Implicit Neural Networks

Autoencoders: User Manual

Béla J. Szekeres, PhD and Ferenc Izsák, PhD

Contents

Construction of the network

Here, we describe a somewhat more general structure, the results of the manuscript can be directly generalized. The neural network is represented as a directed hypergraph. 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 (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

$$a_{i[j]} = \sum_l w_{i[j],l} y_l + b_{i[j]} + \hat{x}_{i[j]}, \tag{1}$$

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

$$y_{i[j]} = f_{i[j]}(a_{i[j]}), \quad y_i = \prod_{j=1}^{n_i} y_{i[j]}. \tag{2}$$

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 1. shows an example of a network structure, and Figure 2. shows the activation value of a neuron in this network.

The formulas in (1)-(2) define the following system of equations.

$$\begin{cases} a(x) = Wy(x) + \hat{x} + b \\ y(x) = f(a(x)) \end{cases}. \tag{3}$$

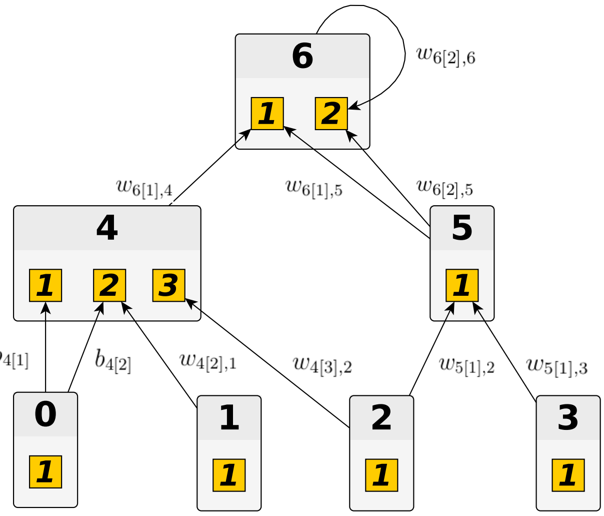

Figure 1: Example: an Implicit network with three input neurons (index $1-3$) and one output neuron (index 6). We can also include a neuron with index 0, this corresponds to the bias, the bias parameters are only given for the 4 index vertex for simplicity.

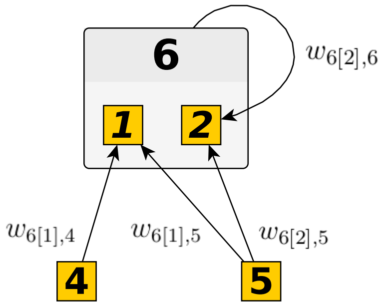

Figure 2: Calculation of the activation value of the neuron with index 6 of the implicit neural network shown in Figure 1. The satisfying equations are: $a_{6[1]} = w_{6[1],4}y_4 + w_{6[1],5}y_5 + b_{6[1]}$, $a_{6[2]} = w_{6[2],6}y_6 + w_{6[2],5}y_5 + b_{6[2]}$, $y_{6[1]} = f_{6[1]}\left(a_{6[1]}\right)$,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

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.

$$y_{i[j]} = f_{i[j]}(a_{i[j]}), \quad y_i = \prod_{j=1}^{n_i} y_{i[j]} \tag{4}$$

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

$$\mathcal{E}(p) = \frac{1}{2}\sum_{j=K-N+1}^{K} \left(y_j(x^{(p)}) - \tilde{t}_j^{(p)}\right)^2, \tag{5}$$

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.,

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

We investigate the task to determine W and b such that the error given by (6) is minimal.

Solving (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) has the form

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

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. 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, which is the unrolling of the small network shown in Figure 1 focusing only the sixth neuron.

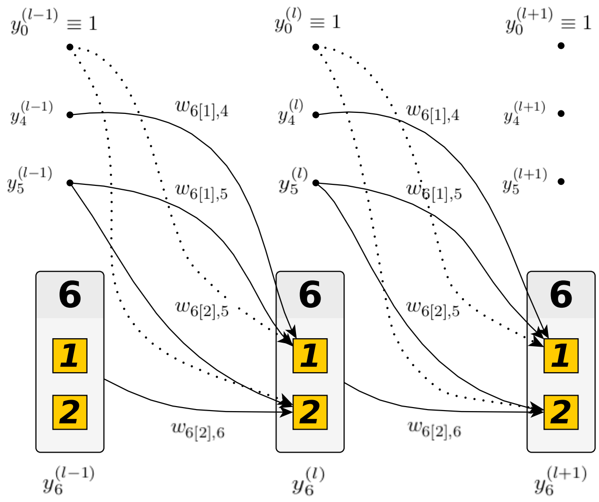

Figure 3: Unrolling of the implicit neural network shown in Figure 1 focusing on the 6th neuron in the $l-1$-th, l-th and $l+1$-th layers, $a_{6[1]}^{(l)} = w_{6[1],4}y_4^{(l-1)} + w_{6[1],5}y_5^{(l-1)} + b_{6[1]}$, $a_{6[2]}^{(l)} = w_{6[2],6}y_6^{(l-1)} + w_{6[2],5}y_5^{(l-1)} + b_{6[2]}$, $y_{6[1]}^{(l)} = f_{6[1]}\left(a_{6[1]}^{(l)}\right)$, $y_{6[2]}^{(l)} = f_{6[2]}\left(a_{6[2]}^{(l)}\right)$, $y_6^{(l)} = y_{6[1]}^{(l)}y_{6[2]}^{(l)}$.

Remark. The algorithms for computing the network and the gradient backpropagation are given in Pseudocodes 1-2 in the Appendix.

User manual for the program

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.

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.

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, then the contents of the file `graph_datas` are given in Code 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 1: The contents of the file `graph_datas`

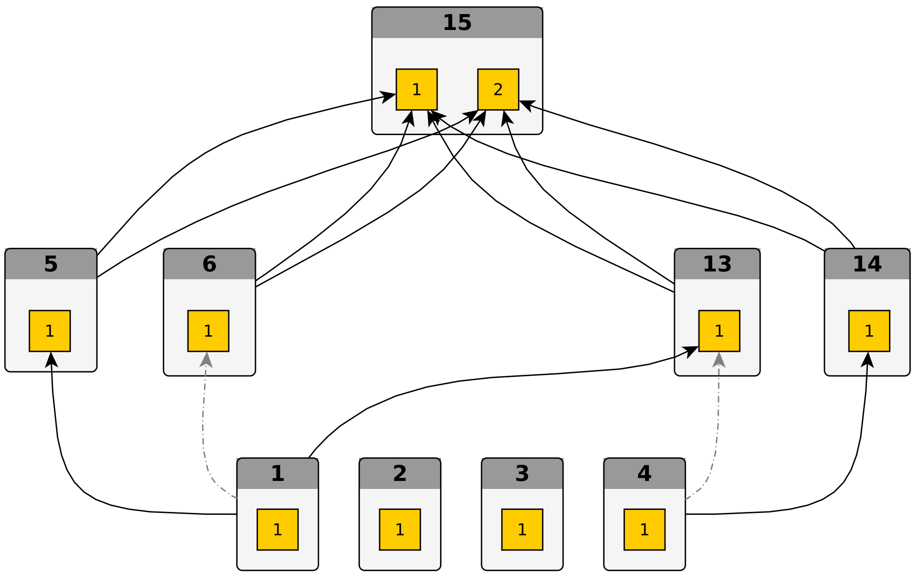

Figure 4: The network in the figure contains 15 neurons. Neurons with index $1-4$ are called input, 15 is output and the remaining neurons are hidden. Each of the input neurons are connected to the single input of the hidden neurons and the hidden neurons are also connected to both inputs of the output 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 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 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 2: The contents of the file `logic_datas`

With these, also the contents of the `fixwb_datas` file can be seen in the Code 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 3: The contents of the file `fixwb_datas`

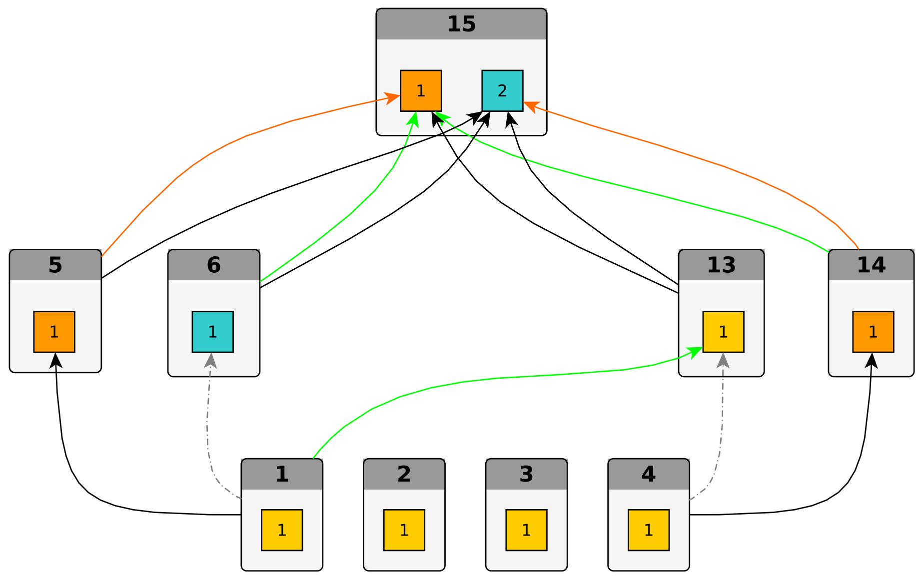

Figure 5: Almos the same network as in Figure 4. The structure is the same, but there are now two groups with shared weights. The first group is represented by green edges, which are composed of the following edges: $(6 \to 15, 1)$, $(1 \to 13, 1)$, and $(14 \to 15, 1)$. The second group is marked in orange and consists of the following edges: $(5 \to 15, 1)$ and $(14 \to 15, 2)$. There are also two bias groups with shared weights. The first group is marked in orange on the inputs, comprised of the first input of neuron 5, the first input of neuron 15, and the first input of neuron 14. The second group is indicated in turquoise, consisting of the first input of neuron 6 and the second input of neuron 15.

In Figure 5, a network with shared weights and biases is presented. The contents of the files representing this, `shared_w_datas` and `shared_b_datas`, can be seen in the text boxes 4 and 5.

```
3 ### 6 15 1; 14 15 1; 1 13 1;
2 ### 5 15 1; 14 15 2;
```

Code 4: The contents of the file `shared_w_datas`

```
3 ### 5 1; 15 1; 14 1;
2 ### 6 1; 15 2;
```

Code 5: The contents of the file `shared_b_datas`

The parameters of **simulparams.txt** file can be found in Tables 1 and 2 with very brief descriptions. The list of the implemented functions and kernels can be seen in Tables 7-8.

Parameter	Description
`seed`	random seed
`shuffle_num`	number of the permutations while shuffing
`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 per epoch of saving the results to file and validating
`input_name`	filename of the training dataset file
`input_name_valid`	filename of the validation dataset file
`input_name_test`	filename of the testin dataset file
`output_name`	output file for monitoring the results
`predict_name_valid`	filename of created predictions by the trained model on the validation set
`predict_name_test`	filename of created predictions by the trained model on the testing set
`test_log`	filename for the predictions on the testing set for logging
`test_log_final`	filename for the predictions on the testing set for logging at the end
`learn_num`	number of training samples, the samples are the rows of the file
`valid_num`	number of validation samples
`test_num`	number of testing samples
`mini_batch_size`	size of minibatches
`neuron_num`	number of neurons
`input_num`	number of input neurons
`output_num`	number of output neurons
`shared_weights_num`	number of the shared weigth groups
`shared_bias_num`	number of the shared bias groups
`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
`shared_w_datas`	filename of the shared weight groups
`shared_b_datas`	filename of the shared bias groups
`alpha`	parameter of L_2-regularisation
`train_lossfunction_type`	type of the loss function
`valid_metric_type`	type of the first validation metric for choosing the best model
`valid_metric_type_2`	second validation metric to choose the threshold in classification
`range_div`	dividing the $[-4, 4]$ interval into this many parts
`optimizer`	chooser for the optimizer
`nesterov`	0 or 1 to using Nesterov momentum in SGD
`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
`clipping`	type of gradient clipping
`clipping_threshold`	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_best_model`	the program creates backups of the best weights to this file

Table 1: Description of the parameters in **./inputs/simulparams.dat** I.

Parameter	Description
`lr_scheduler`	learning rate scheduler
`cyclic_momentum`	cyclic momentum for one cycle learning rate scheduler
`pcr`	identifying the peak in the learning rate scheduler (ratio)
`base_momentum`	base momentum for one cycle learning rate scheduler
`max_momentum`	maximal momentum for one cycle learning rate scheduler
`div_factor`	scaling factor for the learning rates
`final_div_factor`	scaling factor for the learning rates at the end of the training
`step_size`	frequency of scaling the learning rate by div_factor using 3rd `lr_scheduler`
`lr_gamma`	scaling factor for the 3rd learning rate scheduler
`early_stopping`	if larger than 0 then early stopping is applied with this evaluation frequency

Table 2: Description of the parameters in **./inputs/simulparams.dat** II.

Identifier	Function
0	identity
1	sigmoid
2	tanh
3	Leaky ReLU ($\alpha = 0.1$)
4	SiLU
8	cos
9	arctan

Table 3: Types of the activation functions.

Identifier	Optimizer
1	stochastic gradient descent
2	Adam
3	Adamax

Table 4: Types of the optimizers.

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

Table 5: Types of the gradient clippings.

Identifier	Function
0	none
1	onecycle learning rate scheduler
2	cosine annealing
3	exponential learning rate decay

Table 6: Types of the activation functions.

Function name	Description
`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
`calc_gradient_mb_sum_gpu`	calculating the gradients 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_h_gpu`	updating the weights by Gradient descent with momentum
`update_bias_gd_h_gpu`	updating the bias by Gradient descent with momentum
`update_weight_gd_gpu`	updating the weights by Stochastic Gradient descent with momentum
`update_bias_gd_gpu`	updating the bias by Stochastic 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 7: Brief description of the kernels implemented in the GPU version in the *kernels.cu* file.

Kernel name	Description
`f_lr_momentum`	helper function for the learning rate schedulers
`read_parameters`	loading the parameter settings
`read_data`	loading the data
`read_graph`	loading the graph files
`predict`	creating predictions on the validation and on the testing sets
`rand_range_int`	generating random integers
`rand_range`	generating float numbers
`act_fun`	activation functions
`act_fun_diff`	derivative of the activation functions
`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
`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
`copy_dmatrix`	copy float matrix from matrix representation to vector
`copy_imatrix`	copy integer matrix from matrix representation to vector
`initialize_weights`	initialize the weights in the network
`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
`make_predictions`	creating predictions
`make_predictions_ff`	creating predictions for acyclic network
`save_weight_bias`	saving the weights
`load_weight_bias`	loading the weights
`calc_vector_norm`	maximum norm of a vector
`calc_diff_norm`	difference between two vectors
`trapz`	numerical integration with trapezoidal quadrature
`calculate_auc`	calculating AUC
`calculate_tpr_fpr`	calculating FPR and TPR for binary classification
`calculate_mcc`	calculating MCC

Table 8: Brief description of the methods implemented in the GPU version in the *main.cu* file.

Pseudocodes

Algorithm 1: 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 = ∞; y_i = 0 : i = 1,...,K;   a_{i[j]} = 0, f'_{i[j]} = 0, j = 1,...,n_i, i = 1,...,K;;
a = x̂;
while error > tol do
    for j = 1 : K do
        y_j = 1 ;
        for k = 1 : n_j do
            y_j = y_j · 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]} ∏_{l=1,l≠k} f_{j[l]}(a_{j[l]})
        end
    end
    a^{(old)} = a;  a = x̂ ;
    for j = 1 : K do
        for k = 1 : K do
            for l = 1 : n_k do
                if ∃j → 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‖_∞
end
```

Algorithm 2: 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: $\frac{\partial \mathcal{E}(p)}{\partial b_{i[k]}},\ k = 1, \dots, n_i,\ i = 1, \dots, K$, és $\frac{\partial \mathcal{E}(p)}{\partial w_{j[k],i}}$ $\forall i, j = 1, \dots, K,\ k = 1, \dots, n_j$ if $\exists i \to j[k]$ edge in the network.

---

error $= \infty$; $d_{j[k]} = 0 :\ j = 1, \dots, n_i,\ i = 1, \dots, K$;

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**

  $d^{(old)} = dh$; $dh_{i[j]} = 0 :\ j = 1, \dots, n_i,\ i = 1, \dots, K$;

  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

List of Figures

List of Tables