PAPER

Neural Networks Learning Differential Data

Ryusuke MASUOKA[†], *Member*

SUMMARY

In many of machine learning problems, it is essential to use not only the training data, but also a priori knowledge about how the world is constrained. In many cases, such knowledge is given in the forms of constraints on differential data or more specifically partial differential equations (PDEs). Neural networks with capabilities to learn differential data can take advantage of such knowledge and easily incorporate such constraints into the learning of training value data.

In this paper, we report a structure, an algorithm, and results of experiments on neural networks learing differential data.

key words: neural networks, tangent prop, differential data

1. Introduction

In many of machine learning problems, it is essential to use not only the training data, but also a priori knowledge about how the world is constrained. In many cases, such knowledge is given in the forms of constraints on differential data or more specifically partial differential equations (PDEs).

Examples of such cases are given in [8] and [3].

Simard et al. [8] described how the invariance of pattern recognition with respect to such as translations, rotations, scalings, etc. can be interpreted as constraints on first order differential data. The output of the neural network as a pattern recognizer should stay same for such transformations of the input. That constraint is interpreted as that directional derivatives of the output with respect to the directions of transformations have to be zeros.

Hornik et al. [3] identified several areas of applications requiring approximation to an unknown mapping and its derivatives, such as robot learning, deterministic chaos, economics, sensitivity analyses, etc.

In this introduction, we describe two more possible areas of applications where learning constraints on differential data give advantages.

The first possible area of application is simulation of human arm movements by the minimum-torque-change model (see [9]). In [4], Maeda et al. proposed a cascade neural network model for such simulation. This model uses one neural network for each discrete time step, which produces expected torque values at that time step. Then those neural networks are connected to produce difference between expected torque values of consecutive time steps to approximate the differentiation of torque, which in turn is used for the neural networks to learn to minimize the difference. If we use a neural network with elapsed time as input to the network and with coordinates of arm positions as output, we can implement the minimum-torque-change requirement as the third-order differential constraint on the neural network. The neural network is much simpler and the constraint can be implemented more naturally by the neural networks learning differential data.

The other possible area of application is learning physical values, which satisfy partial differential equations (PDEs). Neural networks learning differential data should be especially useful where collecting data is costly. We give an example taken from meteorology. The temperature T and the wind velocity u satisfy the following PDE, where t stands for the time.

$$\frac{\partial T}{\partial t} + (u \cdot \nabla)T = 0 \qquad (1)$$

It is very costly to add observation points for the temperature and the wind velocity. For this kind of problem, we can use a neural network learning differential data, with time and coordinate as input and temperature and wind velocity as output. With such a neural network, we can use both the data from the existing observation points and the constraint given by the PDE for the neural network to learn.

For those examples, neural networks with capabilities to learn differential data can take advantage of such knowledge as constraints on differential data and incorporate such constraints easily into the learning of training value data.

We give a brief history of research on neural networks learning differential data in the following paragraphs.

In the research history of neural networks, there first appeared existence theorems of a multilayer perceptron, which approximates a given function. Funahashi [1] showed that there is a three-layered perceptron approximating any C^0 function with any precision, while Hornik et al. [3] showed there is also a perceptron, which approximates any C^n function with respect to C^n norm. On the other hand, learning of neural network by using a training set of value data was

Manuscript received ? ?, 2000.
Manuscript revised March 12, 2000.

[†]The author is with Intelligent Systems Laboratory, Computer Systems Laboratories, Fujitsu Laboratories Ltd.

justified by Gallant and White [2] who showed that a sequence of perceptron-produced functions which gives the least square error for randomly selected training samples, converges almost surely to a given function and its derivatives. These theorems, assuring the existence of convergent sequence of neural networks, give an important base toward realization of neural networks learning differential data.

However, these theorems are not very useful from a practical point of view, firstly because the number of training data required in the theorem would often be unrealistically large to realize, and secondly because practical learning methods do not necessarily give the minimum for the least square errors. Instead, if available, we should employ differential data themselves as learning data in the learning of neural networks. The learning algorithm of neural networks for first order differential data was first proposed under the name of 'tangent prop' by Simard et al. [8] who applied it to pattern recognition problem.

As our contributions to neural networks learning differential data, we gave results on noise robustness of multilayer perceptron which learns first order differential data in [5].

Then in [6], we have introduced an algorithm of multilayer neural networks learning differential data of arbitrary order. The algorithm proposed let the neural networks to learn differential data higher than first order in such cases as the simulation of human arm movement by minimum-torque-change model mentioned above. But there had been some minor mistakes in details.

In this paper, we propose a correct algorithm of multilayer neural networks learning differential data of arbitrary order. The algorithm employs the back propagation (BP) processes for derivatives of the target function up to the required order together with BP for the target function itself. The algorithm is rather simple for the first order differential data, but becomes rapidly complex as the differential order increases.

The proposed algorithm is then implemented for differential data of *arbitrary* order by coding it in the form of C++ program to show that the proposed algorithm is actually possible. The C++ implementation is checked numerically against the neural network model created in Mathematica [10].

By using the implementation, we carry out preliminary experiments on neural networks learning differential data and give some observations based on the results.

The section 2 gives the structure and the algorithm for the neural networks learning differential data. The section 4 gives results of preliminary experiments on the neural networks learning differential data and observations based on the results.

2. Algorithm for Learning Differential Data of Arbitrary Order

We describe the architecture of an extended multilayer perceptron and its algorithm to learn differential data.

2.1 Definitions

We give notations and their definitions.

n: dimension of the input to the network

$\delta = (a_1, ..., a_n) \in \{N \cup \{0\}\}^n$:
0 stands for $(0, ..., 0)$. We define half order $>$ in $\{N \cup \{0\}\}^n$ as the following.

$$\delta_1 = (a_1, ..., a_n) > \delta_2 = (b_1, ..., b_n)$$
$$\leftrightarrow$$
$$\exists i \quad a_i > b_i \ \wedge \ \forall i \quad a_i \geqq b_i$$

We see $\{N \cup \{0\}\}^n$ as the linear space, so that $\delta_1 + \delta_2$, $c\delta$ are defined as such. We also use δ as a differential operator. For $\delta = (a_1, ..., a_n)$, we define

$$\frac{\partial^{N(\delta)}}{\partial^\delta x} = \frac{\partial^{N(\delta)}}{\partial^{a_1} x_1 \ ... \ \partial^{a_n} x_n} \tag{2}$$

For a vector $x = (x_1, ..., x_n)$, a real number x^δ is defined as follows.

$$x^\delta = x_1^{a_1} \times \cdots \times x_n^{a_n} \tag{3}$$

$\Delta = \{(c_i, \delta_i) \mid i = 1, ..., m, \ \forall i, \ c_i \in N, \delta_i > 0\}$:
In the above definition, $\forall i \neq j \Rightarrow \delta_i \neq \delta_j$. We give the following definitions related to Δ.

$$N(\Delta) = m \tag{4}$$

$$T(\Delta) = \sum_{i=1}^{m} c_i \tag{5}$$

$$Sum(\Delta) = \sum_{i=1}^{m} c_i \times \delta_i \tag{6}$$

$$\Delta(f) = \prod_{(c,\delta)\in\Delta} \left(\frac{\partial^{N(\delta)}}{\partial^\delta x} \right)^c \tag{7}$$

$V(\delta) = \{(d_{\delta,\Delta}, \Delta) \mid \delta = Sum(\Delta)\}$:

Here $d_{\delta,\Delta}$ is a natural number defined by the following equation. f is a C^∞ function from the input space to the real number.

$$\frac{\partial^{N(\delta)}}{\partial^\delta x} \, e^f = \sum_{(d_{\delta,\Delta},\Delta) \, \in \, V(\delta)} d_{\delta,\Delta} \ \Delta(f) \ e^f \tag{8}$$

$p_{l,i}^{\Delta} = \prod_{(c,\delta')\in\Delta} \left(y_{l,i}^{\delta'} \right)^c$:
This is defined by the equation (16) again.

2.2 Framework of the problem

In this section, we will give the framework of the problem. n-dimensional input space, m-dimensional output space [†] and a C^l map from the input space to the output space are given. There are also given several points in the input space and the values of the map itself and differentials [††] of the map with respect to those points. Kinds of values given can vary for each point. The values can include some error or noise.

The problem is to find a neural network that approximates the given mapping under these conditions. We usually fix the network structure and make the network learn the internal values such as weights and thresholds from the given values.

2.3 Network Structure

Figure 1 shows the network structure that we propose to learn the higher order differential data. This network has extended parts in addition to a simple multilayer perceptron. Extended parts are used for propagating and back propagating differential data. This structure is an extension of Jacobian network that was used for tangent prop in [8]. On the left-hand side of figure 1 there are the units in the multilayer perceptron part, which we call the value net. There is corresponding δ net for each differential operator δ. Figure 1 shows one of those δ nets on the right side.

δ net has x^δ, y^δ, and $\sigma^{(m)}$ units for each $x^0 = x$ unit in the value net. $\sigma^{(m)}$ unit has input from the value net, which figure 1 does not show. Connection weights $w_{i,j}$ are same in the value net and δ nets.

We use the same symbol of the unit for denoting the output of the unit. The input to the unit is denoted by prefixing "*net* $\triangleright$" to the symbol of the unit. For example, the output of $x^\delta_{l,i}$ of i-th unit in l-th layer of δ-net is $x^\delta_{l,i}$ and the input to the unit is $net \triangleright x^\delta_{l,i}$.

The output of $x^\delta_{l,i}$ unit in the δ net is the δ differential of the corresponding unit in the value net by the input to the network. The next equation shows the relationship, where x_I is the input vector to the value net.

$$x^\delta_{l,i} = \frac{\partial^{N(\delta)} x^0_{l,i}}{\partial^\delta x_I} \tag{9}$$

The network structure is devised so as to realize the chain rule of the δ differentials by the input vector. Therefore the outputs of units in the output layer of the δ net are the δ differentials of the corresponding units in output layer of the value net by the input vector.

[†]Without any loss of generality, we assume one-dimensional input space in the following treatment.

[††]The differentials are not necessarily along the axes.

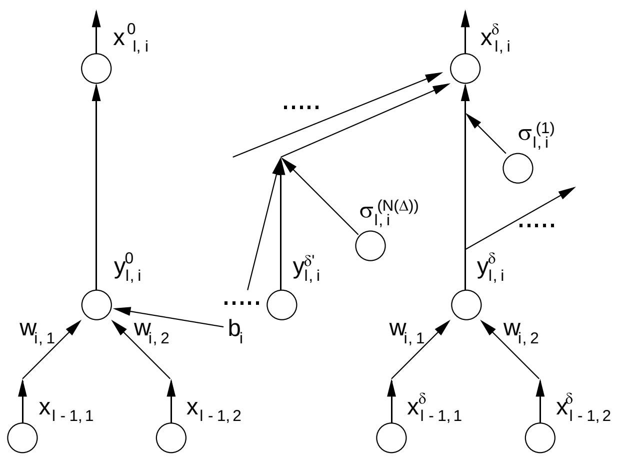

Fig. 1 *Network structure for learning differential data:* On the left-hand side of figure there are the units in the multilayer perceptron part, which we call the value net. There is corresponding δ net for each differential operator δ. This figure shows one of those δ nets on the right side. δ net has x^δ, y^δ, and $\sigma^{(m)}$ units for each $x^0 = x$ unit in the value net. $\sigma^{(m)}$ unit has input from $y^0_{l,i}$ units of the value net, which figure 1 does not show. Connection weights $w_{i,j}$ are same in the value net and δ nets.

We use the symbol ϵ for what is propagated backward through the network. The value of what is propagated backward to the unit is denoted by prefixing "$\epsilon \triangleright$" to the symbol of the unit. For example, what is propagated backward to $x^\delta_{l,i}$ of i-th unit in l-th layer of δ-net is $\epsilon \triangleright x^\delta_{l,i}$.

We sometimes omit the layer in denoting weights $w_{i,j}$ and thresholds b_i. To be correct these should be $w_{\{l,i\},\{l-1,j\}}$ and $b_{\{l,i\}}$.

2.4 Learning Data

This section describes the learning data for the network.

The directions of differentiation for the learning data need not be along axes. But we will limit to the cases of differentiation along axes, since the general cases need inessential and detailed explanation. The learning data is several sets of input and output, which called patterns.

The input part of the learning data for the network has the form of $I = (I^0, ..., I^\delta, ...)$. I^0 is the coordinate of the observation point and this is the same as in ordinary back propagation. I^0 is a vector of the same dimension as of the input space of the value net.

I^δ, the input to the δ net is also a vector of the same dimension as of the input space of the value net. In cases of $N(\delta) = 1$, the input vector is δ itself, which is a vector something like $(0, ..., 0, 1, 0, ..., 0)$. In other cases (i.e. $N(\delta) > 1$), the input vector is 0, the zero vector.

These are derived by thinking that the input to the δ net is a special case of equation (9). The input to δ net is δ differential of the input to the corresponding unit of the value net. In cases of $N(\delta) = 1$, the input vector is δ itself since $\partial x^0_{0,i}/\partial x^0_{0,i} = 1$ and since $\partial x^0_{0,i}/\partial x^0_{0,j} = 0$ if $i \neq j$. In cases of $N(\delta) > 1$, the input vector is 0 since $\partial x^0_{0,i}/\partial x^0_{0,j}\partial x^0_{0,k} = 0$ for the any combination of $\{i, j, k\}$.

The output part of the learning data for the network has the form of $O = (O^0, ..., O^\delta, ...)$. All the elements of the output are the element of the output space or "$*$". $O^\delta = *$ means that there is no differential data for δ and that there will be no back propagation for δ.

2.5 Forward Propagation

First we will give the algorithm for forward propagation. We will show how the input $I = (I^0, ..., I^\delta, ...)$ will be propagated forward through the network.

l stands for the layer in which the unit belongs. It is not essential that there exists a layer structure in the network. Here we assume the layer structure for the ease of explanation. The same algorithm applies for the networks without the layer structure.

2.5.1 Unit in the Value Net (FP)

Here we put $y^0_{l,i} = net \triangleright y^0_{l,i} = net \triangleright x^0_{l,i}$. The value net does not have any y^0 unit as in δ net, but we assume the above because of uniformity of the descriptions between the value net and the δ net.

The algorithm for the forward propagation for the unit in the value net is as follows. σ here is the squashing function for the unit. In this paper we use the same σ for every unit, but the same algorithm applies for the cases where units have different squashing functions.

$$\begin{aligned} net \triangleright x^0_{l,i} &= y^0_{l,i} = net \triangleright y^0_{l,i} \\ &= b_i + \sum_{j \in P(i)} w_{ij} x_{l-1,j} \end{aligned} \tag{10}$$

$$x^0_{l,i} = x_{l,i} = \sigma(net \triangleright x^0_{l,i}) = \sigma(y^0_{l,i}) \tag{11}$$

2.5.2 Unit in the δ Net (FP)

The input and output to the $\sigma^{(m)}_{l,i}$ unit are given by the following equations. $\sigma^{(m)}$ here is the m-th derivative of the function σ. Note that a $\sigma^{(m)}_{l,i}$ unit has $y^0_{l,i}$ for its input as a σ unit.

$$net \triangleright \sigma^{(m)}_{l,i} = y^0_{l,i} \tag{12}$$

$$\sigma^{(m)}_{l,i} = \sigma^{(m)}(y^0_{l,i}) \tag{13}$$

The input and output for $y^\delta_{l,i}$ unit are the same and given by the following equations.

$$y^\delta_{l,i} = net \triangleright y^\delta_{l,i} = \sum_j w_{i,j} x^\delta_{l-1,j} \tag{14}$$

The input and output of $x^\delta_{l,i}$ unit are the same and given by the following equations. Units of this kind are Sigma-Pi units described in [7].

$$\begin{aligned} x^\delta_{l,i} &= net \triangleright x^\delta_{l,i} = \frac{\partial^{N(\delta)}}{\partial^\delta x}(x^0_{l,i}) \\ &= \sum_{(d_{\delta,\Delta},\Delta) \in V(\delta)} d_{\delta,\Delta} \left\{ \sigma^{(T(\Delta))}_{l,i} \prod_{(c,\delta') \in \Delta} \left(y^{\delta'}_{l,i} \right)^c \right\} \end{aligned} \tag{15}$$

The product about Δ will be kept as $p^\Delta_{l,i}$.

$$p^\Delta_{l,i} = \prod_{(c,\delta') \in \Delta} \left(y^{\delta'}_{l,i} \right)^c \tag{16}$$

This is not essential for the algorithm, but it affects the efficiency of the implementation of the algorithm.

<u>proof of the last equation in (15)</u>

If we set $f(x_I) = y^0_{l,i}$ where $y^0_{l,i}$ is being seen as a function of x_I, then $\Delta(f)$ equals to the Π product in equation (15), which is $p^\Delta_{l,i}$. Since $x^0_{l,i} = \sigma(y^0_{l,i}) = \sigma(f(x_I))$, essentially the same mechanism applies to both equations (15) and (8) by the correspondence of the σ function in (15) to the exponential function e in (8). Therefore $d_{\delta,\Delta}$ in equation (15) is exactly the same as the one in equation (8). □

2.6 Error Function

We define the error function by the following equation.

$$E = \sum_{\delta \geqq 0} \alpha^\delta E^\delta \tag{17}$$

Here α^δ is a learning constant corresponding to δ.

For the set of patterns (I_p, O_p) where $p = I^0$, E^δ is calculated as follows. First each I_p is fed into the network, and propagated forward through the network. Then using the output of network and the given output O_p that is sometimes called teacher signal, E^δ is given by the following equation. [†]

$$E^\delta = \frac{1}{2} \sum_p \left(O^\delta_p - x^\delta_p \right)^2 \tag{18}$$

Usual back propagation algorithm does not include the learning constant in the energy, but the constant is used to scale the update values for weights [7]. Our algorithm includes the learning constants in the energy. This is because each δ has an individual learning constant. If we take the former way [††], update values for

[†] Only the subscript for the pattern is given for x in order to avoid too complex equations.

[††] That is to scale the update values by learning constants.

each δ have to be managed and more memory space is necessary. It is unpractical for the higher order differentials. So the following algorithm is designed to include the learning constants before the ϵ's are given to the units in output layer.

2.7 Back Propagation

Using E defined in section 2.6, $w_{i,j}$ is updated as follows.

$$\Delta w_{i,j} = -\frac{\partial E}{\partial w_{i,j}} \tag{19}$$

First we are going to show the back propagation algorithm of ϵ for each type of units. Then we will show the update rules for the weights and thresholds. In the following equations, we assume the case of one pattern and omit the subscript of the pattern.

ϵ's which are propagated backward through the network correspond to what are propagated in ordinary back propagation. That is "the differential of the error function with respect to the input to the unit." We denote that value by $\epsilon \rhd u$ for the unit u. Therefore the definition of $\epsilon \rhd u$ is given by the following equation for the error function E.

$$\epsilon \rhd u = -\frac{\partial E}{\partial net \rhd u} \tag{20}$$

The following sections give the back propagation algorithms for the types of units. This algorithm is for a set of one pattern (I_p, O_p), but it is easily extendable to the case of sets of multiple patterns by summation. It is assumed that the input I_p is fed to the network, and forward propagated through the network.

2.7.1 Units in the output layer (BP)

ϵ's for the units x^δ's in the output layer [†] are given as follows. (In the following equations subscripts for the layer and the unit are omitted to avoid too complex equations.)

First $\epsilon \rhd x^0$, which is ϵ for the units in the output layer of value net, is given by the following equation.

$$\begin{aligned}\epsilon \rhd x^0 &= -\frac{\partial E}{\partial net \rhd x^0}\\ &= \alpha^0 \left(O_p^0 - x_p^0\right) \sigma^{(1)}(net)\end{aligned} \tag{21}$$

In cases of $\delta > 0$, it is given by the following equation.

$$\epsilon \rhd x^\delta = -\frac{\partial E}{\partial net \rhd x^\delta} = \alpha^\delta \left(O_p^\delta - x_p^\delta\right) \tag{22}$$

[†]If the network does not have the layer structure, the output layer is the set of units whose outputs are interpreted as parts of the output of the network. For the δ net, only x^δ units are included. y^δ units and $\sigma^{(m)}$ units are not included.

2.7.2 Units in the Value Net (BP)

For the units in the value net other than in the output layer, the back propagation algorithm is essentially the ordinary back propagation algorithm with ϵ of σ unit.

$$\begin{aligned}\epsilon \rhd x_{l-1,i}^0 &= -\frac{\partial E}{\partial net \rhd x_{l-1,i}^0}\\ &= \sum_j \epsilon \rhd y_{l,j}^0 \; w_{j,i} \; \sigma_{l-1,i}^{(1)}\end{aligned} \tag{23}$$

$$\begin{aligned}\epsilon \rhd y_{l-1,i}^0 &= -\frac{\partial E}{\partial net \rhd y_{l-1,i}^0}\\ &= \epsilon \rhd x_{l-1,i}^0 + \sum_{m \geqq 1} \epsilon \rhd \sigma_{l-1,i}^{(m)}\end{aligned} \tag{24}$$

2.7.3 Units in the δ Net (BP)

For the $x_{l-1,j}^\delta$ units in δ net other than in the output layer, the back propagation algorithm will be given by the following equation.

$$\epsilon \rhd x_{l-1,i}^\delta = -\frac{\partial E}{\partial net_{l-1,i}^\delta} = \sum_j \epsilon \rhd y_{l,j}^\delta \; w_{j,i} \tag{25}$$

For the $\sigma_{l,i}^{(m)}$ units in δ net, the back propagation algorithm will be given by the following equation.

$$\begin{aligned}\epsilon \rhd \sigma_{l,i}^{(m)} &= -\frac{\partial E}{\partial net \rhd \sigma_{l,i}^{(m)}} \quad \sigma_{l,i}^{(m+1)}\\ &= \sum_{\delta : N(\delta) \geqq m} \epsilon \rhd x_{l,i}^\delta\\ &\quad \times \left(\sum_{\Delta : T(\Delta) = m \,\wedge\, Sum(\Delta) = \delta} d_{\delta,\Delta} \quad p_{l,i}^\Delta \right)\\ &\quad \times \sigma_{l,i}^{(m+1)}\end{aligned} \tag{26}$$

For the $y_{l,i}^\delta$ units in δ net, the back propagation algorithm will be given by the following equation. (Note that $\partial net_{l,i}^{\delta'} / \partial net \rhd y_{l,i}^\delta = \partial x_{l,i}^{\delta'} / \partial y_{l,i}^\delta$.)

$$\begin{aligned}\epsilon \rhd y_{l,i}^\delta &= -\frac{\partial E}{\partial net \rhd y_{l,i}^\delta}\\ &= \sum_{\delta' : \delta' \geqq \delta} \epsilon \rhd x_{l,i}^{\delta'}\\ &\quad \sum_{\Delta : Sum(\Delta) = \delta' \wedge \exists c\, (c,\delta) \in \Delta} d_{\delta',\Delta} \sigma_{l,i}^{(T(\Delta))} \; c \left(y_{l,i}^\delta\right)^{c-1}\\ &\quad \left(\prod_{(c'',\delta'') \in (\Delta - \{(c,\delta)\})} \left(y_{l,i}^{\delta''}\right)^{c''} \right)\end{aligned} \tag{27}$$

These algorithms summarized for $\delta > 0$ and $m \geqq 1$ as follows.

$$\epsilon \rhd x_{l-1,i}^\delta = \sum_j \epsilon \rhd y_{l,j}^\delta \; w_{j,i} \tag{28}$$

$$\epsilon \triangleright \sigma_{l,i}^{(m)} = \sum_{\delta:N(\delta)\geqq m} \epsilon \triangleright x_{l,i}^{\delta} \left(\sum_{\Delta:T(\Delta)=m \,\wedge\, Sum(\Delta)=\delta} d_{\delta,\Delta} \quad p_{l,i}^{\Delta} \right) \sigma_{l,i}^{(m+1)} \tag{29}$$

$$\epsilon \triangleright y_{l,i}^{\delta} = \sum_{\delta':\delta'\geqq\delta} \epsilon \triangleright x_{l,i}^{\delta'} \sum_{\Delta:Sum(\Delta)=\delta' \wedge \exists c\ (c,\delta)\in\Delta} d_{\delta',\Delta}\ \sigma_{l,i}^{(T(\Delta))}\ c\ (y_{l,i}^{\delta})^{c-1} \left(\prod_{(c'',\delta'')\in(\Delta-\{(c,\delta)\})} (y_{l,i}^{\delta''})^{c''} \right) \tag{30}$$

2.8 Update rules for weights

In this section, we will show the update rules for weights and thresholds. The same weight shows up in the value net and in the corresponding places of the δ net in figure 1. Therefore the update value for the weight should be the sum of update values for those corresponding weights. Using equations for ϵ obtained in section 2.7, the update rule for the weight $w_{i,j}$ is as follows.

$$\Delta w_{i,j} = -\frac{\partial E}{\partial w_{i,j}} = -\sum_{\delta\geqq 0} \frac{\partial E}{\partial net \triangleright y_{l,i}^{\delta}} \frac{\partial net \triangleright y_{l,i}^{\delta}}{\partial w_{i,j}}$$
$$= \sum_{\delta\geqq 0} \epsilon \triangleright y_{l,i}^{\delta}\ x_{l-1,j}^{\delta} \tag{31}$$

Note that weight updates due to $\sigma^{(m)}$ units ($m \geqq 1$) are included in $\epsilon \triangleright y_{l,i}^{0}$.

Thresholds b_i's [†] are special kinds of weights, which appear only in the value net. Therefore the update rule for the threshold is as follows.

$$\Delta b_i = -\frac{\partial E}{\partial b_i} = \epsilon \triangleright y_{l,i}^{0} \tag{32}$$

Note that bias updates due to $\sigma^{(m)}$ units ($m \geqq 1$) are also included in $\epsilon \triangleright y_{l,i}^{0}$.

In cases we use momentum (c.f. [7]) in the learning algorithm, the update rules are as follows. (Here $\tilde{\Delta} w_{i,j}$ and $\tilde{\Delta} b_i$ stand for previous update values.)

$$\Delta w_{i,j} = \sum_{\delta\geqq 0} \epsilon \triangleright y_{l,i}^{\delta}\ x_{l-1,j}^{\delta} + \beta\ \tilde{\Delta} w_{i,j} \tag{33}$$

$$\Delta b_i = \epsilon \triangleright y_{l,i}^{0} + \beta\ \tilde{\Delta} b_i \tag{34}$$

[†]Thresholds are sometimes called biases.

3. Correspondence to Tangent Prop in First Order Cases

For the first order cases, the following correspondences will establish the equivalence between tangent prop as described in [8] and the algorithm proposed here. The left hand side item of tangent prop of the arrow corresponds to the right hand side item of the algorithm proposed here.

- network $\rightarrow$ value net
- $a_i^l \rightarrow y_{l,i}^0$, $x_i^l \rightarrow x_{l,i}^0$, $b_i^l \rightarrow -\frac{\partial E}{\partial x_{l,i}^0}$, $y_i^l \rightarrow \epsilon \triangleright y_{l,i}^0$
- Jacobian network $\rightarrow$ δ net
- $\alpha_i^l \rightarrow y_{l,i}^{\delta}$, $\xi_i^l \rightarrow x_{l,i}^{\delta}$, $\beta_i^l \rightarrow \epsilon \triangleright x_{l,i}^{\delta}$, $\psi_i^l \rightarrow \epsilon \triangleright y_{l,i}^{\delta}$

4. Experiments

In this section, we give results of preliminary experiments on this algorithm and some observations based on the results.

The algorithm proposed in section 2 is implemented for differential data of an *arbitrary* order by coding it in the form of C++ program. The C++ implementation is checked numerically against the neural network model created in Mathematica [10]. This C++ implementation is used to carry out the experiments described in this section.

We use the following sine function as a target function, which ranges [0.1, 0.9] for the domain of [0.0, 1.0].

$$f(x) = 0.4\ \sin(2\pi x) + 0.5 \tag{35}$$

We carried out three experiments to compare the proposed algorithm with the standard back propagation (BP). First we give the common settings for all three cases and then we give the individual settings for each case.

The neural network and its initial state are the same for all cases. We use a neural network with one input unit, 16 hidden units, and one output unit. The weights and biases are initialized with the values chosen randomly from the interval $[-0.01, 0.01]$. The momentum was set for 0.1.

Here follows individual settings.

(1) Case 1 (Up to second order with three data points):

We trained the network with the algorithm proposed in this paper up to the second order. Learning constants are 1.0, 0.01, and 0.001 for value data, first order differential data, and second order differential data, respectively. Three training data points were randomly selected from the interval $[0, 1]$. The neural networks are trained for the 1,000,000 learning epochs on the training data up to second order from the three training data points.

(2) Case 2 (Standard BP with three data points):

We trained the network with the standard back propagation algorithm. Learning constants is 1.0 for value data. Three training data points are the same ones as in Case 1. The neural networks are trained for the 10,000 learning epochs on the value data from the three training data points.

(3) Case 3 (Standard BP with nine data points):

We trained the network with the standard back propagation algorithm. Learning constants is 1.0 for value data. Nine training data points, 0.1, 0.2, ..., 0.9, were selected uniformly from the interval $[0, 1]$. The neural networks are trained for the 1,000,000 learning epochs on the value data from the nine training data points.

We carry out the experiment in Case 2 to see how well the standard back propagation algorithm performs if the number of data points is the same as in Case 1.

We carry out the experiment in Case 3 to see how well the standard back propagation algorithm performs if the number of data given to the neural network is the same as in Case 1. Since there are value, first order differential, and second order differential data for each data point, there is three times more data for each data point in Case 1 than the standard back propagation cases. Therefore nine data points are selected for Case 3.

Figures 2, 3, 4, show the value, the first order, and the second order outputs of the trained neural networks for three cases. Naturally the first and the second order outputs are much closer to those of target function in Case 1 than Case 2 and Case 3 on the training data points.

Table 1 is given to see the overall performance of the trained neural networks on the interval $[0, 1]$. $\| \|_{2,0}$, $\| \|_{2,1}$, and $\| \|_{2,2}$ distances on the interval $[0, 1]$ between the target function and the trained neural network are given in the table for each case. These distances are given by the following equations where $n(x)$ is the function defined by the neural network.

$$\|f - n\|_{2,l} = \left\{ \int_0^1 \left(\frac{d^l f(x)}{dx^l} - \frac{d^l n(x)}{dx^l} \right)^2 \right\}^{1/2} \quad (36)$$

Figure 5 gives learning curves for all the cases. The graphs show the changes of errors for Case 1, Case 2, and Case 3 from the top respectively.

We describe several observations obtained from those experiments.

First of all, the trained neural network in Case 1 succeeded to learn to approximate all the training data up to second order on the training data points. This also supports the correctness of the proposed algorithm along with the check by Mathematica.

In the table 1, the $\| \|_{2,0}$ distance from the target function of the neural network in Case 1 happens to be smaller than the one in Case 2. This might suggests that the neural networks learning differential data can give better overall performance over the domain than the standard back propagation neural networks if the constraints on differential data are given along with the constraints on value data.

	$\| \|_{2,0}$	$\| \|_{2,1}$	$\| \|_{2,2}$
Case 1	0.0731	0.828	12.4
Case 2	0.171	1.467	8.84
Case 3	0.0448	0.909	13.6

Table 1 $\| \|_{2,0}$, $\| \|_{2,1}$, and $\| \|_{2,2}$ distances: $\| \|_{2,0}$, $\| \|_{2,1}$, and $\| \|_{2,2}$ distances on the interval $[0, 1]$ between the target function and the trained neural network are given for each case.

On the other hand, the $\| \|_{2,2}$ distance from the target function of the neural network in Case 1 happens to be larger than the one in Case 2. This was quite unexpected. The reason behind it may be that learning on both value and first order differential data has affected learning on second-order differential data in Case 1.

The neural network in Case 3 performs better than the network in Case 1 in terms of the $\| \|_{2,0}$ distance probably because its training data in Case 3 consists only of value data.

Another observation is that it took fairly large learning epochs in Case 3, which are comparable to those in Case 1. We expected to have large learning epochs for Case 1 since learning differential data will need subtle adjustments, but not for Case 3 since there is only training value data to learn. This might suggest that the number of learning epochs needed does not depend on the differential order of the data but on the size of the training data. [†]

For the above observations, we need to carry out analyses and further experiments to confirm them.

5. Conclusion

This paper introduced a correct algorithm for multilayer neural networks to learn differential data of arbitrary differential order. Even though the algorithm is complex, this enables to utilize knowledge given in the forms of constraints on differential data along with constraints on value data. We also gave preliminary experiment results comparing our algorithm with standard back propagation, and observations based on them.

It is still early to give general conclusions from those experiments of a small number, but the results seems to suggest the possibilities of the proposed algorithm as an adequate learning algorithm for the cases described in the introduction.

[†]The actual learning time also depends on the time needed for each epoch, which is longer for higher differential orders even if the size of training data is the same. Nevertheless it is an interesting point to consider theoretically.

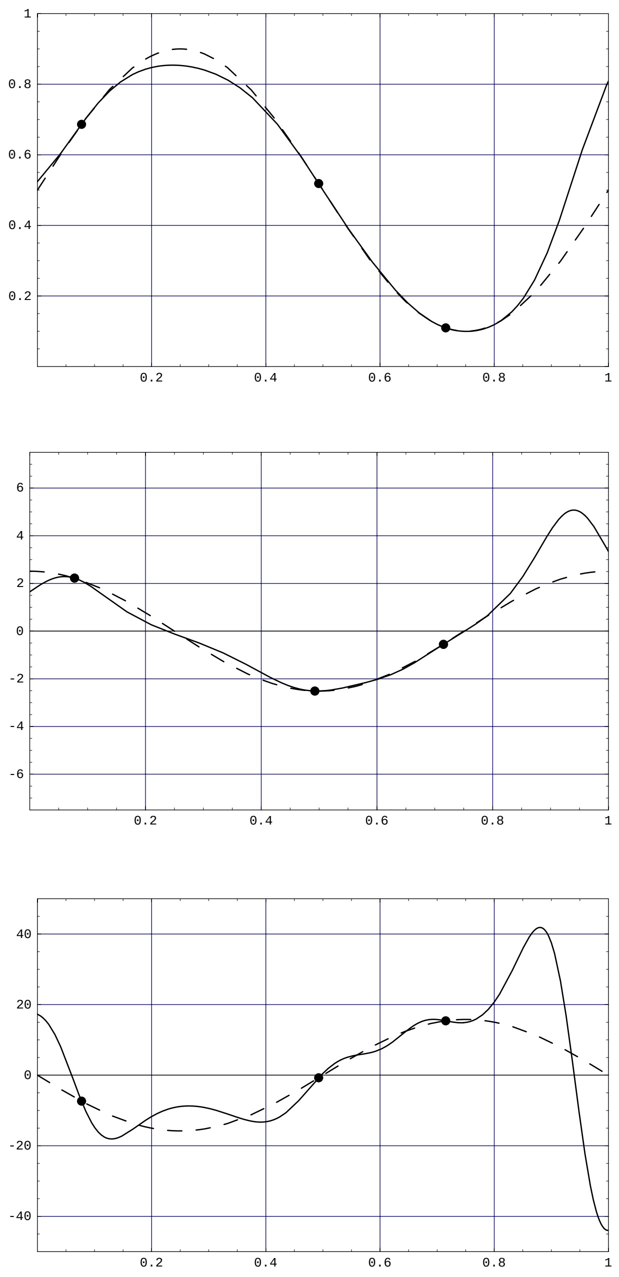

Fig. 2 [Case 1] *Outputs of the trained neural network:* The graphs show the value, first order, and second order outputs of the neural network in Case 1 from the top respectively. The solid lines show the outputs of the trained neural network in Case 1 after 1,000,000 learning epochs. The dotted line is the output of the target function. The dots represent the three training data points.

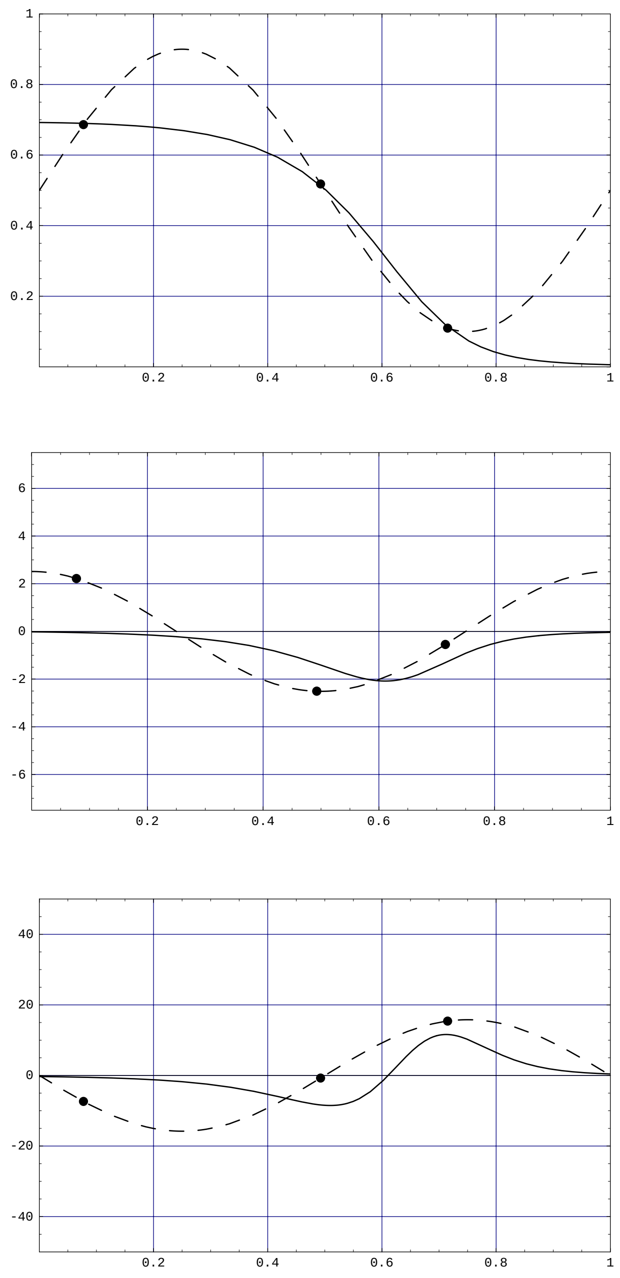

Fig. 3 [Case 2] *Outputs of the trained neural network:* The graphs show the value, first order, and second order outputs of the neural network in Case 2 from the top respectively. The solid lines show the outputs of the trained neural network in Case 2 after 10,000 learning epochs. The dotted line is the output of the target function. The dots represent the three training data points.

Our future plans include carrying out analyses and further experiments on neural networks learning differential data to confirm the obtained observations, to identify convergence properties, and to establish utilities of the proposed algorithm.

References

[1] K. Funahashi, On the approximate realization of continuous mappings by neural networks, Neural Networks, vol. 2, pp.183-192, 1989

[2] A. R. Gallant, and H. White, On learning the derivatives of

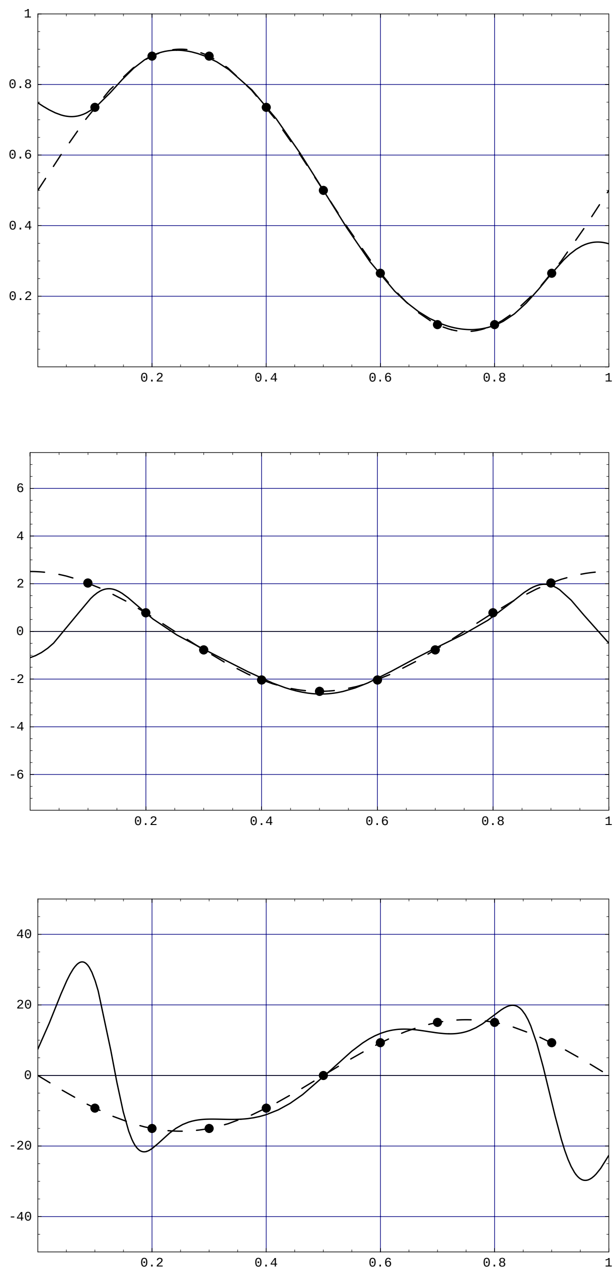

Fig. 4 [Case 3] *Outputs of the trained neural network:* The graphs show the value, first order, and second order outputs of the neural network in Case 3 from the top respectively. The solid lines show the outputs of the trained neural network in Case 3 after 1,000,000 learning epochs. The dotted line is the output of the target function. The dots represent the nine training data points.

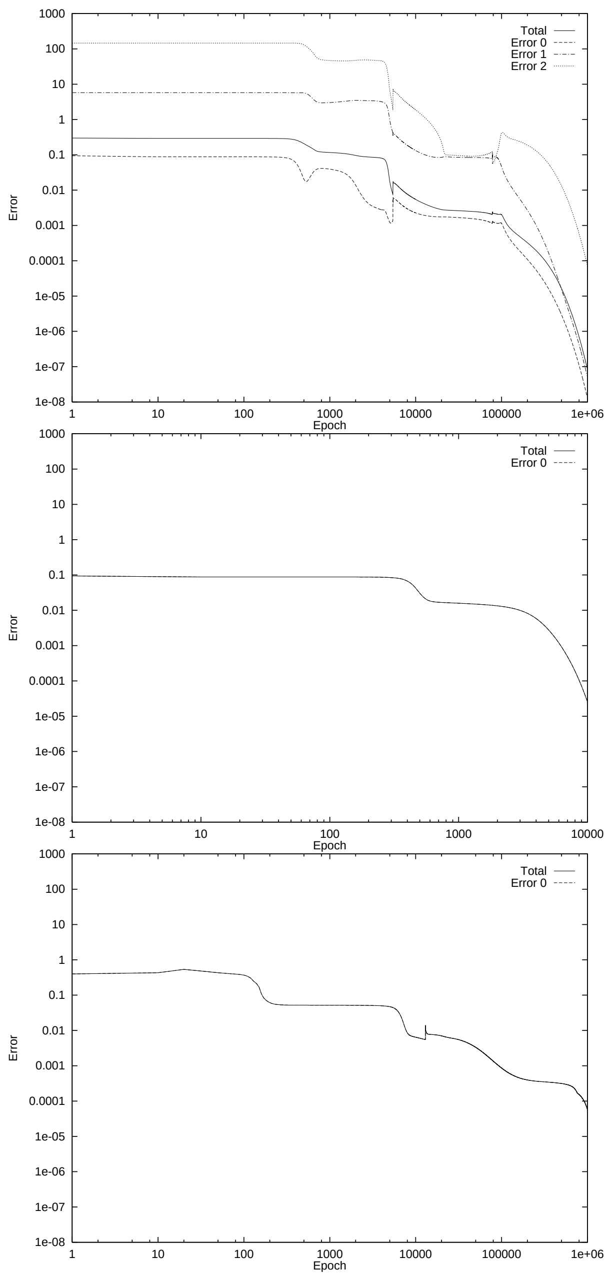

Fig. 5 *Learning curves:* The graphs show the changes of errors for Case 1, Case 2, and Case 3 from the top respectively. The axes of the graphs have logarithmic scaling. “Total”, “Error 0”, “Error 1”, and “Error 2” stand for E in equation 17, value, first order, and second order errors (E^{δ}s in equation 18) respectively. For Cases 2 and 3, “Total” and “Error0” are the same because the learning constant for the cases is 1.0. Note that Case 1 and Case 3 have the learning epochs of 1,00,000 while Case 2 has learning epochs of 10,000.

an unknown mapping with multilayer feedforward networks, Neural Networks, vol.5, pp.129-138, 1992

[3] K. Hornik, M. Stinchcombe, and H. White, Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks, Neural Networks, vol.3, pp.551-560, 1990

[4] Y. Maeda, M. Kawato, Y. Uno, and R. Suzuki, Trajectory formation for human multi-joint arm by a cascade neural network model, IEICE Technical Report, vol. MBE 88-169, pp. 79 - 84, 1989 (In Japanese)

[5] R. Masuoka, Noise robustness of EBNN learning, Proc. of 1993 International Joint Conference on Neural Networks, IEEE, vol.2, pp.1665-1668, 1993

[6] R. Masuoka and M. Yamada, Neural networks which learn from differential data, IEICE Technical Report, vol. NC 94-66, pp. 49-56, 1995

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning internal representations by error propagation, in Parallel Distributed Processing, Vol. I + II, ed. D. E. Rumelhart and J. L. McClelland, pp.318-362, MIT Press, 1986

[8] P. Simard, B. Victorri, Y. LeCun, and J. Denker, Tangent prop – a formalism for specifying selected invariances in an adaptive network, in Advances in Neural Information Processing Systems 4, ed. J. E. Moody, S. J. Hanson, and R. P. Lippmann, pp.895-903, Morgan Kaufmann, San Mateo, CA, 1992

[9] Y. Uno and M. Kawato, Dynamic performance indices for trajectory formation in human arm movements, IEICE Technical Report, vol. NC 94-28, pp. 33 - 40, 1994 (In Japanese)

[10] Mathematica (Wolfram Research, Inc.), http://www.wolfram.com/

Ryusuke Masuoka received the B.S. and M.S. degrees in mathematics from Tokyo University, Tokyo, Japan in 1985 and 1987, respectively. He joined Fujitsu Laboratories Ltd. in 1988. His current research interests include neural networks, simulated annealing, and agent systems.