TUCS

Napsu Karmitsa | Sona Taheri | Kaisa Joki | Pauliina Mäkinen | Adil Bagirov | Marko M. Mäkelä

Hyperparameter-free NN algorithm for large-scale regression problems

Turku Centre *for* Computer Science

TUCS Technical Report
No 1213, November 2020

Hyperparameter-free NN algorithm for large-scale regression problems

Napsu Karmitsa
University of Turku
FI-20014 Turku, Finland
napsu@karmitsa.fi
Sona Taheri
RMIT University
Melbourne, Australia
sona.taheri@rmit.edu.au
Kaisa Joki
University of Turku
FI-20014 Turku, Finland
kjjoki@utu.fi
Pauliina Mäkinen
University of Turku
FI-20014 Turku, Finland
pauliina.e.makinen@gmail.com
Adil Bagirov
Federation University Australia,
Victoria, Australia
a.bagirov@federation.edu.au
Marko M. Mäkelä
University of Turku
FI-20014 Turku, Finland
makela@utu.fi

TUCS Technical Report
No 1213, November 2020

Abstract

In this paper, a new nonsmooth optimization based algorithm for solving large-scale regression problems is introduced. The regression problem is modeled using fully-connected feedforward neural networks with one hidden layer, the piecewise linear activation, and the L_1-loss functions. A novel constructive approach is developed for an automated determination of the proper number of hidden nodes. The limited memory bundle method [Haarala et.al., 2004, 2007] is applied to minimize the nonsmooth objective of the new regression problem. The proposed algorithm is evaluated using real-world data sets with both large number of input features and large number of samples. It is also compared with the well-known backpropagation neural network for regression using TensorFlow. The results demonstrate the superiority of the proposed algorithm as a predictive tool in most data sets used in our numerical experiments.

Keywords: Regression analysis, Neural networks, ReLU, L_1-loss function, Nonsmooth optimization, Nonconvex optimization, Limited memory bundle method.

TUCS Laboratory
Turku Optimization Group (TOpGroup)

1 Introduction

Regression models and methods are used as effective tools for prediction and approximation in many real-world applications. For complex relationships between explanatory and response variables (input and output features), the regression models build with the *neural networks* (NNS) can significantly improve the prediction power in comparison with the other existing regression methods [37]. The most commonly used *neural networks for regression* (NNR) are the feedforward backpropagation network (FFBPN) [43] and the radial-basis network (RBN) [7]. The FFBPN and most of its variants converge to only locally optimal solutions [23, 42], and they only work under the precondition that all the functions involved in the network are differentiable. The RBN and its modifications can be trained without local minima issues [20], but they require a heuristic selection of parameters and hyperparameters.

There are two important components in any NNS: an *activation function* and an error function, conventionally called a *loss function*. An NN with the most simple linear activation function is not able to take into account all the information of the training set resulting in a very poor outcome in the testing phase. On the other hand, smooth (continuously differentiable) nonlinear activation functions may lead to highly complex nonconvex loss functions. Moreover, in [44], it is shown that the usage of smooth activation functions requires many training iterations and a large number of hidden nodes.

In [25], it is theoretically discussed that NNS with nonsmooth activation functions demonstrate a high performance. Among these functions, the *Rectified Linear Unit* (RELU) is considered as the simplest one. Since this function is piecewise linear (i.e. it is not differentiable at all points) its usage leads to the nonsmoothness of the loss function. Instead of minimizing such a loss function directly, it is usually replaced with a smoothed surrogate loss function. However, algorithms based on this approach may become inaccurate once the size of the data and, thus, the number of smoothing parameters, increases. Another commonly used choice to minimize the loss function is the stochastic (sub)gradient descent method (SGD) (see e.g. [11]). Although the SGD is in general fast, it is not accurate and may sometimes require a large number of function and subgradient evaluations. This is due to the fact that the SGD is an extension of the subgradient method for convex problems and NNS lead to nonconvex problems.

The loss function can be defined using the *mean absolute error* (MAE or L_1-norm) or the *mean squared error* (MSE or L_2-norm). There are at least two reasons to choose the L_1- over the L_2-norm: first, the regression models with the L_1-norm are more robust to outliers than those based on the L_2-norm (see, e.g., [26]); and second, the use of the L_2-norm makes the loss function with the RELU activation function more complex than the use of the L_1-norm.

The performance of the NNS is highly sensitive to the choice of the hyperparameters defining the structure of the network and thus, the learning process. Indeed, a poor selection of hyperparameters may lead to inaccurate results [24, 38]. Generally, the hyperparameters can be determined either automatically or manually. Many learning systems are designed that rely on hyperparameter optimization through a combination of grid search and manual search (see e.g. [21, 33, 34]). However, tuning the hyperparameters manually is a tedious and time consuming process. Therefore, it is necessary to automate the calibration of the hyperparameters. Several algorithms with varying levels of automaticity have been proposed for this purpose, for instance, in [14, 16, 22, 40, 35, 45]. Nevertheless, there is no guarantee that the selected number of hidden units (usually the number of nodes) is optimal in these algorithms and the question of good hyperparameter tuning procedure still remains open.

In this paper, we introduce a new approach for modeling and solving regression problems using fully-connected feedforward NNS with the RELU activation function, the L_1-loss function and the L_1-regularization. We call this regression problem the RELU-NNR problem and the algorithm for solving it is named as the LMBNNR algorithm. We consider the NN with only one hidden layer. Nevertheless, the number of hidden nodes is determined automatically using a novel *constructive approach* and an *automated stopping procedure* (ASP). More precisely, the number of hidden nodes is constructed incrementally starting from one node and the ASP, based on the intelligent selection of initial weights and the regularization parameter, is applied at each iteration of the constructive algorithm to stop training if the model is not improving. Thus, there is no tuneable hyperparameters in the proposed algorithm, and it can be considered as a hyperparameter-free method.

The RELU-NNR problem is both nonconvex and nonsmooth[1]. The nonconvexity is addressed as a part of the constructive approach. That is, the solution of the previous iteration is used to construct new initial weights. To solve the underlying nonsmooth optimization problems we apply the limited memory bundle method (LMBM) introduced in [17, 18, 27]. We utilize this method since it is one of the "not so many" algorithms capable of handling large dimensions, nonconvexity, and nonsmoothness. In addition, the LMBM has already proved itself in solving other machine learning problems such as clustering [29, 4], clusterwise linear regression [28] and missing value imputation [30].

The proposed LMBNNR algorithm has some remarkable features including:

- it is hyperparameter-free due to the automated determination of the proper number of nodes;
- it is applicable to large-scale regression problems;
- it is an efficient and accurate predictive tool.

The structure of the paper is as follows. Section 2 provides some theoretical background on nonsmooth optimization, NNS, and regression analysis. The problem statement — the RELU-NNR problem — is given in Section 3. In Section 4, the LMBNNR algorithm is introduced together with the ASP. In Section 5 we evaluate

[1]Note that RELU itself is nonsmooth. Thus, even if we used a smooth loss function and regularization, the underlying optimization problem would still be nonsmooth.

the accuracy of the ASP as well as the performance of the LMBNNR algorithm and compare the proposed algorithm with the backpropagation NNR using TensorFlow. Finally, Section 6 concludes the paper.

2 Theoretical background and notations

In this section we provide some theoretical background and notations that are used throughout the paper.

2.1 Nonsmooth optimization

Nonsmooth optimization refers to the general problem of minimizing (or maximizing) functions that are typically not differentiable at their minimizers (maximizers) [3]. We denote the n-dimensional Euclidean space by $\mathbb{R}^n$, the inner product by $x^\top y = \sum_{i=1}^n x_i y_i$, where $x,\ y \in \mathbb{R}^n$, and the associated norms by $\|x\|_2 = (x^\top x)^{1/2}$ and $\|x\|_1 = \sum_{i=1}^n |x_i|$.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *locally Lipschitz continuous* on $\mathbb{R}^n$ if for any bounded subset $X \subset \mathbb{R}^n$ there exists $L > 0$ such that

$$|f(x) - f(y)| \leq L\|x - y\|_2 \quad \text{for all } x,\ y \in X.$$

The *Clarke's subdifferential* $\partial f(x)$ of a locally Lipschitz continuous function $f : \mathbb{R}^n \to \mathbb{R}$ at a point $x \in \mathbb{R}^n$ is given by [3, 10]

$$\partial f(x) = \operatorname{conv}\left\{ \lim_{i\to\infty} \nabla f(x_i) \mid x_i \to x \text{ and } \nabla f(x_i) \text{ exists} \right\},$$

where "conv" denotes the convex hull of a set. A vector $\xi \in \partial f(x)$ is called a *subgradient*. The point $x^* \in \mathbb{R}^n$ is *stationary*, if $0 \in \partial f(x^*)$. Note that stationarity is a necessary condition for local optimality.

2.2 Neural networks

Generally, NNS can be divided into feedforward or recurrent networks based on their network topology. An NN is *feedforward* if there exists an ordering of nodes such that every node is only connected to nodes further down in the order. If such an ordering does not exist, the network is *recurrent* having one or more feedback loops allowing the information travel in both directions. The feedforward NN can be visualized as a layered network. The first and the last layers are called the *input* and the *output* layers, respectively. Intermediate layers are called *hidden layers*. The dimension of input features determines the size of the input layer. The number and size of the hidden layers — the *hyperparameters* of the network — can be chosen more freely. The output layer consists of only one node in regression problems. In addition to these nodes, each layer of the NNS contains a bias node. A *fully-connected* NNS are

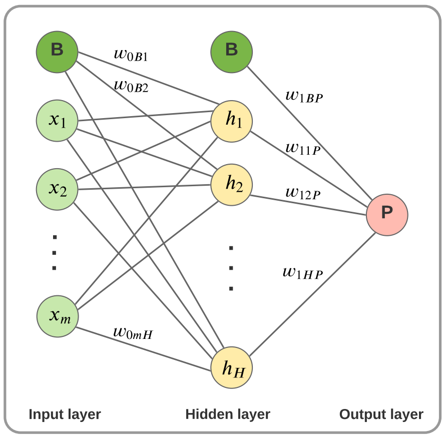

Figure 1: A simple NN model with one output, one hidden layer and H nodes.

comprised of layers consisting of nodes and artificial synapses with *weights* connecting each pair of nodes in the consecutive layers of the NNS [1]. Figure 1 shows the standard formulation of the fully-connected feedforward NNS with one hidden layer.

For a given node, the inputs are multiplied by the weights associated with the node and summed together. This value is referred to as a *summed activation* of the node. The summed activation is then transformed via an activation function and it defines the specific output of the node. The simplest activation function is linear. Although a NN with the linear activation function can be easily trained it cannot learn complex mapping functions. Therefore, in order to learn more complex structures in data the nonlinear activation functions are preferred. Two widely used nonlinear activation functions are the sigmoid and the hyperbolic tangent activation functions [1]. However, a general problem with both these activation functions is that they saturate. That is, the input to the activation function may reach a flat region of the function, so additional changes to the input will have little or no effect on the output.

Lately, the piecewise linear activation function RELU has become the default activation function for many types of NNS due to the computational advantages of its simple structure and, thus, strong training ability. RELU is nonsmooth but it has a simple mathematical form of $f(x) = \max\{0, x\}$. RELU provides more sensitivity to the summed activation of the node and avoids easy saturation. Thus, RELU is preferable for the training of complex relationships in NNS compared to traditional smooth sigmoid and tanh functions.

As a part of the learning process in the NNS, the error for the current state of the model must be estimated repeatedly. This requires the choice of a loss function, that is used to estimate the error in the model such that the weights can be updated to reduce the loss on the next evaluation. The default loss function for regression problems is the MSE (L_2 norm). It is the average of the squared differences between the predicted and the actual values. However, the MSE may not perform well if the distribution of the target variable is mostly Gaussian with many outliers. In such a case, the MAE

(L_1 norm) is an appropriate loss function as it is more robust to outliers. In its turn, the MAE is calculated as an average of the absolute difference between the predicted and actual values.

2.3 Neural networks for regression

Let A be a given data set with n samples: $A = \{(x^i, y_i) \in \mathbb{R}^m \times \mathbb{R} \mid \ i = 1, \dots, n\}$, where $x^i \in \mathbb{R}^m$ are values of m input features and $y_i \in \mathbb{R}$ are their outputs. In regression analysis, the aim is to find a function $\varphi : \mathbb{R}^m \to \mathbb{R}$ such that $\varphi(x^i)$ is a "good approximation" of y_i. In other words, the following *regression error* is minimized

$$\sum_{i=1}^{n} |\varphi(x^i) - y_i|^p, \quad p > 0.$$

Applying NNs to the regression problem can lead significantly higher predictive power compared to a traditional regression. NNs takes several input features (independent variables), multiplies them by their weights (coefficients), and runs them through an activation function and a loss function, which closely resembles the regression error. Once the NN is trained the optimal weights for the model (regression coefficients) are found to fit the data. Therefore, the NN can simulate a regression error. In addition, it can model more and more complex scenarios by increasing the number of nodes. This concept has been proved in the universal approximation theorem stating that a single hidden layer feedforward network of sufficient complexity is able to approximate any given regression function on a compact set to an arbitrary degree [47].

Let us denote the number of hidden nodes in the NNs by H and let w be the weight connection vector and w_s be its element in the index place s, where s is the index triplet $s = abc$. Then w_{abc} states the weight that connects a-th layer's b-th node to $(a+1)$-th layer's c-th node (see Figure 1). In addition, let us denote by P an output index value for the weights which connect the hidden layer's nodes to the NN's output node. It is necessary to use a special notation for the connection weights linking these nodes since a prediction is produced by taking a weighted sum of the output signals of the hidden layer and, in contrast to the hidden nodes, no activation function is applied in the output node. There are no signals coming to the bias terms as an input. Thus, the only connection weights linking these terms are the ones starting from them. In order to assort these differently behaving connection weights, we use B to denote the middle index value for the weights connecting some layer's bias term to some node in the next layer of the NN. The bias terms with this index can be thought as the last node in each layer.

The weight connection vector w for m input features and H hidden nodes has $H(m+2)+1$ components. For the sake of clarity, we organize $w = w^H$ in an "increasing order" as follows:

$$w^H = (w_{011}, w_{012}, \dots, w_{01H}, w_{021}, w_{022}, \dots, w_{02H}, \dots, w_{0m1}, w_{0m2}, \dots, w_{0mH},$$
$$w_{0B1}, w_{0B2}, \dots, w_{0BH}, w_{11P}, w_{12P}, \dots, w_{1HP}, w_{1BP})^\top.$$

3 Nonsmooth optimization model of ReLU-NNR

In this section we formulate the nonsmooth optimization model for the ReLU-NNR problem. For simplicity, using the notations and definitions given in Section 2, we define the following:

$$s_{ij} = w_{0Bj} + \sum_{k=1}^{m} w_{0kj} x_k^i, \qquad i = 1, \dots, n, \; j = 1, \dots, H, \quad \text{and}$$

$$t_i = w_{1BP} + \sum_{j=1}^{H} w_{1jP} \max\{0, s_{ij}\}, \qquad i = 1, \dots, n,$$

where x_k^i denotes the k-th coordinate of the i-th sample. Then the loss function using the L_1-norm is

$$F_H(w) = \sum_{i=1}^{n} |y_i - t_i|. \tag{1}$$

To avoid overfitting that may occur in the learning process of the NN — situation when the NN is so closely fitted to the training set that it is difficult to generalize and make predictions for a new data — we add an extra element to the loss function defined in (1). In most cases, the L_1-regularization is preferable as it reduces the weight values of less important input features, and also, it is robust and insensitive to outliers. Thus, we rewrite (1) as

$$f_H(w) = F_H(w) + \rho_H \|w\|_1, \tag{2}$$

where $\rho_H > 0$ is an automatically updated regularization parameter (to be described later). Therefore, we represent the nonsmooth optimization formulation for the ReLU-NNR as follows:

$$\begin{cases} \text{minimize} & f_H(w) \\ \text{subject to} & w \in \mathbb{R}^{H(m+2)+1}. \end{cases} \tag{3}$$

This optimization problem is nonsmooth and nonconvex. To solve it, we introduce a new algorithm in the next section. First we describe an accessible way to calculate (an approximate) subgradient of the objective f_H given in (2).

Since the objective f_H — obtained as a sum of the function F_H and the convex regularization term $\rho_H \|w\|_1$ — is both nonconvex and nonsmooth and it may not be subdifferentially regular (see [3]), the calculation of the exact subgradient may not be a trivial task at a nondifferentiable point. This is particularly true when we use Clarke's subdifferential. The difficulties arise from F_H which is a sum of nonsmooth nonconvex functions (see (1)). When we calculate an arbitrary subgradient of each nonconvex nonsmooth partial function $|y_i - t_i|$ at a nondifferentiable point w, the sum of these subgradients does not necessarily belong to the subdifferential of F_H [3], and therefore, we can only guarantee to have an approximation of an exact subgradient.

However, this approximation is much easier to calculate than the exact subgradient. In addition, the use of approximated subgradients instead of exact ones has not caused any noticeable problems in the numerical experiments.

Next, we present more closely how the approximated subgradient of f_H is obtained. We start by defining the following index sets:

$$\begin{aligned}
I^+ &= \{\, i \mid y_i - p_i \geq 0,\ i = 1, \ldots, n\}, \\
I^- &= \{\, i \mid y_i - p_i < 0,\ i = 1, \ldots, n\}, \\
J_i^+ &= \{\, j \mid s_{ij} \geq 0,\ j = 1, \ldots, H\} \quad \text{for } i = 1, \ldots, n, \\
J_i^- &= \{\, j \mid s_{ij} < 0,\ j = 1, \ldots, H\} \quad \text{for } i = 1, \ldots, n,
\end{aligned}$$

By utilizing these sets, we can rewrite the function F_H in the form

$$F_H(w) = \sum_{i \in I^+} \big(y_i - t_i\big) + \sum_{i \in I^-} \big(- y_i + t_i\big),$$

where

$$t_i = t_i(w) = \Big(w_{1BP} + \sum_{j \in J_i^+} w_{1jP} s_{ij}\Big).$$

In order to construct an approximation of the subgradient of F_H at a point w, we first give an approximated subgradient $\xi^i \in \mathbb{R}^{H(m+2)+1}$ of the function t_i. This subgradient can be written as

$$\begin{aligned}
\xi^i = (\xi^i_{011}, \xi^i_{012}, \ldots, \xi^i_{01H}, \xi^i_{021}, \xi^i_{022}, \ldots, \xi^i_{02H}, \ldots, \xi^i_{0m1}, \xi^i_{0m2}, \ldots, \xi^i_{0mH} \\
\xi^i_{0B1}, \xi^i_{0B2}, \ldots, \xi^i_{0BH}, \xi^i_{11P}, \xi^i_{12P}, \ldots, \xi^i_{1HP}, \xi^i_{1BP})^\top,
\end{aligned}$$

where

$$\begin{aligned}
\xi^i_{0kj} &= \begin{cases} w_{1jP} x^i_k, & \text{if } j \in J_i^+ \\ 0, & \text{otherwise} \end{cases} \qquad \text{for } k = 1, \ldots, m \ \text{ and } j = 1, \ldots, H, \\
\xi^i_{0Bj} &= \begin{cases} w_{ijP}, & \text{if } j \in J_i^+ \\ 0, & \text{otherwise} \end{cases} \qquad \text{for } j = 1, \ldots, H, \\
\xi^i_{1jP} &= \begin{cases} s_{ij}, & \text{if } j \in J_i^+ \\ 0, & \text{otherwise} \end{cases} \qquad \text{for } j = 1, \ldots, H, \\
\xi^i_{1BP} &= 1.
\end{aligned}$$

The approximated subgradient ξ^F of F_H at a point w can then be given with the formula

$$\xi^F = \sum_{i \in I^+} -\xi^i + \sum_{i \in I^-} \xi^i.$$

For the convex regularization term, one possible exact subgradient $\xi^{reg} \in \mathbb{R}^{H(m+2)+1}$ at a point w can be stated with components

$$\xi_s^{reg} = \begin{cases} \rho_H, & \text{if } w_s \geq 0 \\ -\rho_H, & \text{otherwise} \end{cases} \qquad \text{for } s = 1, \dots, H(m+2)+1.$$

Finally, by combining the above results, the approximated subgradient ξ^f of the objective function f_H at a point w can be obtained by setting

$$\xi^f = \xi^F + \xi^{reg}.$$

Note that ξ^f is an exact subgradient of f_H whenever the calculations are done at a differentiable point and by Rademacher's Theorem a locally Lipschitz continuous function is differentiable almost everywhere [13].

4 The proposed LMBNNR algorithm

In this section we propose the new LMBNNR algorithm for solving the RELU-NNR problem. In addition, we recall the LMBM in the form used here and profess its convergence in the case of the RELU-NNR problem.

The algorithm LMBNNR computes nodes incrementally and uses the ASP to detect the proper number of nodes. More precisely, it starts with one node in the hidden layer and adds a new node to this layer at each iteration of the algorithm. It is worth of noting that each time a node is added to the hidden layer, $m+2$ new connection weights appear. Starting from initial weight $w^1 \in \mathbb{R}^{m+3}$, the LMBNNR algorithm applies the LMBM to solve the underlying RELU-NNR problem. The solution to this problem is employed to generate initial weights for the next iteration. This procedure is repeated until the ASP is activated or the maximum number of hidden nodes $H_{\max}$ is reached. The ASP is designed based on the value of the objective function. That is if the value of the objective is not improved in three subsequent iterations, then the LMBNNR algorithm is stopped. Note that the initialization of weights and the regularization parameter (described below) are determined such a way that we have $f_H \geq f_{H+1}$ for all $H = 1, \dots, H_{\max}$.

We select the initial weights w^1 as

$$w^1 = \Big(\frac{1}{m}, \frac{1}{m}, \dots, \frac{1}{m}, 0, 1, 0\Big)^\top \in \mathbb{R}^{m+3}.$$

Here the first m components are the weights from the nodes of the input layer to the node of the hidden layer. The weights from both the bias terms w_{1BP} and w_{0B1} are set to zero and the weight w_{11P} from the hidden layer to the output is set to one. Figure 2 illustrates the weights in different iterations and the progress of the LMBNNR algorithm.

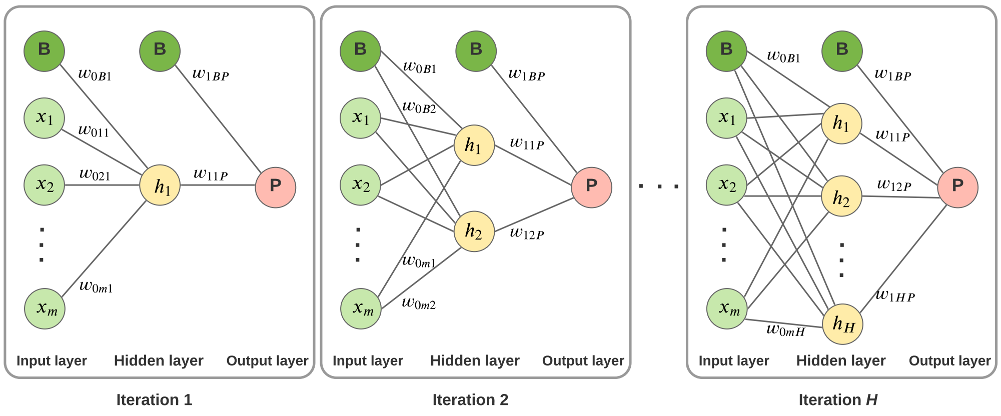

Figure 2: Construction of model in NNR problem by the LMBNNR.

At the subsequent iteration H (i.e. when we add a new node to the hidden layer), $H = 2, \dots, H_{\text{final}}$ of the LMBNNR algorithm — H_{final} is the final number of hidden nodes — we initialize the weights w^H as follows: let $\bar{w}^{H-1}$ be the solution of the previous ReLU-NNR problem. First we set weights from all but the last two hidden nodes to the output layer as

$$\begin{aligned} w^H_{1jP} &= \bar{w}^{H-1}_{1jP} \quad \text{for all } j = 1, \dots, H-2. \quad \text{and} \\ w^H_{1BP} &= \bar{w}^{H-1}_{1BP} \end{aligned} \tag{4}$$

and the weights from the input layer to all but the last two hidden nodes as

$$\begin{aligned} w^H_{0ij} &= \bar{w}^{H-1}_{0ij} \quad \text{for all } i = 1, \dots, m, \ j = 1, \dots, H-2, \text{ and} \\ w^H_{0Bj} &= \bar{w}^{H-1}_{0Bj} \quad \text{for all } j = 1, \dots, H-2. \end{aligned} \tag{5}$$

Note that in the case of $H = 2$ we simply set $w^H_{1BP} = \bar{w}^{H-1}_{1BP}$ and otherwise we skip this step as there is only two nodes in the hidden layer. Then we take the most recent weights obtained at the previous iteration of the LMBNNR algorithm and split these weights to get initial weights connecting the input layer and the last two hidden nodes of the current iteration, that is,

$$\begin{aligned} w^H_{0i(H-1)} &= \frac{1}{2}\bar{w}^{H-1}_{0i(H-1)} \quad \text{and} \quad w^H_{0iH} = w^H_{0i(H-1)} \quad \text{for all } i = 1, \dots, m, \\ w^H_{0B(H-1)} &= \frac{1}{2}\bar{w}^{H-1}_{0B(H-1)} \quad \text{and} \quad w^H_{0BH} = w^H_{0B(H-1)}. \end{aligned} \tag{6}$$

Finally, we set weights from the last two hidden nodes to the output as

$$w^H_{1(H-1)P} = \bar{w}^{H-1}_{1(H-1)P} \quad \text{and} \quad w^H_{1HP} = w^H_{1(H-1)P}. \tag{7}$$

To update the regularization parameter ρ_H for $H = 2, \ldots, H_{\text{final}}$, we use the value of the objective at the previous iteration and the weight connecting the current node of the hidden layer to the output as follows:

$$\rho_H = \rho_{H-1} \cdot \frac{f_{H-1}}{f_{H-1} + |w_{1HP}|}, \tag{8}$$

with $\rho_1 = 1.0$.

Now we are ready to give the LMBNNR algorithm.

Algorithm 1: LMBNNR

```
Data: Data set $A$, the number of input features $m$, and the maximum number of
      hidden nodes $H_{\max} > 0$.
Result: The final number of hidden nodes $H_{\text{final}}$ and the solutions $\bar{w}^H$ to the
        $H$-th ReLU-NNR problem, $H = 1, \ldots, H_{\text{final}}$.
Set the initial weights $w^1 \in \mathbb{R}^{m+3}$ and the regularization parameter $\rho_1 = 1.0$;
Set $H = 1$;
Use the LMBM (Algorithm 2) to solve the ReLU-NNR problem (3) with one
  hidden node starting from $w^1$. Denote the solution by $\bar{w}^1$ and the
  corresponding value of the objective by $f_1$;
while $H < H_{\max}$ do
    Set $H = H + 1$;
    Initialize weights $w^H \in \mathbb{R}^{H(m+2)+1}$ using the previous solution $\bar{w}^{H-1}$ and
      equations (4) – (7);
    Update the regularization parameter $\rho_H$ using (8);
    Use the LMBM to solve the ReLU-NNR problem (3) with $H$ hidden
      nodes starting from $w^H$. Denote the solution by $\bar{w}^H$ and the corresponding
      value of the objective by $f_H$;
    if $H > 2$ then
        if $f_{H-2} = f_H$ then
            Set $H_{\text{final}} = H$;
            STOP with the current model;
Set $H_{\text{final}} = H_{\max}$;
STOP with the current model.
```

REMARK 4.1. With the used initialization of weights w^H and the regularization parameter ρ_H we always have $f_H \geq f_{H+1}$ for $H = 1, \ldots, H_{\max}$. The stopping procedure is activated if the value of the objective is not improved in three subsequent iterations. In other words, if adding more nodes to the hidden layer does not give us a better model after optimization of the weights.

REMARK 4.2. As there is no tuneable hyperparameters in the LMBNNR algorithm there is no need for a validation set either. It is enough to choose $H_{\max}$ big enough (see Section 5 for suitable values).

Next we describe the LMBM, with a slight modification to its original version, for solving the underlying ReLU-NNR problems in the LMBNNR algorithm. This method is called at every iteration of the LMBNNR. For more details of the LMBM we refer to [17, 18, 27].

Algorithm 2: LMBM for ReLU-NNR problems

```
Data: $w_1^H \in \mathbb{R}^{H(m+2)+1}$, $D^0 = I$, $\hat{m}_c \geq 3$, $k_{\max} > 0$, and $\varepsilon > 0$.
Result: Final weight vector $w_k^H$.
Compute $\xi_1 \in \partial f_H(w_1^H)$;
Set $k = 1$, $\tilde{k} = 1$, $d_1 = -\xi_1$, $\tilde{\xi}_1 = \xi_1$, and $\tilde{\beta}_1 = 0$;
while $k \leq k_{\max}$ and the termination condition $-\tilde{\xi}_k^\top d_k + 2\tilde{\beta}_k \leq \varepsilon$ is not met do
    Find step sizes $t_L^k$ and $t_R^k$, and the subgradient locality measure $\beta_{k+1}$;
    Set $w_{k+1}^H = w_k^H + t_L^k d_k$ and $v_{k+1} = w_k^H + t_R^k d_k$;
    Evaluate $f_H(w_{k+1}^H)$ and $\xi_{k+1} \in \partial f_H(v_{k+1})$;
    Store the new correction vectors $s_k = v_{k+1} - w_k^H$ and $u_k = \xi_{k+1} - \xi_{\tilde{k}}$;
    Set $\hat{m}_k = \min\{k, \hat{m}_c\}$;
    if $t_L^k > 0$ then (Serious step)
        Compute the search direction $d_{k+1} = -D^k \xi_{k+1}$, where $D^k$ is calculated
          using the L-BFGS update with $\hat{m}_k$ most recent correction vectors;
        Set $\tilde{k} = k + 1$ and $\tilde{\beta}_{k+1} = 0$;
    else (Null step)
        Determine multipliers $\lambda_i^k$ satisfying $\lambda_i^k \geq 0$ for all $i \in \{1, 2, 3\}$, and
          $\sum_{i=1}^3 \lambda_i^k = 1$ that minimize the function

          $$\begin{aligned}\varphi(\lambda_1, \lambda_2, \lambda_3) =& [\lambda_1 \xi_{\tilde{k}} + \lambda_2 \xi_{k+1} + \lambda_3 \tilde{\xi}_k]^\top D^{k-1} [\lambda_1 \xi_{\tilde{k}} + \lambda_2 \xi_{k+1} + \lambda_3 \tilde{\xi}_k] \\ &+ 2(\lambda_2 \beta_{k+1} + \lambda_3 \tilde{\beta}_k)\end{aligned}$$

          and compute the aggregate values

          $\tilde{\xi}_{k+1} = \lambda_1^k \xi_{\tilde{k}} + \lambda_2^k \xi_{k+1} + \lambda_3^k \tilde{\xi}_k$ and $\tilde{\beta}_{k+1} = \lambda_2^k \beta_{k+1} + \lambda_3^k \tilde{\beta}_k$;
        Compute the search direction $d_{k+1} = -D^k \tilde{\xi}_{k+1}$, where $D^k$ is calculated
          using the L-SR1 update with $\hat{m}_k$ most recent correction vectors;
    Set $k = k + 1$;
```

Remark 4.3. We use a nonmonotone line search [27] to find step sizes t_L^k and t_R^k when Algorithm 2 is combined with Algorithm 1. In addition, as in [2] we use a relatively low maximum number of iterations $k_{\max}$ to avoid overfitting.

Remark 4.4. The search direction in Algorithm 2 is computed using the L-BFGS update after a serious step and the L-SR1 update after a null step. The updating formulae are similar to those in the classical limited memory variable metric methods for smooth optimization [9]. Nevertheless, the correction vectors u_k and s_k are obtained using subgradients instead of gradients and the auxiliary point instead of the new iteration point.

REMARK 4.5. The classical linearization error may be negative in the case of a nonconvex objective function. Therefore, a subgradient locality measure β_k, which is a generalization of the linearization error for nonconvex functions (see e.g. [32]), is used in Algorithm 2.

We now recall convergence properties of the LMBM in case of RELU-NNR problems. The objective function $f_H : \mathbb{R}^{H(m+2)+1} \to \mathbb{R}$ is locally Lipschitz continuous and upper semismooth (see e.g. [6]). In addition, the level set $\{ w^H \in \mathbb{R}^{H(m+2)+1} \mid f_H(w^H) \leq f_H(w_1^H) \}$ is bounded for every starting weight $w_1^H \in \mathbb{R}^{H(m+2)+1}$. Thus, all the assumptions needed for the global convergence of the original LMBM are satisfied provided that the computed subgradients belong to the subdifferential. The theorems on the convergence of the LMBM proved in [18, 27] can be modified for the RELU-NNR problems as follows.

THEOREM 4.1. *If the* LMBM *terminates after a finite number of iterations, say at iteration k, then the weight w_k^H is a stationary point of the* RELU-NNR *problem (3).*

THEOREM 4.2. *Every accumulation point $\bar{w}^H$ of the sequence $\{w_k^H\}$ generated by the* LMBM *is a stationary point of the* RELU-NNR *problem (3).*

5 Numerical experiments

Using some real-world data sets and performance measures, we evaluate the performance of the proposed LMBNNR algorithm and compare it with the well-known backpropagation NNR algorithm utilizing TensorFlow[2].

Data sets and performance measures. Information about the data sets is given in Table 1 and the references therein. The data sets are divided (randomly) to training (80%) and test (20%) sets. To get comparable results we use the same training and test sets with both the methods. We apply the following performance measures: root mean square error (RMSE), mean absolute error (MAE), coefficient of determination (R^2), and Pearson's correlation coefficient (r) (see Appendix for more details).

Used Software and parameters. Computational experiments are carried out on iMac (macOS Mojave 10.14.6), 4.0 GHz Intel(R) Core(TM) i7 machine with 16 GB of RAM. The proposed algorithm LMBNNR is implemented in Fortran 2003. The source code is available at `http://napsu.karmitsa/lmbnnr`. There is no tuneable parameters in the LMBNNR algorithm. Although we set the maximum number of nodes as

$$H_{\max} = \begin{cases} 200, & n \times m < 10^6 \\ 100, & \text{otherwise}, \end{cases} \tag{9}$$

where n and m are the number of samples and features, respectively. In addition, the LMBNNR algorithm always produces the results with the smaller number of nodes as a side product.

[2]`https://www.tensorflow.org/`

Table 1: Data

Data set	No. of samples	No. of features	Reference
Combined cycle power plant	9 568	5	[46, 31]
Airfoil self-noise	1 503	6	[12]
Concrete compressive strength	1 030	9	[48]
Physicochemical properties of protein tertiary structure	45 730	10	[12]
Boston housing data	506	14	[19]
SGEMM GPU kernel performance(1)	241 600	15	[5, 39]
MiniBooNE PID	130 064	50	[12]
Online news popularity	39 644	59	[15]
Residential building data set(1)	372	108	[41]
BlogFeedback(2)	52 397	281	[8]
ISOLET	7 797	618	[12]
Greenhouse gas observing network	2 921	5 232	[36]

(1) Used with the first output feature.

(2) Training data set.

The backpropagation NNR algorithm is implemented using TensorFlow in Google Colab. We use the RELU activation for the hidden layer, the linear activation for the output layer, and the MSE loss function. The reason to choose the MSE is that it usually works better with TensorFlow than the MAE used in the LMBNNR algorithm. This is probably due to the smoothness of the MSE. Naturally, with the proposed algorithm we do not have any problems with the nonsmoothness as it applies the nonsmooth optimization solver LMBM. In TensorFlow, the Keras optimizer SGD with the default parameters and the following three different combinations of *batch size* (the number of samples that will be propagated through the network) and *epochs* (the number of complete passes through the training data) are used[3]:

1. Mini-Batch Gradient Descent (TensorFlow1): batch size $= 32$ and no. of epochs $= 1$. These are the default values for TensorFlow;

2. Batch Gradient Descent (TensorFlow2): batch size = size of the training data and no. of epochs $= 1$. These choices mimic the proposed method;

3. SGD (TensorFlow3): batch size $= 1$ and no. of epochs $= 100$ for data sets with less than 100 000 samples and no. of epochs $= 10$ for a larger data. We reduce the number of epochs in the latter case due to a very long computational times and the fact that the larger number of epochs often leads to NaN loss function value. The aim of this version is to be as stochastic version of TensorFlow as possible.

[3] `https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/`

The number of hidden nodes is set to 2, 5, 10, 50, 100, 200, 500 and 5 000. We run TensorFlow for the ReLU-NNR problems with these numbers repeatedly. Due to the stochastic nature of the SGD used as an optimizer in TensorFlow, we run all the problems ten times and report the average, whereas in the proposed algorithm we solve the ReLU-NNR problem only once per each data.

Results and discussion. Next we present the results of our numerical experiments. The results are given in Tables 2–13 and in Appendix. In these tables we only provide the RMSE of the test set and the used CPU time. The results using other three performance measures are given in Appendix.

To evaluate the reliability of the ASP, we forced the LMBNNR algorithm to solve ReLU-NNR problems up to the maximum number of nodes. Nevertheless, in the tables we report both the results obtained using the maximum number of nodes and also applying the ASP. The best solution with respect to the test set's RMSE in the former case is denoted by "Best", and the solution obtained in the latter case is denoted by "ASP".

The best results (the smallest RSME for the test set) obtained with TensorFlow are given in bolded text to make the tables more illustrative in one sight. Naturally, in the real-world predictions we would not know which result is the best in either of the tested algorithms. To make the comparison more challenging to the proposed LMBNNR algorithm we mainly compare the result that it gives with the ASP to the best result obtained with any version of TensorFlow. Note that for TensorFlow, the results reported (including CPU times) are averaged over the ten runs unless some of the runs lead to NaN loss function values. In this case, the results are averaged over the successful runs. If there is "NaN" in the tables, then all ten runs lead to NaN solution. The results using other three evaluation criteria (MAE, CE, and r), given in Appendix, support the conclusions drawn from the RMSE although MAE often indicates slightly better performance of the LMBNNR algorithm than RMSE. This is probably due to the used loss function and/or the regularization term.

Table 2: RMSE for test set and CPU times in Combined cycle power plant data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	4.433	0.15	8.472	1.45	19.061	0.51	4.582	1036.42
5	4.267	0.64	6.091	0.97	20.547	0.58	4.415	790.27
10	4.216	2.13	5.784	0.78	17.809	0.47	4.278	650.95
50	4.137	39.81	4.712	1.04	18.790	0.66	4.215	679.02
100	4.140	145.54	4.612	0.65	17.213	0.38	**4.167**	**639.65**
200	4.128	623.60	4.566	0.66	17.649	0.69	4.190	691.76
500	–	–	4.534	1.00	16.514	0.63	4.168	761.36
5000	–	–	4.533	1.81	16.710	1.53	4.205	897.52
Best: $H = 198$	4.128	610.95						
ASP: $H = 54$	**4.137**	**45.45**						

Table 3: RMSE for test set and CPU times in Airfoil self-noise data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	4.907	0.03	6.756	0.36	10.129	0.29	4.387	102.65
5	4.407	0.11	6.580	0.35	9.465	0.30	3.688	102.13
10	4.410	0.32	5.888	0.35	8.950	0.33	2.804	101.01
50	4.344	7.46	5.595	0.34	7.114	0.30	2.386	100.87
100	1.818	33.51	5.516	0.40	6.840	0.33	**2.018**	**102.91**
200	1.804	124.42	5.346	0.38	6.967	0.32	2.044	108.73
500	–	–	5.319	0.35	6.743	0.30	2.194	116.60
5000	–	–	5.299	0.42	6.689	0.37	2.035	133.05
Best: $H = 196$	1.804	119.98						
ASP: $H = 107$	**1.818**	**37.50**						

Table 4: RMSE for test set and CPU times in Concrete compressive strength data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	11.407	0.02	17.554	0.37	30.305	0.33	7.659	70.00
5	10.869	0.10	15.736	0.37	23.220	0.32	6.372	74.19
10	6.665	0.36	15.156	0.38	23.949	0.36	5.577	71.78
50	6.429	5.75	13.729	0.39	17.824	0.35	4.918	74.42
100	6.095	23.43	13.277	0.39	18.069	0.31	5.019	72.87
200	6.095	61.74	12.472	0.36	17.365	0.31	4.807	76.91
500	–	–	12.379	0.40	16.515	0.42	4.771	79.92
5000	–	–	12.362	0.45	16.496	0.41	**4.468**	**94.33**
Best: $H = 72$	6.073	10.22						
ASP: $H = 17$	**6.666**	**0.84**						

Table 5: RMSE for test set and CPU times in Physicochemical properties of protein data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	5.304	1.15	5.223	2.29	7.838	0.71	5.127(1)	3368.54
5	5.095	5.24	5.121	2.14	8.366	0.72	NaN	–
10	5.062	18.81	5.082	2.09	8.168	0.73	**4.856**(2)	**3356.05**
50	4.795	383.64	4.975	2.05	6.784	0.73	NaN	–
100	4.795	924.16	4.952	1.86	6.466	0.69	NaN	–
200	4.790	3653.38	4.941	2.31	6.155	0.77	NaN	–
500	–	–	4.972	3.06	6.130	1.07	NaN	–
5000	–	–	4.942	5.79	6.043	4.68	NaN	–
Best: $H = 181$	4.790	2711.36						
ASP: $H = 62$	**4.795**	**557.15**						

(1) 2/10 runs led to NaN loss function value. The results reported are averaged over the existing eight solutions.
(2) 8/10 runs led to NaN loss function value. The results reported are averaged over the existing two solutions.

Table 6: RMSE for test set and CPU times in Boston housing data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	5.547	0.02	10.983	0.47	13.335	0.33	4.675	40.02
5	4.392	0.07	9.345	0.49	12.203	0.33	4.019	33.65
10	3.585	0.27	8.753	0.51	12.663	0.32	3.936	34.16
50	3.895	4.92	8.061	0.49	10.831	0.33	3.784	35.72
100	3.896	12.98	7.702	0.54	10.636	0.29	3.794	37.22
200	3.896	43.41	7.498	0.33	10.516	0.55	**3.609**	**38.10**
500	–	–	7.699	0.47	10.410	0.32	3.656	38.44
5000	–	–	7.625	0.50	9.956	0.37	3.669	47.45
Best: $H = 8$	3.462	0.17						
ASP: $H = 22$	**3.603**	**1.01**						

Table 7: RMSE for test set and CPU times in SGEMM GPU kernel performance data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	223.175	7.15	168.971	8.11	515.647	2.15	148.531	1536.53
5	116.352	30.07	128.186	7.98	549.201	2.18	105.851	1812.15
10	116.352	84.98	116.195	8.02	502.793	2.14	78.309	1839.19
50	77.631	2344.09	108.867	8.09	429.332	2.32	43.959	1669.78
100	75.102	9935.67	106.440	8.09	402.935	2.31	41.323	1809.20
200	–	–	104.307	8.44	385.360	2.52	37.311	1835.75
500	–	–	102.033	10.12	383.167	3.25	34.283	1679.94
5000	–	–	101.886	23.86	367.074	23.83	**28.524**	**2243.39**
Best: $H = 100$	75.102	9935.67						
ASP: $H = 8$	**116.352**	**62.77**						

Table 8: RMSE for test set and CPU times in MiniBooNE PID data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	23.257	14.07	35888.307	4.61	7538.550	1.36	NaN	–
5	14.992	62.92	5419340.282(1)	4.72	5109.595	1.34	NaN	–
10	12.969	204.85	1127040.353(2)	4.46	7753.198	1.35	NaN	–
50	9.565	3269.12	49334345.329(3)	4.54	5167.114	1.43	NaN	–
100	9.517	10983.33	89795765.713(3)	4.98	8497.345	1.62	NaN	–
200	–	–	NaN	–	5295.939	1.61	NaN	–
500	–	–	4089459.935(3)	5.80	3522.259	2.28	NaN	–
5000	–	–	NaN	–	**2849.433**	**15.03**	NaN	–
Best: $H = 95$	9.517	10200.99						
ASP: $H = 98$	**9.517**	**10668.32**						

(1) 2/10 runs led to NaN loss function value. The results are averaged over the existing eight solutions.
(2) 3/10 runs led to NaN loss function value. The results are averaged over the existing seven solutions.
(3) 9/10 runs led to NaN loss function value. The result of the one "successful" run is given here.

Table 9: RMSE for test set and CPU times in Online news popularity data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	12387.232	4.92	**12301.201**	**1.68**	15701.632	0.63	12475.792$^{(1)}$	2664.11
5	12377.297	23.32	12304.054	1.65	18562.269	0.65	NaN	–
10	12363.852	78.77	12351.966	1.62	18095.866	0.69	NaN	–
50	12349.854	1671.91	12670.856	1.77	16461.908	0.67	NaN	–
100	12345.489	6316.63	12714.957	1.78	14731.393	0.68	NaN	–
200	–	–	12937.682	1.80	13977.354	0.74	NaN	–
500	–	–	12519.334	2.10	13089.170	0.91	NaN	–
5000	–	–	12322.533	6.27	12364.057	3.52	NaN	–
Best: $H = 99$	12345.474	6218.10						
ASP: $H = 100$	**12345.489**	**6316.63**						

(1) 7/10 runs led to NaN loss function value. The results are averaged over the existing three solutions.

Table 10: RMSE for test set and CPU times in Residential building data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	238.303	0.09	1163.959	0.54	1635.840	0.32	236.870$^{(1)}$	25.92
5	239.517	0.39	1048.945	0.35	1350.816	0.33	192.679$^{(2)}$	25.93
10	153.462	1.47	1046.231	0.36	1524.374	0.31	155.138$^{(3)}$	26.91
50	156.246	27.89	1014.723	0.35	1217.393	0.32	**148.185**$^{(3)}$	**27.61**
100	156.246	77.18	998.603	0.36	1119.603	0.33	NaN	–
200	156.246	276.56	956.511	0.35	1018.663	0.31	NaN	–
500	–	–	972.945	0.35	998.178	0.33	NaN	–
5000	–	–	775.801	0.45	916.629	0.40	NaN	–
Best: $H = 10$	153.462	1.47						
ASP: $H = 20$	**154.350**	**5.80**						

(1) 2/10 runs led to NaN loss function value. The results reported are averaged over the existing eight solutions.
(2) 1/10 runs led to NaN loss function value. The results reported are averaged over the existing nine solutions.
(3) 4/10 runs led to NaN loss function value. The results reported are averaged over the existing six solutions.

Table 11: RMSE for test set and CPU times in BlogFeedback data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	28.725	20.74	NaN	–	NaN	–	NaN	–
5	27.611	99.42	NaN	–	NaN	–	NaN	–
10	26.832	351.79	NaN	–	NaN	–	NaN	–
50	25.090	8615.21	NaN	–	NaN	–	NaN	–
100	24.960	34350.20	NaN	–	NaN	–	NaN	–
200	–	–	NaN	–	NaN	–	NaN	–
500	–	–	NaN	–	NaN	–	NaN	–
5000	–	–	NaN	–	NaN	–	NaN	–
Best: $H = 100$	24.960	34350.20						
ASP: $H = 95$	**24.961**	**31135.05**						

Table 12: RMSE for test set and CPU times in ISOLET data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	4.896	12.08	5.324	0.71	9.246	0.42	NaN	–
5	4.068	57.26	4.909	0.68	11.008	0.43	NaN	–
10	4.112	203.95	4.761	0.68	9.896	0.42	NaN	–
50	4.332	4582.67	4.686	0.74	9.745	0.45	NaN	–
100	4.320	14882.22	4.662	0.78	9.654	0.46	NaN	–
200	–	–	**4.660**	**0.87**	9.043	0.50	NaN	–
500	–	–	4.833	1.20	9.227	0.66	NaN	–
5000	–	–	NaN	–	8.221	2.900	NaN	–
Best: $H = 6$	4.065	79.48						
ASP: $H = 84$	**4.320**	**12411.66**						

Table 13: RMSE for test set and CPU times in Greenhouse gas observing network data.

	LMBNNR		TensorFlow1		TensorFlow2		TensorFlow3	
H	RMSE	CPU	RMSE	CPU	RMSE	CPU	RMSE	CPU
2	23.170	37.25	**67.654**(1)	0.70	362.121	0.58	NaN	–
5	23.398	179.15	NaN	–	315.623	0.54	NaN	–
10	22.896	675.49	NaN	–	506.358	0.59	NaN	–
50	23.629	10944.60	NaN	–	488.828	0.84	NaN	–
100	23.630	32018.64	NaN	–	455.618	0.68	NaN	–
200	–	–	NaN	–	493.253	0.80	NaN	–
500	–	–	NaN	–	576.689	1.55	NaN	–
5000	–	–	NaN	–	1189.710	10.83	NaN	–
Best: $H = 9$	22.851	552.48						
ASP: $H = 36$	**23.629**	**7615.43**						

(1) 4/10 runs led to NaN loss function value. The results reported are averaged over the existing six solutions.

The predictions with the proposed LMBNNR algorithm in termination are better than those of any version of TensorFlow in 8 data sets out of 12. Namely, in Combined cycle power plant, Airfoil self-noise, Physicochemical properties of protein tertiary structure, Boston housing, MinBooNE PID, BlogFeedback, ISOLET, and Greenhouse gas observing network data.

In other 4 data sets the results obtained by TensorFlow are slightly better than those by LMBNNR. In Online news popularity the LMBNNR algorithm predicts worse than TensorFlow1 with 2 or 5 nodes (see Table 9). However, the deviation in the predictions given by TensorFlow1 as well as the worse predictions obtained with more nodes, and the comparison with respect to MAE (see Appendix Table 21) make the proposed algorithm an advisable choice also in this data set.

In Residential building data the prediction obtained with TensorFlow3 with 50 nodes is slightly more accurate than the one obtained with LMBNNR (see Table 10). However, 4 of 10 runs with TensorFlow3 with 50 nodes leads to NaN solution, thus it may easily happen that instead of an accurate solution one gets no solution at all. In addition, the other evaluation criteria indicate either the better or an equal prediction with LMBNNR to TensorFlow (see Appendix Table 22).

In Concrete compressive strength data the results obtained with LMBNNR are worse than the best results obtained by TensorFlow3 (see Table 4), and it seems that the ASP would have caused a premature termination. We will analyse this termination later on. Nevertheless, it is worth of noting that LMBNNR (with and without termination) produces clearly more accurate predictions than TensorFlow1 and TensorFlow2.

Since our hypothesis is that the proposed LMBNNR algorithm performs the best in the large-scale data sets, the result in SGEMM GPU kernel performance is a disappointment (see Table 7). The smallest RMSE for the test set obtained with the proposed algorithm is considerably more than that of TensorFlow3. Moreover, the ASP would have caused premature termination leading to the prediction that is less accurate than that of TensorFlow1 (with more than 10 nodes). Without the ASP the predictions obtained with LMBNNR (with more than 10 nodes) are more accurate than those of TensorFlow1 and TensorFlow2. Nevertheless, with respect to the MAE the LMBNNR algorithm is ranked next to TensorFlow3 even with the stopping procedure (see Appendix Table 19).

The results of our experiments show that overall the ASP works fairly well. In MiniBooNE PID, Online news popularity[4] and BlogFeedback data sets the ASP triggers within a few iterations after the best solution is obtained. In addition, in Combined cycle power plant, Airfoil self-noise, Physicochemical properties of protein, Boston housing, and Residential building data sets the predictions obtained applying the ASP are very close to the best predictions obtained. However, there is a premature termination with Concrete compressive strength and SGEMM GPU kernel performance data sets (see Tables 4 and 7). The closer look to predictions produced by the LMBNNR algorithm in the former data shows that, indeed, there is a small increase in the RMSE values of the test set from $H = 10$ to $H = 22$, which may explain the premature termination. In the latter data adding nodes between $H = 5$ and $H = 21$ does not affect the prediction at all.

On the other hand, there are delayed terminations in ISOLET and Greenhouse gas observing network data. The late termination is probably caused by the small maximum number of iterations used in the optimization procedure LMBM, which means that it is easy to diminish the objective function value also in the subsequent iterations of the LMBNNR algorithm. However, increasing this number would easily lead to overfitting. Therefore, we can conclude that the ASP works best with the small, medium-sized and relatively large number of features, and we recommend to use the smaller maximum number of nodes than used here with a very large number of features. In both ISOLET and Greenhouse gas observing network data the predictions obtained with LMBNNR are good already with ten nodes.

TensorFlow3 is the most accurate version of TensorFlow when the number of input features is relatively low (see Tables 2–7). However, it usually requires more computational time than the proposed LMBNNR. Moreover, TensorFlow3 fails almost always

[4]To be precise, in Online news popularity data the computation stopped due to maximum number of nodes. The ASP would have triggered with 101 nodes.

when the number of input features is large (see Tables 8–13). Therefore, we can conclude that LMBNNR clearly outperforms this version of TensorFlow.

With TensorFlow2 we almost always get some results out in a short time, however the predictions are not accurate. Obviously, these parameter choices are not suitable for TensorFlow and the main reason to keep this version here is the pure theoretical. Thus, we can easily claim that LMBNNR outperforms this version of TensorFlow.

TensorFlow1 uses the default parameters. It is in general very efficient method using only few seconds to solve an individual ReLU-NNR problem. However, it is able to produce "good enough" predictions in only four data sets out of 12. Out of these four, TensorFlow1 produces (at least with some numbers of nodes) more accurate prediction results than those obtained by the LMBNNR only in two data sets. Therefore, again we can conclude that the proposed algorithm outperforms this version of TensorFlow.

Finally, the results demonstrate that all the tested versions of TensorFlow either fail or give very inaccurate predictions in data sets with large number of samples and/or features like in MiniBooNE PID, BlogFeedback, and Greenhouse gas observation network data while LMBNNR works fine.

It is also worth of noting that finding good hyperparameters for TensorFlow is time consuming and may require a separate validation set to be used. For instant, if we just run TensorFlow with two different numbers of nodes, the CPU time is doubled — not to mention the time needed to prepare the separate runs and validate the results. On the other hand, with LMBNNR all results are obtained within the computational time of the largest number of nodes without the need of separate runs. In addition, there is no need to use a separate validation set to select good hyperparameters in LMBNNR.

6 Conclusions

In this paper, a new neural networks algorithm is proposed to solve regression problems in large data sets. The regression problem is modeled using the neural network with one hidden layer, the piecewise linear activation function known as ReLU, and the L_1-based loss function. Since the loss function is nonsmooth and nonconvex we apply the limited memory bundle method to minimize it. We utilize this method as it is capable of handling large dimensions, nonconvexity, and nonsmoothness very efficiently.

The proposed neural networks algorithm for regression requires no hyperparameter tuning. It starts with one hidden node and gradually adds more nodes at each iteration. The algorithm terminates when the loss function value cannot be improved in several successive iterations or the maximum number of hidden nodes is reached.

The new algorithm is tested using 12 large real-world data sets and compared with the backpropagation algorithm using TensorFlow. In particular, we tested different hyperparameter settings in TensorFlow and used its best results in comparison. The results show that the proposed algorithm clearly outperforms TensorFlow. It provides a better prediction in eight out of 12 data sets and an approximately similar prediction in two out of 12 data sets. Only in two data sets the prediction produced by the proposed algorithm is worse than the best prediction produced by TensorFlow. Although some versions of TensorFlow are notably fast in comparison with the new algorithm, their prediction results are significantly worse. Therefore we can conclude that the proposed algorithm is accurate and efficient for solving regression problems both with large number of samples and large number of input features.

Results presented in this paper show that the use of simple but nonsmooth activation functions in neural networks together with powerful nonsmooth optimization methods can lead to the development of accurate and efficient hyperparameter-free neural network algorithms. The approach proposed in this paper can be extended to model neural networks for classification (supervised learning problems) and to build neural networks with more than one hidden layer in order to develop robust and effective deep learning algorithms. These will be subjects of future research.

Acknowledgments

This work was financially supported by Academy of Finland grants #289500 and #319274, and by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (Project No. DP190100580).

References

[1] AGGARWAL, C. *Neural Networks and Deep Learning.* Springer-Verlag, Berlin, 2018.

[2] AIROLA, A., AND PAHIKKALA, T. Fast kronecker product kernel methods via generalized vec trick. *IEEE Transactions on Neural Networks and Learning Systems 29*, 8 (2018), 3374–3387.

[3] BAGIROV, A. M., KARMITSA, N., AND MÄKELÄ, M. M. *Introduction to Nonsmooth Optimization: Theory, Practice and Software.* Springer, 2014.

[4] BAGIROV, A. M., KARMITSA, N., AND TAHERI, S. *Partitional Clustering via Nonsmooth Optimization: Clustering via Optimization.* Springer, 2020.

[5] BALLESTER-RIPOLL, R., PAREDES, E., AND PAJAROLA, R. Sobol tensor trains for global sensitivity analysis. In arXiv Computer Science / Numerical Analysis e-prints., 2017. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (September 21st, 2020).

[6] BIHAIN, A. Optimization of upper semidifferentiable functions. *Journal of Optimization Theory and Applications 4*, 4 (1984), 545–568.

[7] BROOMHEAD, D., AND LOWE, D. *Radial Basis Functions, Multi-Variable Functional Interpolation and Adaptive Networks*. Malvern, Worcs.: Royals Signals and Radar Establishment, 1988.

[8] BUZA, K. Feedback prediction for blogs. In Data Analysis, Machine Learning and Knowledge Discovery (pp. 145-152). Springer International Publishing., 2014. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (June 11th, 2016).

[9] BYRD, R. H., NOCEDAL, J., AND SCHNABEL, R. B. Representations of quasi-Newton matrices and their use in limited memory methods. *Mathematical Programming 63* (1994), 129–156.

[10] CLARKE, F. H. *Optimization and Nonsmooth Analysis*. Wiley-Interscience, New York, 1983.

[11] DAVIES, D., DRUSVYATSKIY, D., KAKADE, S., AND LEE, J. Stochastic subgradient method converges on tame functions. *Foundations of Computational Mathematics 20* (2020), 119–154.

[12] DUA, D., AND KARRA TANISKIDOU, E. UCI machine learning repository. Available online at <URL: `http://archive.ics.uci.edu/ml`>, University of California, Irvine, School of Information and Computer Sciences, 2017. (November 25th, 2020).

[13] EVANS, L. C., AND GARIEPY, R. F. *Measure theory and fine properties of functions*. CRC Press, Boca Raton, FL, 1992.

[14] FARIS, H., MIRJALILI, S., AND ALJARAH, I. Automatic selection of hidden neurons and weights in neural networks using grey wolf optimizer based on a hybrid encoding scheme. *International Journal of Machine Learning and Cybernetics 10* (2019), 2901–2920.

[15] FERNANDES, K., VINAGRE, P., AND CORTEZ, P. A proactive intelligent decision support system for predicting the popularity of online news. Proceedings of the 17th EPIA 2015 — Portuguese Conference on Artificial Intelligence, September, Coimbra, Portugal., 2015. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (June 11th, 2016).

[16] GENG, Z., AND WANG, Y. Automated design of a convolutional neural network with multi-scale filters for cost-efficient seismic data classification. *Nature Communications 11*, 3311 (2020).

[17] HAARALA, M., MIETTINEN, K., AND MÄKELÄ, M. M. New limited memory bundle method for large-scale nonsmooth optimization. *Optimization Methods and Software 19*, 6 (2004), 673–692.

[18] HAARALA, N., MIETTINEN, K., AND MÄKELÄ, M. M. Globally convergent limited memory bundle method for large-scale nonsmooth optimization. *Mathematical Programming 109*, 1 (2007), 181–205.

[19] HARRISON, D., AND RUBINFELD, D. Hedonic prices and the demand for clean air. *Journal of Environmental Economics and Management 5* (1978), 81–102. Data set available in Kaggle <URL: `https://www.kaggle.com/c/boston-housing/`> (Sepember 21st, 2020).

[20] HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2007.

[21] HINTON, G. A practical guide to training restricted Boltzmann machines. In *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 599–619.

[22] HUANG, D.-S., AND DU, J.-X. A constructive hybrid structure optimization methodology for radial basis probabilistic neural networks. *IEEE Transactions on Neural Networks 19*, 12 (2008), 2099–2115.

[23] HUANG, G.-B. Learning capability and storage capacity of two-hidden-layer feedforward networks. *IEEE Transactions on Neural Networks 14*, 2 (2003), 274–281.

[24] IBNU, C. R. M., SANTOSO, J., AND SURENDRO, K. Determining the neural network topology: A review. In *Proceedings of the 2019 8th International Conference on Software and Computer Applications* (New York, NY, USA, 2019), ICSCA '19, Association for Computing Machinery, pp. 357–362.

[25] IMAIZUMI, M., AND FUKUMIZU, K. Deep neural networks learn non-smooth functions effectively. In *Proceedings of Machine Learning Research* (2019), K. Chaudhuri and M. Sugiyama, Eds., vol. 89, PMLR, pp. 869–878.

[26] KÄRKKÄINEN, T., AND HEIKKOLA, E. Robust formulations for training multilayer perceptrons. *Neural Computation 16*, 4 (2004), 837–862.

[27] KARMITSA, N. Limited memory bundle method and its variations for large-scale nonsmooth optimization. In *Numerical Nonsmooth Optimization: State of the Art Algorithms*, A. M. Bagirov, M. Gaudioso, N. Karmitsa, M. M. Mäkelä, and S. Taheri, Eds. Springer, 2020, pp. 167–200.

[28] KARMITSA, N., BAGIROV, A., TAHERI, S., AND JOKI, K. Limited memory bundle method for clusterwise linear regression. In *Computational Sciences and Artificial Intelligence in Industry*, T. Tuovinen, J. Periaux, and P. Neittaanmäki, Eds. Springer, in-press, 2020.

[29] KARMITSA, N., BAGIROV, A. M., AND TAHERI, S. Clustering in large data sets with the limited memory bundle method. *Pattern Recognition 83* (2018), 245–259.

[30] KARMITSA, N., TAHERI, S., BAGIROV, A. M., AND MÄKINEN, P. Missing value imputation via clusterwise linear regression. *IEEE Transactions on Knowledge and Data Engineering* (2020), in–press.

[31] KAYA, H., TÜFEKCI, P., AND GÜRGEN, S. F. Local and global learning methods for predicting power of a combined gas & steam turbine. In Proceedings of the International Conference on Emerging Trends in Computer and Electronics Engineering ICETCEE 2012, pp. 13–18, March, Dubai., 2012. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (June 11th, 2016).

[32] KIWIEL, K. C. *Methods of Descent for Nondifferentiable Optimization*. Lecture Notes in Mathematics 1133. Springer-Verlag, Berlin, 1985.

[33] LAROCHELLE, H., ERHAN, D., COURVILLE, A., BERGSTRA, J., AND BENGIO, Y. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of ICML* (2007), vol. 227, pp. 473–480.

[34] LECUN, Y., BOTTOU, L., ORR, G., AND MULLER, K.-R. Efficient backprop. In *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer Berlin Heidelberg, 2012, pp. 9–48.

[35] LEUNG, F.-F., LAM, H.-K., LING, S.-H., AND TAM, P.-S. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural Networks 11*, 1 (2003), 79–88.

[36] LUCAS, D., YVER KWOK, C., CAMERON-SMITH, P., GRAVEN, H., BERGMANN, D., GUILDERSON, T., WEISS, R., AND KEELING, R. Designing optimal greenhouse gas observing networks that consider performance and cost. *Geoscientific Instrumentation Methods and Data Systems 4* (2015), 121–137. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (Sepember 21st, 2020).

[37] MALTE, J. Artificial neural network regression models in a panel setting: Predicting economic growth. *Economic Modelling 91* (2020), 148–154.

[38] MAREN, A., HARSTON, C., AND PAP, R. *Handbook of Neural Computing Applications*. Academic Press, 2014.

[39] NUGTEREN, C., AND CODREANU, V. Cltune: A generic auto-tuner for opencl kernels. In MCSoC: 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip. IEEE., 2015. Data set available in UCI

machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (September 21st, 2020).

[40] ODIKWA, H., IFEANYI-REUBEN, N., AND THOM-MANUEL, O. M. An improved approach for hidden nodes selection in artificial neural network. *International Journal of Applied Information Systems (IJAIS) 12*, 17 (2020), 7–14.

[41] RAFIEI, M., AND ADELI, H. A novel machine learning model for estimation of sale prices of real estate units. *ASCE, Journal of Construction Engineering & Management 142*, 2 (2015). Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (September 21st, 2020).

[42] REED, R., AND MARKS, R. J. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. The MIT Press, 1998.

[43] RUMELHART, D., HINTON, G., AND WILLIAMS, R. Learning representations by back-propagating errors. *Nature 323* (1988), 533–536.

[44] SELMIC, R. R., AND LEWIS, F. L. Neural-network approximation of piecewise continuous functions: Application to friction compensation. *IEEE Transactions on Neural Networks 13*, 3 (2002), 745–751.

[45] TSAI, J.-T., CHOU, J.-H., AND LIU, T.-K. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Transactions on Neural Networks 17*, 1 (2006), 69–80.

[46] TÜFEKCI, P. Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods. *International Journal of Electrical Power & Energy Systems 60* (2014), 126–140. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (June 11th, 2016).

[47] WILAMOWSKI, B. M. Neural network architectures. In *The Industrial Electronics Handbook*. CRC Press, 2011.

[48] YEH, I. Modeling of strength of high performance concrete using artificial neural networks. *Cement and Concrete Research 28*, 12 (1998), 1797–1808. Data set available in UCI machine learning repository <URL: `http://archive.ics.uci.edu/ml`> (June 11th, 2016).

Appendix

Performance measures. We now describe the performance measures used in our evaluations and comparisons. Let $e_1, \dots, e_n$ for $n \geq 1$ be actual observed values for some parameter e and $\bar{e}_1, \dots, \bar{e}_n$ be their forecast values. We use the following performance measures:

- *root mean square error*:

$$\mathrm{RMSE} = \Big(\frac{1}{n}\sum_{i=1}^{n}(\bar{e}_i - e_i)^2\Big)^{1/2};$$

- *mean absolute error*:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\bar{e}_i - e_i|;$$

- *coefficient of determination*:

$$R^2 = 1 - \Big(\frac{\sum_{i=1}^{n}(e_i - \bar{e}_i)^2}{\sum_{i=1}^{n}(e_i - e_0)^2}\Big);$$

- *Pearson's correlation coefficient*:

$$r = \frac{\sum_{i=1}^{n}(e_i - e_0)(\bar{e}_i - \bar{e}_0)}{\Big(\sum_{i=1}^{n}(e_i - e_0)^2\sum_{i=1}^{m}(\bar{e}_i - \bar{e}_0)^2\Big)^{1/2}}.$$

In the R^2 and r measures, e_0 is the mean of observed values. The small values of RMSE and MAE measures indicate small deviations of the predictions from actual observations. The R^2 measure ranges from $-\infty$ to 1: $R^2 = 1$ means a perfect prediction, $R^2 = 0$ indicates that the model predictions are as accurate as the mean of the observed data and an efficiency $-\infty < R^2 < 0$ occurs when the observed mean is a better predictor than the model. In the measure r, $\bar{e}_0$ is the mean of forecast values. The range of r is from -1 to 1: $r = 1$ implies that a linear equation describes the relationship between observed and forecast values perfectly, $r = -1$ means that all the samples lie on a line for which a forecast value decreases as an observed value increases and $r = 0$ happens when there is no linear correlation between these values.

Table 14: MAE, R^2 and r for test set in Combined cycle power plant data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	3.487	6.703	15.910	3.634
5	3.342	4.807	16.650	3.470
10	3.176	4.563	14.706	3.332
50	3.076	3.727	16.093	3.226
100	3.068	3.638	14.870	3.182
200	3.059	3.605	15.118	3.221
500	–	3.582	14.315	3.198
5000	–	3.571	14.529	3.249
Best: H=198	3.059			
Term: H=54	3.076			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.934	0.750	–0.256	0.930
5	0.939	0.874	–0.618	0.935
10	0.941	0.887	–0.094	0.939
50	0.943	0.926	–0.213	0.941
100	0.943	0.929	0.004	0.942
200	0.943	0.930	–0.046	0.941
500	–	0.931	0.085	0.942
5000	–	0.931	0.066	0.941
Best: H=198	0.943			
Term: H=54	0.943			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.967	0.867	0.108	0.967
5	0.969	0.935	0.169	0.969
10	0.970	0.942	0.087	0.970
50	0.971	0.962	–0.204	0.972
100	0.971	0.964	0.110	0.972
200	0.971	0.965	–0.042	0.972
500	–	0.965	0.491	0.972
5000	–	0.965	0.738	0.972
Best: H=198	0.971			
Term: H=54	0.971			

Table 15: MAE, R^2 and r for test set in Airfoil self-noise data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	3.519	5.469	7.868	3.308
5	3.010	5.317	7.552	2.819
10	3.011	4.755	7.082	2.187
50	2.921	4.490	5.712	1.833
100	1.306	4.440	5.567	1.549
200	1.295	4.318	5.630	1.580
500	–	4.297	5.454	1.683
5000	–	4.294	5.417	1.532
Best: H=196	1.295			
Term: H=107	1.306			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.467	–0.013	–1.414	0.570
5	0.570	0.034	–1.026	0.696
10	0.569	0.228	–0.851	0.822
50	0.582	0.303	–0.125	0.871
100	0.927	0.325	–0.040	0.909
200	0.928	0.366	–0.080	0.907
500	–	0.373	–0.008	0.890
5000	–	0.378	0.009	0.908
Best: H=196	0.928			
Term: H=107	0.927			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.699	0.033	0.063	0.769
5	0.768	0.245	0.041	0.849
10	0.768	0.478	–0.035	0.911
50	0.784	0.593	0.096	0.940
100	0.963	0.663	0.108	0.956
200	0.964	0.711	–0.012	0.957
500	–	0.723	0.054	0.949
5000	–	0.726	0.164	0.956
Best: H=196	0.964			
Term: H=107	0.963			

Table 16: MAE, R^2 and r for test set in Concrete compressive strength data.

MAE				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	8.611	14.271	22.520	6.099
5	8.349	12.909	18.357	4.991
10	5.167	12.493	18.999	4.286
50	4.851	11.401	14.427	3.811
100	4.505	11.090	14.663	3.866
200	4.505	10.364	14.168	3.680
500	–	10.315	13.593	3.622
5000	–	10.332	13.536	3.425
Best: H=72	4.511			
Term: H=17	5.167			
R^2				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.534	–0.118	–2.544	0.787
5	0.577	0.102	–1.011	0.854
10	0.841	0.170	–1.101	0.888
50	0.852	0.323	–0.153	0.912
100	0.867	0.367	–0.181	0.908
200	0.867	0.442	–0.089	0.915
500	–	0.451	0.021	0.918
5000	–	0.452	0.025	0.928
Best: H=72	0.868			
Term: H=17	0.841			
r				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.767	0.116	–0.017	0.889
5	0.784	0.351	–0.048	0.930
10	0.917	0.441	0.021	0.946
50	0.924	0.594	0.141	0.959
100	0.931	0.655	–0.008	0.959
200	0.931	0.722	0.012	0.962
500	–	0.743	0.181	0.963
5000	–	0.751	0.237	0.967
Best: H=72	0.932			
Term: H=17	0.917			

Table 17: MAE, R^2 and r for test set in Physicochemical properties of protein data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	3.941	4.399	6.651	4.265(1)
5	3.756	4.263	6.723	NaN
10	3.743	4.233	6.587	3.800(2)
50	3.493	4.118	5.732	NaN
100	3.493	4.073	5.710	NaN
200	3.484	4.084	5.424	NaN
500	–	4.122	5.471	NaN
5000	–	4.092	5.423	NaN
Best: H=181	3.484			
Term: H=62	3.493			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.238	0.260	–0.777	0.287(1)
5	0.297	0.289	–0.979	NaN
10	0.305	0.300	–0.844	0.361(2)
50	0.377	0.329	–0.263	NaN
100	0.377	0.335	–0.137	NaN
200	0.378	0.338	–0.027	NaN
500	–	0.330	–0.019	NaN
5000	–	0.338	0.010	NaN
Best: H=181	0.378			
Term: H=62	0.377			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.539	0.511	–0.045	0.551(1)
5	0.583	0.539	–0.008	NaN
10	0.587	0.550	–0.001	0.611(2)
50	0.634	0.577	–0.027	NaN
100	0.634	0.581	–0.010	NaN
200	0.636	0.583	0.055	NaN
500	–	0.579	0.031	NaN
5000	–	0.585	0.168	NaN
Best: H=181	0.636			
Term: H=62	0.634			

(1) 2/10 runs led to NaN loss function value.
(2) 8/10 runs led to NaN loss function value.

Table 18: MAE, R^2 and r for test set in Boston housing data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	3.291	7.388	9.626	3.265
5	2.582	6.662	9.265	2.750
10	2.372	6.229	9.371	2.693
50	2.624	5.519	7.685	2.622
100	2.624	5.163	7.412	2.619
200	2.624	4.904	6.955	2.528
500	–	4.966	7.086	2.532
5000	–	4.917	6.681	2.494
Best: H=8	2.297			
Term: H=22	2.371			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.704	–0.195	–0.783	0.785
5	0.815	0.141	–0.513	0.841
10	0.877	0.239	–0.608	0.849
50	0.854	0.366	–0.145	0.861
100	0.854	0.425	–0.120	0.861
200	0.854	0.455	–0.085	0.874
500	–	0.425	–0.051	0.870
5000	–	0.437	0.040	0.869
Best: H=8	0.885			
Term: H=22	0.875			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.853	0.127	0.067	0.901
5	0.904	0.470	0.298	0.922
10	0.936	0.477	0.042	0.923
50	0.924	0.640	0.099	0.931
100	0.924	0.685	0.169	0.931
200	0.924	0.720	0.138	0.937
500	–	0.698	0.132	0.936
5000	–	0.712	0.542	0.934
Best: H=8	0.941			
Term: H=22	0.936			

Table 19: MAE, R^2 and r for test set in SGEMM GPU kernel performance data.

MAE				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	84.180	106.274	332.182	97.193
5	53.063	75.854	403.268	63.499
10	53.063	63.318	360.189	45.560
50	31.944	61.959	286.637	28.370
100	30.713	61.185	267.953	28.346
200	–	59.193	246.155	24.309
500	–	57.267	221.834	23.878
5000	–	56.482	211.507	19.731
Best: H=100	30.713			
Term: H=8	53.063			
R^2				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.634	0.768	–1.038	0.838
5	0.901	0.878	–1.273	0.916
10	0.901	0.900	–0.892	0.954
50	0.956	0.913	–0.366	0.986
100	0.959	0.917	–0.195	0.987
200	–	0.920	–0.092	0.990
500	–	0.924	–0.079	0.991
5000	–	0.924	0.011	0.994
Best: H=100	0.959			
Term: H=8	0.901			
r				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.825	0.870	0.004	0.924
5	0.951	0.938	–0.016	0.960
10	0.951	0.949	0.005	0.981
50	0.978	0.956	0.007	0.993
100	0.979	0.958	0.007	0.994
200	–	0.960	0.093	0.996
500	–	0.962	–0.091	0.996
5000	–	0.962	0.125	0.997
Best: H=100	0.979			
Term: H=8	0.951			

Table 20: MAE, R^2 and r for test set in MiniBooNE PID data.

MAE				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	1.888	229.634	63.036	NaN
5	1.073	33606.759(1)	48.827	NaN
10	0.816	6994.444(2)	64.216	NaN
50	0.583	305887.491(3)	47.492	NaN
100	0.580	556756.811(3)	64.154	NaN
200	–	NaN	43.483	NaN
500	–	25361.646(3)	31.088	NaN
5000	–	NaN	26.627	NaN
Best: H=95	0.580			
Term: H=99	0.580			

R^2				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.851	–702995.620	–32493.587	NaN
5	0.938	–60903087504.947(1)	–12182.922	NaN
10	0.954	–2103154092.975(2)	–19041.354	NaN
50	0.975	–669919601352.566 (3)	–11304.678	NaN
100	0.975	–2219400412915.230(3)	–28118.869	NaN
200	–	NaN	–10831.922	NaN
500	–	–4603157782.712(3)	–5287.546	NaN
5000	–	NaN	–2588.915	NaN
Best: H=95	0.975			
Term: H=99	0.975			

r				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.995	0.000	0.003	NaN
5	0.996	0.000(1)	–0.058	NaN
10	0.996	0.000(2)	0.005	NaN
50	0.997	0.000(3)	0.026	NaN
100	0.997	0.000(3)	0.016	NaN
200	–	NaN	0.032	NaN
500	–	0.000(3)	0.048	NaN
5000	–	NaN	0.044	NaN
Best: H=95	0.997			
Term: H=99	0.997			

(1) 2/10 runs led to NaN loss function value.
(2) 3/10 runs led to NaN loss function value.
(3) 9/10 runs led to NaN loss function value.

Table 21: MAE, R^2 and r for test set in Online news popularity data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	2460.430	3333.375	7630.585	3800.756(1)
5	2453.002	3249.126	10810.528	NaN
10	2446.060	3464.855	10630.898	NaN
50	2476.646	3642.130	9047.129	NaN
100	2502.860	3632.464	7357.324	NaN
200	–	4408.260	6285.794	NaN
500	–	3609.625	4917.437	NaN
5000	–	3484.741	3420.317	NaN
Best: H=99	2502.809			
Term: H=100	2502.860			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	–0.016	–0.002	–0.690	–0.031(1)
5	–0.014	–0.002	–1.331	NaN
10	–0.012	–0.010	–1.209	NaN
50	–0.009	–0.065	–0.806	NaN
100	–0.009	–0.073	–0.440	NaN
200	–	–0.112	–0.298	NaN
500	–	–0.038	–0.134	NaN
5000	–	–0.005	–0.012	NaN
Best: H=99	–0.009			
Term: H=100	–0.009			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.109	0.042	0.003	0.092(1)
5	0.120	0.066	0.004	NaN
10	0.124	0.074	–0.014	NaN
50	0.117	0.068	0.005	NaN
100	0.111	0.066	–0.001	NaN
200	–	0.063	–0.004	NaN
500	–	0.081	–0.006	NaN
5000	–	0.114	0.011	NaN
Best: H=99	0.111			
Term: H=100	0.111			

(1) 7/10 runs led to NaN loss function value.

Table 22: MAE, R^2 and r for test set in Residential building data.

MAE				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	121.323	787.701	1161.594	149.070(1)
5	122.460	745.838	998.809	126.355(2)
10	87.085	746.882	1179.940	105.905(3)
50	94.572	695.736	850.053	94.044(3)
100	94.572	656.278	782.288	NaN
200	94.572	627.105	708.014	NaN
500	–	607.388	655.423	NaN
5000	–	492.349	568.573	NaN
Best: H=10	87.085			
Term: H=20	93.870			
R^2				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.966	0.169	–0.714	0.964(1)
5	0.966	0.330	–0.107	0.977(2)
10	0.986	0.330	–0.427	0.985(3)
50	0.985	0.375	0.112	0.987(3)
100	0.985	0.392	0.241	NaN
200	0.985	0.440	0.376	NaN
500	–	0.368	0.404	NaN
5000	–	0.636	0.499	NaN
Best: H=10	0.986			
Term: H=20	0.986			
r				
H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.985	0.456	0.320	0.985(1)
5	0.985	0.593	0.280	0.990(2)
10	0.994	0.597	0.160	0.993(3)
50	0.994	0.615	0.466	0.994(3)
100	0.994	0.690	0.547	NaN
200	0.994	0.704	0.632	NaN
500	–	0.673	0.641	NaN
5000	–	0.852	0.708	NaN
Best: H=10	0.994			
Term: H=20	0.994			

(1) 2/10 runs led to NaN loss function value.
(2) 1/10 runs led to NaN loss function value.
(3) 4/10 runs led to NaN loss function value.

Table 23: MAE, R^2 and r for test set in BlogFeedback data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	5.757	NaN	NaN	NaN
5	5.556	NaN	NaN	NaN
10	5.420	NaN	NaN	NaN
50	5.203	NaN	NaN	NaN
100	5.197	NaN	NaN	NaN
200	–	NaN	NaN	NaN
500	–	NaN	NaN	NaN
5000	–	NaN	NaN	NaN
Best: H=100	5.197			
Term: H=95	5.197			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.370	NaN	NaN	NaN
5	0.418	NaN	NaN	NaN
10	0.451	NaN	NaN	NaN
50	0.520	NaN	NaN	NaN
100	0.525	NaN	NaN	NaN
200	–	NaN	NaN	NaN
500	–	NaN	NaN	NaN
5000	–	NaN	NaN	NaN
Best: H=100	0.525			
Term: H=95	0.525			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.635	NaN	NaN	NaN
5	0.672	NaN	NaN	NaN
10	0.693	NaN	NaN	NaN
50	0.730	NaN	NaN	NaN
100	0.733	NaN	NaN	NaN
200	–	NaN	NaN	NaN
500	–	NaN	NaN	NaN
5000	–	NaN	NaN	NaN
Best: H=100	0.733			
Term: H=95	0.733			

Table 24: MAE, R^2 and r for test set in ISOLET data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	3.669	4.270	7.494	NaN
5	2.935	3.831	8.759	NaN
10	2.988	3.665	7.982	NaN
50	3.184	3.642	7.888	NaN
100	3.175	3.607	7.815	NaN
200	–	3.605	7.301	NaN
500	–	3.784	7.502	NaN
5000	–	NaN	6.648	NaN
Best: H=6	2.925			
Term: H=84	3.175			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.550	0.463	–0.682	NaN
5	0.689	0.547	–1.444	NaN
10	0.682	0.574	–0.850	NaN
50	0.647	0.587	–0.791	NaN
100	0.649	0.591	–0.754	NaN
200	–	0.592	–0.539	NaN
500	–	0.558	–0.666	NaN
5000	–	NaN	–0.315	NaN
Best: H=6	0.690			
Term: H=84	0.649			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.755	0.676	0.050	NaN
5	0.838	0.743	0.118	NaN
10	0.838	0.763	0.086	NaN
50	0.826	0.774	0.147	NaN
100	0.827	0.779	0.142	NaN
200	–	0.782	0.210	NaN
500	–	0.800	0.242	NaN
5000	–	NaN	0.392	NaN
Best: H=6	0.840			
Term: H=84	0.827			

Table 25: MAE, R^2 and r for test set in Greenhouse gas observing network data.

MAE

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	17.584	32.923$^{(1)}$	238.605	NaN
5	17.802	NaN	204.267	NaN
10	17.397	NaN	313.008	NaN
50	17.890	NaN	294.423	NaN
100	17.890	NaN	278.436	NaN
200	–	NaN	300.219	NaN
500	–	NaN	325.884	NaN
5000	–	NaN	664.811	NaN
Best: H=8	17.584			
Term: H=36	17.890			

R^2

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.950	0.509$^{(1)}$	–20.987	NaN
5	0.949	NaN	–11.939	NaN
10	0.952	NaN	–27.330	NaN
50	0.948	NaN	–24.579	NaN
100	0.948	NaN	–19.414	NaN
200	–	NaN	–25.177	NaN
500	–	NaN	–33.636	NaN
5000	–	NaN	–147.049	NaN
Best: H=8	0.951			
Term: H=36	0.948			

r

H	LMBNNR	TensorFlow1	TensorFlow2	TensorFlow3
2	0.975	0.700$^{(1)}$	0.557	NaN
5	0.975	NaN	0.667	NaN
10	0.976	NaN	0.740	NaN
50	0.974	NaN	0.832	NaN
100	0.974	NaN	0.832	NaN
200	–	NaN	0.753	NaN
500	–	NaN	0.795	NaN
5000	–	NaN	0.838	NaN
Best: H=8	0.975			
Term: H=36	0.974			

(1) 4/10 runs led to NaN loss function value.

ISBN 978-952-12-4005-8
ISSN 1239-1891