Hyun S, Cheong Y-G (2021) Analysis of autoencoders for network intrusion detection. Sensors 21(13):4294. https://doi.org/10.3390/s21134294
18. Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: synthetic minority over-sampling technique. J Artif Int Res 16(1):321–357
19. Stolfo S, Fan W, Lee W, Prodromidis A, Chan P (1999) KDD Cup 1999 data. UCI machine learning repository. https://doi.org/10.24432/C51C7N
20. Loshchilov, I, Hutter F (2017) SGDR: stochastic gradient descent with warm restarts
21. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: an imperative style, high-performance deep learning library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems, Curran Associates Inc, Red Hook, NY, USA, pp 8026–8037
22. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. https://www.tensorflow.org/