Enhancing approximation abilities of neural networks by training derivatives

V.I. Avrutskiy

Abstract-A method to increase the precision of feedforward networks is proposed. It requires a prior knowledge of a target function derivatives of several orders and uses this information in gradient based training. Forward pass calculates not only the values of the output layer of a network but also their derivatives. The deviations of those derivatives from the target ones are used in an extended cost function and then backward pass calculates the gradient of the extended cost with respect to weights, which can then be used by any weights update algorithm. Despite a substantial increase in arithmetic operations per pattern (if compared to the conventional training), the extended cost allows to obtain 140–1000 times more accurate approximation for simple cases if the total number of operations is equal. This precision also happens to be out of reach for the regular cost function. The method fits well into the procedure of solving differential equations with neural networks. Unlike training a network to match some target mapping, which requires an explicit use of the target derivatives in the extended cost function, the cost function for solving a differential equation is based on the deviation of the equation's residual from zero and thus can be extended by differentiating the equation itself, which does not require any prior knowledge. Solving an equation with such a cost resulted in 13 times more accurate result and could be done with 3 times larger grid step. GPU-efficient algorithm for calculating the gradient of the extended cost function is proposed.

Index Terms-Neural networks, high order derivatives, partial differential equations, function approximation.

I. INTRODUCTION

N EURAL networks can be used as universal approximators [1]-[4] in a wide range of [1]dimensional cases they successfully overcome the curse of dimensionality [5], [6], thus being an excellent remedy for problems like voice recognition [7], [8] and pattern classification [9]-[11]. Low-dimensional applications are not that famous since many alternatives are available. For example, in solving differential equations [12] (usually 1D [13], [14] or 2D [15], sometimes 3D [16]) neural networks inevitably have to compete with other methods like finite differences, where functions are described by their values on a set of points and in each point, in theory, those values can be accurate within machine precision. Such quality of approximation is not readily achievable for neural networks. Widely developed techniques for training them [17]–[21] are mostly focused on problems like classification, which are not very sensitive to the actual output since small disturbances can hardly produce a shift in class. For the case of direct function approximation, any deviation of an output decreases the accuracy.

V.I. Avrutskiy is with the Department of Aeromechanics and Flight Engineering of Moscow Institute of Physics and Technology, Institutsky lane 9, Dolgoprudny, Moscow region, 141700, e-mail: avrutsky@phystech.edu

This paper proposes a method of utilizing information about target derivatives that increases the precision of neural networks. For some low-dimensional cases, it allows deviations from targets to come close to the rounding error of single precision used during the training, thus addressing the gap between describing a function by an array of values and by a neural network. The concept of using derivatives for approximation [22] is quite common and was investigated for neural networks in numerous studies [23]–[29], however, the implementations of training in said papers included only low order derivatives and used somewhat small architectures, since the conditions of tests did not lead to precision gains of few orders of magnitude. Even though requirements for architectures of neural networks to approximate derivatives are usually modest [4], extra layers are sometimes necessary [30].

1

Due to the necessity to train high order derivatives, which are not popular in applications nor implementations, this paper also includes an algorithm capable of an efficient high-order forward and backward procedures for arbitrary feedforward networks. It is derived directly from formulas for derivative transformation under a change of coordinates that are created by connections between layers, and wherever possible reductions are made. Somewhat similar algorithms can be found in papers on solving differential equations with neural networks [15], [25], [31], however, no papers on this subject contain a universal algorithm for any order of derivatives and arbitrarily deep networks. Essentially the presented procedure is an equivalent to automatic differentiation [32], although software with similar capabilities is currently not quite optimized and fast research prototypes [33] are not yet available for GPUs. As the parallel computations are crucial for neural networks, all formulas are written in terms of matrix multiplications which are implemented on GPUs with highly efficient routines and element-wise operations which are parallelized most easily.

II. MOTIVATION

A particular approach [12], [15] to solving differential equations was investigated: a single solution of one equation corresponds to one neural network which is treated as a smooth function. Its inputs are chosen as independent variables of the equation and the output is supposed to be the solution's value. A simplified procedure of obtaining such a network is as follows. At first, the weights are randomly initialized as they would be for the regular training [34], [35]. Then, the network is substituted into the equation in place of the unknown function, which requires calculating the derivatives of the output with respect to the input. Since the network is not

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

TABLE I The precision of various derivatives after 1000 epochs of the extended training with order 3

Derivative	0	1	2	3
rms	0.07%	0.16%	0.63%	1.72%

yet a solution, after substitution the residual exists. The next step is weights tuning. It requires finding the gradient of the residual with respect to the weights, which then can be used by a weights update algorithm. Two previous steps alternate each other until the residual becomes acceptable. The distinction from the conventional network training thus lies in the cost function which can contain the derivatives of the network.

Due to the lack of an algorithm that could handle cost functions with arbitrary derivatives in the previous papers on this subject, it was implemented and simple tests were conducted on a network with two inputs x, y, one output z and a few hidden layers¹. It was found that any derivative D(z) of the output with respect to the inputs at least up to the 5th order can be trained to approximate the derivative D(f) of an analytical expression f(x, y) using a cost function $e = [D(z) - D(f)]^2$ (here and further per-pattern cost functions e will be used, the actual training cost E is the sum of e over all input patterns). The precision of approximation for this derivative did not depend on the order or the type of the derivative, and after 1000 epochs, the square root of the average cost was about 2.5% of the standard deviation of D(f). For the next test, a network was trained to fit the function and its derivatives simultaneously, using a cost $e = \sum c_D^2 [D(z) - D(f)]^2$. The sum was running through all the 9 possible derivatives of order ≤ 3 and the values themselves. Coefficients c_D are the inversed standard deviations of the corresponding targets D(f). Training with this type of cost function will be referred to as an extended one of the order 3. One could expect a lower or similar precision for each derivative and thus the root of the averaged cost to be $\sqrt{9+1} \cdot 2.5\% \simeq 8\%$. However, that was not the case. After 1000 epochs it reached 3.7%, thus making the deviation of each derivative lower. The gradient of the cost with respect to the weights increased roughly as the number of additional terms, but it still vanished with the same exponential rate as it was propagated backward. The particular precisions are presented in Table I: the root mean square of deviation for each derivative is measured in the percentage of the standard deviation of the corresponding target derivative and averaged along the same orders. The observed increase of precision for values (0th derivative) led to a further investigation on how derivatives can be used to boost the precision most effectively.

III. RESULTS

In all cases RProp [36] is used for weights updating with parameters $\eta_+ = 1.2$, $\eta_- = 0.5$. Weights are forced to stay in [-20, 20] interval, no min/max bonds for steps are imposed. Initial steps Δ_0 are set to $2 \cdot 10^{-4}$ unless otherwise stated. When the number of epochs is greater than 5000, steps Δ that were reduced to zero are set back to 10^{-6} after each 8% of epochs. Weights matrices are initialized [34], [35] with random

¹the network and training conditions are described in the next section



Fig. 1. The function generated by a piece of random 2D Fourier series

TABLE II The precision of values after 1000 epochs of the extended training of order d

d	0	1	2	3	4	5
rms	2.5%	0.8%	0.09%	0.07%	0.28%	0.5%

values from range $\pm 2/\sqrt{||\kappa||}$, where $||\kappa||$ is the number of senders. Thresholds are from ± 0.1 range. All layers but the input and output are nonlinear with a sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{1}$$

unless otherwise stated. All input patterns are processed in one batch. Root mean square values are always divided by the standard deviations of the corresponding functions.

A. 2D function approximation

The target function is generated by a Fourier series:

$$f(x,y) = \sum_{n=1}^{10} \sum_{k=1}^{10} \frac{r_{nk}}{n \cdot k} \sin nx \cos ky + \dots$$

Random coefficients r_{nk} are uniformly distributed in [-1, 1]. Three similar terms with other combinations of sine and cosine have separate coefficients and are omitted for brevity. The total number of parameters is 400. The region for approximation is a square $[-1, 1]^2$. The particular realization is shown on Fig. 1. The input is generated as the vertices of a Cartesian grid with the outer points lying on the boundary, and has 729 points unless otherwise stated. In each point all derivatives are calculated analytically. After training the performance is measured on a grid with 9025 points. The network is a fully connected perceptron with the following configuration of layers (asterisks denote linear activation functions): 2^{*}, 128, 128, 128, 128, 1^{*}.

Table II was created as an attempt to determine the optimal order of the extended training. It varies the maximum order of derivatives used in the cost $e = \sum c_D^2 [D(z) - D(f)]^2$, the training was run for 1000 epochs. The first and the second derivatives boost the precision of values, however, when the order becomes higher than 3, it starts to decrease. Even inclusion of the third derivatives becomes ineffective if one takes into account the additional computational burden. Consider the 4th order training: the cost contains 15 terms, only one of which is related to the precision of values. Moreover, the relative magnitude of those terms is increasing with the order, thus, it is understandable that the training is less focused on the zero order term. By choosing smaller c_D coefficients for the 4th order terms, it is possible to increase the precision of values, but no greater than it was for d = 3. However, it was found that if during the training c_D for higher orders are abruptly set to zero, the precisions of lower orders quickly increase. With further experiments, the following process was constructed: the extended training is started with an order d. After a certain number of epochs, it is terminated and reinitialized with the order d-1 and so on, up until only values alone (d = 0) are trained. An equal number of epochs was chosen for all d+1 steps of this procedure. The training set remains the same throughout the process. This procedure will be referred to as an exclusion training of the order d. The reinitialization of RProp after turning off higher orders needs to be made with $\Delta = 10^{-5}$, otherwise networks are disturbed too much. A gradual decrease of c_D instead of an abrupt drop worked a bit worse. Inclusion of new derivatives, on the contrary, leads to the opposite effect: the precision degrades heavily until new derivatives are properly trained, and then slowly reaches its typical value for the new order.

Table III summarizes the results obtained by various orders of the exclusion training. The median (denoted by tilde) of the root mean square of deviations for values z - f(x, y) is taken across 25 training attempts for each order d. Distribution functions for precisions can be seen on fig. 2. Durations in (kilo) epochs are chosen to equalize the total number of arithmetic operations with the base level taken as 1750 epochs for each step of the order 5 on a grid with 729 points. Due to an overfitting encountered for lower orders, the number of points had to be increased, however, the least favorable scenario for higher order training was chosen: the number of samples was compensated to prevent overfitting, but the number of epochs (now marked by an asterisk) was chosen as if the number of samples remained the same. As soon as the overfitting is stopped, a further increase in number of points does not affect the precision. Training with d = 4 and d = 5 demonstrated no overfitting with 729 patterns even though the later steps of the training completely discarded high-order terms (provided the weights were not disturbed too much, $\Delta = 10^{-5}$). The complexity for various orders is discussed in section V.

B. Autoencoder for 3D curve

Targets are the points of 3D space located on 1D curve:

$$(\cos t, \sin t, t/\pi) \equiv (f_1, f_2, f_3).$$
 (2)

It is possible to reconstruct their 1D representation using a network with 1-neuron linear layer [37] inserted in the middle:

TABLE III

2D Function approximation: the precision of the exclusion training of order d, the gain over the one of d-1. The number of kilo epochs (KE) equalizes the total amount of operations

d	0	1	2	3	4	5
$\widetilde{\text{rms}}$	$2.3 \cdot 10^{-3}$	$2 \cdot 10^{-4}$	$3 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$5.2 \cdot 10^{-6}$	$2.2 \cdot 10^{-6}$
gain	-	11.5	6.6	3	1.9	2.4
KE	107*	26.5^{*}	10.5*	5.2*	2.9	1.75



Fig. 2. 2D Function approximation: cumulative distribution functions for $\log_{10}(\operatorname{rms}(z - f(x, y)))$ for different orders of the exclusion training. The best result of a lower order barely reaches the worst result of a higher order.

TABLE IV 3D Autoencoder: the precision of the exclusion training of order d, the gain over the one of d - 1. The number of kilo epochs (KE) equalizes the total amount of operations

d	0	1	2	3	4	5
$\widetilde{\text{rms}}$	$6.6 \cdot 10^{-4}$	$4.1 \cdot 10^{-4}$	$3.7 \cdot 10^{-5}$	$2.2 \cdot 10^{-5}$	$6.6 \cdot 10^{-6}$	$4.7 \cdot 10^{-6}$
gain	-	1.5	11	1.7	3.4	1.3
KE	29*	9.5*	4.6*	2.6*	1.7	1.2

 $3^*, 64, 64, 1^*, 64, 64, 3^*$, provided it is trained to replicate the input (other layers are smaller as this task is less demanding). This network can be used to generate new points on the curve [38], [39]. The training and test sets are generated by formula (2) with 64 and 1184 equidistant values of $t \in [-2\pi, 2\pi]$ respectively unless otherwise stated. The derivatives of (f_1, f_2, f_3) with respect to t are calculated analytically. The network's output is a vector z_i so the cost includes two sums:

$$e = \sum_{D,i} c_{D,i}^2 [D(z_i) - D(f_i)]^2 = \sum_{j=1}^{a} \sum_{i=1}^{3} c_{ji}^2 [\frac{\partial^j}{\partial t^j} z_i - \frac{\partial^j}{\partial t^j} f_i]^2.$$

Table IV summarizes the results. The median (denoted by tilde) of the root mean square of deviations for the values of the first component: $z_1 - f_1$ is taken across 25 training attempts for each order d. Two other components have similar precisions and gains. Durations in (kilo) epochs equalize the total number of arithmetic operations with the base level taken as 1200 epochs for each step of the order 5 exclusion training on 64 input vectors. The overfitting for orders ≤ 3 is compensated by increasing the number of vectors to 128, but similarly to the previous case the number of epochs was not lowered. The complexity is discussed in section V.

C. Solving differential equations

Previous examples require a prior calculation of the target derivatives in order to increase the precision. Usually this kind of information is not obtained easily, especially for real world applications. However, for solving differential equations this technique can be used without any extra information. The minimizing procedure is based on the equation itself which can be differentiated as many times as necessary without any prior knowledge about the solution.

Consider a generic boundary value problem inside a 2D region Γ with a boundary $\partial\Gamma$ for a function u(x, y):

$$U(x, y, u, u_x, u_y, \ldots) = 0, \left. u \right|_{\partial \Gamma} = f$$

There are different approaches, but the most accurate results are obtained using the method from [15]. A new function v is introduced with relation

$$u(x,y) = v(x,y) \cdot \phi(x,y) + f.$$
(3)

Here ϕ is a smooth function carefully chosen to vanish on $\partial \Gamma$:

$$\phi|_{\partial\Gamma} = 0$$

and not to vanish anywhere inside the region. Usually, it is the simplest analytical expression that is zero on the boundary and has the maximum value of the order of 1 somewhere inside Γ . For trivial boundaries like the circle of radius 1, one can choose $\phi = 1 - x^2 - y^2$. After the substitution, the equation is written for v:

$$V(x, y, v, v_x, v_y, \ldots) = 0.$$

And the boundary condition is

$$|v|_{\partial\Gamma} < \infty$$

The function v can now be approximated with a neural network with two inputs and one output. As long as its values do not diverge during the training the boundary condition is satisfied. A discrete grid in the region Γ is created and according to [15] a cost function $e = V^2$ can be minimized with respect to the weights for all the grid points. Here, however, an additional derivatives of V up to a certain order are included in the cost

$$e = \sum [D(V)]^2.$$

For example, up to the second order:

$$V^{2} + \left(\frac{\partial V}{\partial x}\right)^{2} + \left(\frac{\partial V}{\partial y}\right)^{2} + \left(\frac{\partial^{2} V}{\partial x^{2}}\right)^{2} + \left(\frac{\partial^{2} V}{\partial x \partial y}\right)^{2} + \left(\frac{\partial^{2} V}{\partial y^{2}}\right)^{2}$$

The method was tested on a boundary value problem inside a circle $\Gamma: x^2 + y^2 \le 1$ for Poisson's equation with a nonlinear source

$$u_{xx} + u_{yy} = u^2 + \frac{3}{2}u^3, \ u|_{\partial\Gamma} = -2$$

Numerical results are compared against the analytical solution:

$$u_{\rm a} = \frac{4}{x^2 + y^2 - 3}.$$

A Cartesian grid inside $[-1, 1]^2$ is created with spacing λ . All points outside Γ are excluded and additional points from the boundary with spacing λ are added. This task is similar to the one from the subsection A, however, it is simpler and does not require as many hidden neurons. For differential equations deeper networks seem to perform better. The configuration

TABLE V

POISSON'S EQUATION: THE RMS OF RESIDUAL V FOR THE EXCLUSION TRAINING OF ORDER d, THE GAIN OVER THE ONE OF d - 1, THE RMS OF DEVIATION FROM THE ANALYTICAL SOLUTION u_A . THE NUMBER OF KILO EPOCHS (KE) EQUALIZES THE TOTAL AMOUNT OF OPERATIONS

d	0	1	2	3
$\widetilde{\mathrm{rms}}(V)$	$3 \cdot 10^{-3}$	$5.4 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$	$9.3 \cdot 10^{-5}$
gain	-	5.5	3	2
$\widetilde{\mathrm{rms}}(u-u_{\mathrm{a}})$	$2.8 \cdot 10^{-5}$	$9 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$2.1 \cdot 10^{-6}$
λ	0.052	0.1	0.125	0.15
grid size	1210	352	233	157
KE	1.4	1.6	1.2	1

was chosen as 2^* , 64, 64, 64, 64, 64, 64, 1^* . In this case, instead of giving low-order training extra advantages whenever it encounters overfitting, the total number of arithmetics is to be equalized and the increase of sample points will be accounted for. Since this results in a trade-off between the number of epochs and the size of a grid, it was verified that in all cases using grids that produce overfitting in order to increase the number of epochs decreased the quality of the solution. Networks are verified against the analytical solution on a Cartesian grid with about 8000 points. The results are presented in Table V. One can see that the deviation from analytical solution for d = 3 comes somewhat close to its minimum possible value determined by the rounding error. If it was the only source, rms would be around $0.36 \cdot 10^{-6} = 10^{-6.44}$. The complexity for various orders is discussed in section V.

IV. Algorithm

This section is focused on implementing a gradient algorithm that can be used to minimize a more general form of a cost function for a neural network. The named function now depends not only on the network's output, but also on the derivatives of the output with respect to the input. It is valid for feedforward networks with any number of hidden layers and the derivatives of any order. Presented without thorough derivation, which can be found in the author's paper [40].

A. Notation

The quantity that each neuron obtains from the previous layer before applying its nonlinear sigmoid mapping will be referred to as the neuron activity. All neuron activities of a layer are gathered in a matrix with the letter z like $z^{\kappa\alpha}$. The index α runs through the number of patterns and κ through the neurons of that layer. The norm of indices such as $||\alpha||$ and $||\kappa||$ denotes the total number of enumerated objects. Here it refers to the number of input patterns and the number of neurons in the layer respectively. Different layers are denoted by different Greek letters which also appear in weights matrices connecting those layers: $W^{\theta\kappa}$ would be a matrix that is used to pass from the layer with neurons denoted by κ to the layer with those denoted by θ . The input layer is denoted by β and the output layer by ω . In addition to neurons activities $z^{\kappa\alpha}$, each layer has the derivatives of those activities with respect to certain variables. Those derivatives are matrices of the same size $||\kappa|| \times ||\alpha||$. When variables are

known explicitly, they are denoted by Latin letters a, b, c, d. The corresponding derivatives of $z^{\kappa\alpha}$ are denoted by lower indices:

$$\frac{\partial}{\partial a} z^{\kappa\alpha} = z_a^{\kappa\alpha}, \frac{\partial^2}{\partial a^2} z^{\kappa\alpha} = z_{aa}^{\kappa\alpha}$$

and so on. For formulas that allow an arbitrary number of variables, multi-index notation [41] is used. Namely, if the order of variables is fixed: (a, b, c, d, ...), any derivative can be written using a multi-index $s = (s_1, s_2, ...)$, which is a tuple of integers each component of which is the order of the derivative with respect to the corresponding variable. Thus,

$$\partial^s \equiv \frac{\partial^{|s|}}{\partial a^{s_1} \partial b^{s_2} \dots}.$$

The absolute value of a multi-index is the sum of its components $|s| = \sum_{i} s_{i}$. The difference of two multi-indices is defined element-wisely.

B. Forward pass

The first step is to initialize $z^{\beta\alpha}$, $z^{\beta\alpha}_a$, $z^{\beta\alpha}_{aa}$, $z^{\beta\alpha}_b$, ... at the input layer of the network. $z^{\beta\alpha}$ is simply the matrix of $||\alpha||$ input vectors of length $||\beta||$. Let us say *a* is the first component of the input, then for any α : $z_a = (1, 0, 0, ...)$ and the input matrix is $z^{\beta\alpha}_a = \delta_{\beta-1}$ (the Kronecker delta). All higher derivatives that include *a* like $z^{\beta\alpha}_{ab}$, $z^{\beta\alpha}_{aa}$ are zero. Therefore, initializing derivatives with respect to the input of a neural network is trivial. If *b* is a variable that all components of the input vector depend on, for example, let $||\beta|| = 2$ and α be fixed: $z = (\sin b, \cos b)$, then $z_b = (\cos b, -\sin b)$. If *b* depends on α (see section III.B) this derivative varies for other input vectors. Higher derivatives are non zero and should be calculated accordingly.

After the input layer matrix is generated and its derivatives are calculated, they are to be propagated forward. Consider two generic successive layers with indices κ and θ . The values themselves are propagated by a nonlinear sigmoid and a matrix multiplication (denoted by \times):

$$z^{\theta\alpha} = t^{\theta} + W^{\theta\kappa} \times \sigma(z^{\kappa\alpha}). \tag{4}$$

Here t^{θ} is a threshold vector added to each column of the matrix product to the right of it. The scalar function σ is applied independently to each element of $z^{\kappa\alpha}$. Due to linearity of (4) and weights with thresholds being constant, one can apply a differential operator to both sides to establish the rule for derivative propagation from κ to θ :

$$\partial^s z^{\theta\alpha} = W^{\theta\kappa} \times \left[\partial^s \sigma(z^{\kappa\alpha})\right]. \tag{5}$$

The term $\partial^s \sigma(z^{\kappa\alpha})$ is to be obtained using the chain rule. For example, the first order derivative with respect to *a* is

$$z_a^{\theta\alpha} = W^{\theta\kappa} \times \frac{\partial}{\partial a} \sigma(z^{\kappa\alpha}) = W^{\theta\kappa} \times \left[\sigma'(z^{\kappa\alpha}) \cdot z_a^{\kappa\alpha}\right].$$
(6)

The expression in square brackets is an element-wise product. In similar formulas the sigmoid argument as well as "." sign will be omitted, so it is written as $\sigma' z_a^{\kappa\alpha}$. The chain rule expresses the derivative of the next layer $z_a^{\theta\alpha}$ in terms of

the derivative of the previous layer $z_a^{\kappa\alpha}$. The second order derivative propagation:

$$z_{ab}^{\theta\alpha} = W^{\theta\kappa} \times \frac{\partial^2}{\partial a \partial b} \sigma(z^{\kappa\alpha}) =$$
$$= W^{\theta\kappa} \times [\sigma'' z_a^{\kappa\alpha} z_b^{\kappa\alpha} + \sigma' z_{ab}^{\kappa\alpha}].$$
(7)

For non-mixed second derivatives like $\partial^2/\partial a^2$, one can simplify the square brackets to $\sigma'' [z_a^{\kappa\alpha}]^2 + \sigma' z_{aa}^{\kappa\alpha}$ (the second power of square brackets is also an element-wise operation). The same holds for further formulas: all Latin variables are considered distinct, and in case they are not, it is useful to simplify expressions first to avoid unnecessary arithmetic and/or memory operations. Any mixed second derivative depends on three terms of a previous layer: said second derivatives depend only on two terms: the first and the second derivatives with respect to that variable. The third order:

$$z_{abc}^{\theta\alpha} = W^{\theta\kappa} \times \left[\sigma^{\prime\prime\prime} z_{a}^{\kappa\alpha} z_{b}^{\kappa\alpha} z_{c}^{\kappa\alpha} + \sigma^{\prime\prime} \cdot (z_{ab}^{\kappa\alpha} z_{c}^{\kappa\alpha} + z_{ac}^{\kappa\alpha} z_{b}^{\kappa\alpha} + z_{bc}^{\kappa\alpha} z_{a}^{\kappa\alpha}) + \sigma^{\prime} z_{abc}^{\kappa\alpha} \right].$$
(8)

The forth order:

$$z_{abcd}^{\theta\alpha} = W^{\theta\kappa} \times \left[\sigma^{IV} z_a^{\kappa\alpha} z_b^{\kappa\alpha} z_c^{\kappa\alpha} z_d^{\kappa\alpha} + \sigma^{IV} (z_{ab}^{\kappa\alpha} z_c^{\kappa\alpha} z_d^{\kappa\alpha} + z_{ac}^{\kappa\alpha} z_b^{\kappa\alpha} z_d^{\kappa\alpha} + z_{ac}^{\kappa\alpha} z_d^{\kappa\alpha} + z_{ad}^{\kappa\alpha} z_b^{\kappa\alpha} z_c^{\kappa\alpha} + z_{bc}^{\kappa\alpha} z_a^{\kappa\alpha} z_d^{\kappa\alpha} + z_{bd}^{\kappa\alpha} z_a^{\kappa\alpha} z_c^{\kappa\alpha} + z_{cd}^{\kappa\alpha} z_a^{\kappa\alpha} z_b^{\kappa\alpha} \right] + \sigma^{\prime\prime} (z_a^{\kappa\alpha} z_{bcd}^{\kappa\alpha} + z_b^{\kappa\alpha} z_{acd}^{\kappa\alpha} + z_c^{\kappa\alpha} z_{abd}^{\kappa\alpha} + z_{d}^{\kappa\alpha} z_{abc}^{\kappa\alpha} + z_{abc}^{\kappa\alpha} z_{bc}^{\kappa\alpha} + z_{ac}^{\kappa\alpha} z_{bd}^{\kappa\alpha} + z_{ad}^{\kappa\alpha} z_{bc}^{\kappa\alpha} + \sigma^{\prime} z_{abcd}^{\kappa\alpha} \right].$$
(9)

The fifth order simplified for the cases considered in this paper:

$$z_{aaaaa}^{\theta\alpha} = W^{\theta\kappa} \times \left[\sigma^{V} [z_{a}^{\kappa\alpha}]^{5} + \sigma^{IV} 10 [z_{a}^{\kappa\alpha}]^{3} z_{aa}^{\kappa\alpha} + \sigma^{\prime\prime\prime} (15 z_{a}^{\kappa\alpha} [z_{aa}^{\kappa\alpha}]^{2} + 10 [z_{a}^{\kappa\alpha}]^{2} z_{aaa}^{\kappa\alpha}) + \sigma^{\prime\prime} (10 z_{aa}^{\kappa\alpha} z_{aaa}^{\kappa\alpha} + 5 z_{a}^{\kappa\alpha} z_{aaaa}^{\kappa\alpha}) + \sigma^{\prime} z_{aaaaa}^{\kappa\alpha} \right], (10)$$

$$\begin{split} z_{aaaab}^{\theta\alpha} = & W^{\theta\kappa} \times \left[\sigma^{V} [z_{a}^{\kappa\alpha}]^{4} z_{b}^{\kappa\alpha} + \sigma^{IV} (4[z_{a}^{\kappa\alpha}]^{3} z_{ab}^{\kappa\alpha} \\ & + 6[z_{a}^{\kappa\alpha}]^{2} z_{aa}^{\kappa\alpha} z_{b}^{\kappa\alpha}) + \sigma^{\prime\prime\prime} (6[z_{a}^{\kappa\alpha}]^{2} z_{aab}^{\kappa\alpha} + 3[z_{aa}^{\kappa\alpha}]^{2} z_{b}^{\kappa\alpha} \\ & + 12 z_{a}^{\kappa\alpha} z_{aa}^{\kappa\alpha} z_{ab}^{\kappa\alpha} + 4 z_{a}^{\kappa\alpha} z_{b}^{\kappa\alpha} z_{aaa}^{\kappa\alpha}) + \sigma^{\prime\prime} (4 z_{a}^{\kappa\alpha} z_{aaab}^{\kappa\alpha} \\ & + 6 z_{aa}^{\kappa\alpha} z_{aab}^{\kappa\alpha} + 4 z_{ab}^{\kappa\alpha} z_{aaa}^{\kappa\alpha} + z_{b}^{\kappa\alpha} z_{aaaa}^{\kappa\alpha}) + \sigma^{\prime} z_{aaaab}^{\kappa\alpha} \right], \end{split}$$

$$\begin{split} z_{aaabb}^{\theta\alpha} = & W^{\theta\kappa} \times \left[\sigma^{V} [z_{a}^{\kappa\alpha}]^{3} [z_{b}^{\kappa\alpha}]^{2} + \sigma^{IV} (3z_{a}^{\kappa\alpha} z_{aa}^{\kappa\alpha} [z_{b}^{\kappa\alpha}]^{2} \\ & + 6 [z_{a}^{\kappa\alpha}]^{2} z_{ab}^{\kappa\alpha} z_{b}^{\kappa\alpha} + [z_{a}^{\kappa\alpha}]^{3} z_{bb}^{\kappa\alpha}) + \sigma^{\prime\prime\prime} (3z_{a}^{\kappa\alpha} z_{aa}^{\kappa\alpha} z_{ab}^{\kappa\alpha} z_{bb}^{\kappa\alpha} \\ & + 3 [z_{a}^{\kappa\alpha}]^{2} z_{abb}^{\kappa\alpha} + 6 z_{a}^{\kappa\alpha} z_{aab}^{\kappa\alpha} z_{b}^{\kappa\alpha} + 6 z_{aa}^{\kappa\alpha} z_{abb}^{\kappa\alpha} + 6 z_{aa}^{\kappa\alpha} z_{abb}^{\kappa\alpha} + 6 z_{aa}^{\kappa\alpha} z_{abb}^{\kappa\alpha} + 3 [z_{a}^{\kappa\alpha}]^{2} + 6 z_{a}^{\kappa\alpha} (z_{ab}^{\kappa\alpha}]^{2}) + \sigma^{\prime\prime} (z_{bb}^{\kappa\alpha} z_{aaa}^{\kappa\alpha} \\ & + 6 z_{ab}^{\kappa\alpha} z_{aab}^{\kappa\alpha} + 3 z_{aa}^{\kappa\alpha} z_{abb}^{\kappa\alpha} + 2 z_{b}^{\kappa\alpha} z_{aaab}^{\kappa\alpha} + 3 z_{a}^{\kappa\alpha} z_{aabb}^{\kappa\alpha}) \\ & + \sigma^{\prime} z_{aaabb}^{\kappa\alpha} \Big]. \end{split}$$

In general, a derivative ∂^s uses those and only those derivatives ∂^r from the previous layer, for which s - r has no negative components. For example, if a cost function would use only $\partial^5/\partial a^5$, one still needs to calculate $\partial^4/\partial a^4$, $\partial^3/\partial a^3$, $\partial^2/\partial a^2$, $\partial/\partial a$ and the values themselves for all layers, as seen from formula (10).

C. Backward pass

After obtaining the final layer matrices $z^{\omega\alpha}$, $z_a^{\omega\alpha}$, $z_{aa}^{\omega\alpha}$, ..., one can calculate a net cost function E, usually as the sum of a per-pattern cost function e:

$$E = \sum_{\alpha} e(z^{\omega\alpha}, z_a^{\omega\alpha}, z_{aa}^{\omega\alpha}, \ldots),$$

and more importantly, its derivatives with respect to the elements of all used matrices

$$\frac{\partial E}{\partial z^{\omega\alpha}}, \frac{\partial E}{\partial z_a^{\omega\alpha}}, \frac{\partial E}{\partial z_{aa}^{\omega\alpha}}, \dots,$$

which can also be written with a multi-index

$$\frac{\partial E}{\partial (\partial^r z^{\omega \alpha})}$$

Those derivatives have indices ω and α , which means they are matrices of the same size as the matrices with respect to which the derivative of E is calculated, in this case $||\omega|| \times ||\alpha||$. They are propagated backwards using the following relation (now one moves from θ to κ):

$$\frac{\partial E}{\partial (\partial^r z^{\kappa \alpha})} = \sum_{s} \binom{s}{r} \cdot \partial^{s-r} \sigma'(z^{\kappa \alpha}) \cdot \left([W^{\theta \kappa}]^T \times \frac{\partial E}{\partial (\partial^s z^{\theta \alpha})} \right)$$
(11)

where sum is taken through all s that were used for the forward propagation and for which s - r has no negative components. Operator ∂^{s-r} is applied only to $\sigma'(z^{\kappa\alpha})$. Following the notation of [41], the binomial coefficient is:

$$\binom{s}{r} = \prod_{i} \frac{s_i!}{r_i!(s_i - r_i)!}$$

Expression (11) is essentially a formula for a derivative upon the transformation of arguments. Consider a simplified scenario with only two derivatives $\partial/\partial a$, $\partial^2/\partial a^2$. Let $||\alpha|| = 1$ and κ be fixed, $\theta \in [\theta_1 \dots \theta_n]$. *E* can be considered as the function of values and their derivatives on either of two layers θ or κ . For a change of variables

$$E(z^{\theta_1}, z_a^{\theta_1}, z_{aa}^{\theta_1}, \dots, z^{\theta_n}, z_a^{\theta_n}, z_{aa}^{\theta_n}) \to E(z^{\kappa}, z_a^{\kappa}, z_{aa}^{\kappa}),$$

the following holds:

$$\frac{\partial E}{\partial z_a^{\kappa}} = \sum_{\theta=\theta_1}^{\theta_n} \frac{\partial E}{\partial z^{\theta}} \frac{\partial z^{\theta}}{\partial z_a^{\kappa}} + \frac{\partial E}{\partial z_a^{\theta}} \frac{\partial z_a^{\theta}}{\partial z_a^{\kappa}} + \frac{\partial E}{\partial z_{aa}^{\theta}} \frac{\partial z_{aa}^{\theta}}{\partial z_a^{\kappa}}.$$
 (12)

The derivatives of E with respect to $z^{\theta}, z^{\theta}_{a}$ and z^{θ}_{aa} were calculated on the previous backward step (or initialized at the output layer). The remaining terms can be calculated by differentiating forward pass formulas. Namely, according to (4) $\partial z^{\theta}/\partial z^{\kappa}_{a}$ is zero, since the values of the next layer z^{θ} do not depend on the derivatives of the previous one. The term $\partial z^{\theta}_{a}/\partial z^{\kappa}_{a}$ is obtained by applying $\partial/\partial z^{\kappa}_{a}$ to (6) (and $\partial z^{\theta}_{aa}/\partial z^{\kappa}_{a}$ is obtained by applying it to (7) with a = b):

$$\frac{\partial}{\partial z_a^{\kappa}} z_a^{\theta} = \frac{\partial}{\partial z_a^{\kappa}} \sum_{\tilde{\kappa}} W^{\theta \tilde{\kappa}} \sigma'(z^{\tilde{\kappa}}) z_a^{\tilde{\kappa}} = W^{\theta \kappa} \sigma'(z^{\kappa}).$$
(13)

The matrix multiplication written as a sum over $\tilde{\kappa}$ vanishes as the derivative of a summand is non zero only if $\tilde{\kappa} = \kappa$. Terms like $\sigma'(z^{\kappa})$ are the derivatives of $\partial^s \sigma(z^{\kappa})$ (the square brackets in expressions like (9)) with respect to one of their terms $\partial^r z^{\kappa}$. Their general form is $\binom{s}{r} \partial^{s-r} \sigma'(z^{\kappa})$. It is derived in [40] by the analysis of Faà di Bruno's formula [42]. The sum over θ in (12) can be written as a vector-matrix product (the pattern index α turns it into a matrix-matrix product) so its second term is $\partial E/\partial z_a^{\theta} \times [W^{\theta\kappa} \cdot \sigma'(z^{\kappa})]$. The term $\sigma'(z^{\kappa})$ has no θ and can be taken out. A transpose and rearrangement to match the dimensions lead to expression (11). The gradients of E with respect to weights and thresholds are as follows:

$$\frac{\partial E}{\partial W^{\theta\kappa}} = \sum_{s} \frac{\partial E}{\partial (\partial^{s} z^{\theta\alpha})} \times [\partial^{s} \sigma(z^{\kappa\alpha})]^{T}, \qquad (14)$$

$$\frac{\partial E}{\partial t^k} = \sum_{\alpha} \frac{\partial E}{\partial z^{\kappa \alpha}}.$$
(15)

The sum in (14) runs through all used derivatives. The expression for the gradient of E with respect to the weights $W^{\theta\kappa}$ between layers κ and θ includes matrices like $z_a^{\kappa\alpha}$ which are calculated during the forward pass for layer κ . They emerge from the term $\partial^s \sigma(z^{\kappa\alpha})$. Matrices $\partial E/\partial(\partial^s z^{\theta\alpha})$ are obtained for layer θ during the backward pass. The expression (14) is also a consequence of a variable change. Let $||\alpha|| = 1$ and $W^{\theta\kappa}$ be fixed, $\theta \in [\theta_1 \dots \theta_n]$, $s \in [s_1 \dots s_m]$. The change is

$$E(\partial^{s_1} z^{\theta_1}, \partial^{s_1} z^{\theta_2}, \dots, \partial^{s_m} z^{\theta_n}) \to E(W^{\theta\kappa}).$$

The derivative of E is transformed as

$$\frac{\partial E}{\partial W^{\theta\kappa}} = \sum_{s=s_1}^{s_m} \sum_{\tilde{\theta}=\theta_1}^{\theta_n} \frac{\partial E}{\partial (\partial^s z^{\tilde{\theta}})} \frac{\partial (\partial^s z^{\theta})}{\partial W^{\theta\kappa}}.$$

According to the forward pass formula (5) the latter term is

$$\frac{\partial(\partial^s z^{\theta})}{\partial W^{\theta\kappa}} = \delta_{\theta\tilde{\theta}} \cdot \partial^s \sigma(z^{\kappa}).$$

Upon substitution, the Kronecker delta removes the sum over $\tilde{\theta}$. Pattern index α creates a matrix multiplication and transpose matches the dimensions, which leads to expression (14).

V. COMPLEXITY

A. Forward pass

Propagation of the derivative ∂^s from layer κ with $||\kappa||$ neurons to layer θ with $||\theta||$ neurons, according to (5), requires an element-wise calculation of the term $\partial^s \sigma(z^{\kappa\alpha})$ spawned by the chain rule and one matrix multiplication. The number of operations for the latter is $(2||\kappa|| - 1) \cdot ||\theta|| \cdot ||\alpha||$, and the total number of element-wise operations is $\rho||\kappa|| \cdot ||\alpha||$, where ρ is the amount of arithmetics per pattern per neuron of layer κ .

To find ρ , that is the number of operations required to calculate the square brackets in expressions like (8) and (9) without taking into account the upper indexes, one may notice that each term is a product of the sigmoid derivative with the order from 1 to |s| and some expression in parentheses. Arithmetics required for the first derivative of the sigmoid (1) can be found by writing it as a polynomial of the σ itself:

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} = \sigma - \sigma^2.$$

By differentiating this further, one can obtain all required expressions as polynomials of σ . Since higher orders are

TABLE VI THE NUMBER OF OPERATIONS #OP PER PATTERN PER NEURON FOR ALL PARENTHESES OF THE DERIVATIVE ∂^s

s	#op	s	#op	s	#op	s	#op
(0)	0	(1,1)	1	(4)	11	(5)	20
(1)	0	(3)	4	(3,1)	19	(4,1)	38
(2)	1	(2,1)	6	(2,2)	22	(3,2)	54

used only together with lower ones, their polynomials can be simplified and expressed in terms of lower order derivatives to decrease the number of operations. Namely, up to the sixth order:

$$\sigma' = \sigma(1 - \sigma) \quad \sigma'' = \sigma'(1 - 2\sigma)$$
$$\sigma''' = \sigma'(1 - 6\sigma') \quad \sigma^{IV} = \sigma''(1 - 12\sigma')$$
$$\sigma^{V} = \sigma' - 30(\sigma'')^{2} \quad \sigma^{VI} = \sigma'' + (\sigma^{IV} - \sigma'')(5 - 30\sigma').$$

The total number of products in the square brackets is equal to the order |s|. This, together with the operations required to calculate the derivatives of σ , but without taking into account the arithmetic of the expressions in parentheses, leads to the following number of extra² operations per pattern per neuron for |s| from 1 through 6: 3, 8, 13, 18, 23, 30. To evaluate the number of operations required to calculate the parentheses, one can notice the following: the parentheses that are multiplied by the k^{th} derivative of the sigmoid are the sum of products of z with lower indices which from all possible unique partitions of s into k parts. If some partitions are not unique (if two or more Latin variables are the same), one should multiply the corresponding product by the number of occurrences. Expressions like (8) and (9) are the worst-case scenario since all the variables are distinct, therefore, any partition is unique. Since partitions are symmetrical with respect to a permutation of variables, Table VI is sufficient for evaluating the number of operations for each derivative used in this paper.

B. Weights gradient

Expression (14) for the weights gradient is a sum of matrix products of two terms, one of which was calculated during the backward pass and another is an element-wise part of (5) that can be cached during the forward pass. The number of operations for each matrix multiplication is $||\kappa|| \cdot ||\theta|| \cdot (2||\alpha|| - 1)$.

C. Backward pass

Formula (11) contains a sum for each backpropagated derivative r. However, for different r, calculations for the summands overlap significantly. Except for the combinatorial coefficient, each summand is an element-wise product of two terms, one of which is a matrix product and another one is very similar to expression $\partial^s \sigma(z^{\kappa\alpha})$ encountered in the forward pass (5). The only difference is the presence of σ' instead of σ . One can notice that for r = 0 multi-index s - r has no negative components for any s, therefore, all possible derivatives of $\sigma'(z^{\kappa\alpha})$ have to be calculated for r = 0 but

7

TABLE VII The amount of element-wise arithmetics for the extended training of order d as the percentage of matrix arithmetics for the examples from Section III

d	0	1	2	3	4	5
2D Function	1%	2%	4%	6%	10%	17%
3D Autoencoder	2%	4%	10%	15%	22%	30%
Poisson's equation	7%	12%	19%	31%	-	-

 TABLE VIII

 THE RELATIVE COMPLEXITY OF ONE EPOCH OF THE EXTENDED TRAINING

 OF ORDER d FOR A NETWORK WITH LAYER CONFIGURATION

 n^*, n, n, n, n^* FOR ONE AND TWO VARIABLES FOR DIFFERENTIATION

order	0	1	2	3	4	5
a	1	$2 + \frac{2}{5n}$	$3 + \frac{6}{n}$	$4 + \frac{15}{n}$	$5 + \frac{28}{n}$	$6 + \frac{48}{n}$
a, b	1	$3 + \frac{2}{5n}$	$6 + \frac{13}{n}$	$10 + \frac{45}{n}$	$15 + \frac{120}{n}$	$21 + \frac{280}{n}$

then can be reused for higher r. In fact, all of those terms can be evaluated during the forward pass when the parentheses of expressions like (8) and (9) are already calculated and only need to be multiplied by higher derivatives of σ . Thus, the number of an additional operations per pattern per neuron for |s-r| from 0 through 5 is 0, 1, 3, 4, 7, 9 plus from 2 to 5 operations to increase the maximum order of the derivative of σ by one, but only once for the whole bundle of the propagated derivatives. As for the matrix multiplication on the right of (11), all unique products have to be calculated for r = 0. For higher r, those computationally expensive products should be reused. The only unaccounted part of operations left for $\partial E/\partial(\partial^r z^{\kappa\alpha})$ is the multiplications by the binomial coefficients and the summation over all s for which s - r has no negative components. For example, the maximum number of summands is equal to the number of propagated derivatives, which in this paper ≤ 21 . One can roughly estimate the ratio between the element-wise and matrix operations as $\frac{\rho}{6\theta}$. For the cases considered in Section III this value never exceeds 0.1, however, caching $z^{\kappa\alpha}$, $\partial^s \sigma(z^{\kappa\alpha})$ and $\partial^s \sigma'(z^{\kappa\alpha})$ is required. Since this would triple the memory complexity, and the portion of element-wise operations is relatively low, only the neuron activities $z^{\kappa\alpha}$ are cached. Terms $\partial^s \sigma(z^{\kappa\alpha})$ and $\partial^s \sigma'(z^{\kappa\alpha})$ are calculated on demand. Table VII shows the portion of the element-wise operations measured in the percentage of matrix operations provided only the neuron activities $z^{\kappa\alpha}$ are cached. To calculate the equalizing number of epochs for different order training one can simply compare the total amount of matrix operations and then slightly correct it using this table. Note that the exclusion training consists of steps that are the extended training. Table VIII generalizes the relative complexity of the extended training for one and two variables derivatives with respect to which are propagated. The first summand is the total number of derivatives. The second summand reflects the element-wise portion of operations. For simplicity, all values were rounded to the nearest integer.

D. Hardware efficiency.

Some technical details are required to get the maximum hardware efficiency. This study uses CUDA with cuBLAS

²the evaluation of σ itself is required for any order. The inclusion of the corresponding number of operations very slightly advantages high-order training and therefore can be ignored in the scope of this paper

[43], that is, a GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS). Its function cublasSgemm is used to compute matrix multiplications. The rest of the operations are element-wise and can be implemented by a C-like code that describes the computations on one element which are then parallelized automatically across the whole matrix. The code should be written in such a way that delays between the calls of cublasSgemm and other functions are minimal, and matrices which they operate on are as large as possible. The heights of matrices are obviously fixed by the network's configuration, so they can only be as long as possible, i.e. it is preferably to process all input patterns in one batch. An efficient method to make matrices longer is to use pattern index α to stack the matrices of different derivatives together. In the forward pass formula (5), the matrix multiplication $W^{\theta\kappa} \times [\partial^s \sigma(z^{\kappa\alpha})]$ is supposed to be called for each s separately. But if one first evaluates all (let us say n) different terms $\partial^s \sigma(z^{\kappa\alpha})$ and stores them in the memory as a single matrix with a new index $\tilde{\alpha} \in [1 \dots n ||\alpha||]$ then cublasSgemm can be called only once and will result in a similarly stacked $\partial^s z^{\theta \tilde{\alpha}}$ matrix. This requires the column-wise storage of matrices and an explicit memory allocation but, for example, made the calculations of the 5th order for 2D function approximation few times more efficient. Backward pass formula (11) allows a similar enhancement.

As for the delay between the functions calls, two options are available: either memory for the matrices of all derivatives of all layers is permanently allocated on the GPU or intermediate data saves/loads occur in the background while some matrices are propagated from one layer to another. The first option is suitable for systems with enough GPU memory. The second option can be implemented in any system with fast enough communications between the device and the host. If the time for a matrix $||\alpha|| \times ||\kappa||$ to load is less than the time required to multiply it from the right by a matrix $||\kappa|| \times ||\theta||$ then one can hide all memory operations behind the computations.

A trick to reduce the memory complexity of formula (11) is worth mentioning. Instead of calculating matrix products $[W^{\theta\kappa}]^T \times \partial E / \partial (\partial^s z^{\theta\alpha})$ and storing them in the cache one can put them in the memory dedicated for $\partial E/\partial(\partial^r z^{\kappa\alpha})$ with r = s and then calculate $\partial E / \partial (\partial^r z^{\kappa \alpha})$ in a proper order. Namely, as soon as the evaluation is started for any r, one has to multiply a matrix product residing in that memory by $\sigma'(z^{\kappa\alpha})$, thus, it cannot be reused. To tackle this issue, one should start with r = (0, 0, 0, 0) and, thus, spoil s = (0, 0, 0, 0), which is not used for any other r since the components of s - r can not be negative. Then one can pick any first order derivative, for example, r = (1, 0, 0, 0) and spoil s = (1, 0, 0, 0), which is not a problem, since the only other case when (1, 0, 0, 0) - r has no negative components is when r = (0, 0, 0, 0), but that term has already been calculated. All the first order derivatives are calculated, then all the second order ones and so on, and no conflicts are encountered.

E. Performance test

CUDA C code was written using the proposed suggestions. It was tested on fully connected perceptrons with 7 layers

TABLE IX RUNNING TIMES FOR THE REGULAR TRAINING

n	Theano	TensorFlow	CUDA C
128	17.4	10	3.1
256	19.9	11.8	5.7
512	27.6	22.2	14.9

of the same width n. RProp with the regular cost was run for 1000 epochs with 2048 input patterns processed in one batch. It is compared against Keras 2.0.8 with backends theano [44] 0.9.0 and TensorFlow [45] 1.3.0, default settings are used. The system is Deep Learning AMI for Amazon Linux, version 3.3 run on EC2 p2.xlarge instance with one GK210 core of Tesla K80 available. The driver version is 375.66, CUDA version is 8.0. The results are gathered in Table IX. This paper uses neural networks with n equal to 64 and 128. Provided TensorFlow can scale its performance for cases when many derivatives are being propagated, the gain is around 300%. Even for the regular training where standard neural network libraries should be quite efficient, the proposed code is about 3 times faster for the networks used in this paper. For cases where many derivatives are to be calculated, naive implementations of automatic differentiation would probably be much slower.

VI. CONCLUSION

A training process that enhances the approximation abilities of fully connected feedforward neural networks was presented. It is based on calculating extra derivatives of the network and comparing them with the target ones to evaluate the weights gradient. It was demonstrated to work well for lowdimensional cases. Using derivatives up to the 5th order, the precision of approximation for 2D analytical function was increased 1000 times. Among all derivatives, the first and the second contributed the most to the relative increase of accuracy. Computational costs per pattern increase significantly (see table VIII), however, it seems that there are no conditions under which the conventional training could catch up with the proposed one provided a network capacity is sufficient. Highorder training was found to be more demanding in this regard. For 2D approximation lowering the number of neurons in all hidden layers from 128 to 64 and then to 32 increased the root mean square of the deviation of values for the 5th order extended training from $2.2 \cdot 10^{-6}$ to $2.4 \cdot 10^{-5}$ and then to $2.4 \cdot 10^{-4}$, for the 2nd order from $3.7 \cdot 10^{-5}$ to $1 \cdot 10^{-4}$ and then to $5 \cdot 10^{-4}$ and for the regular training from $2.3 \cdot 10^{-3}$ to $3.0 \cdot 10^{-3}$ and then to $5.3 \cdot 10^{-3}$. Increasing the number of neurons higher than 128 was not beneficial for any order.

For real neural network applications like classification one can imagine hard times calculating high-order derivatives. From this point of view, solving partial differential equations can benefit much more from the proposed enhancements, as all information about extra derivatives can be obtained via simple differentiation of the equation itself. In the presented example of solving partial differential equation, the precision of the regular method was quite acceptable, however, it required a grid with three times smaller spacing. Even though the extended training achieved 13 times smaller error for the same computational cost, the increase of a spacing λ might be more important. As the number N of dimensions increases, the grid size grows as $(1/\lambda)^N$. If a similar increase of the grid spacing persists for high-dimensional partial differential equations, the extended training could be even more advantageous.

ACKNOWLEDGMENT

Author owes a great debt of gratitude to his scientific advisor E.A. Dorotheyev. Special thanks for invaluable support are due to Y.N. Sviridenko, A.M. Gaifullin, I.A. Avrutskaya and I.V. Avrutskiy without whom this work would be impossible. Sincere gratitude is extended to Neural Networks and Deep Learning lab, MIPT and M.S. Burtsev personally.

REFERENCES

- K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [2] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 2, no. 4, pp. 303–314, 1989.
- [3] V. Kurkova, "Kolmogorov's theorem and multilayer neural networks," *Neural networks*, vol. 5, no. 3, pp. 501–506, 1992.
- [4] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.
- [5] A. R. Barron, "Universal approximation bounds for superpositions of a sigmoidal function," *IEEE Transactions on Information theory*, vol. 39, no. 3, pp. 930–945, 1993.
- [6] —, "Approximation and estimation bounds for artificial neural networks," *Machine Learning*, vol. 14, no. 1, pp. 115–133, 1994.
- [7] L. Deng and X. Li, "Machine learning paradigms for speech recognition: An overview," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 5, pp. 1060–1089, 2013.
- [8] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 369–376.
- [9] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.
- [10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [11] G. P. Zhang, "Neural networks for classification: a survey," IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 30, no. 4, pp. 451–462, 2000.
- [12] M. Kumar and N. Yadav, "Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: a survey," *Computers & Mathematics with Applications*, vol. 62, no. 10, pp. 3796–3811, 2011.
- [13] A. Meade and A. A. Fernandez, "The numerical solution of linear ordinary differential equations by feedforward neural networks," *Mathematical and Computer Modelling*, vol. 19, no. 12, pp. 1–25, 1994.
- [14] A. Malek and R. S. Beidokhti, "Numerical solution for high order differential equations using a hybrid neural network optimization method," *Applied Mathematics and Computation*, vol. 183, no. 1, pp. 260–271, 2006.
- [15] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE Transactions* on Neural Networks, vol. 9, no. 5, pp. 987–1000, 1998.
- [16] I. Lagaris, A. Likas, and D. Fotiadis, "Artificial neural network methods in quantum mechanics," *Computer Physics Communications*, vol. 104, no. 1-3, pp. 1–14, 1997.
- [17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [18] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, 2013, pp. 1058–1066.

- [19] J.-R. Chang and Y.-S. Chen, "Batch-normalized maxout network in network," arXiv preprint arXiv:1511.02583, 2015.
- [20] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," arXiv preprint arXiv:1302.4389, 2013.
- [21] D. Mishkin and J. Matas, "All you need is a good init," arXiv preprint arXiv:1511.06422, 2015.
- [22] R. Murray-Smith and B. A. Pearlmutter, "Transformations of gaussian process priors," in *Deterministic and Statistical Methods in Machine Learning*. Springer, 2005, pp. 110–123.
- [23] P. Simard, Y. LeCun, J. Denker, and B. Victorri, "Transformation invariance in pattern recognitiontangent distance and tangent propagation," *Neural networks: tricks of the trade*, pp. 549–550, 1998.
- [24] H. Drucker and Y. Le Cun, "Improving generalization performance using double backpropagation," *IEEE Transactions on Neural Networks*, vol. 3, no. 6, pp. 991–997, 1992.
- [25] S. He, K. Reif, and R. Unbehauen, "Multilayer neural networks for solving a class of partial differential equations," *Neural networks*, vol. 13, no. 3, pp. 385–396, 2000.
- [26] G. W. Flake and B. A. Pearlmutter, "Differentiating functions of the jacobian with respect to the weights," in Advances in Neural Information Processing Systems, 2000, pp. 435–441.
- [27] P. Cardaliaguet and G. Euvrard, "Approximation of a function and its derivative with a neural network," *Neural Networks*, vol. 5, no. 2, pp. 207–220, 1992.
- [28] A. Pukrittayakamee, M. Hagan, L. Raff, S. T. Bukkapatnam, and R. Komanduri, "Practical training framework for fitting a function and its derivatives," *IEEE transactions on neural networks*, vol. 22, no. 6, pp. 936–947, 2011.
- [29] E. Basson and A. P. Engelbrecht, "Approximation of a function and its derivatives in feedforward neural networks," in *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.* 99CH36339), vol. 1. IEEE, 1999, pp. 419–421.
- [30] E. D. Sontag, "Feedback stabilization using two-hidden-layer nets," *IEEE Transactions on neural networks*, vol. 3, no. 6, pp. 981–990, 1992.
- [31] J. Berg and K. Nyström, "A unified deep artificial neural network approach to partial differential equations in complex geometries," *Neurocomputing*, vol. 317, pp. 28–41, 2018.
- [32] A. Griewank and A. Walther, Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM, 2008.
- [33] J. M. Siskind and B. A. Pearlmutter, "Efficient implementation of a higher-order language with built-in ad," arXiv preprint arXiv:1611.03416, 2016.
- [34] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.
- [35] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [36] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The rprop algorithm," in *Neural Networks*, 1993., IEEE International Conference on. IEEE, 1993, pp. 586–591.
- [37] S. Petridis and M. Pantic, "Deep complementary bottleneck features for visual speech recognition," in 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2016, pp. 2304–2308.
- [38] H. Bourlard and Y. Kamp, "Auto-association by multilayer perceptrons and singular value decomposition," *Biological cybernetics*, vol. 59, no. 4, pp. 291–294, 1988.
- [39] Y. Bengio et al., "Learning deep architectures for ai," Foundations and trends[®] in Machine Learning, vol. 2, no. 1, pp. 1–127, 2009.
- [40] V. Avrutskiy, "Backpropagation generalized for output derivatives," arXiv preprint arXiv:1712.04185, 2017.
- [41] X. Saint Raymond, Elementary introduction to the theory of pseudodifferential operators. Routledge, 2018.
- [42] M. Hardy, "Combinatorics of partial derivatives," the electronic journal of combinatorics, vol. 13, no. 1, p. 1, 2006.
- [43] C. Nvidia, "Cublas library," NVIDIA Corporation, Santa Clara, California, vol. 15, no. 27, p. 31, 2008.
- [44] J. Bergstra, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, I. Goodfellow, A. Bergeron, Y. Bengio, and P. Kaelbling, "Theano: Deep learning on gpus with python," 2011.
- [45] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.