Eötvös Loránd University

Faculty of Science

Department of Applied Analysis and Computational Mathematics

Implicit neural networks: theory and practice

Mathematics Expert in Data Analytics and Machine Learning
postgraduate specialization program

Supervisor:
Ferenc Izsák, PhD
Associate Professor

Author:
Béla J. Szekeres, PhD

Budapest, 2022

Contents

Chapter 1

Introduction

In the last years, neural networks became the most popular tool in a wide range of scientific computing. The usefulness of this approach was demonstrated for problems where large measurement, simulation or observation data are available. Accordingly, a number of neural networks were constructed to assist various learning algorithms, see the review paper in [4]. The vast majority of these networks are feedforward in the sense that the neurons or perceptrons are organized into consecutive layers. This approach has a solid background. First, it was established that a corresponding composition of elementary functions can approximate a wide class of output functions. In concrete terms, the approximation properties can depend on the depth and size of the layers in the neural network. Also, the so-called activation function, which is responsible for the non-linearity, has a crucial role. For the details, we refer to the classical result in [6] and the tedious review papers in [5] and [3], where further details and references can be found. Second, an entire armory of programming tools were developed to implement such neural networks supported with efficient computational tools, which compute gradients of the corresponding functions applying backpropagation algorithms. For the description of the original approach, we refer to [15] and a for a detailed overview to [8].

Whenever the corresponding computational procedures result in a very powerful computational tool, we should keep in mind that originally, real-life biological systems served as the motivation of such networks. These, in many cases, contain a complex structure, which cannot be given with a feedforward structure. For example, in a human brain, a very complex network of neurons can be found with a billion of links between and inside of some important areas. A corresponding computational model is given by the graph neural networks, where a number of loops can emerge, see the review paper [7] for further literature. At the same time, there is no straightforward computing algorithm for these structures. One can obviously define inputs and outputs but the feedback in the loops can modify step-by-step the intermediate value at each node. This may affect also at the output. Even if this value is stabilized and will be well-defined, it is not obvious how to evaluate it and based on this, how to use backpropagation to train and finally use such a network. In a network or a graph representation of any complex system, cycles or loops correspond to feedback, which is an important cornerstone of their operation. Such systems can be analyzed in the framework of the implicit neural networks since the output of the nodes during the training procedure cannot be given explicitly. At the same time, in an implicit representation, it is by far not clear whether such a form is well-posed, or practically, how we can evaluate nodal values in a given graph neural network. In [9, 10], this problem was investigated depending on the matrices of parameters and special choice of activation functions. This question was also investigated earlier, see, e.g. [16, 17, 18]. Here we should also mention Boltzmann machines, which are also special implicit neural networks, see [2].

In this contribution, we raise the question of training such neural networks. How can we compute the gradient of the loss function if we know only its well-posedness? How should we modify the standard backpropagation algorithm? We will answer these questions, and show real-life examples, where after implementing the corresponding procedure, our results can be applied.

The structure of the thesis is the following. After the short literature review in this chapter, we investigate to implicit artificial neural networks in the second chapter. A manuscript of the results presented here has been submitted to the journal Neurocomputing with my supervisor and is currently under review [1]. In the third chapter, we look at a more general implicit structure, where a neuron can have multiple inputs. Here, after summing the signals at the inputs and activating them separately, the neuron activation value is the product of them. Note that convolutional networks are special cases of this architecture, and are also building blocks of LSTM networks.

In both chapters, our strategy is the same. We first describe formally the problem to be solved. We then assign to the problem a finite implicit graph or hypergraph neural network, which may contain cycles. We assign to this network an infinitely deep feedforward neural network in which we apply the gradient backpropagation. In this way, we introduce implicit neural networks in a very illustrative way. A limit transition makes then possible to compute the desired derivatives with respect to the parameters of the neural network. Finally, the usefulness of the presented computational technique will be demonstrated and analyzed in real-life examples.

Chapter 4 is also an important part of the thesis. This chapter contains the user documentation or user manual for the program that to perform the numerical experiments described in Chapters 2 and 3. In this way, the thesis ends where the work began, because my original aim was to create a program to simulate neural networks in which the (hyper)graphs are given by the user in a very flexible way. The theoretical results of the thesis were inspired during this programming process.

Chapter 2

Implicit neural networks

In this chapter, we generalize the evaluation of neural networks and the gradient backpropagation based learning to neural networks represented by graphs containing even directed cycles. Our theoretical results on the computation of the gradient are also supported by numerical experiments.

2.1 Construction of the network

In general, a neural network is represented as a directed graph [19], where each edge has a certain weight. This corresponds to the strength of a synapse along this edge in the original model. A network is called feedforward or acyclic if the corresponding graph is acyclic. Similarly, cyclic directed graphs correspond to the so-called cyclic or implicit neural networks. The total number of vertices (which are called also the

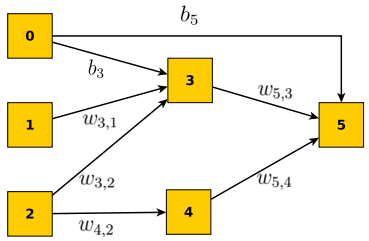

Figure 2.1: Neural network with two input neurons (index 1 and 2) and one output neuron (index 5).

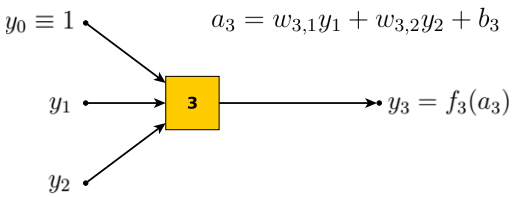

Figure 2.2: Calculating the output y_3 of a neuron with index 3. The neuron with index 0 corresponds to the bias parameter $y_0 \equiv 1$.

neurons) is K. Let y_j denote the activation value of the j-th neuron and $f_j : \mathbb{R} \to \mathbb{R}$ be the corresponding activation function. Common examples are, e.g., $f_j(a) = \tanh(a)$ or $f_j(a) = a$. Evaluation of the network for an input vector $x \in \mathbb{R}^L$ is as follows. Let initially $y_j = 0$, $\forall j = 1, \dots, K$. If a neuron with index j receives a stimulus of magnitude y_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, the cumulated input a_j of this neuron is

$$a_j = \sum_l w_{j,l} y_l + b_j + \hat{x}_j. \tag{2.1}$$

Here, $\hat{x}$ denotes the mapping of the vector x to the input of the neurons using the operator $\hat{:}\mathbb{R}^L \to \mathbb{R}^K$. Accordingly, the activation value of the neuron with index j is

$$y_j = f_j(a_j). \tag{2.2}$$

Assume for simplicity that the indexing of the neurons is such that the last N neurons are the output ones. Simple examples are shown in Figures 2.1 and 2.2, respectively.

2.2 Problem statement

Indeed, the formulas in (2.1)-(2.2) define the following system of K equations:

$$\begin{cases} a(x) = Wy(x) + \hat{x} + b \\ y(x) = f(a(x)). \end{cases} \tag{2.3}$$

Here, summarized, $a(x), y(x), b \in \mathbb{R}^K$, $W \in \mathbb{R}^{K \times K}$, $\hat{\cdot} : \mathbb{R}^L \to \mathbb{R}^K$ and $x \in \mathbb{R}^L$. Also, the functions $f_j : \mathbb{R} \to \mathbb{R}$ are given for $j = 1, \ldots, K$, which are used to define the vector $f(a) \in \mathbb{R}^K$ the way $f(a) = (f_1(a_1), \ldots, f_K(a_K))^T$, when $a = (a_1, \ldots, a_K)^T$. Sometimes, we simplify the notation and omit the x-dependence of the terms in (2.3).

We also have M pairs of training samples (x, t), where $x \in \mathbb{R}^L$ are input and $t \in \mathbb{R}^N$ are target vectors. At the m-th pair of samples, i.e., at the input $x^{(m)}$, the error is defined as

$$\mathcal{E}(m) = \frac{1}{2} \sum_{j=K-N+1}^{K} \left(\tilde{t}_j^{(m)} - y_j(x^{(m)})\right)^2, \tag{2.4}$$

where $\tilde{t}_j^{(m)}$ denotes the corresponding component of the m-th training sample $t^{(m)} = (t_1^{(m)}, \ldots, t_N^{(m)}) \in \mathbb{R}^N$ of the target vector to be compared with the value of $y_j(x^{(m)})$. That is, let the operator $\tilde{\cdot} : \mathbb{R}^N \to \mathbb{R}^K$ defined by the formula $\tilde{t} = (0, \ldots, 0, t_1, \ldots, t_N)^T$. The average error $\mathcal{E}$ over all pairs of training samples is their average over the entire data set, i.e.,

$$\mathcal{E} = \frac{1}{M} \sum_{m=1}^{M} \mathcal{E}(m). \tag{2.5}$$

We investigate the task to determine W and b such that the error given by (2.5) is minimal.

Solving equation (2.3) by a fixed-point iteration yields the vector $a = (a_1, \ldots, a_K)^T$ of neuron input values and the vector $y = (y_1, \ldots, y_K)^T$ of activation values. Figure 2.3 shows a sample computation of the activation value of a neuron in a cyclic network.

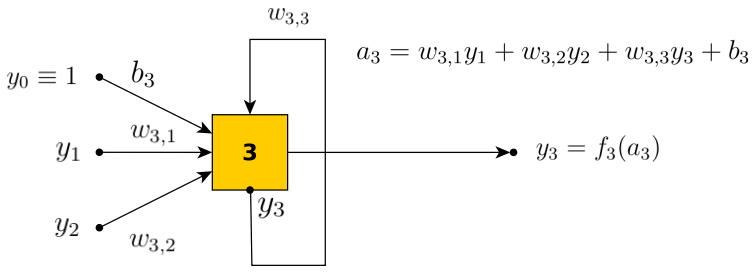

Figure 2.3: Example of computing the value of a neuron in a cyclic neural network. The neuron of index 3 has three conventional inputs, plus a loop edge leading back to itself and the output of the neuron is also its input.

A single step in the fixed point iteration for solving (2.3) has the form

$$a^{(l)} = Wy^{(l-1)} + b + \hat{x}, \quad y^{(l)} = f(a^{(l)}), \quad l \geq 2 \quad \text{and } a^{(1)} = \hat{x} \in \mathbb{R}^K. \tag{2.6}$$

An important observation is that the iteration in (2.6) delivers a feedforward neural network of infinite number of layers with K neurons in each layer. In this framework, the fixed point iteration can be interpreted as 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 $b \in \mathbb{R}^K$ is the bias vector. Such an interpretation is shown in Figure 2.4, which is the unrolling of the small network shown in Figure 2.3 focusing only on the 3rd neuron. The algorithm for computing the network is given in Pseudocode 1 in the Appendix.

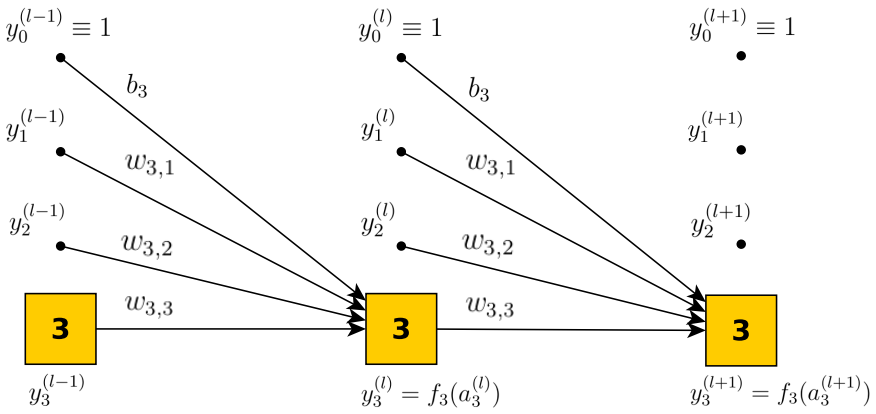

Figure 2.4: Unrolling of the cyclic neural network shown in Figure 2.3 focusing on the 3rd neuron in the $l-1$-th, l-th and $l+1$-th layers, $a_3^{(l)} = w_{3,1}y_1^{(l-1)} + w_{3,2}y_2^{(l-1)} + w_{3,3}y_3^{(l-1)} + b_3$, and $a_3^{(l+1)} = w_{3,1}y_1^{(l)} + w_{3,2}y_2^{(l)} + w_{3,3}y_3^{(l)} + b_3$.

2.3 Further notations

Summarized, we use the following notations in the infinite network:

- The initial vector of the iteration is $a^{(1)} = \hat{x} \in \mathbb{R}^K$. Let $a_i^{(l)}(x)$ denote the input value of the i-th neuron in the l-th layer and use $a_i(x) = \lim\limits_{l\to\infty} a_i^{(l)}(x)$ provided that this exists and is finite. In vector form, we have $a^{(l)}(x) = \left(a_1^{(l)}(x), \ldots, a_K^{(l)}(x)\right)^T$ and $a(x) = (a_1(x), \ldots, a_K(x))^T$. Sometimes, for simplicity, we omit the arguments x.

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

$$f(a^{(l)})(x) = y^{(l)}(x) = \left(y_1^{(l)}(x), \ldots, y_K^{(l)}(x)\right)^T$$

  and $y(x) = (y_1(x), \ldots, y_K(x))^T$.

- Parallel with the formula (2.4), we also introduce $d^{(\infty)} \in \mathbb{R}^K$ with

$$d_j^{(\infty)} = \begin{cases} \left(y_j - \tilde{t}_j^{(m)}\right) f_j{}'(a_j), & K-N < j \le K \\ 0, & 1 \le j \le K-N. \end{cases}$$

- We use the notation

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

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

2.4 Theoretical results

As discussed previously, we transform the original cyclic network into an infinitely deep feedforward one. We apply the gradient backpropagation learning method to this network.

To minimize the error function (2.5) using some gradient-based method, we need to determine the partial derivatives $\frac{\partial \mathcal{E}(m)}{\partial w_{j,i}}$ and $\frac{\partial \mathcal{E}(m)}{\partial b_j}$. In the following Theorem, we express these in concrete terms. We make use the method in [15] 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.

(i) Equation (2.3) has a unique solution and the iteration in (2.6) is convergent such that we also have

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

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

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

(iii) The linear mapping DW^T is a contraction in some matrix norm, i.e. $\|DW^T\| < 1$.

Theorem 1. *With the assumptions (i)-(iii), the system of equations*

$$d = \left(I - DW^T\right)^{-1} d^{(\infty)} \tag{2.7}$$

has a unique solution. Furthermore, the partial derivatives of the error function can be given as

$$\frac{\mathcal{E}(m)}{\partial b_j} = d_j \quad \textit{and} \quad \frac{\mathcal{E}(m)}{\partial w_{j,i}} = y_i d_j. \tag{2.8}$$

Proof. Consider the first $R \geq 2$ layers of the infinite forward-connected network constructed previously. Let $a \in \mathbb{R}^K$ given as the initialization of the fixed point iteration in (2.6). With these, we have

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

or componentwise, $a_j^{(l),R} = \sum_k w_{j,k} y_k^{(l-1),R} + b_j + \hat{x}_j^{(m)}$.

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

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

where we denote the a-dependence of the error. We perform the gradient backpropagation on this truncated network. The partial derivative $d_i^{(l),R} = \frac{\partial \mathcal{E}_R(m,a)}{\partial a_i^{(l),R}}$ 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

$$d_j^{(R),R} = \begin{cases} \left(y_j^{(R),R} - \tilde{t}_j^{(m)}\right) f_j{}'(a_j^{(R),R}), & K - N < j \leq K \\ 0, \quad 1 \leq j \leq K - N. \end{cases} \tag{2.9}$$

for the output $(K - N < j \leq K)$ and non-output $(1 \leq j \leq K - N)$ neurons, respectively.
For $1 \leq l < R$, correspondig to the gradient backpropagation algorithm, we have

$$d^{(l),R} = \frac{\partial \mathcal{E}_R(m, a)}{\partial a^{(l+1),R}} \cdot \frac{\partial a^{(l+1),R}}{\partial a^{(l),R}} = D^{(l)} W^T d^{(l+1),R}. \tag{2.10}$$

For calculating $\frac{\partial \mathcal{E}_R(m,a)}{\partial b_j}$, we have to sum up the above vectors $d^{(l),R}$ as shown in the following identity:

$$\frac{\partial \mathcal{E}_R(m, a)}{\partial b_j} = \sum_{k=0}^{R-1} \frac{\partial \mathcal{E}_R(m, a)}{\partial a_j^{(R-k),R}} \frac{\partial a_j^{(R-k),R}}{\partial b_j} = \sum_{k=0}^{R-1} d_j^{(R-k),R}. \tag{2.11}$$

Note that this principle is similar to the so-called Backpropagation Through Time [24]. According to the identity in (2.10), we have

$$d^{(R-k),R} = \left(\prod_{l=0}^{k-1} D^{(R-l)} W^T\right) d^{(R),R}. \tag{2.12}$$

Therefore, writing (2.12) into the equation (2.11) we arrive to the equation

$$\frac{\partial \mathcal{E}_R(m, a)}{\partial b_j} = \sum_{k=0}^{R-1} \left[\left(\prod_{l=0}^{k-1} D^{(R-l)} W^T\right) d^{(R),R}\right]_j. \tag{2.13}$$

Observe that this should be true also for the fixed point $a \in \mathbb{R}^K$ of the iteration in (2.6). Since in this case,

the diagonal matrices $D^{(l)}$ $1 \leq l \leq R$ coincide, denoting their common value with D, equation (2.13) is simplified to

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial b_j} = \sum_{k=0}^{R-1} \left[\left(DW^T\right)^k d^{(R),R} \right]_j . \tag{2.14}$$

Taking the limit $R \to \infty$ in equation (2.14) and using assumptions (i) and (ii), we get the equation

$$\begin{aligned} \frac{\partial \mathcal{E}(m,a)}{\partial b_j} &= \lim_{R\to\infty} \frac{\partial \mathcal{E}_R(m,a)}{\partial b_j} \\ &= \lim_{R\to\infty} \sum_{k=0}^{R-1} \left[\left(DW^T\right)^k d^{(R),R} \right]_j = \left[\left(I - DW^T\right)^{-1} d^{(\infty)} \right]_j . \end{aligned} \tag{2.15}$$

Clearly, by the contraction condition of the theorem, there exists the inverse of $\left(I - DW^T\right)$.

We turn now to the statement for $\frac{\mathcal{E}(m,a)}{\partial w_{j,i}}$.

Similarly to the principle in (2.11), we obtain the following equation for arbitrary $a \in \mathbb{R}^K$.

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial w_{j,i}} = \sum_{k=0}^{R-2} \frac{\partial \mathcal{E}_R(m,a)}{\partial a_j^{(R-k),R}} \frac{\partial a_j^{(R-k),R}}{\partial w_{j,i}} = \sum_{k=0}^{R-2} d_j^{(R-k),R} y_i^{(R-k-1),R}. \tag{2.16}$$

We can apply the formula (2.12) again in (2.16), so that we get

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial w_{j,i}} = \sum_{k=0}^{R-2} \left[\left(\prod_{l=0}^{k-1} D^{(R-l)} W^T \right) d^{(R),R} \right]_j y_i^{(R-k-1),R}. \tag{2.17}$$

We assume again, that $a \in \mathbb{R}^K$ is the limit in (2.6). Therefore, $D^{(l)} \equiv D$ holds $\forall \leq l \in \mathbb{N}$. With these, we can rewrite (2.17) as

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial w_{j,i}} = \sum_{k=0}^{R-2} \left[\left(DW^T\right)^k d^{(R),R} \right]_j y_i. \tag{2.18}$$

Then we perform the limit transition with respect to the number of the layers again, but now in equation (2.18), so we get the equation

$$\begin{aligned} \frac{\partial \mathcal{E}(m,a)}{\partial w_{j,i}} &= \lim_{R\to\infty} \frac{\partial \mathcal{E}_R(m,a)}{\partial w_{j,i}} \\ &= \lim_{R\to\infty} \sum_{k=0}^{R-2} \left[\left(DW^T\right)^k d^{(R),R} \right]_j y_i = \left[\left(I - DW^T\right)^{-1} d^{(\infty)} \right]_j y_i. \end{aligned} \tag{2.19}$$

which completes the proof of the statement in the theorem. □ □

Remark. The algorithm of the calculation of the gradient can be found in the Pseudocode 2 in the Appendix.

2.5 Numerical simulations

In this section, we perform an experimental comparison between implicit and conventional feed-forward neural networks. Their performance are related in case of real-world multiclassification tasks. We use the cross-entropy loss function in each experiments, therefore, we modify the error function in (2.4) as follows.

$$\mathcal{E}(m) = \sum_{j=K-N+1}^{K} \tilde{t}_j^{(m)} \log o_j(y(x^{(m)})), \tag{2.20}$$

where $\tilde{t}_j^{(m)}$ denotes the j-th component of the m-th training sample $t^{(m)} = (t_1^{(m)}, \dots, t_N^{(m)}) \in \mathbb{R}^N$ of the target vector. This is compared with $o_j(y(x^{(m)}))$, where the softmax function o is applied the vector $y = [y_{K-N+1}, \dots, y_K] \in \mathbb{R}^N$ depending on the input. With these, omitting the arguments, $d^{(\infty)} \in \mathbb{R}^K$ should be easily modified to get

$$d_j^{(\infty)} = \begin{cases} \left(o_j - \tilde{t}_j^{(m)}\right) f_j{}'(a_j), & K-N < j \le K \\ 0, & 1 \le j \le K-N. \end{cases}$$

Our aim is to revisit and supplement the numerical results of the work in [9]. For the construction of the network, only dense layers are applied. We use the Adam optimizer [22] with the default parameter setting for feed-forward networks. We choose a batchsize of 100 for each simulation.

Above to the special gradient descent algorithm based on Theorem 1, we applied a splitting technique for training the weights in the network. First, we have adjusted the weights corresponding to the feed-forward edges of the network. Afterwards, we have optimized all weights. Since in the second half, we have started from a pre-trained network, we applied here a smaller learning rate 0.0025 compared to the default default value 0.001. This method can significantly speed up the learning of implicit networks. However, the computational cost is still much more demanding, see Table 2.1.

MNIST experiments

MNIST is a digit classification dataset, where the input data consist of images of handwritten digits between 0 and 9 [26]. The size of these greyscale images is 28×28 pixels. The training and the testing set contains 60000 and 10000 images, respectively. They are reshaped into 784 dimensional vectors and the entries are scaled to the range $[0, 1]$. The architecture we use for the neural network as a reference is a four-layer feedforward neural network with layer sizes $784 - 60 - 40 - 10$ using leaky ReLU activation in the hidden units. The comparisons were carried out with a three-layer implicit neural network with layer sizes $784 - 100 - 10$ using leaky ReLU activation in the hidden units, where the hidden block is fully connected to itself. For the leaky ReLU activation function, we use every time the parameter setting $\alpha = 0.1$. The obtained evaluations on the test set can be seen in Figures 2.5-2.6.

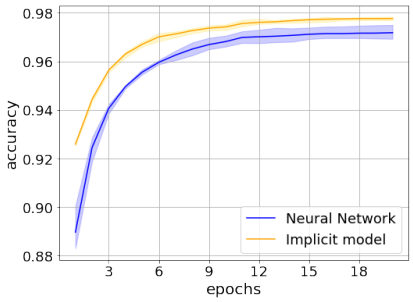

Figure 2.5: Performance comparison on the MNIST dataset regarding accuracy. An average of 5 independent runs is shown. Best accuracy for the implicit network: 0.9781, for the feed-forward neural network: 0.9719. The shaded region is enclosed between the maximum and minimum values over the runs, while the boldface curve displays the average.

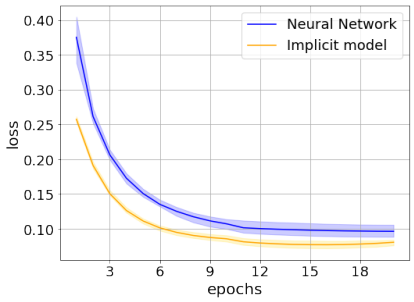

Figure 2.6: Performance comparison on MNIST dataset regarding loss generated from 5 independent runs. Smallest loss for the implicit network: 0.0770, and for the feed-forward neural network: 0.0963. The shaded region is enclosed between the maximum and minimum values over the runs, while the boldface curve displays the average.

GTRSB experiments

The benchmark problem called German Traffic Sign Recognition Benchmark (GTRSB) contains images of traffic signs consisting of 43 classes [25]. Originally, they are rgb images of size 32×32 pixels, which are turned into gray-scale. First, they were reshaped into 1024-dimensional vectors and then rescaled into the range $[0, 1]$. We have a number of 39208 training and 12630 test samples, respectively. The architecture we use for the neural network as a reference is a four-layer feedforward neural network with layer sizes $1024 - 300 - 100 - 43$ using leaky ReLU activation in the hidden units. The comparisons were carried out with a similar implicit structure using the same principle as in the previous subsection. We use a three-layer implicit neural network with layer sizes $1024 - 400 - 43$ using leaky ReLU activation in the hidden units, where the hidden block is fully connected to itself. For the leaky ReLU activation function, we use every time the parameter setting $\alpha = 0.1$. The obtained evaluations on the test set can be seen in Figures 2.7-2.8.

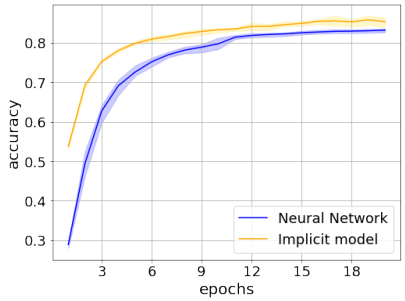

Figure 2.7: Performance comparison on GTRSB dataset for the accuracy generated from 5 independent runs. Average best accuracy for the implicit network: 0.8608, for the feed-forward neural network: 0.8317. The shaded region is enclosed between the maximum and minimum values over the runs, while the boldface curve displays the average.

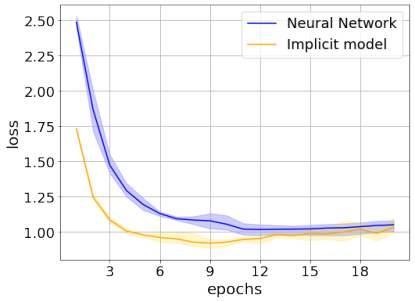

Figure 2.8: Performance comparison on GTRSB dataset for the loss generated from 5 independent runs. Mean of the smallest losses for the implicit neural network: 0.9082, for the feed-forward neural network: 1.0138. The shaded region is enclosed between the maximum and minimum values over the runs, while the boldface curve displays the average.

	Feed-forward network	Implicit network
Evaluating the network	4	34.69
Calculating the gradient	4	17.00

Table 2.1: Average of iteration numbers in the GTRSB experiments.

2.6 Conclusion

In this chapter, we generalized the evaluation of neural networks and the corresponding learning process. We have extended these algorithms to networks including directed cycles. By an illustrative approach, such networks were represented by a special Graph Neural Network, the so-called Implicit Neural Network. The implementation of the algorithms was applied in two series of numerical experiments were carried out using the MNIST and GTRSB datasets. We found that the implicit network overperforms the conventional feed-forward one in course of these experiments. At the same time, the computational complexity of the implicit approach may be significantly larger compared to the one of the conventional feedforward network.

Chapter 3

A more general implicit architecture

In this chapter, we generalize the evaluation of neural networks and the learning based on gradient backpropagation to special hypergraph neural networks, including directed cycles, in which a neuron can have multiple inputs and the value of the neuron is the product of the activation values of these inputs, to obtain a new network architecture. The theoretical results on the calculation of the gradient are supported by numerical experiments.

3.1 Construction of the network

The neural network is represented as a directed hypergraph [11]. A neuron in a network is a subset of (hyper)vertices, called the inputs of the neuron. We assume that two different neurons have no common input, and that any vertex is the input of a neuron. The edges go from the neurons to the vertices, i.e. to the inputs of other neurons, and have weights. Let there be K neurons in the network, of which N are output neurons, for simplicity let these be the ones with the largest index. We call a neuron input-type (or input neuron) if we write a component of the input vector x to one of the inputs of the neuron, but starting from (3.1) we reinterpret this.

The operation of the network can be understood as the propagation of electrical stimuli again, similar to the processes in the brain. The weights interpreted on the edges of the graph of the network can be represented as the strength of the synapse corresponding to a given edge. The greater the weight of the edge, the more the stimulus spreads through it. The evaluation of the network for an input vector $x \in \mathbb{R}^L$ is as follows. Let the i-th neuron have n_i inputs, denote the value of the j-th input by $a_{i[j]}$ and let $n = \sum_{i=1}^{K} n_i$. Then the index $i[j]$ can be thought of as an integer between 1 and n.

Given this input, let the activation function $f_{i[j]} : \mathbb{R} \to \mathbb{R}$. Frequently used examples are $f_{i[j]}(a) = \tanh(a)$ or $f_{i[j]}(a) = a$. If the j-th input of the neuron with index i receives a stimulus of magnitude y_l with the weight $w_{i[j],l}$ and a constant stimulus $b_{i[j]}$ (also called bias) applied to it, the cumulated input $a_{i[j]}$ is

$$a_{i[j]} = \sum_l w_{i[j],l} y_l + b_{i[j]} + \hat{x}_{i[j]}, \tag{3.1}$$

where the operator $\hat{:} \mathbb{R}^L \to \mathbb{R}^n$ is used to write a vector $x \in \mathbb{R}^L$ to the input of the neurons. We will not optimize this mapping and will usually write input values for only a few neurons, which can be called input neurons. The activation value of the j-th input of the i-th neuron and the activation value of the i-index

neuron are given by

$$y_{i[j]} = f_{i[j]}(a_{i[j]}), \quad y_i = \prod_{j=1}^{n_i} y_{i[j]}. \tag{3.2}$$

We call a network cyclic, or implicit, if the directed graph representing the connections of the neurons in the network contains a directed cycle, otherwise it is called feedforward, or acyclic. Figure 3.1. shows an example of a network structure, and Figure 3.2. shows the activation value of a neuron in this network.

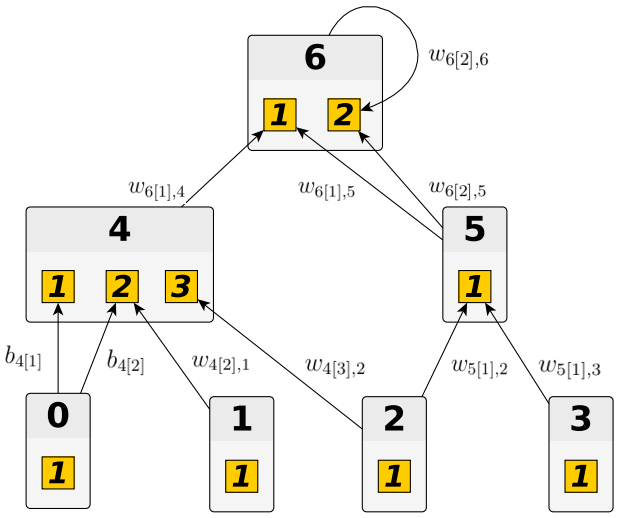

Figure 3.1: Example: an Implicit network with three input neurons (index $1-3$) and one output neuron (index 6). We can also include a neuron with index 0, this corresponds to the bias, the bias parameters are only given for the 4 index vertex for simplicity.

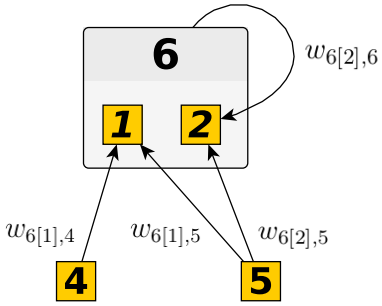

Figure 3.2: Calculation of the activation value of the neuron with index 6 of the implicit neural network shown in Figure 3.1. The satisfying equations are: $a_{6[1]} = w_{6[1],4}y_4 + w_{6[1],5}y_5 + b_{6[1]}$, $a_{6[2]} = w_{6[2],6}y_6 + w_{6[2],5}y_5 + b_{6[2]}$, $y_{6[1]} = f_{6[1]}\left(a_{6[1]}\right)$, $y_{6[2]} = f_{6[2]}\left(a_{6[2]}\right)$, $y_6 = y_{6[1]}y_{6[2]}$.

3.2 Problem statement

The formulas in (3.1)-(3.2) define the following system of equations.

$$\begin{cases} a(x) = Wy(x) + \hat{x} + b \\ y(x) = f(a(x)) \end{cases}. \tag{3.3}$$

Here, summarized $a(x), b \in \mathbb{R}^n$, $y(x) \in \mathbb{R}^K$, $W \in \mathbb{R}^{n\times K}$, $\hat{\cdot} : \mathbb{R}^L \to \mathbb{R}^n$ and $x \in \mathbb{R}^L$. Let the dimension n of the first equation be divided into K disjoint sets corresponding to the inputs of the neurons. The sets themselves correspond to the neurons. The activation values of the neurons are defined in the following line.

$$y_{i[j]} = f_{i[j]}(a_{i[j]}), \quad y_i = \prod_{j=1}^{n_i} y_{i[j]} \tag{3.4}$$

We also have M pairs of training samples (x, t) where $x \in \mathbb{R}^L$ are input and $t \in \mathbb{R}^N$ are target vectors. At the p-th pair of samples, i.e., at the input $x^{(p)}$, the error is defined as

$$\mathcal{E}(p) = \frac{1}{2} \sum_{j=K-N+1}^{K} \left(y_j(x^{(p)}) - \tilde{t}_j^{(p)}\right)^2, \tag{3.5}$$

where $\tilde{t}_j^{(p)}$ denotes the corresponding component of p-th target vector $t^{(p)} = (t_1^{(p)}, \ldots, t_N^{(p)}) \in \mathbb{R}^N$, which should be compared with the value of $y_j(x^{(p)})$. That is, let the operator $\tilde{} : \mathbb{R}^N \to \mathbb{R}^K$ defined by the formula $\tilde{t} = (0, \ldots, 0, t_1, \ldots, t_N)^T \in \mathbb{R}^K$. The average error $\mathcal{E}$ over all pairs of training samples is their average over the entire dataset, i.e.,

$$\mathcal{E} = \frac{1}{M} \sum_{p=1}^{M} \mathcal{E}(p). \tag{3.6}$$

We investigate the task to determine W and b such that the error given by (3.6) is minimal.

Solving (3.3) by a fixed point iteration yields the vector $a = (a_1, \ldots, a_n)^T$ of neuron input values and the vector $y = (y_1, \ldots, y_K)^T$ of activation values. A single step in the fixed point iteration for solving (3.3) has the form

$$a^{(l)} = Wy^{(l-1)} + b + \hat{x}, \quad y^{(l)} = f(a^{(l)}), \quad l \geq 2 \quad \text{and } a^{(1)} = \hat{x} \in \mathbb{R}^K. \tag{3.7}$$

An important observation is that the iteration in (3.7) delivers a feedforward neural network of infinite number of layers with K neurons in each layer. In this framework, the fixed point iteration can be interpreted as the layer-wise computation with the original input. The weights of the edges passing between each two adjacent layers are given by the matrix $W \in \mathbb{R}^{n \times K}$ and $b \in \mathbb{R}^n$ is the bias vector. Such an interpretation is shown in Figure 3.3, which is the unrolling of the small network shown in Figure 3.1 focusing only the sixth neuron.

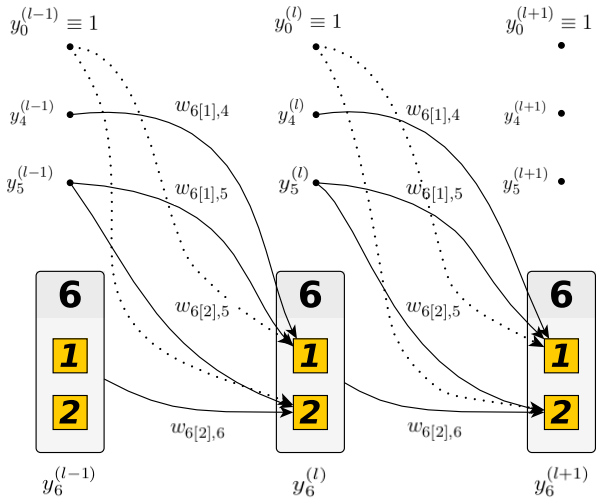

Figure 3.3: Unrolling of the implicit neural network shown in Figure 3.1 focusing on the 6th neuron in the $l-1$-th, l-th and $l+1$-th layers, $a_{6[1]}^{(l)} = w_{6[1],4}y_4^{(l-1)} + w_{6[1],5}y_5^{(l-1)} + b_{6[1]}$, $a_{6[2]}^{(l)} = w_{6[2],6}y_6^{(l-1)} + w_{6[2],5}y_5^{(l-1)} + b_{6[2]}$, $y_{6[1]}^{(l)} = f_{6[1]}\left(a_{6[1]}^{(l)}\right)$, $y_{6[2]}^{(l)} = f_{6[2]}\left(a_{6[2]}^{(l)}\right)$, $y_6^{(l)} = y_{6[1]}^{(l)}y_{6[2]}^{(l)}$.

Remark. The algorithm for computing the network is given in Pseudocode 3 in the Appendix.

3.3 Further notations

Summarized, we use the following notations in the infinite network:

- The initial vector of the iteration is $a^{(1)} = \hat{x} \in \mathbb{R}^n$. Let $a^{(l)}_{i[j]}(x)$ denote the j-th input value of the i-th neuron in the l-th layer and use $\lim_{l\to\infty} a^{(l)}_{i[j]}(x)$ provided that this exists and is finite. In vector form, we have $a^{(l)}(x) = \left(a^{(l)}_{1[1]}(x), \ldots, a^{(l)}_{1[n_1]}(x), \ldots, a^{(l)}_{K[1]}(x) \ldots, a^{(l)}_{K[n_K]}(x)\right)^T$ and $a(x) = \left(a_{1[1]}(x), \ldots, a_{1[n_1]}(x), \ldots, a_{K[1]}(x) \ldots, a_{K[n_K]}(x)\right)^T$. Sometimes, for simplicity, we omit the arguments x.

- Let $y^{(l)}_i(x)$ denote the activation value of the i-th neuron in the l-th layer with the input vector x. We use the notation $y_i(x) = \lim\limits_{l\to\infty} y^{(l)}_i(x)$, provided that this exists and is finite. Accordingly, we use

$$y^{(l)}_{i[j]}(x) = f^{(l)}_{i[j]}(a^{(l)}_{i[j]}(x)), \quad y^{(l)}_i(x) = \prod_{j=1}^{n_i} y^{(l)}_{i[j]}(x), \quad \text{and} \quad y^{(l)}(x) = \left(y^{(l)}_1(x), \ldots, y^{(l)}_K(x)\right)^T$$

  and similarly $y(x) = (y_1(x), \ldots, y_K(x))^T$.

- To compute the partial derivatives of the error function (3.5), introduce the following auxiliary values on the input-output vector pair $(x^{(p)}, t^{(p)})$:

$$d^{(\infty)}_{i[k]} = \begin{cases} \left(y_i(x^{(p)}) - \tilde{t}^{(p)}_i\right) f'_{i[k]}\left(a_{i[k]}\right) \prod_{n=1, n\neq k} f_{i[n]}\left(a_{i[n]}\right), & K - N < i \leq K \\ 0, \quad 1 \leq i \leq K - N \end{cases},$$

  where k runs from 1 to n_i in each case. Also denote by $d^{(\infty)} \in \mathbb{R}^n$ the column vector of these.

- We use the notation

$$D^{(l)}_{i[k]} = f'_{i[k]}(a^{(l)}_{i[k]}) \prod_{n=1, n\neq k} f_{i[n]}(a^{(l)}_{i[n]})$$

  for the utility value of the k-th input of i-th neuron in the l-th layer and $D_{i[k]} = \lim_{l\to\infty} D^{(l)}_{i[k]}$ provided that it exists and is finite. We also define the matrix $D \in \mathbb{R}^{n\times K}$ such that the value of the i-th column in the $i[k]$-th row is $D_{i[k]}$, the other elements of the matrix are 0. Similarly, we can obtain the matrix $D^{(l)} \in \mathbb{R}^{n\times K}$.

3.4 Theoretical results

As discussed previously, we transform the original cyclic network into an infinitely deep feedforward one. We apply the gradient backpropagation learning method this network.

To minimize the error function (3.6) using some gradient-based method, we need to determine the partial derivatives $\frac{\partial \mathcal{E}(p)}{\partial w_{j[k],i}}$ és $\frac{\partial \mathcal{E}(p)}{\partial b_{j[k]}}$. In the following Theorem, we express these in concrete terms. We make use the method in [15] 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.

(i) Equation (3.3) has a unique solution and the iteration (3.7) is convergent.

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

(iii) The linear mapping DW^T is a contraction in some norm, i.e. $\|DW^T\| < 1$.

Theorem 2. *With the assumptions (i)-(iii), the system of equations*

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

has a unique solution. Furthermore, the partial derivatives of the error function can be given as

$$\frac{\mathcal{E}(p)}{\partial b_{j[k]}} = d_{j[k]} \quad és \quad \frac{\mathcal{E}(p)}{\partial w_{j[k],i}} = d_{j[k]} y_i. \tag{3.8}$$

Proof: Consider the first $R \geq 2$ layers of the infinite forward-connected network constructed previously. Let $a \in \mathbb{R}^n$ given as the initialization of the fixed point iteration in (3.7).

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

$$\mathcal{E}_R(p, a) = \frac{1}{2} \sum_{j=K-N+1}^{K} \left(y_j^{(R),R}(x^{(p)}) - \tilde{t}_j^{(p)}\right)^2,$$

where we denote the a-dependence of the error.

With these, we have

$$a^{(l),R} = W y^{(l-1),R} + b + \hat{x}^{(p)} \quad \text{and} \quad a^{(1),R} = a$$

or componentwise, $a_{j[i]}^{(l),R} = \sum_k w_{j[i],k} y_k^{(l-1),R} + b_{j[i]} + \hat{x}_{j[i]}^{(p)}$.

We perform the gradient backpropagation on this truncated network. The partial derivative $d_{i[k]}^{(l),R} = \frac{\partial \mathcal{E}_R(m,a)}{\partial a_{i[k]}^{(l),R}}$ 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

$$d_{i[k]}^{(R),R} = \begin{cases} \left(y_i^{(R),R}(x^{(p)}) - \tilde{t}_i^{(p)}\right) f'_{i[k]}(a_{i[k]}^{(R),R}) \prod_{n=1, n\neq k} f_{i[n]}(a_{i[n]}^{(R),R}), & K-N < i \leq K \\ 0, \quad 1 \leq i \leq K-N \end{cases},$$

for the output ($K - N < i \leq K$) and non-output ($1 \leq i \leq K - N$) neurons, respectively.
For $1 \leq l < R$, correspondig to the gradient backpropagation algorithm, we have

$$d^{(l),R} = \frac{\partial \mathcal{E}_R(m, a)}{\partial a^{(l+1),R}} \cdot \frac{\partial a^{(l+1),R}}{\partial a^{(l),R}} = D^{(l)} W^T d^{(l+1),R}. \tag{3.9}$$

For calculating $\frac{\partial \mathcal{E}_R(m,a)}{\partial b_{i[k]}}$, we have to sum up the above vectors $d^{(l),R}$ as shown in the following identity:

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial b_{i[k]}} = \sum_{l=0}^{R-1} \frac{\partial \mathcal{E}_R(m,a)}{\partial a_{i[k]}^{(R-l),R}} \frac{\partial a_{i[k]}^{(R-l),R}}{\partial b_{i[k]}} = \sum_{l=0}^{R-1} d_{i[k]}^{(R-l),R}. \tag{3.10}$$

Note that this principle is similar to the one in Backpropagation Through Time [24]. According to the identity in (3.9), we have

$$d^{(R-l),R} = \left(\prod_{s=0}^{l-1} D^{(R-s)} W^T \right) d^{(R),R}. \tag{3.11}$$

Therefore, writing (3.11) into the equation (3.10) we arrive to the next equation.

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial b_{i[k]}} = \sum_{l=0}^{R-1} \left[\left(\prod_{s=0}^{l-1} D^{(R-s)} W^T \right) d^{(R),R} \right]_{i[k]}. \tag{3.12}$$

Observe that this should be true also for the fixed point $a \in \mathbb{R}^n$ of the iteration in (3.7). Since in this case, the diagonal matrices $D^{(l)}$ $1 \leq l \leq R$ coincide, denoting their common value with D, equation (3.12) is simplified to

$$\frac{\partial \mathcal{E}_R(m,a)}{\partial b_{i[k]}} = \sum_{l=0}^{R-1} \left[\left(DW^T \right)^l d^{(R),R} \right]_{i[k]}. \tag{3.13}$$

Taking the limit $R \to \infty$ in equation (3.13) and using assumptions (i) and (ii), we get the equation

$$\begin{aligned} \frac{\partial \mathcal{E}(m,a)}{\partial b_{i[k]}} &= \lim_{R\to\infty} \frac{\partial \mathcal{E}_R(m,a)}{\partial b_{i[k]}} \\ &= \lim_{R\to\infty} \sum_{l=0}^{R-1} \left[\left(DW^T \right)^l d^{(R),R} \right]_{i[k]} = \left[\left(I - DW^T \right)^{-1} d^{(\infty)} \right]_{i[k]}. \end{aligned} \tag{3.14}$$

Clearly, by the contraction condition of the theorem, there exists the inverse of $\left(I - DW^T\right)$.

We have now seen the first part of the claim we want to prove, i. e. the first part in formula (3.8). The proof of the second part of the formula (3.8) can be done in the same way as in Theorem 1. □.

Remark. The algorithm of the calculation of the gradient can be found in the Pseudocode 4 in the Appendix.

3.5 Numerical simulations

In this section, we perform an experimental comparison between implicit and conventional feed-forward neural networks. We use the binary cross-entropy loss function for N classes in our experiments, therefore, we modify the error function in (3.5) as follows.

$$\mathcal{E}(p) = -\sum_{j=K-N+1}^{K} \tilde{t}_j^{(p)} \log y(x^{(p)}) + \left(1 - \tilde{t}_j^{(p)}\right) \log\left(1 - y(x^{(p)})\right), \tag{3.15}$$

here $\tilde{t}_j^{(p)}$ denotes the j-th component of the p-th training sample $t^{(p)} = (t_1^{(p)}, \dots, t_N^{(p)}) \in \mathbb{R}^N$ of the target vector, this is compared with $y(x^{(p)})$, where the vector $y = [y_{K-N+1}, \dots, y_K] \in \mathbb{R}^N$ depending on the input. With these, $d^{(\infty)} \in \mathbb{R}^n$ should be easily modified to get

$$d_{i[k]}^{(\infty)} = \begin{cases} \frac{y_j - \tilde{t}_j^{(p)}}{y_j(1-y_j)} f_j{}'(a_j) \prod_{n=1, n\neq k} f_{i[n]}(a_{i[n]}), & K-N < i \leq K \\ 0, \quad 1 \leq i \leq K-N. \end{cases}$$

We propose a new simple implicit architecture, as shown in Figure 3.4.

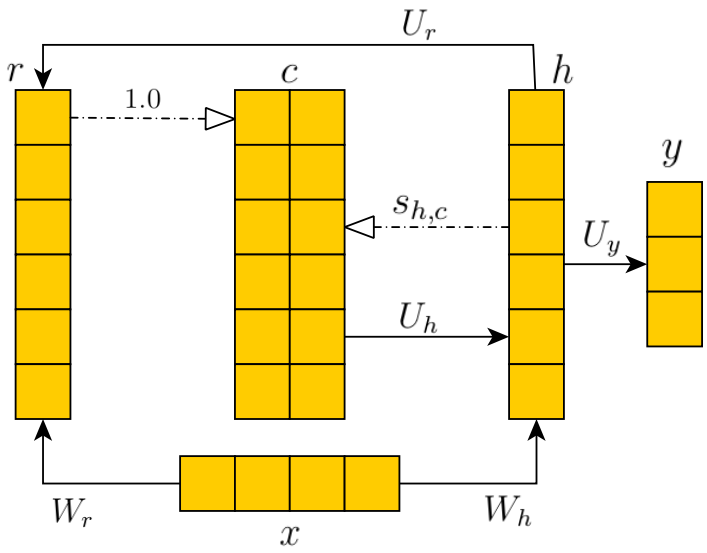

Figure 3.4: Schematic diagram of our proposed network architecture.

The system of equations corresponding to our proposed network arcitecture is the following:

$$\begin{cases} r = \phi\left(W_r x + U_r h + b_r\right) \\ c = r \odot \sigma\left(h \odot s_{h,c} + b_c\right) \\ h = \sigma\left(W_h x + U_h c + b_h\right) \\ y = \phi_o\left(U_y h + b_y\right). \end{cases} \tag{3.16}$$

Here, an explanation of the notation used in the formula follows with given cell number $H \in \mathbb{N}$. The transition matrices are $W_r, W_h \in \mathbb{R}^{H\times L}$, $s_{h,c} \in \mathbb{R}^H$, and $U_r, U_h \in \mathbb{R}^{H\times H}$. Furthermore, $r, c, h \in \mathbb{R}^H$, $y \in \mathbb{R}^N$ is the output and the input vector is $x \in \mathbb{R}^L$. The bias vectors are $b_r, b_h, b_c \in \mathbb{R}^H$ and $b_y \in \mathbb{R}^L$. Finally, ϕ is the tanh function, ϕ_h is the activation function at the output, σ is the sigmoid function and $\odot$ denotes the elementwise product.

Numerical experiments

Our choice for numerical experiments in this section is the HTRU2 dataset [27]. This dataset describes a sample of pulsar candidates. The task is binary classification, we have 17898 examples, 75% of the total data set is chosen as the teaching set, i.e. 13423 examples. Each example contains 8 input variables. For a detailed description of the variables, see [27]. After standardizing, the architecture we use for the neural network as a reference is four-layer feedforward neural network with layer sizes $8 - 40 - 20 - 1$ using leaky ReLU activations in the hidden units with parameter 0.1. Comparisons are made using the implicit architecture proposed in the previous section with the parameter setting $H = 20$. In both network, the output activation is the sigmoid function. We use the Adam optimizer with its standard parameters, and we choose a batchsize of 128 for the simulations.The obtained results on the set can be seen in Figures 3.5-3.6 and in Table 3.1.

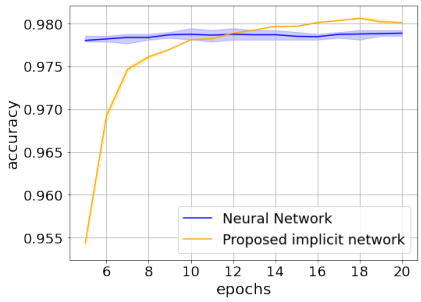

Figure 3.5: Performance comparison on HTRU2 dataset for the accuracy generated from 5 independent runs. Average best accuracy for the proposed implicit network: 0.9806, for the feed-forward neural network: 0.9794. The shaded region is enclosed between the maximum and minimum values over the runs, while the boldface curve displays the average.

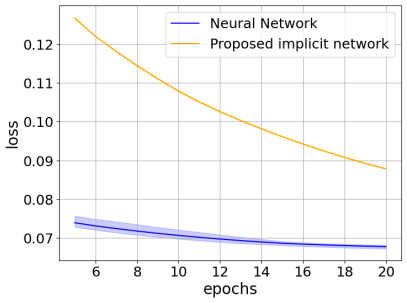

Figure 3.6: Performance comparison on HTRU2 dataset for the loss generated from 5 independent runs. Mean of the smallest losses for the proposed implicit network: 0.0678, for the feed-forward neural network: 0.0878. The shaded region is enclosed between the maximum and minimum values over the runs, while the boldface curve displays the average.

	Feed-forward network	Proposed Implicit network
Evaluating the network	4	33.1
Calculating the gradient	4	54.9

Table 3.1: Average of iteration numbers in the HTRU2 experiments.

3.6 Conclusion

In this chapter, we generalized the evaluation of neural networks and the corresponding learning process by taking the activation value of the neurons as the product of the activations of their cumulated inputs. We have extended these algorithms to networks including directed cycles similarly to Chapter 2. The implementation of the algorithms was applied in numerical experiments were carried out using the HTRU2 dataset. We found that the proposed implicit architecture overperforms the conventional feed-forward one in this experiment a little bit. However, further experiments are needed. At the same time, the computational complexity of the implicit approach may be significantly larger compared to the one of the conventional feedforward network.

Chapter 4

User manual for the programs

This chapter contains a short user documentation of the implemented programs, we will not go into all the implemented features in detail. The source code is open, free to use, available at `https://github.com/szbela87/neural`. The programs are under continuous development, as I use some versions for my work.

4.1 CPU version

Structure of the program The source code of the program, which was written in C is contained in the `main.c` file. The compilation can be done with *make*, the code is parallelized with *OpenMP*. On *Ubuntu 20.04.2*, the packages needed to compile are *gcc, openmp*, make*.

main.c: it contains all the source codes, which simulates a neural network. Reads the data from a file and the run parameters: first the simulation parameters from the **./inputs/simulparams.dat** file, and then the other parameters stored in this file that are needed for the simulation.

File with the variable name `input_name` containing the input-output data pairs: The j-th input-output data pair is located on the j-th line in the file (first the input data, then the output data), of course, in the same order on each line, separated by a space. For example, if you have 3 pieces of input data and 1 piece of output data, and the input data is 1.0, 2.0 and 4.0 and the output data is 3.9, then the corresponding line in the file is *1.0 2.0 4.0 3.9* .

The structure of the computation graph stored in the file `graph_datas`: Line j of the file describes neuron j. The first number is the number of neighbours of the neuron, followed by `###` characters. In the next block we list the neighbours of the neuron separated by `;` by first giving the identifier of the neighbouring neuron and then the identifier of the corresponding input of the target neuron. This block is also terminated by a sequence of `###` characters, followed by the number of inputs to the neuron, also followed by `###`. Next, we list the types of activation functions for the inputs, separated by spaces. It is also assumed that there are rows corresponding to input neurons at the beginning of the file and rows corresponding to output neurons at the end of the file.

The structure of the modifiability switches stored in the file `logic_datas`: Line j of the file describes neuron j, containing ones and zeros. Each switch describes the modifiability of the edges given in `graph_datas` (`1`=modifiable). The list of modifiability of weights and the bias values on the inputs are again separated by the `###` string.

The structure of the file describing the fixed weights stored in the file `fixwb_datas`: Line j of the file describes the neuron j, and sets the values of the weights set in `logic_datas` as unmodifiable in the appropriate order. Again, the list of fixed weights and input distortions is separated by the `###` string.

Let's look at these files through an example. Consider the network shown in Figure 4.1, then the contents of the file `graph_datas` are given in Code 4.1.

```
10 ### 5 1; 6 1; 7 1; 8 1; 9 1; 10 1; 11 1; 12 1; 13 1; 14 1; ### 1 ### 0
10 ### 5 1; 6 1; 7 1; 8 1; 9 1; 10 1; 11 1; 12 1; 13 1; 14 1; ### 1 ### 0
10 ### 5 1; 6 1; 7 1; 8 1; 9 1; 10 1; 11 1; 12 1; 13 1; 14 1; ### 1 ### 0
10 ### 5 1; 6 1; 7 1; 8 1; 9 1; 10 1; 11 1; 12 1; 13 1; 14 1; ### 1 ### 0
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
1 ### 15 1; 15 2; ### 1 ### 2
0 ###  ### 2 ### 1 2
```

Code 4.1: The contents of the file `graph_datas`

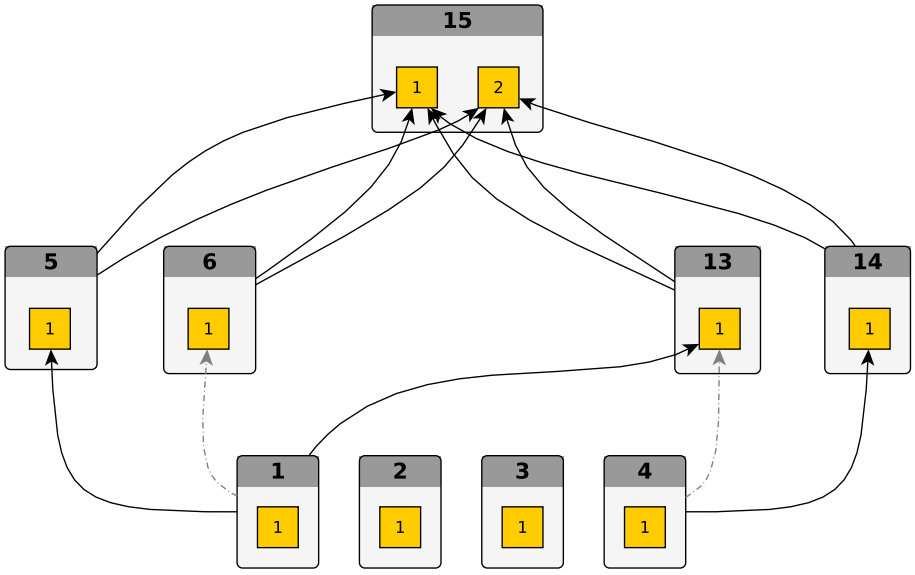

Figure 4.1: The network in the figure contains 15 neurons. Neurons with index $1-4$ are called input, 15 is output and the remaining neurons are hidden. Each of the input neurons are connected to the single input of the hidden neurons and the hidden neurons are also connected to both inputs of the output neuron. The activation function of the input neurons is the identity, the activation function of the first input of the output neuron is *tanh*. The second input of the output neuron has sigmoid activation function. The activation function of the hidden neurons is the *tanh* function.

We are investigating the network given in Figure 4.1 now. Assume that the edges between the hidden neurons and the second input of the output neuron are unmodifiable and have a value of 0.3 each and the bias values on the input neurons are also immutable with values 0.0. Then the contents of the `logic_datas` file can be seen in Code 4.2.

```
1 1 1 1 1 1 1 1 1 1 ### 0
1 1 1 1 1 1 1 1 1 1 ### 0
1 1 1 1 1 1 1 1 1 1 ### 0
1 1 1 1 1 1 1 1 1 1 ### 0
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
1 0 ### 1
### 1 1
```

Code 4.2: The contents of the file `logic_datas`

With these, also the contents of the `fixwb_datas` file can be seen in the Code 4.3

```
### 0.0
### 0.0
### 0.0
### 0.0
0.3 ###
0.3 ###
0.3 ###
0.3 ###
0.3 ###
0.3 ###
0.3 ###
0.3 ###
0.3 ###
0.3 ###
###
```

Code 4.3: The contents of the file `fixwb_datas`

Parameter	Description
`seed`	random seed
`thread_num`	number of parallel threads
`tol_fixit`	threshold for fixed point iterations
`maxiter_grad`	number of epochs
`maxiter_fix`	maximum number of iterations in fixed point iterations
`initdx`	a multiplication factor for weight initialization
`sfreq`	frequency of saving the results to file
`input_name`	filename of the dataset file
`output_name`	output file for monitoring the results
`predict_name`	filename of created predictions by the trained model
`data_num`	number of data samples
`learn_num`	number of training samples, the first `learn_num` rows are the training set
`mini_batch_size`	size of minibatches
`neuron_num`	number of neurons
`input_num`	number of input neurons
`output_num`	number of output neurons
`graph_datas`	the file containing the hypergraph which represents the network
`logic_datas`	file of the mutability of the weights
`fixwb_datas`	file with the immutable weights
`alpha`	parameter of L_2-regularisation
`lossfunction_type`	type of the loss function
`optimizer`	chooser for the optimizer
`grad_alpha`	learning rate in the stochastic gradient descent
`adam_alpha`	a parameter in Adam/Adamax/RAdam optimizers
`adam_beta1`	a parameter in stochastic gradient/Adam/Adamax/RAdam optimizers
`adam_beta2`	a parameter in Adam/Adamax/RAdam optimizers
`adam_eps`	a parameter in Adam optimizer
`ff_optimization`	if true, the the program turns on the PERT optimizer for acyclic networks
`chunker`	type of gradient clipping
`chunk_treshold`	threshold for gradient clipping
`loaddatas`	if true, then the program loads the saved weights stored in `load_backup`
`load_backup`	the program loads the saved weights from this file, if `loaddatas` is true
`save_backup`	the program creates backups from the weights to this file
`zero_optim_param`	setting to zero the learned paramaters of the optimizer
`numgrad`	if true, then the program calculates the numerical gradient, too
`numgrad_eps`	helping value for calculating the numerical gradients by finite difference schemes

Table 4.1: Description of the parameters in **./inputs/simulparams.dat**.

Identifier	Function
`0`	identity
`1`	sigmoid
`2`	tanh
`3`	ReLU
`4`	SiLU

Table 4.2: Types of the activation functions.

Identifier	Optimizer
1	stochastic gradient descent
2	Adam
3	Adamax
4	RAdam

Table 4.3: Types of the optimizers.

Identifier	Clipping type
0	no clipping
1	simple chunking above a threshold in absolute value
2	chunking above a threshold in maximum norm

Table 4.4: Types of the gradient clippings.

Function name	Description
read_parameters	loading the parameter settings
read_data	loading the data
read_graph	loading the graph files
rand_range_int	generating random integers
rand_range	generating float numbers
initialize_weights	initialize the weights in the network
act_fun	activation functions
act_fun_diff	derivative of the activation functions
calc_network_one_sample	calculating the network
calc_network_one_sample_ff	calculating the network with optimalization for acyclic network
calc_gradient_one_sample	calculating the network
calc_gradient_one_sample_ff	calculating the gradient with optimalization for acyclic network
calc_diff_matrices	maximum difference between two matrices
calc_gradient_mini_batch	calculating the gradient on a minibatch
calc_num_gradient_mini_batch	calculating the numerical gradient on a minibatch
calc_network_mini_batch	calculating the network on a minibatch
update_weights_bias_gd	updating the weights by stochastic gradient descent
update_weights_bias_adam	updating the weights by Adam optimizer
update_weights_bias_adamax	updating the weights by Adamax optimizer
update_weights_bias_radam	updating the weights by RAdam optimizer
dmax	maximum of two float numbers
imax	maximum of two integers
allocate_dmatrix	allocating float matrices in LOL representation
allocate_imatrix	allocating integer matrices in LOL representation
deallocate_dmatrix	deallocating float matrices
deallocate_imatrix	deallocating integer matrices
print_progress_bar	display the progress bar
save_weight_bias	saving the weights
load_weight_bias	loading the weights
matrix_norm	maximum norm of a matrix
calc_error	calculating the loss function
print_graph	printing the graph to the screen
program_failure	function for exception handling
random_normal	generating random float numbers from a Gaussian distribution
softmax	softmax function
make_predictions	creating predictions

Table 4.5: Brief description of the methods implemented.

4.2 GPU version

I also created a version of the program parallelized by GPU to speed up the calculations.

Structure of the program The source code of the program, which was written in C is contained in the `main.cu`, `kernels.cu` and `kernels.cuh` files. The compilation can be done by running first the *make clean* and then the *make* command, the code is parallelized by *CUDA*. On *Ubuntu 20.04.2*, the packages needed to compile are *nvcc* and *make*.

main.cu: it contains some of the source codes, which simulates a neural network. We mostly call the CUDA kernels (included in **kernels.cu**) from these functions. Reads the data from a file and the run parameters: first the simulation parameters from the **./inputs/simulparams.dat** file, and then the other parameters stored in this file that are needed for the simulation.

kernels.cu and **kernels.cuh**: it contains the CUDA kernels needed to simulate the neural network.

> Giving the parameters and the graph of the network are exactly the same as in the CPU version, the same files (**simulparams.dat**, `graph_datas`, `logic_datas`, `fixwb_datas`) are appropriate here as well.

However, there are some differences: the thread parameter for *OpenMP* `thread_num` is not used, and only the first version of gradient clipping is implemented. The list of the implemented functions and kernels can be seen in Tables 4.7-4.8.

Comparisons

Some comparisons have been made for acyclic networks between the CPU and the GPU program carried out on the MNIST dataset with parameter settings in the second chapter. My desktop computer has a *Xeon X5670* processor and a *Geforce GTX 1080* video card. The results are shown in Table 4.6.

Number of trainable parameters	CPU version running time $[s]$	GPU version running time $[s]$	Speedup
$79,510$	21	21	1.01
$318,010$	58	61	0.95
$1,276,810$	312	146	2.13
$5,592,010$	1311	369	3.55
$13,596,010$	4857	1376	3.52

Table 4.6: Comparisons between running times with different size of acyclic networks.

Function name	Description
`calc_gradient_mb_sum_gpu`	calculating the gradients on a minibatch
`calc_gradient_mb_sum_gpu_w`	calculating the gradients for the weights on a minibatch
`calc_gradient_mb_sum_gpu_b`	calculating the gradients for the bias on a minibatch
`weight_transpose_gpu`	transposing the weight matrix
`add_bias_bcast`	adding the bias to the inputs
`calc_grad_help_0_gpu`	calculating the help vectors needed by the gradient calculations – 1st part
`calc_grad_help_gpu`	calculating the help vectors needed by the gradient calculations – 2nd part
`calc_neuron_mb_gpu`	calculating the activation values
`update_weight_gd_gpu`	updating the weights by Gradient descent with momentum
`update_bias_gd_gpu`	updating the bias by Gradient descent with momentum
`update_weight_adam_gpu`	updating the weights by Adam optimizer
`update_bias_adam_gpu`	updating the bias by Adam optimizer
`update_weight_adamax_gpu`	updating the weights by Adamax optimizer
`update_bias_adamax_gpu`	updating the bias by Adamax optimizer
`copy_input_gpu`	copying the input data to the inputs of the input neurons
`set_zero_gpu`	setting all components of a vector to zero
`act_fun_gpu`	activation functions
`act_fun_diff_gpu`	derivative of the activation functions
`atomicMaxf`	atomic maximum function
`maxnorm`	calculating the max norm of a vector
`maxnormDiff`	calculating the max norm for the difference between two vectors
`calc_gradient_mb_gpu`	Calculating the gradients in the network
`calc_network_mb_gpu`	Calculating the input values in the network
`calc_gradient_mb_gpu_ff`	gradient calculation for the acyclic case (for back propagation)
`calc_network_mb_gpu_ff`	calculating the input values in the network for the acyclic case
`calc_neuron_mb_gpu_ff`	calculating the activation values for the acyclic case
`divide_gpu`	dividing a vector by a number
`l1norm`	L_1-norm of a vector with reduction
`l1normdiff`	L_1-norm for the difference between two vectors with reduction
`my_atomicAdd`	atomic addition function
`reg_weight_gpu`	L_2-regularization for the weights
`reg_bias_gpu`	L_2-regularization for the weights (we don't use it)
`clipping_weight_gpu`	gradient clipping for weights
`clipping_bias_gpu`	gradient clipping for bias

Table 4.7: Brief description of the kernels implemented in the GPU version.

Kernel name	Description
`read_parameters`	loading the parameter settings
`read_data`	loading the data
`read_graph`	loading the graph files
`rand_range_int`	generating random integers
`rand_range`	generating float numbers
`initialize_weights`	initialize the weights in the network
`act_fun`	activation functions
`act_fun_diff`	derivative of the activation functions
`calc_gradient_mini_batch`	calculating the gradient on a minibatch
`calc_network_mini_batch`	calculating the network on a minibatch
`calc_gradient_mini_batch_ff`	calculating the gradient on a minibatch for acyclic network
`calc_network_mini_batch_ff`	calculating the network on a minibatch for acyclic network
`dmax`	maximum of two float numbers
`imax`	maximum of two integers
`allocate_dmatrix`	allocating float matrices in LOL representation
`allocate_imatrix`	allocating integer matrices in LOL representation
`deallocate_dmatrix`	deallocating float matrices
`deallocate_imatrix`	deallocating integer matrices
`print_progress_bar`	display the progress bar
`save_weight_bias`	saving the weights
`load_weight_bias`	loading the weights
`matrix_norm`	maximum norm of a matrix
`calc_error`	calculating the loss function
`print_graph`	printing the graph to the screen
`program_failure`	function for exception handling
`random_normal`	generating random float numbers from a Gaussian distribution
`softmax`	softmax function
`make_predictions`	creating predictions
`make_predictions_ff`	creating predictions for acyclic network

Table 4.8: Brief description of the methods implemented in the GPU version.

Acknowledgements

The author 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. During this work the author used the supercomputer infrastructure of the Governmental Agency for IT Development in Hungary (KIFÜ).

I also would like to thank my supervisor, Ferenc Izsák for taking on the role of my supervisor and for the opportunity to work together again. Hopefully, not for the last time. And last but not least I would also like to thank my Darling for her endless patience for the work that often takes me into the night.

Appendix

Algorithm 1: Calculation of the network

Input: $x \in \mathbb{R}^L$ input data, tol > 0 is the tolerance threshold, maxiter is the maximum number of iterations and $W \in \mathbb{R}^{K \times K}$ the weight matrix

---

Output: $a, y, f' \in \mathbb{R}^K$

---

error $= \infty$; $a_i = 0, y_i = 0, f_i' = 0, \quad i = 1, \ldots, K$; iter $= 0$; $a = \hat{x}$;

while error $>$ tol *and* iter $<$ maxiter **do**

  for $j = 1 : K$ **do**

    $y_j = f_j(a_j)$; $f_j{}' = f_j{}'(a_j)$

  end

  iter $=$ iter $+ 1$; $a^{(old)} = a$; $a = \hat{x}$;

  for $j = 1 : K$ **do**

    for $k = 1 : K$ **do**

      if $\exists j \to k$ *edge* **then**

        $a_k = a_k + w_{k,j} y_j$

      end

    end

    $a_j = a_j + b_j$

  end

  error $= \|a^{(old)} - a\|$

end

return a, y, f'

Algorithm 2: Gradient calculation

Input: $a, y, f' \in \mathbb{R}^K$, $x^{(m)} \in \mathbb{R}^L$, $t^{(m)} \in \mathbb{R}^N$ target vector, tol > 0 tolerance threshold, maxiter is the maximum number of iterations and $W \in \mathbb{R}^{K \times K}$ the weight matrix

---

Output: $\frac{\partial \mathcal{E}(m)}{\partial b_i}, \forall i = 1, \ldots, K$ 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.

---

iter $= 0$; error $= \infty$; $d_i = 0, dh_i = 0, \quad i = 1, \ldots, K$;

for $i = K - N + 1 : K$ **do**

  $dh_i = \left(y_i - \tilde{t}_i^{(m)}\right) f_i{}'$

end

$d = dh$;

while error $>$ tol *and* iter $<$ maxiter **do**

  iter $=$ iter $+ 1$; $d^{(old)} = dh$; $dh_i = 0, \quad i = 1, \ldots, K$;

  for $j = 1 : K$ **do**

    for $k = 1 : K$ **do**

      if $\exists j \to k$ *edge* **then**

        $dh_j = dh_j + d_k^{(old)} w_{k,j} f_j{}'$

      end

    end

  end

  $d = d + dh$; error $= \|dh\|$

end

for $i = 1 : K$ **do**

  for $j = 1 : K$ **do**

    if $\exists i \to j$ *edge* **then**

      $\frac{\mathcal{E}(m)}{\partial w_{j,i}} = d_j y_i$

    end

  end

  $\frac{\partial \mathcal{E}(m)}{\partial b_i} = d_i$

end

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.

Algorithm 3: Network calculation in the more general case

Input: $x \in \mathbb{R}^L$, tol > 0 tolerance threshold, maxiter is the maximum number of iterations and the weight matrix $w \in \mathbb{R}^{n \times K}$, $b \in \mathbb{R}^n$

Output: $a, df \in \mathbb{R}^n$, $y \in \mathbb{R}^K$

---

error $= \infty$; $y_i = 0 : \ i = 1, \ldots, K$; $\quad a_{i[j]} = 0, f'_{i[j]} = 0, \ j = 1, \ldots, n_i, \ i = 1, \ldots, K$;;

$a = \hat{x}$;

while error $>$ tol **do**

  for $j = 1 : K$ **do**

    $y_j = 1$;

    for $k = 1 : n_j$ **do**

      $y_j = y_j \cdot f_{j[k]}(a_{j[k]})$;

      $f'_{j[k]} = f'_{j[k]}(a_{j[k]})$

    end

    for $k = 1 : n_j$ **do**

      $df_{j[k]} = f'_{j[k]} \prod_{l=1, l \neq k} f_{j[l]}(a_{j[l]})$

    end

  end

  $a^{(old)} = a; \ a = \hat{x}$;

  for $j = 1 : K$ **do**

    for $k = 1 : K$ **do**

      for $l = 1 : n_k$ **do**

        if $\exists j \to k[l]$ *edge* **then**

          $a_{k[l]} = a_{k[l]} + w_{k[l],j} y_j$

        end

      end

    end

    for $k = 1 : n_j$ **do**

      $a_{j[k]} = a_{j[k]} + b_{j[k]}$

    end

  end

  error $= \|a^{(old)} - a\|_\infty$

end

Algorithm 4: Gradient calculation in the more general case

Input: $x^{(p)} \in \mathbb{R}^L$, $t^{(p)} \in \mathbb{R}^N$, tol > 0 tolerance threshold, maxiter maximum number of iterations, $a, df \in \mathbb{R}^n$, $y \in \mathbb{R}^K$, $w \in \mathbb{R}^{n \times K}$, $b \in \mathbb{R}^n$

Output: $\frac{\partial \mathcal{E}(p)}{\partial b_{i[k]}}$, $k = 1, \ldots, n_i$, $i = 1, \ldots, K$, és $\frac{\partial \mathcal{E}(p)}{\partial w_{j[k],i}}$ $\forall i, j = 1, \ldots, K$, $k = 1, \ldots, n_j$ if $\exists i \to j[k]$ edge in the network.

---

error $= \infty$; $d_{j[k]} = 0 : \ j = 1, \ldots, n_i, \ i = 1, \ldots, K$;
for $i = K - N + 1 : K$ **do**
  for $j = 1 : n_i$ **do**
    $dh_{i[j]} = \left(y_i - t_i^{(p)}\right) df_{i[j]}$; $d_{i[j]} = dh_{i[j]}$
  end
end
while error $>$ tol **do**
  $d^{(old)} = dh$; $dh_{i[j]} = 0 : \ j = 1, \ldots, n_i, \ i = 1, \ldots, K$;
  for $i = 1 : K$ **do**
    for $k = 1 : n_i$ **do**
      for $j = 1 : K$ **do**
        for $l = 1 : n_j$ **do**
          if $\exists i \to j[l]$ *edge* **then**
            $dh_{i[k]} = dh_{i[k]} + d_{j[l]}^{(old)} w_{j[l],i} df_{i[k]}$
          end
        end
      end
    end
  end
  $d = d + dh$; error $= \|dh\|$
end
for $i = 1 : K$ **do**
  for $j = 1 : K$ **do**
    for $k = 1 : n_j$ **do**
      if $\exists i \to j[k]$ *edge* **then**
        $\frac{\partial \mathcal{E}(p)}{\partial w_{j[k],i}} = d_{j[k]} y_i$
      end
    end
  end
  for $k = 1 : n_i$ **do**
    $\frac{\partial \mathcal{E}(p)}{\partial b_{i[k]}} = d_{i[k]}$
  end
end

Statement

I, the undersigned, Béla J. Szekeres (NEPTUN code: FOTP4V), a student at Eötvös Loránd University, hereby declare and certify with my own signature that I have prepared this thesis without any unauthorized assistance, I have prepared it myself and used only those sources that I have provided in my thesis. Any part that I have taken from another source, either verbatim or in the same sense, but rephrased, has been clearly marked by indicating the source in accordance with the current regulations. I am the original author of this thesis and this independent intellectual work complies with the "Organizational and Operational Regulations of Eötvös Loránd University, II. Volume, Student Requirements System, With Amendments. 2017. September 1st. Regulation: 74/A–74/C. §".

Date

Student

List of Figures

List of Tables

Bibliography

[1] Izsák, F., Szekeres, B. J.: Implicit artificial neural networks: a graph neural network approach. Neurocomputing (2022), Submitted

[2] Ackley, D. H., Hinton, G. E., Sejnowski, T. J.: A learning algorithm for boltzmann machines. Cognitive Science 9, 1, (1985), 147–169. DOI:https://doi.org/10.1016/S0364-0213(85)80012-4

[3] Daubechies, I., DeVore, R., Foucart, S., Hanin, B., Petrova G.: Nonlinear Approximation and (Deep) ReLU Networks. Constructive Approximation. DOI: https://doi.org/10.1007/s00365-021-09548-z

[4] Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks 61, (2015) 85–117, ISSN 0893-6080, DOI: https://doi.org/10.1016/j.neunet.2014.09.003.

[5] Pinkus, A.: Approximation theory of the MLP model in neural networks. Acta Numerica, 8, (1999) 143–195. doi:10.1017/S0962492900002919

[6] Leshno, M., Lin, V., Pinkus, A., and Schocken, S.: Multilayer Feedforward Networks with a Non-Polynomial Activation Function Can Approximate Any Function. New York University Stern School of Business Research Paper Series (1993). DOI: 10.1016/S0893-6080(05)80131-5

[7] Zhang, Z., Cui, P. and Zhu, W.: Deep Learning on Graphs: A Survey. IEEE Transactions on Knowledge and Data Engineering, 34 (1), (2022) 249–270. DOI: 10.1109/TKDE.2020.2981333.

[8] Werbos, P. J.: Backwards Differentiation in AD and Neural Nets: Past Links and New Opportunities (2006). DOI: DOI:10.1007/3-540-28438-9_2

[9] El Ghaoui, L., Gu, F., Travacca, B., Askari, A., and Tsai, A.: Implicit deep learning. SIAM J. Math. Data Sci., 3(3): 930–958 (2021). DOI: https://doi.org/10.1137/20M1358517

[10] Bai, S., Kolter, J. Z., and Koltun, V.: Deep Equilibrium Models. Advances in Neural Information Processing Systems 32 (2019) 688–699.

[11] Ausiello, G., Luigi, L.:. Directed hypergraphs: Introduction and fundamental algorithms - A survey. Theoretical Computer Science 658, (2017) 293–306. doi:10.1016/j.tcs.2016.03.016

[12] Nadkarni, P. M., Ohno-Machado, L., Chapman, W. W.: Natural language processing: an introduction. Journal of the American Medical Informatics Association. 18, 5 (2011) 544–551. https://doi.org/10.1136/amiajnl-2011-000464

[13] Haddadnia, J., Ahmadi, M.: N-feature neural network human face recognition. Image and Vision Computing. 22, 12 (2004) 1071–1082. https://doi.org/10.1016/j.imavis.2004.03.011.

[14] LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature 521, (2015) 436-444. https://doi.org/10.1038/nature14539

[15] Rumelhart, D. E., Hinton, G. E., Williams, R. J.: Learning representations by back-propagating errors. Nature. 323, (1986) 533–536. https://doi.org/10.1038/323533a0

[16] Bianchini, M., Gori, M., Sarti, L., Scarselli, F.: Recursive Neural Networks and Graphs: Dealing with Cycles. Lecture Notes in Computer Science. 3931, (2005) 38–43. DOI: 10.1007/11731177_6

[17] Bianchini, M., Gori, M., Sarti, L., Scarselli, F.: Recursive processing of cyclic graphs. IEEE Transactions on Neural Networks with 17(1), (2006) 10–18. doi: 10.1109/TNN.2005.860873.

[18] Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE Transactions on Neural Networks. 20, 1 (2009) 61–80. https://doi.org/10.1109/TNN.2008.2005605

[19] Bondy, J. A., Murty, U. S. R.: Graph Theory. Springer, ISBN 978-1-84628-969-9, 2008.

[20] Fiesler, E.: Neural Network Classification and Formalization, Computer Standards & Interfaces. 16, (1994) 231–239.

[21] Guresen, E., Kayakutlu, G.: Definition of artificial neural networks with comparison to other networks. Procedia Computer Science. 3 (2011) 426–433. https://doi.org/10.1016/j.procs.2010.12.071.

[22] Kingma, D. P., Ba, J.: Adam: A Method for Stochastic Optimization. CoRR, abs/1412.6980 (2015).

[23] Malcolm, D. G., Roseboom, J. H., Clark, C. E., Fazar, W.: Application of a Technique for Research and Development Program Evaluation. Operations Research. 7, 5, (1959) 646–669. https://doi.org/10.1287/opre.7.5.646

[24] Werbos, P. J.: Backpropagation through time: what it does and how to do it, in Proceedings of the IEEE, 78(10): 1550-1560, (1990), DOI: 10.1109/5.58337.

[25] Stallkamp, J., Schlipsing, M., Salmen, J., Igel, C.: The german traffic sign recognition benchmark: a multi-class classification competition, in The 2011 international joint conference on neural networks, IEEE, (2011) 1453–1460, DOI:10.1109/IJCNN.2011.6033395

[26] Lecun, Y., Bottou, L., Bengio Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86, 11, (1998) 2278–2324. DOI:10.1109/5.726791.

[27] Lyon, R., J., Stappers, B. W., Cooper, S., Brooke, J. M., Knowles, J. D.: Fifty Years of Pulsar Candidate Selection: From simple filters to a new principled real-time classification approach. MNRAS, 459, 1, (2016) 1104–1123. DOI:https://doi.org/10.1093/mnras/stw656.