Constraining Neural Networks to Fit Target Slopes

Ryusuke Masuoka
Pattern Information Processing Laboratory
Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku Kawasaki 211, Japan
E-mail: ryusk@flab.fujitsu.co.jp

Sebastian Thrun
University of Bonn
Institut für Informatik III
Römestr. 164, D-5300 Bonn, Germany
E-mail: thrun@uran.informatik.uni-bonn.de

Tom M. Mitchell
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
E-mail: mitchell@cs.cmu.edu

June 11, 1993

Contents

1 Introduction **3**

2 Derivation of the Algorithms **3**
2.1 Definition and Terminology 4
2.2 Value-Jacobian Net 6
2.3 Forward Propagation 6
2.4 Error functions 8
2.5 Backpropagation (δ rule and ε rule) 10
2.6 Tangent Prop with the EBNN Slope Error 13
2.7 Weight and Bias Update 14
2.8 What Is Learned by the EBNN Slope Error 14

3 Comparisons between Backpropagation and Tangent Prop **15**
3.1 Complexity 15
3.1.1 Space Complexity 16
3.1.2 Time Complexity 17
3.1.3 Batch Learning 18
3.2 Foundation for Comparisons 18

3.3 A Simple Case . . . 19
3.4 A General Case . . . 22

4 Experiments **24**
4.1 How It Works : A Sample Case . . . 26
4.2 Sample Complexity . . . 32
4.3 Scaling Up and Irrelevant Features . . . 34
4.4 Noise in Target Values and Target Slopes . . . 34

5 Conclusion **37**

List of Figures

1 Value-Jacobian net . . . 7
2 Input and output for learning slopes . . . 11
3 Value error . . . 20
4 Slope error . . . 21
5 Value error and slope error combined . . . 21
6 Value errors on two points . . . 22
7 General case . . . 23
8 Surface of the target function . . . 27
9 Error curve of backpropagation . . . 28
10 Surface learned by backpropagation . . . 29
11 Error curves of EBNN tangent prop . . . 30
12 Surface learned by EBNN tangent prop . . . 31
13 Graphs of generalization errors . . . 32
14 Graphs of speedup effect . . . 33
15 Graphs of scaling up effect . . . 35
16 Graphs of mistake bound . . . 36
17 Performance of backpropagation . . . 38
18 Performance of tangent prop (1) . . . 39
19 Performance of tangent prop (2) . . . 40

Abstract

A variety of methods have recently been proposed for constraining neural networks to fit various constraints while being trained. One such approach is to constrain the function approximated by the network to fit desired slopes, or derivatives. Such slopes may be provided by the designer, as in Simard's character recognizer network which was constrained so that the slope of the output with respect to translations, rotations, etc. of the input should be zero. Alternatively, target slopes may be generated automatically by program as in Explanation Based Neural Network (EBNN) learning.

This paper explores the role of slope information in constraining neural network learning. In particular, we present a mathematical framework and algorithms for using slopes to constrain learning. This approach is then compared with standard backpropagation, in terms of the quality of learned generalizations, as well as time and space complexity. We provide an analytical comparison of learning to fit target slopes with the alternative of learning to fit extra synthesized points. Scaling properties and the robustness to training errors are also considered. Analytical and experimental results show that training using slopes leads to improved generalization at higher computational cost.

1 Introduction

Much recent research on neural network learning concerns the question of how to constrain learning in large networks. One promising approach is to constrain training by forcing the network to fit target *slopes*, or *derivatives*, of the learned function, in addition to the usual constraint that it must fit target *values* of the function. This approach of constraining networks to fit target slopes has been used, for example, in [Simard *et al.*, 1992], to constrain a character recognition network to produce outputs that are invariant to certain transformations of the inputs (e.g., rotation of the input character). It has also been used in the Explanation Based Neural Network (EBNN) alogorithm ([Mitchell and Thrun, 1993]) which uses prior knowledge encoded in previously learned networks to derive slope constraints that must be satisfied by the function to be learned.

The goal of this paper is to explore the use of slope information for constraining neural network backpropagation learning. We describe tangent prop algorithm presented in [Simard *et al.*, 1992] somewhat in a more general and detailed way for the later analysis. We examine the role of slope constraints in reducing the generalization error in learning, as well as the complexity of these algorithms and their robustness to errors in the training data. A combination of analytical and experimental results is presented, demonstrating that using slopes to constrain learning results in improved generalization at a higher computational cost.

Two related algorithms have been proposed for using slopes to constrain neural network learning. One is tangent prop proposed by Simard, Victorri, Le Cun, and Denker ([Simard *et al.*, 1992]) and the other is part of the EBNN learning procedure proposed by Mitchell and Thrun ([Mitchell and Thrun, 1993]). Although they use different error functions, the essential part of these two algorithms is mathematically identical. We call both procedures *tangent prop*. For experimental analyses, we use error functions used in the EBNN learning procedure along with tangent prop. We call this the *EBNN tangent prop*.

In this paper, first we provide a mathematical framework for tangent prop. Second, by estimating space and time complexity, we show how costly it is compared to back propagation. Third, through mathematical analysis of structures of minimal sets of energy functions, we elucidate the merit of learning slopes compared to standard backpropagation only with value patterns (extra patterns scheme or synthesized patterns scheme). Then using an illustrative example learning task of two-variable nonlinear function, we show what happens in learning slopes compared to usual back propagation. Lastly, we give experimental results about scaling up with respect to the number of irrelevant features and robustness to noise.

2 Derivation of the Algorithms

Tangent prop using Jacobian net was introduced by Simard, Victorri, Le Cun, and Denker ([Simard *et al.*, 1992]). First we describe the derivation of the formula in detail. Then we show some of idiosyncrasies of

tangent prop with the EBNN slope error, including a more efficient algorithm and a fact that it is essentially learning normals to the surface of the target function.

Before going into the details of the algorithm, we point out that the mathematical idea behind the algorithm is to use the *Sobolev norm* instead of L^2 norm in backpropagation. Let $f_t : x = (x_1, ..., x_I) \in R^I \rightarrow f_t(x) \in R^K$ be the target function $f_n : x \in R^I \rightarrow f_n(x) \in R^K$ be the function computed by the neural network. Let $|a-b|$ be the Euclidean distance between two points a and b in R^K. Then a finite point version of L_2 norm of

$$||f_t - f_n||^2_{L_2} = \int |f_t - f_n|^2 dx \tag{1}$$

is used as the error function for backpropagation. A finite point version of Sobolev (W_2^1) norm

$$||f_t - f_n||^2_{W_2^1} = ||f_t - f_n||^2_{L_2} + \int \sum_{i=1}^{I} \left| \frac{\partial f_t}{\partial x_i} - \frac{\partial f_n}{\partial x_i} \right|^2 dx \tag{2}$$

is used as EBNN slope error in the EBNN learning. [1] (See [Mazja, 1985] for Sobolev norms and Sobolev spaces.)

2.1 Definition and Terminology

The following definitions are used throughout this paper. A Jacobian net, a bar unit, and a underbar units are defined in Section 2.2.

[1] We are ignoring constants as they are not essential here.

nn	the function computed by the neural network
x_i, $x_{\underline{i}}$, $x_{\overline{i}}$	*activations* of unit i, underbar unit $\underline{i}$, and bar unit $\overline{i}$ respectively
net_i, $net_{\underline{i}}$, $net_{\overline{i}}$	*net-inputs* of the unit i, underbar unit $\underline{i}$, and bar unit $\overline{i}$ respectively
$y_{\overline{i}}$	defined by Equation (8)
w_{ij}	*weight* from unit j to unit i
b_i	*bias* of unit i
I, K	*numbers* of input units and output units respectively
H, N	*numbers* of all hidden units and all units respectively. (Note $I + H + K = N$)
n	*maximum number* of all units connected to or connected from any unit in the value net
Q_p	*number* of the directions for the pattern p
$L(i)$	the *layer* unit i is in ($L(i) = 0$ if i is an input unit)
$P(i)$	the *set of preceding units* of unit i, given by $P(i) = \{j \mid L(j) = L(i) - 1\}$
$S(i)$	the *set of succeeding units* of unit i, given by $P(i) = \{j \mid L(j) = L(i) + 1\}$
$\sigma(\cdot)$	*squashing function*, here $\sigma(z) := (1 + e^{-z})^{-1}$
$\vec{\xi}$	*vector* in the input space, $\vec{\xi} = (\xi_1, ..., \xi_I)$
$\vec{e_i}$	*i-th axial direction vector* in the input space R^I, $\vec{e_i} = (0, ..., 0, \underbrace{1}_{i}, 0, ..., 0)$
$\tilde{x}_k$	*desired output value* of output unit k of the value net
δ_k	defined by Equation (28)
β_i^k, $\beta_{\vec{\xi}}^k$	*output Jacobian* of $\overline{k}$ with respect to input Jacobian $\vec{e_i}$ and to $\vec{\xi}$ respectively
$\tilde{\beta}_i^k$, $\tilde{\beta}_{\vec{\xi}}^k$	*desired output Jacobian* of $\overline{k}$ with respect to input Jacobian $\vec{e_i}$ and to $\vec{\xi}$ respectively
ε_u, $\varepsilon_{l+\underline{l}}$, $\varepsilon_{\overline{l}}$	defined by Equation (33), (36), (37) respectively
E_{bp}, E_{tp}, E_{ebnn}	*total errors* for back propagation, tangent prop, and EBNN learning respectively
E_{v}, E_{s}, $E_{\text{ebnn-s}}$	*value, slope, and EBNN slope errors* respectively
$\overline{E}_{\text{v}}$, $\overline{E}_{\text{s}}$, $\overline{E}_{\text{ebnn-s}}$	*normalized value, slope and EBNN slope errors* respectively
η_{v}, η_{s}, $\eta_{\text{ebnn-s}}$	*stepsize* for value, slope, and EBNN slope error respectively
α	*momentum coefficient*
Δw_{ij}, $\Delta_{\text{v}} w_{ij}$	*weight changes* with respect to total and value errors
$\Delta_{\text{s}} w_{ij}$, $\Delta_{\text{ebnn-s}} w_{ij}$	*weight changes* with respect to slope and EBNN slope errors
$\tilde{\Delta}_{\text{ebnn-s}} w_{ij}$	*auxiliary variable* for EBNN slope errors
$\overline{\Delta} w_{ij}$	*last (total) weight change*
Δb_i, $\Delta_{\text{v}} b_i$	*bias changes* with respect to total and value
$\Delta_{\text{s}} b_i$, $\Delta_{\text{ebnn-s}} b_i$	*bias changes* with respect to slope and EBNN slope errors
$\overline{\Delta} b_i$	*last (total) bias change*
$\rho_{jj'}^k$	*second order auxiliary gradient*

We will use the names in the left column of the following table with their meanings in the corresponding rows.

name	meaning
"input value" "point in the input space"	input to the value net
"output value"	output of the value net
"input Jacobian" "direction in the input space"	input to the Jacobian net
"output Jacobian"	output of the Jacobian net
"desired output value"	desired output of the value net
"desired output Jacobian"	desired output of the Jacobian net
"input"	pair of an input value and an input Jacobian
"output"	pair of an output value and an output Jacobian
"value pattern"	pair of an input value and an output value
"Jacobian pattern"	pair of an input Jacobian and an output Jacobian
"pattern"	pair of an input and output
"value pattern set"	set of value patterns
"Jacobian pattern set"	set of Jacobian patterns
"pattern set"	set of patterns
"EBNN tangent prop"	tangent prop with the EBNN slope error

2.2 Value-Jacobian Net

Learning constrained by slopes is most easily understood by imagining a network that contains additional virtual units to encode slopes. We will call this elaborated network the value-Jacobian net, as explained in this section. This discussion, though it is modified and more detailed, is based on the discussion in [Simard *et al.*, 1992].

We call the original back propagation neural network, the *value net*. We add an underbar unit $\underline{l}$ and a bar unit $\overline{l}$ for each unit l in the value net and connect them in the following fashion. (See Figure 1.)

The underbar unit $\underline{l}$ gets inputs from the same units as the unit l. The weights between the underbar unit $\underline{l}$ and these inputs are always identical to the weights between the unit l and these units. We use the symbol w_{lj} instead of $w_{\underline{l}j}$. The bias of the underbar unit $\underline{l}$ is also set to the same value as for the unit l and notated b_l instead of $b_{\underline{l}}$.

The bar unit $\overline{l}$ has inputs from the underbar unit $\underline{l}$ and the bar units $\overline{j}$'s corresponding to the preceding units of the unit l. Connections from the bar units $\overline{j}$ converge and produce $y_{\overline{l}}$. Then it forms a (unit-weight) sigma-pi connection ([Rumelhart *et al.*, 1986]) with the connection from the underbar unit $\underline{l}$. The weight for the connection from the unit $\overline{j}$ is set to the value of w_{lj} and notated w_{lj} instead of $w_{\overline{lj}}$.

We call the network of units $\overline{l}$'s and $\underline{l}$'s, the *Jacobian net* after ([Simard *et al.*, 1992]) and the complex of the value net and Jacobian net, the value-Jacobian net. Weights and biases which appear in the Jacobian net have the same values to corresponding weights and biases in the value net. The input units of the Jacobian net are those bar units which correspond the input units of the value net. The output units of the Jacobian net are those bar units which correspond the output units of the value net.

2.3 Forward Propagation

We will explain the forward propagation in the value-Jacobian net.

For forward propagation, we give a value input $v \in R^I$ for the value net and a Jacobian input $\vec{\xi} \in R^I$ for the Jacobian net.

The forward propagation in the value net is independent from the Jacobian net and the same as in a

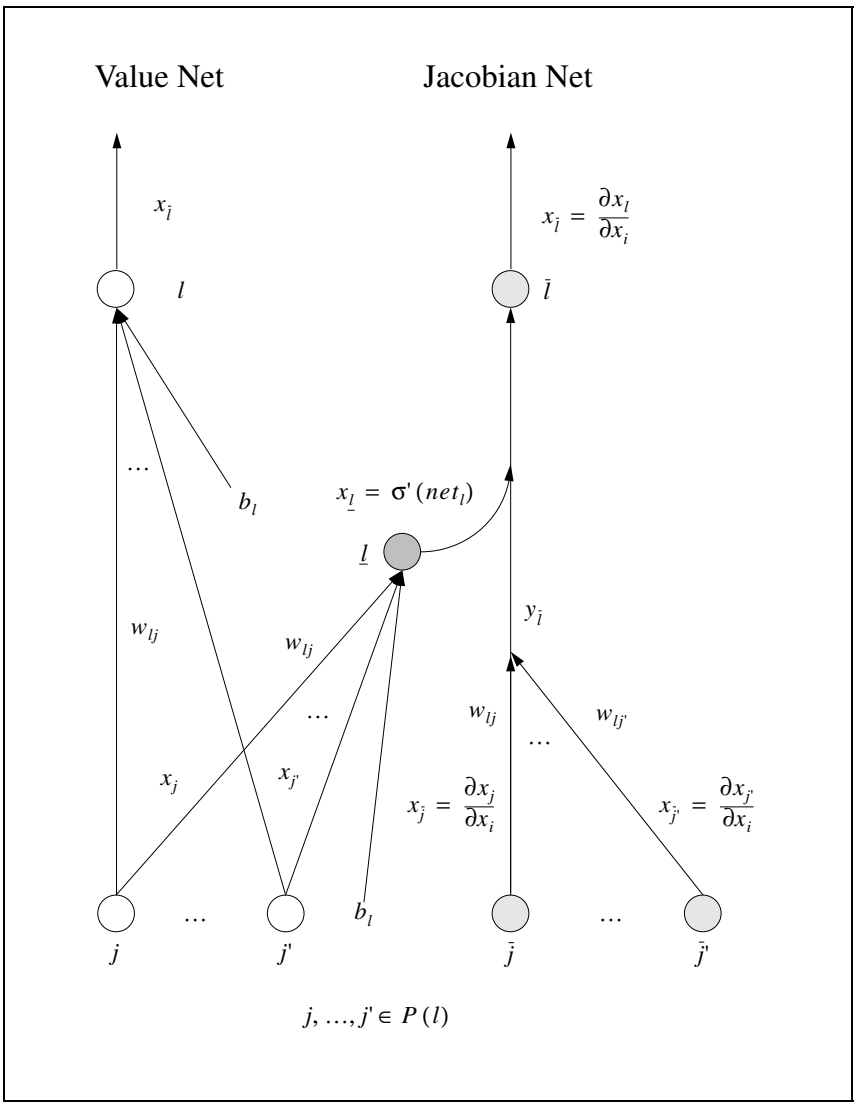

Figure 1: **Value-Jacobian net:** From a part of the original backpropagation network including the units l and $j, ..., j'$, the corresponding part of value-Jacobian net is elaborated. In this case, i-th axial directional vector $\vec{e_i}$ is fed to the Jacobian net as an input.

standard backpropagation net. It is given by: [2]

$$net_i = b_i + \sum_{j \in P(i)} w_{ij} x_j \tag{3}$$

$$x_i = \sigma(net_i) \tag{4}$$

The forward propagation in the Jacobian net is given by following rules for underbar units and bar units.

The unit $\underline{l}$ has exactly the same inputs as the unit l. The activation function of $\underline{l}$ is the derivative σ' of activation function σ of l.

$$net_{\underline{l}} = net_l = b_l + \sum_{j \in P(l)} w_{lj} x_j \tag{5}$$

$$x_{\underline{l}} = \sigma'(net_{\underline{l}}) = \sigma'(net_l) \tag{6}$$

The unit $\overline{l}$ is a linear sigma-pi unit ([Rumelhart *et al.*, 1986]) with the identity function as the activation function and receives inputs from the unit $\underline{l}$ and bar units of preceding units of the unit l.

$$x_{\overline{l}} = net_{\overline{l}} = x_{\underline{l}} \sum_{j \in P(l)} w_{lj} x_{\overline{j}} = x_{\underline{l}} y_{\overline{l}} \tag{7}$$

In the above equation, we defined :

$$y_{\overline{l}} = \sum_{j \in P(l)} w_{lj} x_{\overline{j}} \tag{8}$$

Weights appeared in the above equations are the same ones in the value net.

By setting up the nets this way, we will have the directional derivative $\sum_{i \text{ is input}} (\partial x_l / \partial x_i) \times \xi_i$ of the activation of the unit l with respect to the vector $\vec{\xi} = (\xi_1, ..., \xi_I)$ in the input space as the activation $x_{\overline{l}}$ of the bar unit $\overline{l}$ if we give the training data for the value net and $(\xi_1, ..., \xi_I)$ for the Jacobian net as input. (See Figure 2.) This is true for input units of the Jacobian net and it can be also shown to be true for units in other parts of the Jacobian net by using mathematical induction:

$$\begin{aligned} \sum_{i \text{ is input}} \frac{\partial x_l}{\partial x_i} \xi_i &= \sum_{i \text{ is input}} \sum_{j \in P(l)} \frac{\partial x_l}{\partial x_j} \frac{\partial x_j}{\partial x_i} \xi_i = \sum_{j \in P(l)} w_{lj} \sigma'(net_l) \sum_{i \text{ is input}} \frac{\partial x_j}{\partial x_i} \xi_i \\ &= \sum_{j \in P(l)} w_{lj} x_{\underline{l}} x_{\overline{j}} = x_{\overline{l}} \end{aligned} \tag{9}$$

Note the followings. $\beta^k_{\vec{\xi}}$ is the activation of the bar unit $\overline{k}$ of the Jacobian net with $\vec{\xi}$ as the input Jacobian. Relationships between β^k_i and $\beta^k_{\vec{\xi}}$ are :

$$\beta^k_{\vec{\xi}} = \sum_{i \text{ is input}} \beta^k_i \xi_i \tag{10}$$

$$\beta^k_i = \beta^k_{\vec{e}_i} \tag{11}$$

2.4 Error functions

We distinguish three types of errors: The *value error*:

$$E_{\mathrm{v}} := \frac{1}{2} \sum_{p \text{ is pattern}} \sum_{k \text{ is output}} (\tilde{x}_k - x_k)^2 \tag{12}$$

[2] Though we used same σ for all the units in the value net, it is not necessary for them to be same. The backpropagation algorithms for the value-Jacobian net still holds with appropriate care of which σ is in use.

the *slope error* with respect to the set of the directions $\{\{\vec{\xi}_q\}_{q=1,\ldots,Q_p}\}_p$ in the input space:

$$E_{\mathrm{s}} := \frac{1}{2} \sum_{p \text{ is pattern}} \sum_{q=1,\ldots,Q_p} \sum_{k \text{ is output}} (\tilde{\beta}^k_{\vec{\xi}_q} - \beta^k_{\vec{\xi}_q})^2 \tag{13}$$

and the *EBNN slope error* (which is a special case of slope errors):

$$E_{\text{ebnn-s}} := \frac{1}{2} \sum_{p \text{ is pattern}} \sum_{i \text{ is input}} \sum_{k \text{ is output}} (\tilde{\beta}^k_i - \beta^k_i)^2 \tag{14}$$

For the sake of clarity we do *not* label any value with its pattern label p. Note that $\{\vec{\xi}_q\}_{q=1,\ldots,Q_p}$ can be a totally different set for a different pattern p. We also sometimes use the following normalized error functions.

$$\overline{E}_{\mathrm{v}} := \frac{1}{K} E_{\mathrm{v}} \tag{15}$$

$$\overline{E}_{\mathrm{s}} := \frac{1}{K} E_{\mathrm{s}} \tag{16}$$

$$\overline{E}_{\text{ebnn-s}} := \frac{1}{I \times K} E_{\text{ebnn-s}} \tag{17}$$

The total error for each algorithm, which a gradient descent method search the (local) minimum of, is given below. The total error for backpropagation is :

$$E_{\mathrm{bp}} = \eta_{\mathrm{v}} E_{\mathrm{v}} \tag{18}$$

The total error for tangent prop is :

$$E_{\mathrm{tp}} = \eta_{\mathrm{v}} E_{\mathrm{v}} + \eta_{\mathrm{s}} E_{\mathrm{s}} \tag{19}$$

The total error for EBNN learning is :

$$E_{\mathrm{ebnn}} = \eta_{\mathrm{v}} E_{\mathrm{v}} + \eta_{\text{ebnn-s}} E_{\text{ebnn-s}} \tag{20}$$

Any other terms can be added to these total errors. Possible terms include regularizers such as

$$c \left(\sum |w_{lj}|^2 + \sum |b_l|^2 \right) \tag{21}$$

with some positive constant c.

We set the update values for weights and biases with respect to each energy as follows.

$$\Delta_{\mathrm{v}} w_{lj} = -\frac{\partial E_{\mathrm{v}}}{\partial w_{lj}} \tag{22}$$

$$\Delta_{\mathrm{v}} b_l = -\frac{\partial E_{\mathrm{v}}}{\partial b_l} \tag{23}$$

$$\Delta_{\mathrm{s}} w_{lj} = -\frac{\partial E_{\mathrm{s}}}{\partial w_{lj}} \tag{24}$$

$$\Delta_{\mathrm{s}} b_l = -\frac{\partial E_{\mathrm{s}}}{\partial b_l} \tag{25}$$

$$\Delta_{\text{ebnn-s}} w_{lj} = -\frac{\partial E_{\text{ebnn-s}}}{\partial w_{lj}} \tag{26}$$

$$\Delta_{\text{ebnn-s}} b_l = -\frac{\partial E_{\text{ebnn-s}}}{\partial b_l} \tag{27}$$

Finite step gradient descent based on the linear combination of these energy functions can be realized by linearly combining these update values accordingly.

2.5 Backpropagation (δ rule and ε rule)

Here we derive the backpropagation algorithms to calculate update values $\Delta_{\rm v} w_{lj}$, $\Delta_{\rm v} b_l$ and $\Delta_{\rm s} w_{lj}$,$\Delta_{\rm s} b_l$, corresponding to $E_{\rm v}$ and $E_{\rm s}$ respectively. The next section derives the algorithm to calculate $\Delta_{\text{ebnn-s}} w_{lj}$, $\Delta_{\text{ebnn-s}} b_l$ corresponding to $E_{\text{ebnn-s}}$.

In this and the next section we consider the case in which the number Q_p (see Equation 13) of directions for the pattern is 1 and in which the pattern set consists of one pattern. For the case in which the pattern set consists of more than one direction or more than one pattern, one can simply repeat the algorithm for each slope error or the algorithm for each pattern in the pattern set, add up the update values for weights and biases, and update weights and biases with the results at the end.

As for $\Delta_{\rm v} w_{lj}$, $\Delta_{\rm v} b_l$, there is no contribution to the calculation of $E_{\rm v}$ from the Jacobian net. Therefore the standard back propagation algorithm derived in [Rumelhart *et al.*, 1986] can be used. Here we summarize the algorithm. δ_j for a unit j in the value net is defined by:

$$\delta_j = -\frac{\partial E_{\rm v}}{\partial net_j} \tag{28}$$

δ's for output units k's are

$$\delta_k = (\tilde{x_k} - x_k)\sigma'(net_k) \tag{29}$$

The backpropagation rule for δ (δ rule) for hidden units is

$$\delta_j = \sigma'(net_j) \sum_{l \in S(j)} \delta_l w_{lj} \tag{30}$$

And update values for weights and biases are

$$\Delta_{\rm v} w_{lj} = -\frac{\partial E_{\rm v}}{\partial w_{lj}} = \delta_l x_j \tag{31}$$

$$\Delta_{\rm v} b_l = -\frac{\partial E_{\rm v}}{\partial b_l} = \delta_l \tag{32}$$

To derive backpropagation rules for slope errors, let the value-Jacobian net be propagated forward with the input value v for the value net and the $\vec{\xi} = (\xi_1, ..., \xi_I)$ for the Jacobian net. Then we calculate the backpropagation rule of δ for the slope error. In order to avoid confusion we call δ for the slope error, "ε" and the δ rule for the slope error, the "ε rule." Therefore ε_u for u being a unit j in the value net, an underbar unit $\underline{j}$, or an bar unit $\overline{j}$ in the Jacobian net is given by:

$$\varepsilon_u = -\frac{\partial E_{\rm s}}{\partial net_u} \tag{33}$$

As the slope error is given by Equation 13, the ε's for output units k's and $\overline{k}$'s are the following. (See Figure 2 and note that we are assuming $Q_p = 1$.)

$$\varepsilon_{\overline{k}} = (\tilde{\beta}_i^k - x_{\overline{k}}) \tag{34}$$

$$\varepsilon_k = 0 \tag{35}$$

(Note that the unit $\overline{k}$ has the identity function as its activation function.)

We will derive ε-rule (i.e. δ rule for the value net and the Jacobian net combined). A unit j in the value net has the connections to the units in $\{l, \underline{l} | l \in S(j)\}$.

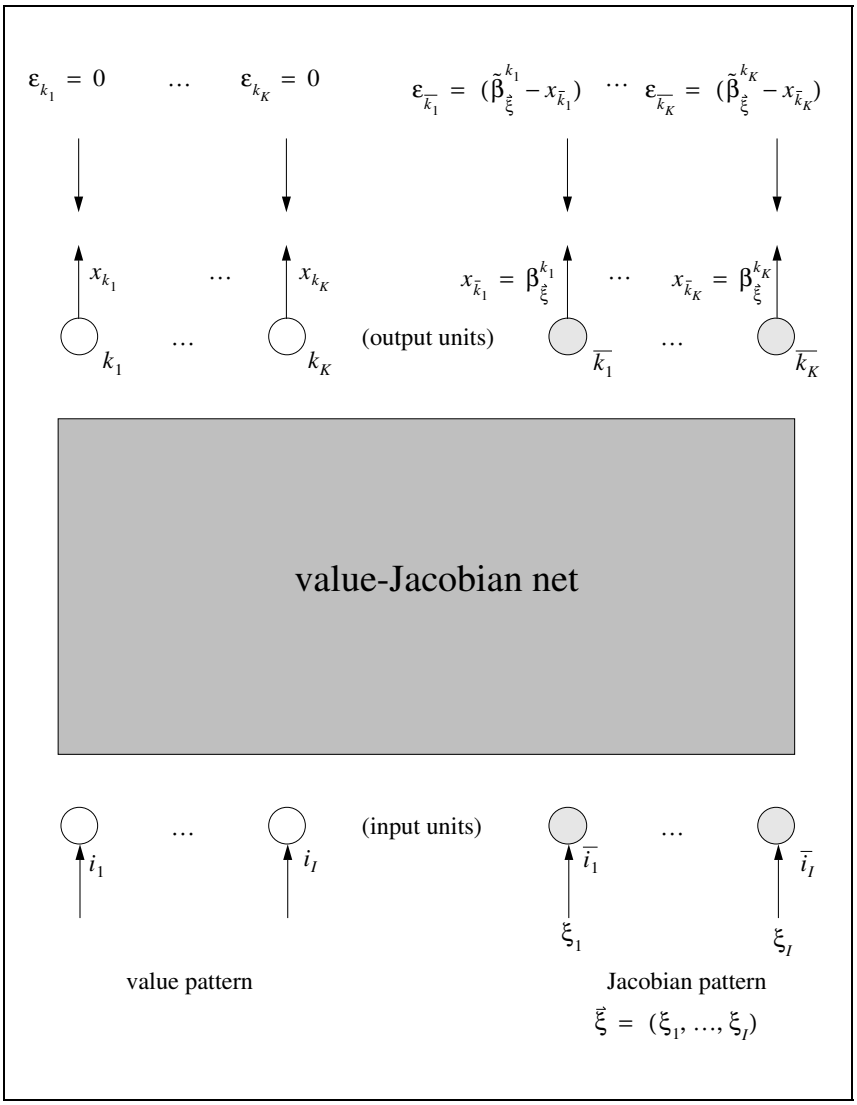

Figure 2: **Input and output for learning slopes:** The outputs of the value-Jacobian net and the slope errors which are propagated backward through the value-Jacobian net are shown. As a Jacobian input, $\vec{\xi} = (\xi_1, ..., \xi_I)$ is being given to the Jacobian net.

$$
\begin{aligned}
\varepsilon_j &= -\frac{\partial E_{\rm s}}{\partial net_j} = \sum_{l \in S(j)} \left[-\frac{\partial E_{\rm s}}{\partial net_l} - \frac{\partial E_{\rm s}}{\partial net_{\underline{l}}} \right] \frac{\partial net_l}{\partial net_j} \\
&= \sigma'(net_j) \sum_{l \in S(j)} [\varepsilon_l + \varepsilon_{\underline{l}}] w_{lj} = x_{\underline{j}} \sum_{l \in S(j)} [\varepsilon_l + \varepsilon_{\underline{l}}] w_{lj}
\end{aligned}
$$

An underbar unit $\underline{l}$ in the Jacobian net has the sigma-pi connections to the unit $\overline{l}$.

$$
\begin{aligned}
\varepsilon_{\underline{l}} &= -\frac{\partial E_{\rm s}}{\partial net_{\underline{l}}} = -\frac{\partial E_{\rm s}}{\partial net_{\overline{l}}} \frac{\partial net_{\overline{l}}}{\partial net_{\underline{l}}} \\
&= \varepsilon_{\overline{l}} \frac{\partial}{\partial net_{\underline{l}}} [x_{\underline{l}} y_{\overline{l}}] = \varepsilon_{\overline{l}} \sigma''(net_l) y_{\overline{l}}
\end{aligned}
$$

A bar unit $\overline{j}$ in the Jacobian net has the sigma-pi connections to the units $\{\overline{l} | j \in P(l)\}$.

$$
\begin{aligned}
\varepsilon_{\overline{j}} &= -\frac{\partial E_{\rm s}}{\partial net_{\overline{j}}} = \sum_{l \in S(j)} -\frac{\partial E_{\rm s}}{\partial net_{\overline{l}}} \frac{\partial net_{\overline{l}}}{\partial net_{\overline{j}}} \\
&= \sum_{l \in S(j)} \varepsilon_{\overline{l}} \underbrace{\frac{\partial x_{\overline{j}}}{\partial net_{\overline{j}}}}_{=1} \frac{\partial net_{\overline{l}}}{\partial x_{\overline{j}}} = \sum_{l \in S(j)} \varepsilon_{\overline{l}} w_{lj} x_{\underline{l}}
\end{aligned}
$$

The same w_{lj} appears in the three places, two connections from unit j to l and $\underline{l}$ and one sigma-pi connection from unit $\overline{j}$ and $\underline{l}$ to $\overline{l}$. So the increment to the weight w_{lj} is :

$$
\begin{aligned}
\Delta_{\rm s} w_{lj} &= -\frac{\partial E_{\rm s}}{\partial w_{lj}} \\
&= -\frac{\partial E_{\rm s}}{\partial net_l} \frac{\partial net_l}{\partial w_{lj}} - \frac{\partial E_{\rm s}}{\partial net_{\underline{l}}} \frac{\partial net_{\underline{l}}}{\partial w_{lj}} - \frac{\partial E_{\rm s}}{\partial net_{\overline{l}}} \frac{\partial net_{\overline{l}}}{\partial w_{lj}} \\
&= [\varepsilon_l + \varepsilon_{\underline{l}}] x_j + \varepsilon_{\overline{l}} x_{\underline{l}} x_{\overline{j}}
\end{aligned}
$$

The same b_l appears in the two places, two connections to l and $\underline{l}$. So the increment to the bias b_l is :

$$
\begin{aligned}
\Delta_{\rm s} b_l &= -\frac{\partial E_{\rm s}}{\partial b_l} \\
&= -\frac{\partial E_{\rm s}}{\partial net_l} \frac{\partial net_l}{\partial b_l} - \frac{\partial E_{\rm s}}{\partial net_{\underline{l}}} \frac{\partial net_{\underline{l}}}{\partial b_l} \\
&= \varepsilon_l + \varepsilon_{\underline{l}}
\end{aligned}
$$

We define $\varepsilon_{l+\underline{l}}$ and $\varepsilon_{\overline{\underline{l}}}$ as follows. [3]

$$
\varepsilon_{l+\underline{l}} = \varepsilon_l + \varepsilon_{\underline{l}} \tag{36}
$$

$$
\varepsilon_{\overline{\underline{l}}} = \varepsilon_{\overline{l}} x_{\underline{l}} \tag{37}
$$

[3] $\varepsilon_{l+\underline{l}}$ can be thought as an ε of the following unit. The unit (say, it is the unit $\ddot{l}$) gets inputs from those units $j, ..., j'$ connected to the unit l. The output of the unit is $net_l = \sum_j w_{lj} x_j$. The units l and $\underline{l}$ get their inputs from the unit $\ddot{l}$ instead of from the units $j, ..., j'$ connected to l and $\underline{l}$ originally. We did not include the unit $\ddot{l}$ in our explanation because it would be too tedious. But this is the reason why backpropagation rules for ε's can be written in $\varepsilon_{l+\underline{l}}$ and $\varepsilon_{\overline{l}}$.

By using $\varepsilon_{l+\underline{l}}$, the above equations are summarized as follows. For an output unit k,

$$\varepsilon_{\overline{k}} = (\tilde{\beta}_i^k - x_{\overline{k}}) \tag{38}$$

$$\varepsilon_{\underline{k}} = \varepsilon_{\overline{k}} x_{\underline{k}} \tag{39}$$

$$\varepsilon_{k+\underline{k}} = \varepsilon_{\overline{k}} \sigma''(net_k)\, y_{\overline{k}} \tag{40}$$

for a hidden units j,

$$\varepsilon_{\overline{j}} = \sum_{l \in S(j)} \varepsilon_{\underline{l}} w_{lj} \tag{41}$$

$$\varepsilon_{\underline{j}} = \varepsilon_{\overline{j}} x_{\underline{j}} \tag{42}$$

$$\varepsilon_{j+\underline{j}} = x_{\underline{j}} \sum_{l \in S(j)} \varepsilon_{l+\underline{l}} w_{lj} + \varepsilon_{\overline{j}} \sigma''(net_j)\, y_{\overline{j}} \tag{43}$$

and for units l, j,

$$\Delta_{\rm s} w_{lj} = \varepsilon_{l+\underline{l}}\, x_j + \varepsilon_{\underline{l}}\, x_{\overline{j}} \tag{44}$$

$$\Delta_{\rm s} b_l = \varepsilon_{l+\underline{l}} \tag{45}$$

2.6 Tangent Prop with the EBNN Slope Error

The EBNN slope error function is given by Equation 14. Instead of picking one direction in the input space, EBNN adds up slope errors with respect to all axial direction vectors and there are several points we can take advantage of to make this calculation more efficient. (Rather than simply repeating the above algorithm described in the previous sections.) Forward and backward propagation algorithms for tangent prop with the EBNN slope error can be modified as the following. We call this "tangent prop with the EBNN slope error" the *EBNN tangent prop*.

As for forward propagation, the Jacobian net is given $\{\ \vec{e_i} = (0, ..., 0, \underbrace{1}_{i}, 0, ..., 0)\}_{i=1,...,I}$ as an input. For the input $\vec{e_i}$ and the unit j connected to input units, it is more efficient just to put

$$x_{\overline{j}} = w_{ji} \tag{46}$$

instead of the straightforward forward propagation (Equation 7) of the Jacobian net.

The same and other tricks can be used for backpropagation. For each training data, we first propagate forward the value net with the training data and propagate δ's backward through the value net to calculate $\Delta_{\rm v} w_{lj}$'s and $\Delta_{\rm v} b_l$'s for weights and biases. Second, we fix the activation of each unit in the value net and the activation of each underbar unit in the Jacobian net and reset $\tilde{\Delta}_{\text{ebnn-s}} w_{lj}$'s and $\Delta_{\text{ebnn-s}} b_l$'s to 0. ($\tilde{\Delta}_{\text{ebnn-s}} w_{lj}$'s are auxiliary variables to calculate $\Delta_{\text{ebnn-s}} w_{lj}$'s.) Then repeat the following procedure for the number of input units. For i-th repetition :

1. Propagate forward the input Jacobian $\vec{e_i} = (0, ..., \underbrace{1}_{i}, ..., 0)$ through the Jacobian net.

2. Propagate ε's backward through the whole value-Jacobian net.

3. For a unit j which is not an input unit and a unit l which is connected from j, add $\varepsilon_{\overline{l}}\, x_{\underline{l}}\, x_{\overline{j}}$ to each $\tilde{\Delta}_{\text{ebnn-s}} w_{lj}$.

$$\tilde{\Delta}_{\text{ebnn-s}} w_{lj} \longleftarrow \tilde{\Delta}_{\text{ebnn-s}} w_{lj} + \varepsilon_{\overline{l}}\, x_{\overline{j}} \tag{47}$$

For the input unit i and a unit l which is connected from i,

$$\tilde{\Delta}_{\text{ebnn-s}} w_{li} \longleftarrow \tilde{\Delta}_{\text{ebnn-s}} w_{li} + \varepsilon_{\underline{l}} \tag{48}$$

For an input unit j such that $j \neq i$ and a unit l which is connected from j, this step is omitted.

4. Add $\varepsilon_{l+\underline{l}}$ to each $\Delta_{\text{ebnn-s}} b_l$.

$$\Delta_{\text{ebnn-s}} b_l \longleftarrow \Delta_{\text{ebnn-s}} b_l + \varepsilon_{l+\underline{l}} \tag{49}$$

After these repetitions, we put

$$\Delta_{\text{ebnn-s}} w_{lj} = \Delta_{\text{ebnn-s}} b_l x_j + \tilde{\Delta}_{\text{ebnn-s}} w_{lj} \tag{50}$$

2.7 Weight and Bias Update

As usual with gradient descent search, weights and biases are primarily updated with the gradients of the error functions and sometimes with the momentum terms.

Updating rules for standard backpropagation are:

$$w_{lj} \longleftarrow w_{lj} + \eta_{\text{v}} \Delta_{\text{v}} w_{lj} + \alpha \overline{\Delta} w_{lj} \tag{51}$$

$$b_l \longleftarrow b_l + \eta_{\text{v}} \Delta_{\text{v}} b_l + \alpha \overline{\Delta} b_l \tag{52}$$

Updating rules for tangent prop are:

$$w_{lj} \longleftarrow w_{lj} + \eta_{\text{v}} \Delta_{\text{v}} w_{lj} + \eta_{\text{s}} \Delta_{\text{s}} w_{lj} + \alpha \overline{\Delta} w_{lj} \tag{53}$$

$$b_l \longleftarrow b_l + \eta_{\text{v}} \Delta_{\text{v}} b_l + \eta_{\text{s}} \Delta_{\text{s}} b_l + \alpha \overline{\Delta} b_l \tag{54}$$

Updating rules for EBNN learning are:

$$w_{lj} \longleftarrow w_{lj} + \eta_{\text{v}} \Delta_{\text{v}} w_{lj} + \eta_{\text{ebnn-s}} \Delta_{\text{ebnn-s}} w_{lj} + \alpha \overline{\Delta} w_{lj} \tag{55}$$

$$b_l \longleftarrow b_l + \eta_{\text{v}} \Delta_{\text{v}} b_l + \eta_{\text{ebnn-s}} \Delta_{\text{ebnn-s}} b_l + \alpha \overline{\Delta} b_l \tag{56}$$

Here $\eta_{\text{v}} > 0$, $\eta_{\text{s}} > 0$, and $\eta_{\text{ebnn-s}} > 0$ denote the *stepsizes* of the gradient descent procedure. Starting with some initial values for weights and biases (usually randomly drawn), the gradient descent update with momentum terms is iterated.

2.8 What Is Learned by the EBNN Slope Error

To see what is learned by the EBNN slope error, we assume the network has only one output unit for the sake of simplicity. Then the learning task is approximation of a target function $f : U \in R^I \rightarrow R$ by a neural network with I input units and one output unit, We refer to the function computed by the neural network as $nn(x_{i_1}, ..., x_{i_I})$. We define two I dimensional surfaces in $I+1$ dimensional Euclidean space as follows:

$$S_f = \{(x_{i_1}, ..., x_{i_I}, f(x_{i_1}, ..., x_{i_I})\} \tag{57}$$

$$S_{nn} = \{(x_{i_1}, ..., x_{i_I}, nn(x_{i_1}, ..., x_{i_I})\} \tag{58}$$

Normal vectors to these surfaces are given by

$$N_f = (-\nabla_X(f), 1) \tag{59}$$

$$N_{nn} = (-\nabla_X(nn), 1) \tag{60}$$

where $X = (x_{i_1}, ..., x_{i_I})$. Then we can rewrite the EBNN slope error by these normal vectors. Let $\{d_p\}_p$ be the input value pattern set (set of input values of elements in value pattern set).

$$E_{\text{ebnn-s}} = \frac{1}{2}\sum_p \left[|\nabla_X(f) - \nabla_X(nn)|^2\right]_{X=d_p} = \frac{1}{2}\sum_p \left[|N_f - N_{nn}|^2\right]_{X=d_p} \tag{61}$$

On the other hand E_{v} can be thought as the square of the distance between the points where two normals N_f and N_n are normal to.

Therefore learning normal vectors to the surface of the target function including the origins of these vectors is the essence of EBNN tangent prop. It learns the full tangent spaces of the target function not a particular direction in the tangent space.

3 Comparisons between Backpropagation and Tangent Prop

What is a fair comparison between standard backpropagation and tangent prop? There are many cases where you can get slope information along with value information at a very small cost, such as tangent prop applied to pattern recognition and EBNN learning applied to robot learning. In those cases, it is justified to use extra resources (space and time) in order to utilize additional information and to get more accurate results. But is that all? Is slope information just additional information? Is slope information not different from, for example, adding new value patterns?

In this section, we tried to answer above questions from a theoretical point of view. First, we will calculate space and time complexity for back propagation and for the algorithms of learning slopes. Then we give a mathematical formalization of the problem as a foundation for comparison. Using the above formalization and through the arguments of a simple case and a fully general case, we will show that adding a slope error function gives good characteristics to the set of global minima and some advantage over adding additional value error function for new value patterns. [4]

3.1 Complexity

In this section we compare the space and time complexity of backpropagation, tangent prop with one direction per pattern, and EBNN tangent prop. The rough estimates will show that the space requirement for tangent prop with one direction is about $\frac{4}{3}$ and that the space requirement for EBNN learning is about twice that for backpropagation. The time requirement for tangent prop with one direction per pattern is about $\frac{11}{4}$ times and that the time requirement for tangent prop with the EBNN slope error is about $2I + \frac{7}{2}$ times of that for backpropagation (where I is the number of network inputs).

The following analysis assumes that we use momentum terms in updating weights and biases. We also assume that we use on-line learning for the first two sections. Therefore we assume that the pattern set consists of only one pattern, one value pattern for backpropagation, one pair of a value pattern and a Jacobian pattern for tangent prop, and a value pattern and I (the number of input units) Jacobian patterns for EBNN tangent prop.

Even though we incorporate in the tables the algorithm from Section 2.6 for the estimate of the time requirement for EBNN tangent prop, the rough estimate (Equation 70) has dropped most of it. The reader should note that in some cases the algorithm from Section 2.6 gives several times speedup over the standard tangent prop algorithm.

We follow the notation of Section 2.1. As this analysis depends on the maximum number, n, of units in one layer it might give unnecessarily large bounds if there is too large an imbalance in the numbers of units in different layers.

[4] In this section we will set aside the problem of how to reach the global minima (i.e. learnability issue) and concentrate on the structure of the set of global minima.

3.1.1 Space Complexity

In the following analysis, we ignore the memory needed for keeping parameters (learning rates, stepsizes, ...) because the main purpose here is to see how much memory is needed as the size of network grows.

The space complexity for backpropagation is (The unit for this table is the memory needed to store one real value):

x_l	$(I + H + K)$
net_l	$(H + K)$
w, b	$(n + 1)(H + K)$
$\Delta_{\mathrm{v}} w$ and $\Delta_{\mathrm{v}} b$	$(n + 1)(H + K)$
$\overline{\Delta} w$ and $\overline{\Delta} b$	$(n + 1)(H + K)$
δ_l	$(H + K)$
Total	$I + (3n + 6)H + (3n + 6)K$

The addition to the space complexity for tangent prop with one direction is :

$\Delta_{\mathrm{s}} w$ and $\Delta_{\mathrm{s}} b$	$(n + 1)(H + K)$
$x_{\overline{l}}$ $(= net_{\overline{l}})$	$(I + H + K)$
$x_{\underline{l}}$	(H + K)
$\epsilon_{l+\underline{l}}$, $\epsilon_{\overline{l}}$	$2(H + K)$
$y_{\overline{l}}$	$(H + K)$
Total	$I + (n + 7)H + (n + 7)K$

The further addition to the space complexity for EBNN tangent prop is :

$\Delta_{\text{ebnn-s}} w$ and $\Delta_{\text{ebnn-s}} b$	$(n + 1)(H + K)$
$\tilde{\Delta}_{\text{ebnn-s}} w_{ij}$	$n(H + K)$
$\varepsilon_{\overline{l}}$, σ''_l	$2(H + K)$
Total	$(2n + 3)H + (2n + 3)K$

σ''_l's are needed for keep the value of the second derivatives of activation functions for hidden and output units.

Writing SC_{bp}, SC_{tp}, and SC_{ebnn} for each total space complexity, we can conclude the following

$$\begin{aligned} SC_{\mathrm{bp}} &= I + (3n + 6)H + (3n + 6)K && (62) \\ SC_{\mathrm{tp}} &= SC_{\mathrm{bp}} + I + (n + 7)H + (n + 7)K \\ &= 2I + (4n + 13)H + (4n + 13)K && (63) \\ SC_{\mathrm{ebnn}} &= SC_{\mathrm{tp}} + (2n + 3)H + (2n + 3)K \\ &= 2I + (6n + 16)H + (6n + 16)K && (64) \end{aligned}$$

If we can assume $n \simeq I \simeq H \simeq K$, then

$$\begin{aligned} SC_{\mathrm{bp}} &\simeq 6n^2 + 13n && (65) \\ SC_{\mathrm{tp}} &\simeq 8n^2 + 28n && (66) \\ SC_{\mathrm{ebnn}} &\simeq 12n^2 + 34n && (67) \end{aligned}$$

Comparing the fastest growing terms of these equations, we can give rough estimates. The space requirement for tangent prop with one direction is about $\frac{4}{3}$ and that for EBNN tangent prop is about twice that for backpropagation.

3.1.2 Time Complexity

In the following analysis of time complexity of algorithms, time needed to access memory is not included.

We first give the tables of computational resources for updating weights and biases and for the calculation of σ, σ', and σ''. We made these tables independent because the contributions of these differ by conditions such as whether we use on-line learning or batch learning, and whether we use the lookup table for σ, σ', and σ'' or not.

Time complexity for updating weights and biases is :

	$+ - \times /$	Equations
backpropagation	$(4n+4)(H+K)$	51, 52
tangent prop with one direction	$(6n+6)(H+K)$	53, 54
EBNN tangent prop	$(6n+6)(H+K)$	55, 56

Time complexity for the calculation of σ, σ', and σ'' is :

	σ	σ'	σ''	Equations
backpropagation	$(H+K)$	$(H+K)$	0	4, 30
tangent prop with one direction	0	0*	$(H+K)$	6, 40, 43
EBNN tangent prop	0	0*	$(H+K)$†	6, 40, 43

The time complexity for backpropagation is :

	$+ - \times /$	Equations
forward propagation	$2n(H+K)$	3, 4
backward propagation (δ)	$2nH+2K$	29, 30
$\Delta_v w$	$n(H+K)$	31
$\Delta_v b$	0	32
Total	5nH+(3n+2)K	

The time complexity for tangent prop with one direction is :

	$+ - \times /$	Equations
forward propagation ($\overline{l}$)	$2n(H+K)$	7, 8
forward propagation ($\underline{l}$)	0	5, 6
backward propagation ($\varepsilon_{\overline{l}}$, $\varepsilon_{\overline{j}}$)	$2nH+2K$	38, 39, 41, 42
backward propagation ($\varepsilon_{l+\underline{l}}$)	$(2n+3)H+2K$	40, 43
$\Delta_s w$	$3n(H+K)$	44
$\Delta_s b$	0	45
Total	(9n+3)H+(5n+4)K	

We define H_0 as the number of the hidden units connected to the input units and H_1 as the number of hidden units not connected the input units. So $H = H_0 + H_1$ holds.

The time complexity of backpropagation for EBNN tangent prop is :

	$+ - \times /$	Equations
forward propagation ($\overline{l}$)	$2nI(H_1+K)$	7, 8, 46
forward propagation ($\underline{l}$)	0	5, 6
backward propagation ($\varepsilon_{\overline{l}}$, $\varepsilon_{\overline{j}}$)	$2nIH+2IK$	38, 39, 41, 42
backward propagation ($\varepsilon_{l+\underline{l}}$)	$(2n+3)IH+2IK$	40, 43
$\Delta_{\text{ebnn-s}} w$	$(I+2n)H_0+(2nI+2n)(H_1+K)$	47, 48, 50
$\Delta_{\text{ebnn-s}} b$	$I(H+K)$	49
Total	$[(4n+5)I+2n]H_0+[(8n+4)I+2n]H_1$ + [(4n+5)I+2n]K	

*This calculation can be done as a part of backpropagation by the value net.

†This calculation is necessary only once for each pattern.

Writing $TC_{\rm bp}$, $TC_{\rm tp}$, and $TC_{\rm ebnn}$ for each total time complexity (except updating and σ's), If we can assume $n \simeq I \simeq H \simeq K$, then

$$\begin{aligned} TC_{\rm bp} &\simeq 8n^2 + 2n & (68)\\ TC_{\rm tp} &\simeq TC_{\rm bp} + 14n^2 + 7n \\ &= 22n^2 + 9n & (69)\\ TC_{\rm ebnn} &\simeq TC_{\rm bp} + 16n^2 I + 14nI + 6n^2 \\ &\simeq (16I + 28)n^2 + 2n & (70)\end{aligned}$$

Comparing the fastest growing terms of these equations, we can give rough estimates. The time requirement for tangent prop with one direction is about $\frac{11}{4}$ times and that for EBNN tangent prop is about $2I + \frac{7}{2}$ times of that for backpropagation.

3.1.3 Batch Learning

In case of the batch learning, one needs extra space and time.

For space complexity, one needs memories to accumulate weight and bias changes ($\Delta_{\rm v}w$, $\Delta_{\rm v}b$, $\Delta_{\rm s}w$, $\Delta_{\rm s}b$, $\Delta_{\text{ebnn-s}}w$, $\Delta_{\text{ebnn-s}}b$). These are $(n+1)(H+K)$ units for backpropagation and $2(n+1)(H+K)$ units for tangent prop with one direction and EBNN tangent prop.

For time complexity, one needs time to accumulate the above weight and bias changes. These are $(n+1)(H+K)$ units/pattern for back propagation and $2(n+1)(H+K)$ units/pattern for tangent prop and EBNN learning. (A unit here is the time for the calculation of one of $+ - \times /$.)

Notice that updating weights and biases costs much less for batch learning because updating these happens only after all patterns are propagated forward and backward.

3.2 Foundation for Comparisons

Let $f_t : U \subset R^I \to R^m$ be the target function that has to be learned. Let the neural network in consideration have r parameters (that are weights and biases). For $w \in R^r = W$, define $f_n(w) : U \subset R^I \to R^m$ as the function computed by the neural network with parameters set to w. Let ge be *generalization error* such that [5]

$$ge(f) = \int_U |f_t(p) - f(p)|^2 \, dp \tag{71}$$

The strategy of standard backpropagation is to minimize the mean square error between the values of the functions f_t and $f_n(w)$ on the training patterns by gradient descent. The strategy of the slope-constrained methods is to use the value *and slope* errors combined instead of the value only error.

For a fixed set of points $P = \{p_1, p_2, ..., p_r\} \subset U$ we define functionals (as functions on W) $e_{\rm v}$ and $e_{\rm s}$ and sets as follows :

$$e_{\rm v}(w) \quad = \quad e_{\rm v}(f_n(w)) = \sum_{i=1}^{r} |f_t(p_i) - f_n(w)(p_i)|^2 \tag{72}$$

[5] The problem is more generally formulated as follows: Fix a function f_t in $\{f : U \subset R^I \to R^m\}$. Define a functional ge over the set $V \subset \{f : U \subset R^I \to R^m\}$ whose codomain is R. Assume that $ge(f) \geq 0$ for any $f \in V$ and that $ge(f_t) = 0$. For example, ge could be the L^p-distance or L^∞-distance from the function f_t with appropriate measure on U. The measure might be used to express the probability distribution of data. We call the functional ge the *generalization error*. The problem is to find a function $f \in V$ with the property that $ge(f_t) = \min_{f \in V} ge(f)$ with access to information of f_t on a finite set of points in U.

With these definitions in place, we can think of V as a version space for this problem and ge as fitness of these hypotheses (i.e. functions). By identifying the set of weights (including biases) $w \in R^n = W$ and the function $f_n(w)$ computed by the neural network with the weight w, we can identify the version space V and the weight space W. By identifying this way, these functionals ge, $e_{\rm v}$ and $e_{\rm s}$ on the version space V can be thought as functions on the weight space W.

$$e_{\rm s}(w) \quad = \quad e_{\rm s}(f_n(w)) = \sum_{i=1}^{r}\sum_{i=1}^{I}\left|\frac{\partial f_t}{\partial x_i}(p_i) - \frac{\partial f_n(w)}{\partial x_i}(p_i)\right|^2 \tag{73}$$

$$Min(ge) = \{w \in W \,|\, ge(w) \text{ is minimum}\} \tag{74}$$
$$Min(e_{\rm v}) = \{w \in W \,|\, e_{\rm v}(w) \text{ is minimum}\} \tag{75}$$
$$Min(e_{\rm s}) = \{w \in W \,|\, e_{\rm s}(w) \text{ is minimum}\} \tag{76}$$
$$Min(e_{\rm v} + e_{\rm s}) = \{w \in W \,|\, e_{\rm v}(w) + e_{\rm s}(w) \text{ is minimum}\} \tag{77}$$

Note that we used $e_{\rm s}$ instead of more appropriate $e_{\text{ebnn-s}}$ for the sake of simplicity of notation. as we use only EBNN slope error in this section.

These four sets $Min(ge)$, $Min(e_{\rm v})$, $Min(e_{\rm s})$, $Min(e_{\rm v} + e_{\rm s})$ are unrelated in general. But if there is a weight w in the weight space that satisfies $e_{\rm v}(w) + e_{\rm s}(w) = 0$, then because of non-negativeness of $e_{\rm v}$ and $e_{\rm s}$ the following holds.

$$Min(e_{\rm v}) \cap Min(e_{\rm s}) = Min(e_{\rm v} + e_{\rm s}) \tag{78}$$

Actually if the number of parameters (that are weights and biases) are enough, then with high probability we can guarantee a point w in the weight space such that $e_{\rm v}(w) + e_{\rm s}(w) = 0$. Usually the following is thought to hold.

$$\begin{aligned} Min(e_{\rm v}) \supset Min(e_{\rm v}) \cap Min(e_{\rm s}) &\simeq Min(e_{\rm v} + e_{\rm s}) \\ Nbd(Min(e_{\rm v} + e_{\rm s})) &\supset Min(ge) \end{aligned} \tag{79}$$

Here $Nbd(Min(e_{\rm v} + e_{\rm s}))$ is a neighborhood of the set $Min(e_{\rm v} + e_{\rm s})$.

If one training rule converges to an arbitrary point in $Min(e_{\rm v})$, and a second training rule converges to an arbitrary point in $Min(e_{\rm v} + e_{\rm s})$, then the second will have a better chance of having found a point near $Min(ge)$. In other words, minimizing $e_{\rm v} + e_{\rm s}$ leads to a higher probability of correct generalization.

Of course, exactly the same line of arguments holds if we take a value error $e'_{\rm v}$ for another set of points $P' = \{p'_1, p'_2, ..., p'_{r'}\} \subset U$ instead of $e_{\rm s}$. We can say that minimizing $e_{\rm v} + e'_{\rm v}$ leads to a higher probability of correct generalization than minimizing $e_{\rm v}$ alone.

Also if we take these new points near the original points, we might even be able to mimic the effect of slope information. For example, if we add a value pattern, $\{\{x_1 + \Delta x_1, x_2, ..., x_I\}, f(x_1 + \Delta x_1, x_2, ..., x_I)\}$ along with the original value pattern $\{\{x_1, x_2, ..., x_I\}, f(x_1, x_2, ..., x_I)\}$ to the pattern set, the pattern set has implicit information of $\partial f / \partial x(x_1, x_2, ..., x_I)$ in it.

As the time complexity of tangent prop is about $\frac{11}{4}$ times with one direction (or $2I + \frac{7}{2}$ times with the EBNN slope error) more than that of standard backpropagation, it is in a sense fair to compare tangent prop with standard backpropagation with more training patterns. We call the method the *extra pattern scheme*, in which we apply the standard backpropagation with nearby patterns of the original patterns added to the pattern set. In the following sections we see the qualitative difference between tangent prop and extra pattern scheme.

In the extra pattern scheme, we need additional information about nearby patterns. To get the information might be costly even if they are near the original patterns in the pattern set. We can devise another scheme in which we use the standard backpropagation and the same information as tangent prop. We use the value and slope information of the original pattern to extrapolate the values of nearby patterns and add them to the value pattern set. We call this method the *synthesized pattern scheme*. The same line of arguments in the following sections apply for comparison between tangent prop and synthesized pattern scheme as one between tangent prop and extra point scheme and we limit our comparison to the latter.

3.3 A Simple Case

We see that tangent prop and extra pattern scheme (which is explained in the end of the previous section) are qualitatively different in the following simple example.

Let us take the example of approximating the function $f(x) = \sigma(x, 1, 0) = 1/(1 + e^{-x})$ by $\sigma(x, w, b) = 1/(1 + e^{-(wx+b)})$. We can think of it as a function approximation by a single threshold unit with a weight w and a bias b. Whatever the reasonable choice of ge may be, $Min(ge) = (w, b) = (1, 0)$ because two functions $f(x)$ and $\sigma(x, w, b)$ are identical if and only if $(w, b) = (1, 0)$. We take $x = 0.5$ as a training example. (Taking another point is essentially the same.) In this case, $e_{\rm v}(w, b)$ has a surface like Figure 3. The curve (actually the line) on the surface is the projection of $Min(e_{\rm v}) = \{(w, b)|0.5(w - 1) - b = 0\}$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. In Figure 4 we plot the surface of $e_{\rm s}(w, b)$. The

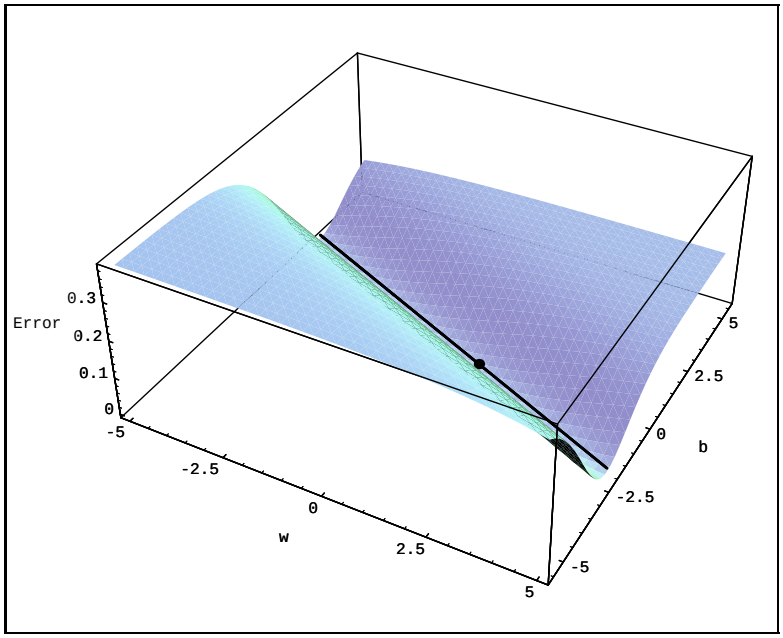

Figure 3: **Value error:** $e_{\rm v}(w, b)$ is plotted. The curve (actually the line) on the surface is the projection of $Min(e_{\rm v}) = \{(w, b)|0.5(w - 1) - b = 0\}$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$.

curve on the surface is the projection of $Min(e_{\rm s})$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. With either $e_{\rm v}$ or $e_{\rm s}$ only, there is no guarantee where on the corresponding curve we will end up. But if we add $e_{\rm v}$ and $e_{\rm s}$ together, then the surface looks like Figure 5. The curves on the surface is the projection of $Min(e_{\rm v}) \cup Min(e_{\rm s})$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. In this case, Equation (79) holds, actually $Min(e_{\rm v} + e_{\rm s}) = Min(ge) = \{(1, 0)\}$. In order to approximate the function correctly using value error only, we need two different points as examples. Using $e_{\rm v} + e_{\rm s}$ we could approximate the function correctly even from one point.

It is true that we are extracting more information from one point when we are using $e_{\rm v} + e_{\rm s}$ instead of $e_{\rm v}$. Let us see what happens if we take $e'_{\rm v}$ with respect to a point in the neighborhood of the point $x = 0.5$ instead of $e_{\rm s}$ and let us see what is qualitative difference between these two. The surface of $e_{\rm v} + e'_{\rm v}$ with $e_{\rm v}$ for $x = 0.5$ and $e'_{\rm v}$ for $x = 0.6$ looks like Figure 6. Two lines on the surface is the projection of $Min(e_{\rm v})$ and $Min(e'_{\rm v})$ on the surface. Two lines are almost identical and the error surface is almost identical to Figure 3 except that the scale on Error axis has doubled. The surface of Figure 6 is not much different from the surface of Figure 3 qualitatively.

On the other hand in Figure 5, two curves are much more perpendicular even though two curves are constructed from information on *one* point. And the surface of Figure 5 is quite different from Figure 3.

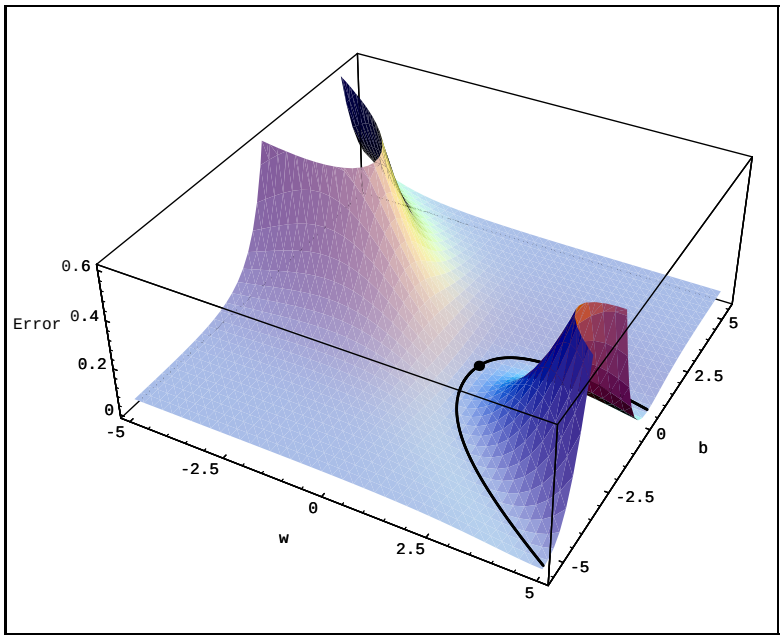

Figure 4: **Slope error:** $e_{\mathrm{v}}(w,b)$ is plotted. The curve on the surface is the projection of $Min(e_{\mathrm{s}})$ onto the surface. The point on the surface is the projection of $(w,b)=(1,0)$.

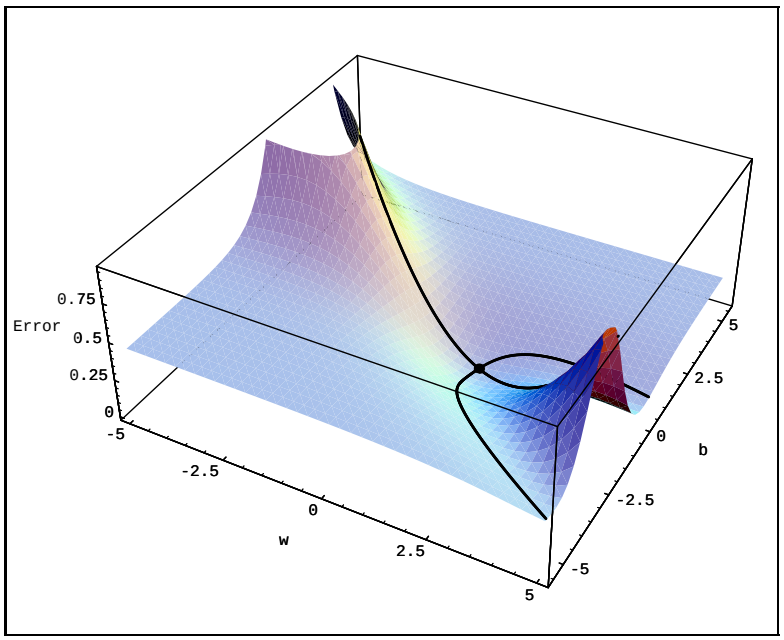

Figure 5: **Value error and slope error combined:** $e_{\mathrm{v}}+e_{\mathrm{s}}$ is plotted. The curves on the surface is the projection of $Min(e_{\mathrm{v}})\cup Min(e_{\mathrm{s}})$ onto the surface. The point on the surface is the projection of $(w,b)=(1,0)$. Note that two curves are much more perpendicular than the curves in Figure 6

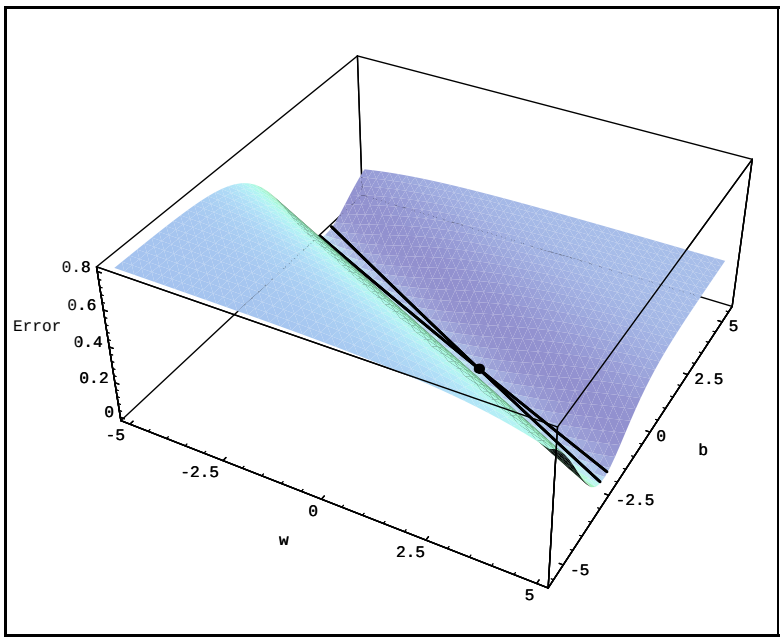

Figure 6: **Value errors on two points:** $e_{\mathrm{v}} + e'_{\mathrm{v}}$ is plotted. Two lines on the surface is the projection of $Min(e_{\mathrm{v}})$ and $Min(e'_{\mathrm{v}})$ on the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. Note that these two lines are almost identical.

From this observation, we can expect that adding new points near the original points of training examples does not contribute much to finding minima and that tangent prop is much more noise-tolerant than extra pattern scheme. (Experimetal results about noise tolerance of tangent prop are given in Section 4.4.)

3.4 A General Case

In this section, we extend the analysis in the previous section to a more general case with reasonable conditions. (See Figure 7.) We show even in a general case that the additional value error function of a nearby point of the original point has less information about the target function than the slope error function of the original point.

Let f_t be the target function and f_n be the function computed by the neural network. We consider the case of a single point $p_0 \in U$ (U is the input space) and a single output unit. Therefore we are considering the case of a single pattern $\{p_0, f_t(p_0)\}$. We notate e^0_{v} for the value error and e^0_{s} for the slope error with respect to p_0. We also consider to move p_0 a little. Let us take a differentiable curve $\{p(t) = (p^1(t), ..., p^I(t)) | t \in [0, 1]\}$ in the input space such that $p(0) = p_0$. We notate $\overline{e_{\mathrm{v}}}(t)$ for the value error for the single point $p(t)$. We fix the p_0 and $\{p(t) | t \in [0, 1]\}$ and think all the functions are defined on the weight space W.

Here are three assumptions we make.

1. Let $\min(e^0_{\mathrm{v}}) = \min(e^0_{\mathrm{s}}) = 0$ and $\min(\overline{e_{\mathrm{v}}}(t)) = 0$ for $t \in [0, 1]$.
2. $Min(e^0_{\mathrm{v}})$, $Min(e^0_{\mathrm{s}})$, and $\{Min(\overline{e_{\mathrm{v}}}(t))\}_{0 \leq t \leq 1}$ have at least one point in common. That is :
$$Min(e^0_{\mathrm{v}}) \cap Min(e^0_{\mathrm{s}}) \cap_{0 \leq t \leq 1} Min(\overline{e_{\mathrm{v}}}(t)) \ni \exists w_0 \tag{80}$$
3. $(\nabla_w f_n)(w_0, p_0)$ and $\left(\nabla_w \frac{\partial f_n}{\partial x_i}\right)(w_0, p_0)$'s are non-zero vectors for the above weight w_0.

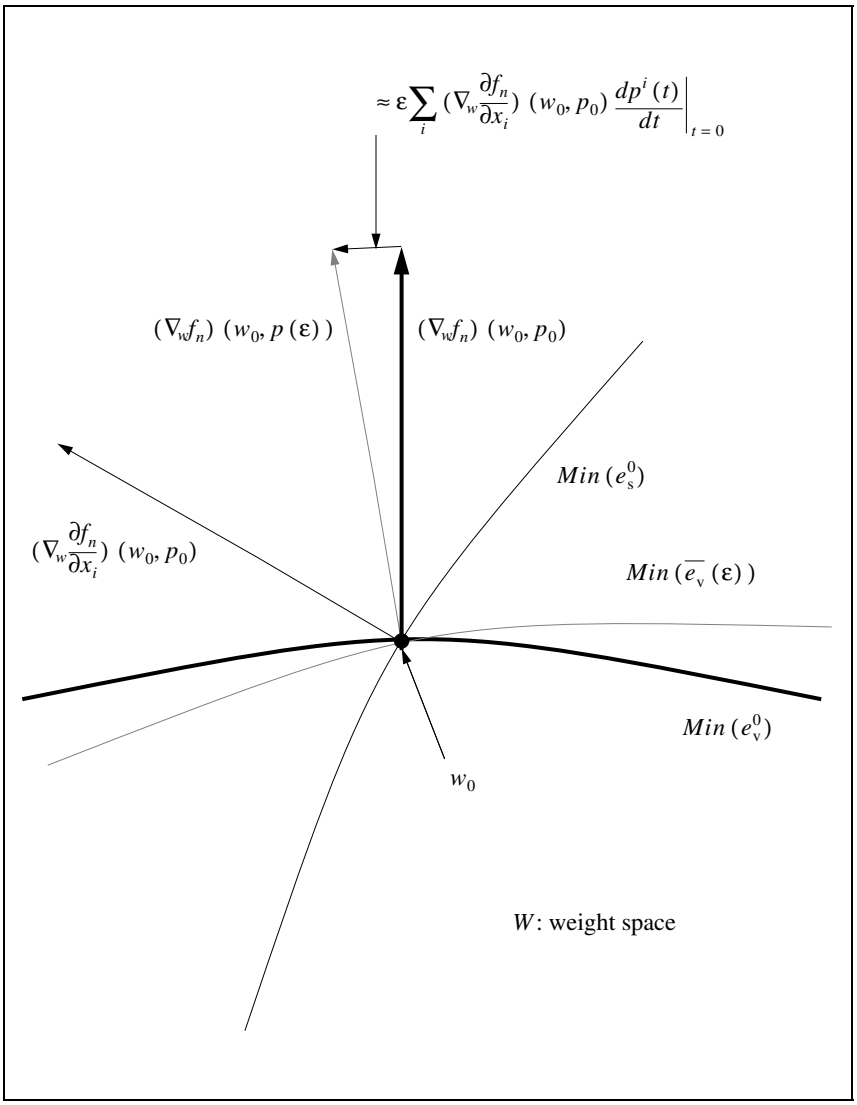

Figure 7: **General case:** The figure shows the normal vectors (shown as arrows) to the surfaces $Min(e_{\rm v}^0)$, $Min(e_{\rm s}^0)$, and $Min(\overline{e_{\rm v}}(\epsilon))$. The difference between the normal vectors to two surfaces $Min(e_{\rm v}^0)$ and $Min(\overline{e_{\rm v}}(\epsilon))$ is a linear combination of the normals to $Min(e_{\rm s}^0)$.

From the first assumption, we can derive that $f_n(w)(p_0) \equiv f_t(p_0)$ on $Min(e_{\rm v}^0)$. From this and the third assumption, $Min(e_{\rm v}^0)$ is essentially a hyperplane in the neighborhood of w_0 and the vector $(\nabla_w f_n)(w_0, p_0)$ is a normal to $Min(e_{\rm v}^0)$ at w_0. Similarly we can conclude that $\{Min(\overline{e_{\rm v}}(t))\}_{0 \le t \le 1}$ and $Min(e_{\rm s}^0)$ are essentially hyperplanes in the neighborhood of w_0, that $(\nabla_w f_n)(w_0, p(t))$ is a normal to $Min(\overline{e_{\rm v}}(t))$ for $t \in [0,1]$, and that $\left(\nabla_w \frac{\partial f_n}{\partial x_i}\right)(w_0, p_0)$'s are normals to $Min(e_{\rm s}^0)$ at w_0.

We see how $Min(\overline{e_{\rm v}}(t))$ changes in the neighborhood of w_0. In order to do so, we see the change of the normal to $Min(\overline{e_{\rm v}}(t))$ at $t = 0$.

$$\left.\frac{\partial}{\partial t}(\nabla_w f_n)(w_0, p(t))\right|_{t=0} = \sum_i \left(\nabla_w \frac{\partial f_n}{\partial x_i}\right)(w_0, p_0) \left.\frac{dp^i(t)}{dt}\right|_{t=0} \tag{81}$$

So the rate of the change to the normal to $Min(\overline{e_{\rm v}}(t))$ at $t = 0$ is a linear combination of normals to $Min(e_{\rm s}^0)$ at w_0. That means that $Min(\overline{e_{\rm v}}(\epsilon)))$ is less distinguishable from $Min(e_{\rm v}^0)$ than $Min(e_{\rm s}^0)$ in the neighborhood of w_0.

To use the sum of $e_{\rm v}^0$ and the value errors $e_{\rm v}^i$'s with respect to points $\{p_i = p_0 + (0, ..., \epsilon_i, ..., 0) | i = 1, .., I\}$ has similar effect to use $e_{\rm v} + e_{\rm s}$. But the differences in normals to these value error surfaces for these points are very slight and these differences are essentially linear combinations of normals to $Min(e_{\rm s}^0)$ at w_0.

Therefore we can reconfirm the expectation for the simple case in the previous section. The differences in normals being slight, the differences in the structures of the surfaces of $e_{\rm v}^0$ and $e_{\rm v}^i$'s are slight and a little bit of noise makes these surfaces indistinguishable.

We can put this the other way around. Taking the surface normal to all of $\left(\nabla_w \frac{\partial f_n}{\partial x_i}\right)(w_0, p_0)$'s at w_0 (i.e. taking $Min(e_{\rm s}^0)$) is more robust to noise and efficient. Using $e_{\rm s}^0$ has similar but more efficient and robust effect on the structure of the minimum set of the error function than the effect by using the value errors of points near p_0.

4 Experiments

Experiments in this section are intended to explore following questions:

- In practice, does slope-constrained training improve generalization?
- How robust is slope-constrained training to errors in training values and in training slopes?
- How does the generalization ability of slope-constrained training scale with the number of inputs to the target function?

In order to see these effects of slope information to learning procedure, we use EBNN tangent prop (that is tangent prop with the EBNN slope error) as slope-constrained training method for all the experiments.

In the following experiments, we tried very hard to make sure that the set of the conditions for experiments is reasonable one. Instead of running all imaginable experiments, which is impossible, we tried to make reasonable choices of these factors and experimented. In order to do so, we run a number of preliminary experiments prior to those described in this paper. We also run several experiments by changing several key parameters after those in this paper to see that the tendency described in this paper is not by luck.

Even so, we have to admit these experiments included in this section are not at all complete. The results of these experiments depend on many factors including (but not limited to):

- Parameters for learning procedures [6]

[6] Actually the efficiency of algorithm also depends on the search algorithm of these parameters. (ex. trial and error, heuristics, genetic algorithms, a method using second derivatives of energy function...)

- Number of epochs [7]
- Number of patterns [8]
- Functions to be approximated
- Initial weights and biases
- Sequence of patterns (or probabilistic distribution of patterns)
- Structure of noise (or probabilistic distribution of noise)

Although it is virtually impossible to explore all these factors. we have attempted to control for reach of these factors to the degree possible given our computational resources.

The common conditions for experiments in Section 4.2, 4.3, and 4.4 are the followings.

- We drew random numbers from the uniform probabilistic distribution on $[-0.01, 0.01]$ to set initial values of weights and biases. The same initial values were used for all the experiments.
- All the patterns are drawn randomly from the uniform probabilistic distribution on $[0, 1] \times [0, 1]$.
- Each trial is run for 200,000 epochs. Every 200 epochs, the generalization error on the 26×26 equidistant lattice points on $[0, 1] \times [0, 1]$ is calculated. [9] The best generalization error during 200,000 epochs for each trial is recorded.
- The "bp to ebnn factor", b is used to divide the learning rate a between the learning rate a_{v} for the value error and the learning rate $a_{\text{ebnn-s}}$ for the EBNN slope error for EBNN tangent prop.

$$a_{\mathrm{v}} = \frac{b}{b+1} \times a \tag{82}$$

$$a_{\text{ebnn-s}} = \frac{1}{b+1} \times a \tag{83}$$

  In the experiments in this paper, b is taken to be 8.0.
- The stepsizes, η's, in these experiments are used with respect to normalized error functions. (See Equations 15, 16, 17) So when we say the learning rate for the value error is to be a_{v} and the learning rate for the EBNN slope error is to be $a_{\text{ebnn-s}}$, then the actual stepsizes are as follows (K is the number of output units and I is the number of input units.):

$$\eta_{\mathrm{v}} = a_{\mathrm{v}} \times \frac{1}{K} \tag{84}$$

$$\eta_{\text{ebnn-s}} = a_{\text{ebnn-s}} \times \frac{1}{I \times K} \tag{85}$$

- The learning rate is taken to be 2.0. The momentum is taken to be 0.9.
- Ten trials are run for the same condition by giving a different set of patterns through changing the seed for the random number generator. [10] These ten different seeds for patterns are kept same through the experiments.

[7] To see asymptotic behavior, one has to carry on the experiment to the infinite number epochs. On the other hand, when there is time constraints, one should consider to optimize the result in in the given time constraints.

[8] Sometimes, this number of patters is the parameter to be experimented on.

[9] For the network with I input units more than 2, instead of calculating errors on all $(26)^I$ lattice points, which is impractical, we decided to use $(26)^2$ points drawn from the uniform probabilistic distribution over $[0,1]^I$. We calculated errors on these points, averaged them, and used the average as the generalization error.

[10] The seed for the random number generator decides the sequence of random numbers generated by the generator.

- The performance for a particular learning method under certain conditions is measured by the average of the best generalization error of the ten trials.

In order to clarify, the relationship of stepsizes with the above parameters, we give the following example. In the case of the "bp to ebnn factor" $b = 8.0$, the learning rate $a = 2.0$, the number of the input units $I = 2$, and the number of the output units $K = 1$, for standard backpropagation we have

$$\eta_{\mathrm{v}} = 2.0 \times \frac{1}{1} = 2.0 \tag{86}$$

For EBNN tangent prop we have

$$\eta_{\mathrm{v}} = \frac{8.0}{8.0+1} \times 2.0 \times \frac{1}{1} = \frac{16}{9} \tag{87}$$

$$\eta_{\text{ebnn-s}} = \frac{1.0}{8.0+1} \times 2.0 \times \frac{1}{2 \times 1} = \frac{1}{9} \tag{88}$$

4.1 How It Works : A Sample Case

In this section, we pick up one function approximation task and try standard backpropagation and EBNN tangent prop to see how things work in detail.

We used the following nonlinear function for a learning problem.

$$g(x,y) = 0.5e^{-10((x-0.2)^2+(y-0.5)^2)} + 0.25\sin(8xy) + 0.25 \tag{89}$$

The task is to approximate this function. The network configuration is two input units (for x and y), six hidden units, and one output unit (for $g(x,y)$). The network is fully connected between layers and each unit except input units has a bias.

One pattern for standard backpropagation consists of input value, x and y, and desired value output, $g(x,y)$, for the output unit. One pattern for EBNN tangent prop consists of input value, x and y, desired value output, $g(x,y)$, for the output unit of the value net and desired output Jacobian, $\frac{\partial g}{\partial x}(x,y)$ and $\frac{\partial g}{\partial y}(x,y)$ for Jacobian inputs $\vec{e_1}$, $\vec{e_2}$ respectively.

In this particular experiment, we used the learning rate 0.5 and made the network learn for 10,000 epochs. We use 10 patterns. The input value of first pattern is taken to be $(0.5, 0.5)$. The input values of the rest of 9 patterns were taken from the uniform distribution on $[0,1] \times [0,1]$. Figure 8 is the surface plot of the target function. The points on the surface are the projections of input values onto the surface. The lines coming out from the points are the normals to the surface. [11]

The graph of the normalized value error $\overline{E}_{\mathrm{v}}$ (see Equation 15) for ordinary back propagation is given by Figure 9. The surface learned by learning values only is given by Figure 10. The dotted lines are the normals to the surface of target function and the solid lines are the normals to this surface learned by backpropagation. Note that these normals are quite different each other.

The graphs of $\frac{8}{9}$ of the normalized value error $\overline{E}_{\mathrm{v}}$, $\frac{1}{9}$ of the normalized EBNN slope error $\overline{E}_{\text{ebnn-s}}$ (see Equation 17), and the sum $\frac{8}{9}\overline{E}_{\mathrm{v}} + \frac{1}{9}\overline{E}_{\text{ebnn-s}}$ of them for tangent prop with the EBNN slope error are given by Figure 11. The surface learned by EBNN tangent prop given by Figure 12. The dotted lines are the normals to the surface of target function and the solid lines are the normals to this surface learned by EBNN learning. Note that in this case these two normals for each point are very close except for one point. Because they are so close, some of the dotted lines are hidden by solid lines.

We estimated the quality of each learned function after 10,000 epochs by the average of squares of value errors from the target function on 200×200 equidistant lattice points. We also call this the *generalization error*. [12] It was 0.0102 for the function learned by backpropagation and it was 0.0063 for the function

[11] These normals are normalized to be the same length in order to make the figure easy to understand. In the following two figures of surfaces, all the normals are also normalized to be the same length.

[12] The only difference from the definition of *generalization error* in Section 4 is the number of the lattice points.

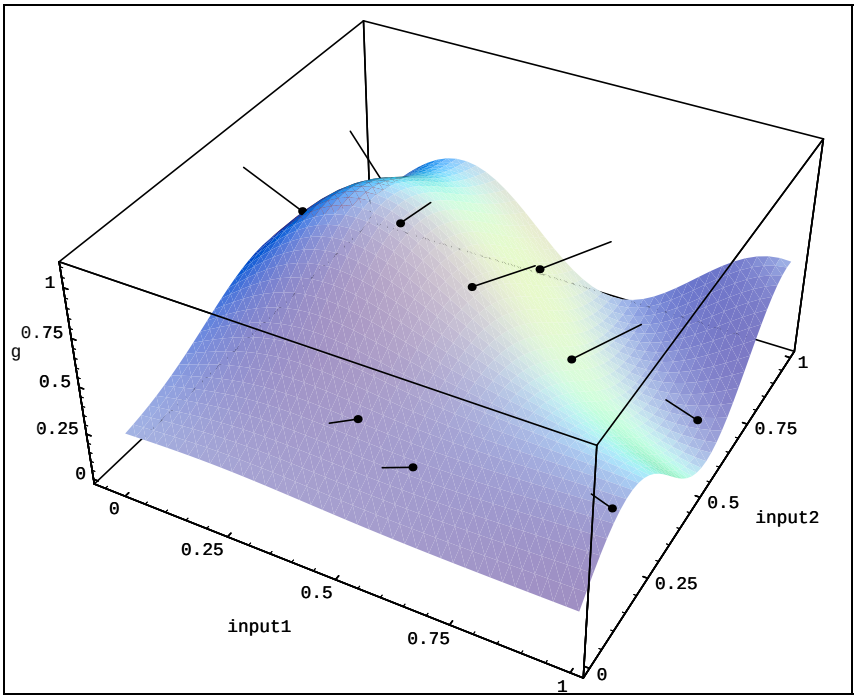

Figure 8: **Surface of the target function:** The surface of the target function is shown. The points on the surface are the projections of input values onto the surface. The lines coming out from the points are the unit normals to the surface.

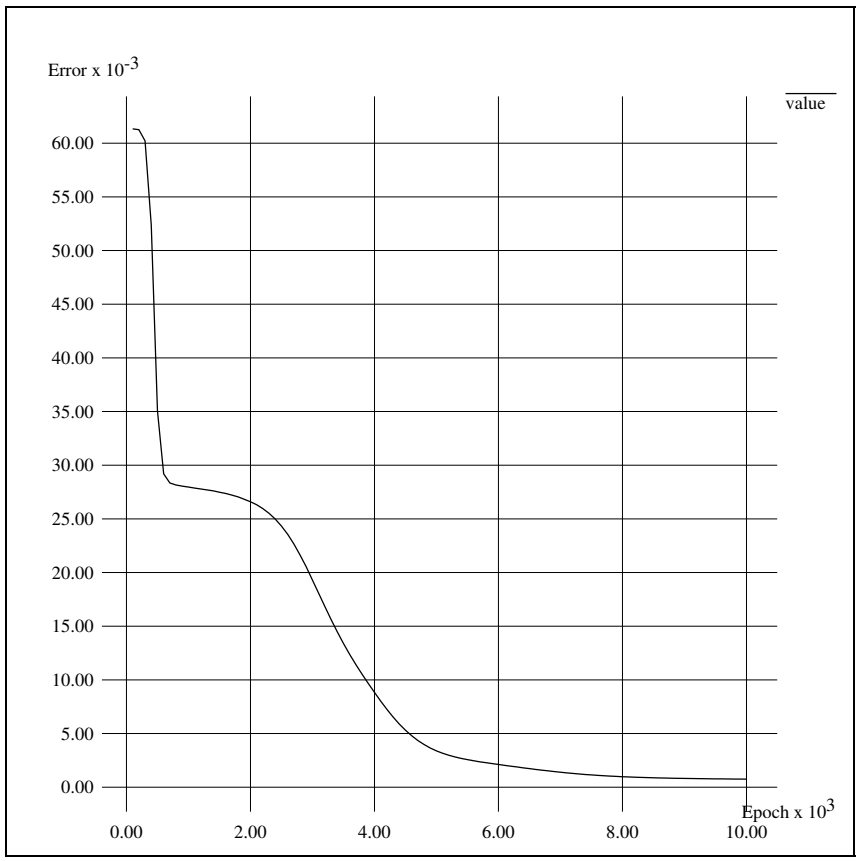

Figure 9: **Error curve of backpropagation:** The change of the normalized value error, $\overline{E}_{\mathrm{v}}$, is plotted against the number of epochs.

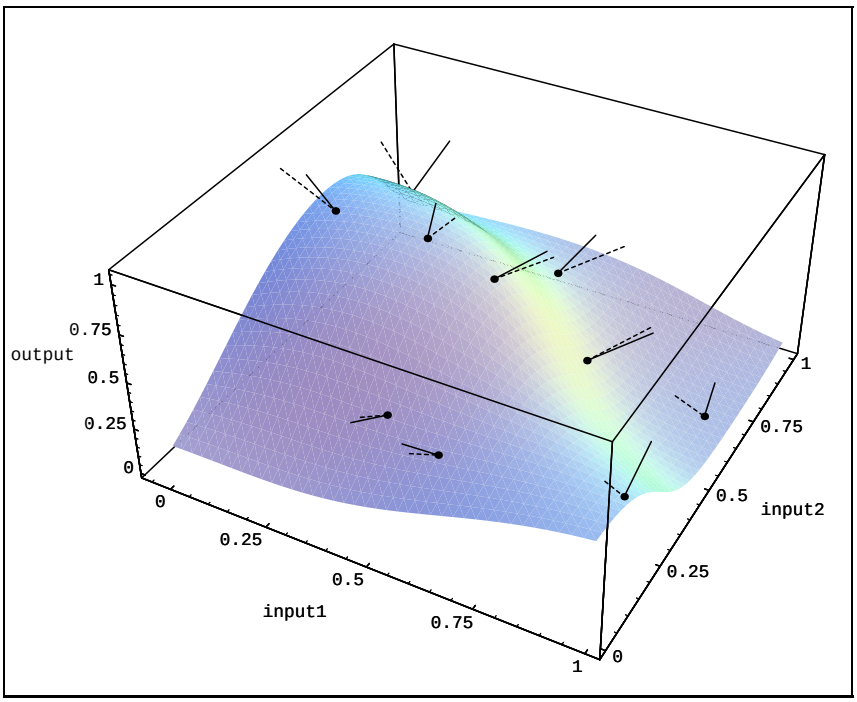

Figure 10: **Surface learned by backpropagation:** The surface learned by backpropagation is shown. The dotted lines are the normals to the surface of the target function and the solid lines are the normals to this surface learned by backpropagation. Note that these normals are quite different each other.

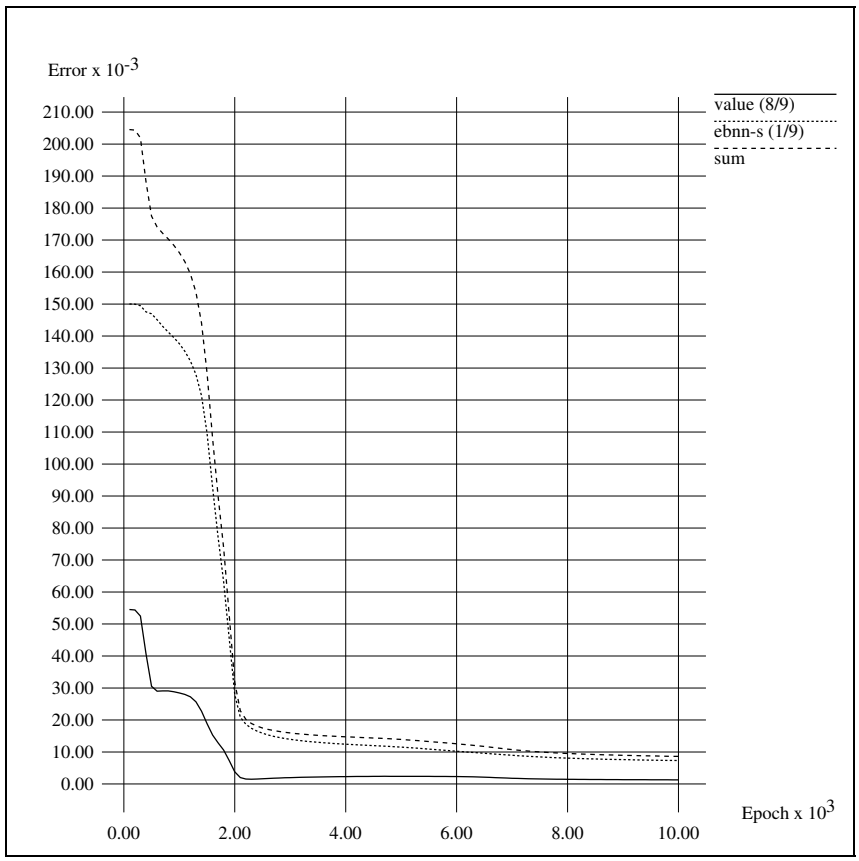

Figure 11: **Error curves of EBNN tangent prop:** The $\frac{8}{9}$ of the normalized value error, $\overline{E}_{\mathrm{v}}$, $\frac{1}{9}$ of the normalized EBNN slope error, $\overline{E}_{\text{ebnn-s}}$, and the sum (total) of them are plotted against the number of epochs for tangent prop with the EBNN slope error.

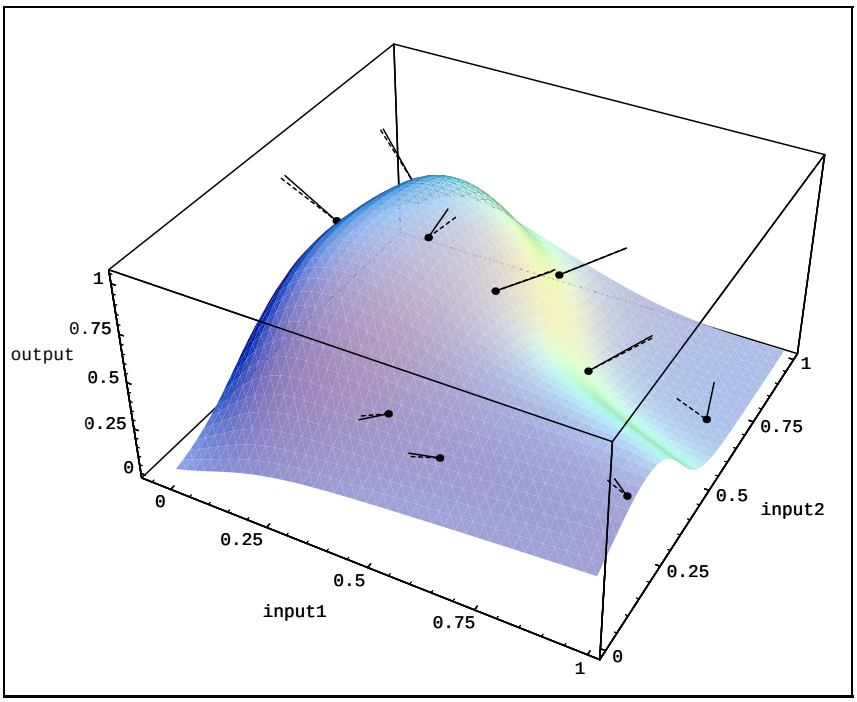

Figure 12: **Surface learned by EBNN tangent prop:** The surface learned by EBNN tangent prop is shown. The dotted lines are the normals to the surface of target function and the solid lines are the normals to this surface learned by EBNN learning. Note that in this case these two normals for each point are very close except for one point. Because they are so close, some of the dotted lines are hidden by solid lines.

learned by EBNN tangent prop. [13] The generalization errors for learning procedures by backpropagation and EBNN tangent prop are plotted against the number of epochs in Figure 13. Generalization error for EBNN tangent prop is faster to drop and the final value of it is smaller.

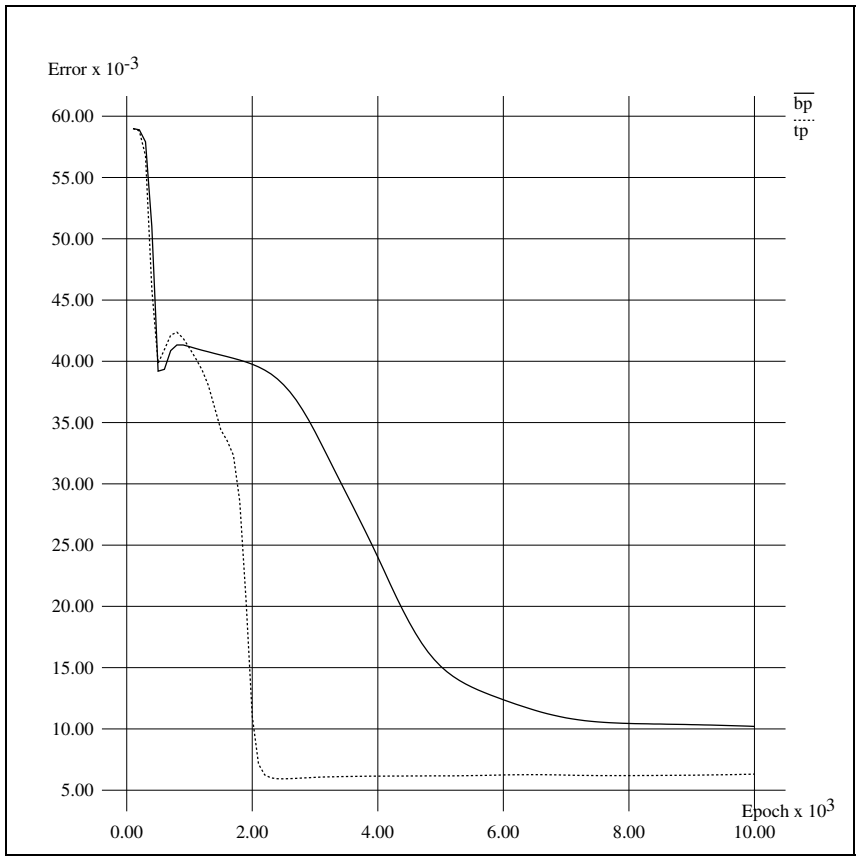

Figure 13: **Graphs of generalization errors:** The generalization errors for learning procedures by backpropagation and EBNN tangent prop are plotted against the number of epochs.

4.2 Sample Complexity

In this section, we will explain the results of experiments to see speedup effect of tangent prop (with the EBNN slope error) over backpropagation learning in terms of the number of required training patterns.

We use the g function of Equation 89 in Section 4.1. We use 2 inputs for x and y. The other general settings for the experiments are described in the beginning of Section 4.

We run the experiments for 1,2,3,4,5,10,20,30,40, and 50 patterns. We plotted the performances of backpropagation and EBNN tangent prop against the number of patterns in Figure 14. Except for small number of patterns and for more than 50 patterns, EBNN tangent prop outperforms backpropagation. By crossing the graphs horizontally, one can see the required number of patterns to achieve certain performance

[13] We have to remind that it is not always the case that EBNN tangent prop outperforms backpropagation. But you will find in the following sections, statistical results supporting that EBNN tangent prop does better than backpropagation on average.

for EBNN tangent prop is 1 to 2 times smaller than that of backpropagation in the region where EBNN tangent prop outperforms backpropagation.

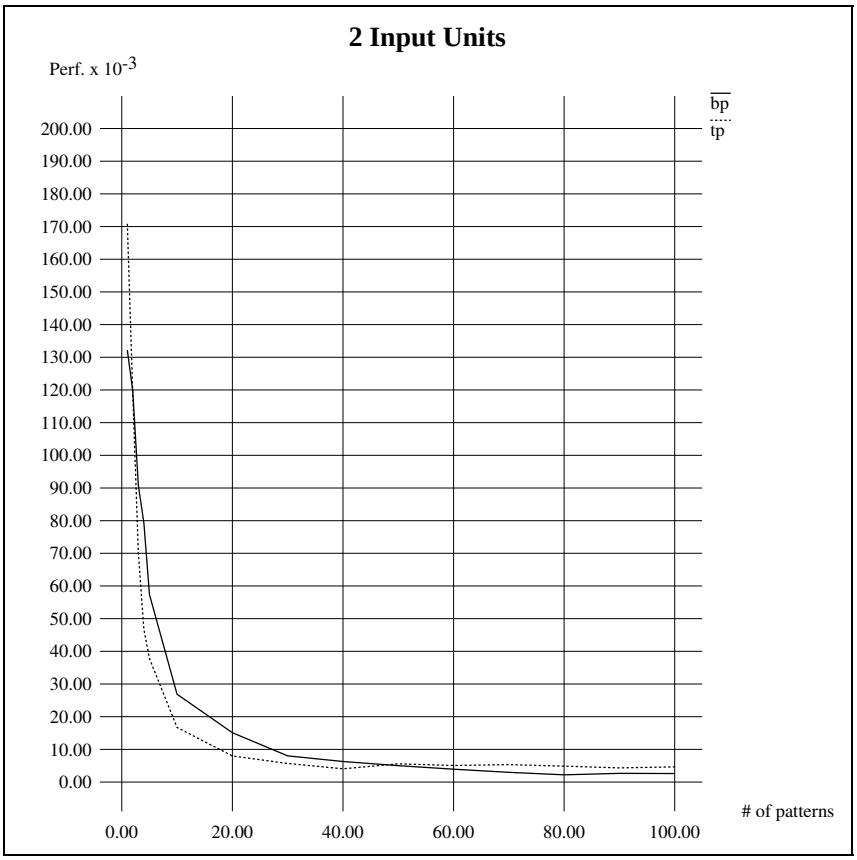

Figure 14: **Graphs of speedup effect:** We plotted the performances of backpropagation and tangent prop with the EBNN slope error against the number of patterns for 1,2,3,4,5,10,20,30,40, and 50 patterns and join them by the lines. Except for small number of patterns and for more than 50 patterns, EBNN tangent prop outperforms backpropagation. By crossing the graphs horizontally, one can see the required number of patterns to achieve certain performance for EBNN tangent prop is 1 to 2 times smaller than that of backpropagation in the region where EBNN tangent prop outperforms backpropagation.

Neural networks used in both methods are same and so are the potentials of approximation. So in each particular trial backpropagation might outperform EBNN tangent prop. [14] But as in Equation 79, there must be significant difference in performance between backpropagation and EBNN tangent prop By averaging over 10 trials, we can see the apparent tendency that EBNN tangent prop outperforms backpropagation as Figure 14 shows it.

For first several patterns and for 50 patterns, EBNN tangent prop is outperformed by backpropagation. We assume the reasons are the following. Backpropagation approximates only values at points while EBNN tangent prop approximates also slopes at points. While there are a few points, with backpropagation the whole surface stays at the level of somewhat around the given values, But in EBNN tangent prop, fitting

[14] Actually in each individual trial, it often happened that backpropropagation outperforms EBNN tangent prop for certain areas.

slopes makes other parts of surface deviate from the target surface. As soon as the points are dense enough, EBNN tangent prop starts to outperform the backpropagation.

But then, the performance of backpropagation catches up that of EBNN tangent prop at 50 patterns. After examining the data of learning procedure, we observed that EBNN tangent prop stops improving on either the value error or the EBNN slope error in the early stage of learning procedures. We believe what is happening here is that there are too many constraints for the number of parameters (i.e. weights and biases) in the neural network. Therefore we believe the reason why EBNN tangent prop is outperformed by backpropagation is because of the above and because we estimate the performance in terms of the generalization error which is essentially the value error.

4.3 Scaling Up and Irrelevant Features

In order to see how irrelevant dimensions affect the learning, we set up the following experiments.

We use the g function of Equation 89 in Section 4.1. We used 2 inputs for x and y as relevant inputs and other 0, 3, or 8 inputs as irrelevant inputs. (So the number of the total input units are 2, 5, and 10 respectively.) Let $x_1, ...$ be irrelevant inputs. We take the following $\overline{g}$ function as the target function.

$$\overline{g}(x, y, x_1, ...) = g(x, y) \tag{90}$$

As input patterns, we draw the values for $\{x, y, x_1, ...\}$ from uniform distribution over $[0, 1]$. As desired value outputs, we gave $g(x, y)$. As desired Jacobian outputs, we gave $(\frac{dg}{dx}, \frac{dg}{dy}, 0, ..., 0)$.

We did essentially the same experiment in Section 4.2 for each number of input units. We collected the performance of each method on 1, 2, 3, 4, 5, 10, 20, 30, 40, and 50 patterns. The performances are plotted against the number of patterns for the networks of 2, 5, and 10 inputs in Figure 15. We can see that the performance of backpropagation degenerates rapidly as the number of irrelevant input units increases, while EBNN tangent prop stays at the same level.

We used above results to produce Figure 16. We are considering the following situation. We get one pattern at a time. Each time we make the networks learn the patterns we have accumulated up to the point and make a prediction. These graphs show the accumulated mistakes (summed errors) done by the networks up to the point. The detailed explanation of what are plotted is the following. In these graphs, we plotted the integrated value of the square root of the average performance up to the number of patterns. Therefore, for the number i the following value is plotted. (Here I.A.P. stands for Integrated Average Performance.)

$$I.A.P. = \sum_{i=1}^{n} \sqrt{\text{the average performance for } i \text{ patterns}} \tag{91}$$

Where the average performance is missing, we interpolated its value linearly from the values known. [15]

These results show that EBNN tangent prop is more robust to the number of irrelevant input units. But, of course, there is no magic in tangent prop. The merit of tangent prop is that it can utilize more information when there are more input units. When the irrelevant dimension of irrelevant inputs gets large, the information tangent prop can utilize increases with the irrelevant dimension while the information backpropagation can utilize stays the same. The point is that tangent prop can use that extra information from these points.

4.4 Noise in Target Values and Target Slopes

In the frameworks like EBNN learning in which target slopes is constructed from incomplete and/or inaccurate world models, noise in target slopes are inevitable. In this section we explain the results of experiments

[15] As the Figure 16 is produced from the results in Figure 15, the graphs in Figure 16 give the same conclusion that EBNN learning is more robust with respect to the number of irrelevant input units.

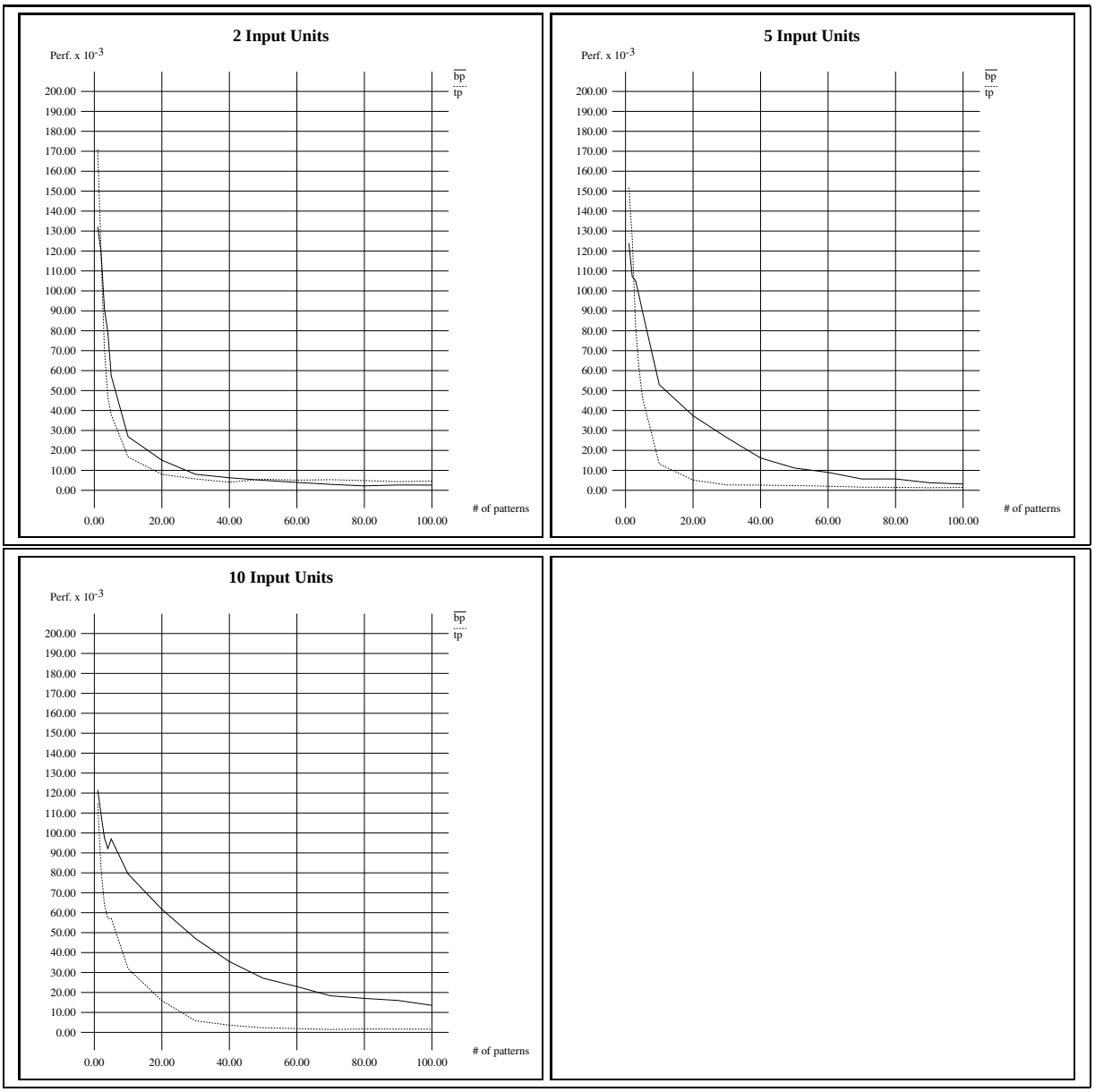

Figure 15: **Graphs of scaling up effect:** The performances are plotted against the number of patterns for the networks of 2, 5, and 10 inputs for 1, 2, 3, 4, 5, 10, 20, 30, 40, and 50 patterns. We can see that the performance of backpropagation degenerates rapidly as the number of irrelevant input units increases, while EBNN tangent prop stays at the same level.

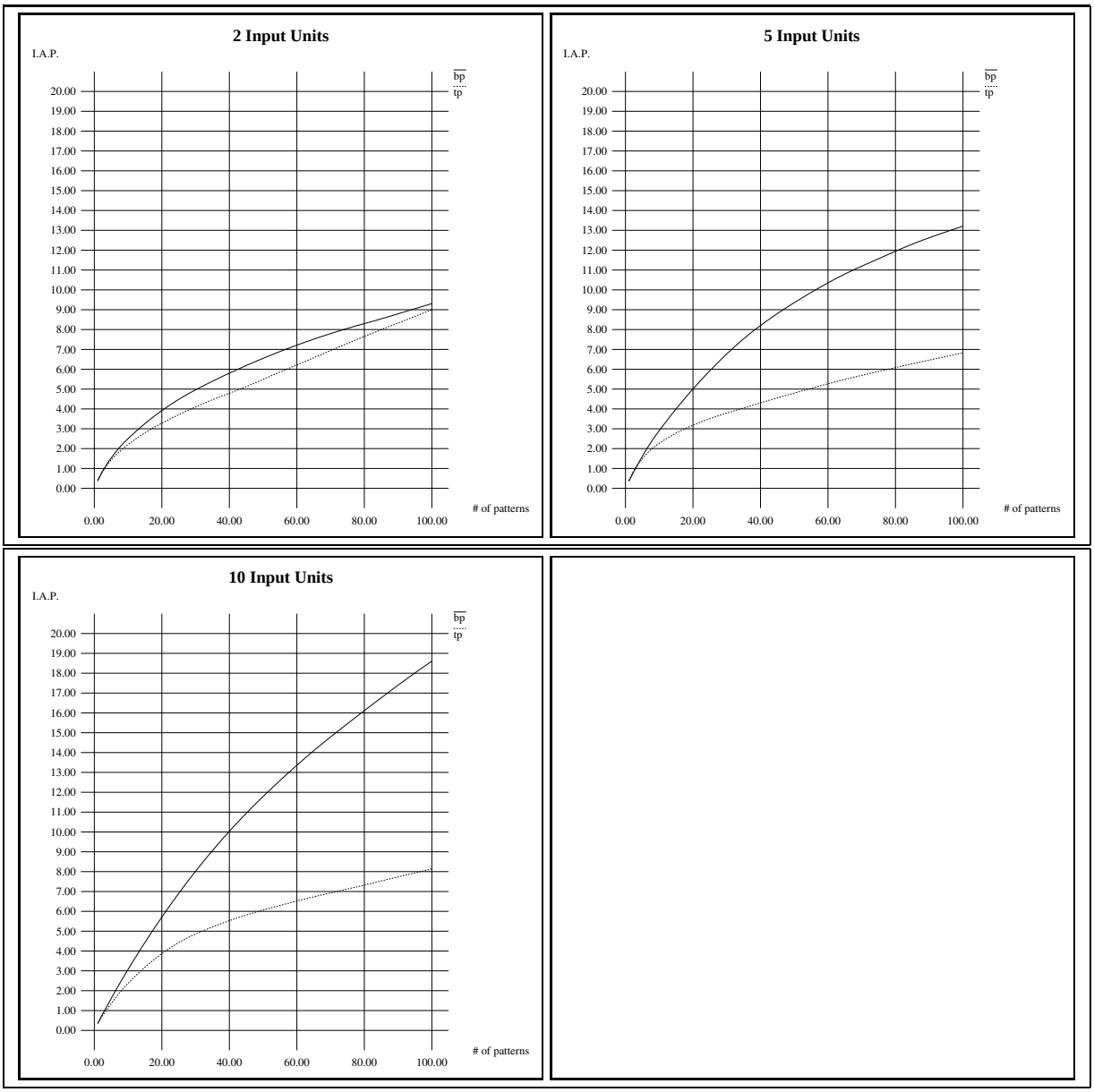

Figure 16: **Graphs of mistake bound:** In these graphs, we plotted I.A.P., the integrated value of the square root of the average performance up to the number of patterns. These graphs shows that EBNN tangent prop is more robust with respect to the number of irrelevant input units.

to see how noise in target values and noise in target slopes affect backpropagation and EBNN tangent prop. The experimental results are cited from [Masuoka, 1993].

Again we use the g function of Equation 89 in Section 4.1 as the target function and used fixed 20 patterns for all the experiments. By numerical integration, the averages of the absolute values of derivatives are given by :

$$\int_0^1 \int_0^1 \left| \frac{dg}{dx} \right| dxdy \simeq 0.893483 \tag{92}$$

$$\int_0^1 \int_0^1 \left| \frac{dg}{dy} \right| dxdy \simeq 1.01263 \tag{93}$$

They are both around 1.0. So we decided to give noises of the same multitude to derivatives by both x and y. When we say the value (or respectively slope) noise level is 0.1, the random value is drawn from the normal probabilistic distribution with its variation 0.1 and with its mean 0 and it is added to each desired value (or respectively Jacobian) output. [16] When the resulting desired output value is less than 0.0 or greater than 1.0, it is reset to 0.0 or 1.0 accordingly.

First we run standard backpropagation, plotting the performances as the value noise level is varied from 0.0 to 1.0 with the step 0.05 with the 20 patterns. The result is Figure 17.

We then run EBNN tangent prop, plotting performance as the slope noise level is varied from 0.0 to 2.0 with step 0.1 with the 20 patterns (with zero value noise). The result is Figure 18.

We also run EBNN tangent prop, changing both the value and slope noise levels as in the previous two experiments. The result is Figure 19.

From these experiments, we can conclude the following empirically.

From Figure 17 and the cross section of Figure 19 for which *slope noise level* = 0.0, EBNN tangent prop generalizes better than standard backpropagation even if value noise is present.

By observing the cross sections of Figure 19 for which value noise level is constant, we can conclude that the greater the noise level for desired output values, the less the slope noise level matters. Also from the above cross sections and Figure 17, we can also conclude the slope noise level for which EBNN tangent prop has the same performance as standard backpropagation increases, as the value noise level increases.

In Figure 18, the curve of the performance by EBNN tangent prop shows a graceful degeneration as the slope noise level increases. The crossover point of these two curves in Figure 18 shows when EBNN tangent prop is degenerated enough so as to have the same performance of backpropagation. That happened around where the slope noise level is 1.2. Considering that the average absolute values of derivatives are around 1.0, we conclude (in this case) that EBNN tangent prop is quite robust against noises.

This results support the observations in Sections 3.3 and 3.4 that the minimum set for the EBNN slope error has quite different characteristics from that of the value error and that using the EBNN slope error would be robust.

5 Conclusion

In this paper, we provided a detailed description of the algorithm of tangent prop. We expect the readers find it easy to implement these algorithms now. We also gave estimates for time and space complexity of these algorithms compared to standard backpropagation. We expect the readers can judge when to use these algorithms based on these estimates and their own estimates for other restraints. [17] Then by examining structures of minimal sets of energy functions mathematically, we could establish a qualitative difference of slope information and value information and that tangent prop should be more robust and efficient than using additional value information for backpropagation. Lastly, we reported the results of experiments. We

[16] We also tried the uniform probabilistic distribution like one on $[-0.1, 0.1]$ and we got very similar results described here.

[17] Such as costs for getting the information and the accuracy of the information.

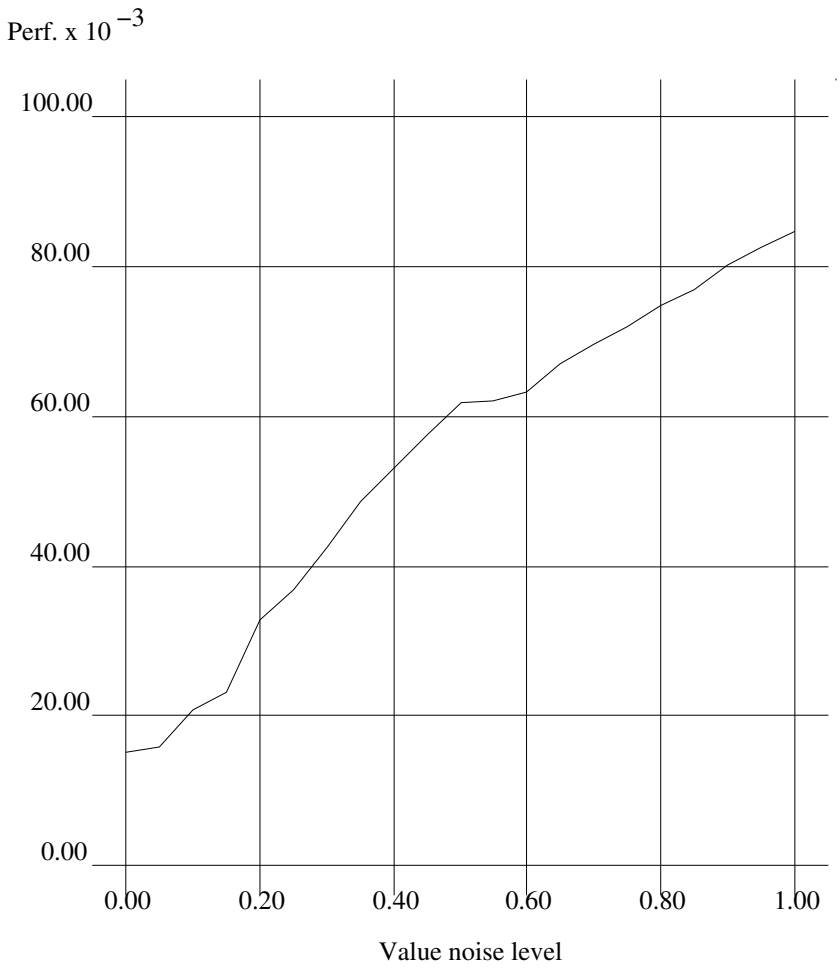

Figure 17: **Performance of backpropagation:** The performance of standard backpropagation against the value noise level is plotted. Noise is added only to desired output values. The value noise level is varied from 0.0 to 1.0 in increments of 0.05.

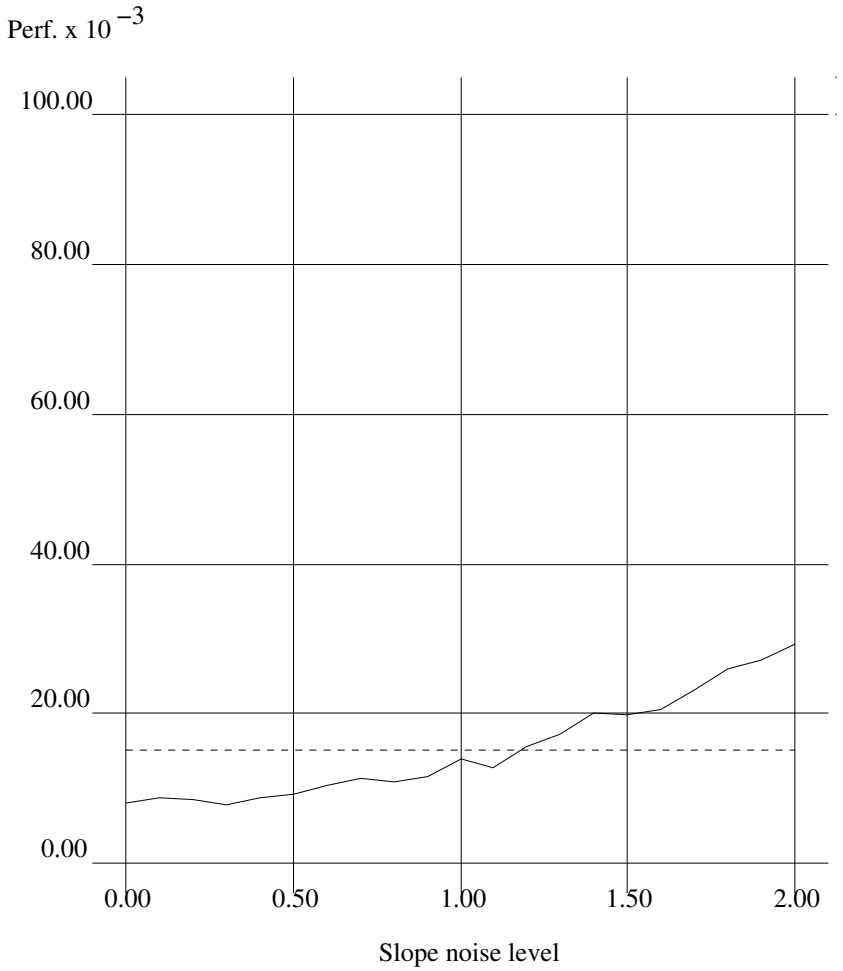

Figure 18: **Performance of tangent prop (1):** The performance of EBNN tangent prop against the slope noise level is plotted as the solid line. Noise is added only to desired output Jacobians. The slope noise level is changed from 0.0 to 2.0 with stepsize 0.1. The dotted line is the performance of standard backpropagation with no noise in the training patterns. (See Figure 17 where the value noise level is 0.0.) Notice that solid line and the dotted line cross around where the noise level is 1.2. That is the crossover point where the desired output Jacobians are degraded enough to match the performance of standard backpropagation.

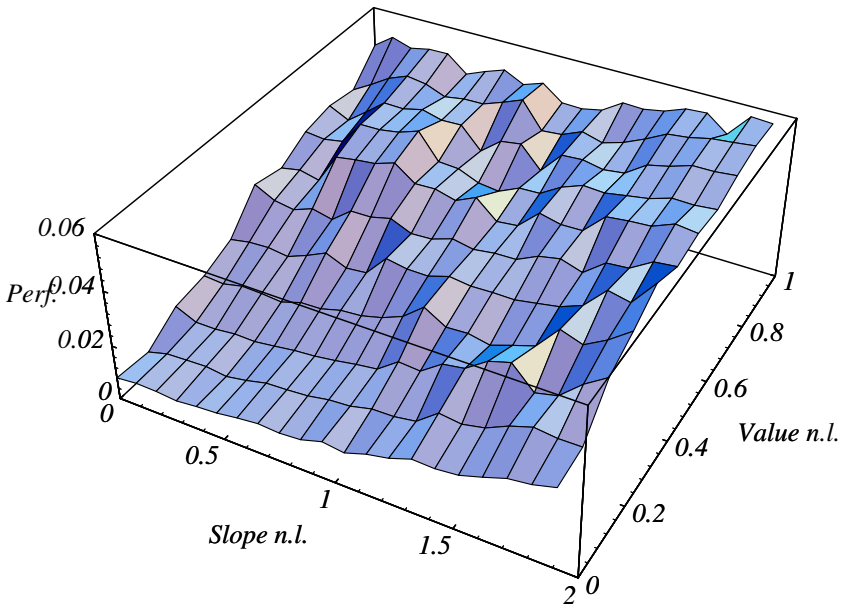

Figure 19: **Performance of tangent prop (2) :** The performance of EBNN tangent prop against both the value and slope noise levels is plotted.

gave an illustrative example to compare EBNN learning and backpropagation. Then we gave a result to show the speedup effect of EBNN tangent prop compared to standard backpropagation. We also gave the results which show the robustness of EBNN tangent prop with respect to the number of irrelevant input units and noise in target values and target slopes.

Acknowledgments

We thank XXX and YYY for proofreading, ZZZ for inspiring discussion and suggestions. This research was sponsored in part by AAA under Contract BBB and by a grant from CCC Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the DDD or of EEE.

References

[Masuoka, 1993] Ryusuke Masuoka. Noise robustness of ebnn learning. In *IJCNN'93-NAGOYA*. Not Known Publisher, 1993.

[Mazja, 1985] V. G. Mazja. *Sobolev Spaces*. Springer Series in Soviet Mathematics. Springer-Verlag, 1985.

[Mitchell and Thrun, 1993] Thomas M. Mitchell and Sebastian Thrun. Explanation-based neural network learning for robot control. In Stephen J. Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 287–294. Morgan Kaufmann, San Mateo, CA, 1993.

[Rumelhart *et al.*, 1986] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing. Vol. I + II*, pages 318–362. MIT Press, 1986.

[Simard *et al.*, 1992] Patrice Simard, Bernard Victorri, Yann LeCun, and John Denker. Tangent prop – a formalism for specifying selected invariances in an adaptive network. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 895–903. Morgan Kaufmann, San Mateo, CA, 1992.