Neural Networks
Learning Differential Data

Ryusuke Masuoka
Department of Mathematical Sciences
The University of Tokyo

March 2000

Abstract

Learning systems that learn from previous experiences and/or provided examples of appropriate behaviors, allow the people to specify *what* the systems should do for each case, not *how* systems should act for each step. That eases system users' burdens to a great extent.

It is essential in efficient and accurate learning for supervised learning systems such as neural networks to be able to utilize knowledge in the forms of such as logical expressions, probability distributions, and constraint on differential data along with provided desirable input and output pairs.

Neural networks, which can learn constraint on differential data, have already been applied to pattern recognition and differential equations. Other applications such as robotics have been suggested as applications of neural networks learning differential data.

In this dissertation, we investigate the extended framework introduce constraints on differential data into neural networks' learning. We also investigate other items that form the foundations for the applications of neural networks learning differential data.

First, new and very general architecture and an algorithm are introduced for multilayer perceptrons to learn differential data The algorithm is applicable to learning differential data of orders not only first but also higher than first and completely localized to each unit in the multilayer perceptrons like the back propagation algorithm.

Then the architecture and the algorithm are implemented as computer programs. This required high programming skills and great amount of care. The main module is programmed in C++.

The implementation is used to conduct experiments among others to show convergence of neural networks with differential data of up to third order.

Along with the architecture and the algorithm, we give analyses of neural networks learning differential data such as comparison with extra pattern scheme, how learnings work, sample complexity, effects of irrelevant features, and noise robustness.

A new application of neural networks learning differential data to continuous action generation in reinforcement learning and its experiments using the implementation are described. The problem is reduced to realization of a random vector generator for a given probability distribution, which corresponds to solving a differential equation of first order.

In addition to the above application to reinforcement learning, two other possible applications of neural networks learning differential data are proposed. Those are differential equations and simulation of human arm. For differential equations, we propose a very general framework, which unifies differential equa-

tions, boundary conditions, and other constraints. For the simulation, we propose a natural neural network implementation of the minimum-torque-change model.

Finally, we present results on higher order extensions to radial basis function (RBF) networks of minimizing solutions with differential error terms, best approximation property of the above solutions, and a proof of C^l denseness of RBF networks.

Through these detailed accounts of architecture, an algorithm, an implementation, analyses, and applications, this dissertation as a whole lays the foundations for applications of neural networks learning differential data as learning systems and will help promote their further applications.

Acknowledgment

I would first like to thank my dissertation advisor, Michio Yamada. I gratefully acknowledge his warm encouragement and patient guidance through my slow process of preparing this dissertation.

I would also like to thank my other thesis committee members. In particular, I am grateful to Takashi Omori for valuable feedbacks.

I would like to thank sincerely Takushirou Ochiai, who kindly invited me to take this doctorate course.

I am deeply indebted to Tom Mitchell and Sebastian Thrun. This thesis has begun from my stay with them at CMU as a visiting scientist. Their research on Explanation Based Neural Network (EBNN) inspired my work.

I have also benefited from interaction with Hiroyuki Okada and Hiroshi Yamakawa. They provided me a very interesting problem in reinforcement learning, which has developed into a chapter in this dissertation.

I would like to express my thanks to members of Intelligent Systems Laboratory of Fujitsu Laboratories Ltd. for discussions, comments, and support. Discussions with Shinya Hosogi and Yoshiharu Maeda helped me consider the possible applications. Special thanks go to Noriko Maeda and Kyoko Tanaka for their support. Without their help on bibliography, I could not have compiled this dissertation.

Special thanks go to many people in Fujitsu Laboratories Ltd. for their support, especially to Shigeru Satoh, Jun-ichi Tanahashi, Hiromu Hayashi, Kazuhiro Matsuo, Kazuo Asakawa, Tomoharu Mohri, and Fumihiro Maruyama. Particularly Kazuo Asakawa encouraged and helped me to start this doctorate course.

Finally I would like to acknowledge my thanks to my family. My parents, Yutaka and Hiromi Masuoka supported me with their encouragement and comprehension. My wife, Takako provided me constant support which helped me to overcome the many difficulties and discouragement on the way to completing this dissertation. My sons, Sei and Sou have been a constant source of joy and gave me the strength I needed to go through the preparation of this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Learning Systems

Learning systems will affect all mankind greatly in the future. Learning systems allow the people to specify *what* the systems should do for each case, not *how* systems should act for each step. That eases system users' burdens to a great extent.

We define a "learning system" as a system that can decide in a reasonable way what to do in a particular situation from previous experiences and/or provided examples of appropriate behaviors even though the situation may not be experienced by the system before.

Systems that work only as ordered and learning systems (even those systems that learn partly) are completely different existences. Methods to give orders to systems without learning capabilities, such as programming languages, scripting languages, graphical user interfaces (GUIs), and aural interfaces, will be substantially more sophisticated in the future. But, however refined those methods may become, systems without learning capabilities burden us with tasks of specifying *how* systems should act for each step. It also requires system users to understand how the system works, which is not essential to achieve objectives.

On the other hand, it is only necessary to indicate *what* the systems should do for each case with learning systems. Then learning systems learn to adjust, configure, and/or program themselves appropriately. This is especially useful when the systems or target domains of the systems are so complex, huge, and/or rapidly changing that human beings are just not able to fathom them. For example, Internet is one of such domains. Learning systems will be indispensable for Internet users in the future.

Learning systems relieve the people from complex and tedious tasks adjusting, configuring, and programming systems. It will also be possible by learning for learning systems to be able to handle gradually more abstract specifications of goals.

Learning systems need learning algorithms. Learning algorithms (and frameworks) include clustering, reinforcement learning, many logic-based systems, neural networks, fuzzy systems, genetic algorithms, etc.

One of the categorizations of those learning algorithms is that of unsupervised/supervised learning. We focus our discussion on supervised learning from

here on.

Supervised learning is learning from provided examples. Examples often take the form of desired input and output pairs in supervised learning. Being presented desired input and output pairs as examples, learning systems learn the relationship (or the function) between input and output from those examples. For example, pairs of sensory data as input and desired motor control signals as output are given to a learning system so that it learns to output the appropriate motor control signals for individual sensory data.

For learning systems, the problem is not about where the input data is present, but where the input data is missing. Usually supervised learning is ill posed problem. If there are no specifications, the learning system will interpolate or extrapolate the output data according to the internal constraints of the learning systems.

There are forms of knowledge that can be provided to the learning systems other than desired input and output pairs. Such knowledge is the constraints of the domain or how the world under consideration is (or seems to be) constrained. Human beings also use these kinds of knowledge to learn or to recognize. Actually these kinds of knowledge are essential for human beings to learn or to recognize since learning and recognition in real world are always ill posed problems. These kinds of knowledge might sometimes be called biases and it happens in some cases that they work against learning and recognition processes. Even so, these kinds of knowledge help efficient processes of learning for learning systems in many cases. Especially for limited domains where exact constraints are known, it should be always right to count on them.

The forms of these kinds of knowledge include the followings:

- Knowledge in the logical forms
  - Rules and facts asserted in logical expressions
  - Fuzzy rules
- Probability distribution
  - Markov Random Field
- Constraints on differential data
  - Differential equations
  - Constraints on values calculated from differential data

If these kinds of knowledge can be combined in supervised learning, learning process will be accelerated and the result will be more accurate in most of the cases.

For example, fuzzy rules are incorporated into the neural networks by prewiring the neural networks to be used for giving better initial states and limiting the search space of the neural networks [21, 27, 28].

Introducing knowledge represented by probability distribution is also very useful. Simulated annealing is one of the popular methods to mix this kind of knowledge as probability distributions in optimization problems of target functions. Simulated annealing is applied to many applications in the area of Computer Aided Design (CAD) [15] and simulated annealing with Markov Random Field (MRF) has been applied to pattern restoration [7] successfully.

Constraints on differential data are another main sources of knowledge, which can be utilized for learning systems. Physical values often satisfy differential equations.

Other forms of constraints on differential data are used for learning systems. Those constraints include that certain directional derivatives have to be zeros for outputs of pattern recognizers [36]. In robotics, some of constraints take forms of minimization of integral of squared differential value [11, 38].

In this dissertation, we investigate the framework to introduce constraints on differential data into neural networks as learning systems.

1.2 Applications

There are many possible applications where constraints on differential data are available along with desired input and output pairs. Neural networks, which learn differential data, are very apt learning system for such applications.

Examples of such applications are given in [36] and [13].

Simard et al. [36] described how the invariance of pattern recognition with respect to transformations such as translations, rotations, and scalings, 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 zero.

Hornik et al. [13] identified several areas of applications requiring approximation to an unknown mapping and its derivatives, such as robot learning, deterministic chaos, economics, and sensitivity analyses.

There are also researches [17, 4, 39, 16] on the application of neural networks to solving differential equations. Neural networks are applied to first and second order differential equations such as a linear Poisson equation, one of thermal conduction with non-linear heat generation, and one in plasma equilibrium problem.

1.3 History of Related Research

In this section, we give a brief history of research on neural networks learning differential data.

In the research history of neural networks, there first appeared existence theorems of a multilayer perceptron that approximates a given function. Funahashi [5] showed that there is a three-layered perceptron approximating any C^0 function with any precision, while Hornik et al. [13] showed there is also a perceptron, which approximates any C^n function with respect to C^n norm. On the other hand, constraining neural networks by using a training set of value data was justified by Gallant and White [6] 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 minima for the least square errors. Instead, if available, we should employ differential data themselves as learning data in constraining neural networks. The constraining algorithm for neural networks learning first order differential data was first proposed under the name of 'tangent prop' by Simard et al. [36] who applied it to a pattern recognition problem.

In this dissertation, we propose an algorithm of multilayer neural network learning differential data of arbitrary order [24, 23]. The algorithm employs the back propagation [34] processes for derivatives of the target function up to the required order together with back propagation 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. Here in this dissertation, we show that the proposed algorithm is possible to implement for differential data of an *arbitrary* order by coding it in the form of C++ program.

Here we also discuss an introduction of differential data into radial basis function (RBF) networks. RBF was first discussed by Poggio [31] as a basis of smooth approximation to a function by solving a variational problem of least square error with a smoothing term. Park and Sandberg [29, 30] proved that RBF networks are dense in $L^p(1 \leq p \leq \infty)$, and Chen and Chen [2] showed that a necessary and sufficient condition for a function of one variable to be qualified as a mother function (activation function) is that the function is not an even polynomial. Further, in 1998, Li [18] proved that RBF networks are dense in C^n space. In this dissertation, we first give an RBF network by solving a variational problem with a differential error terms, and then prove its denseness in C^n in an alternative way to that of Li.

1.4 Objectives and Contents of this Dissertation

In this section, we describe the objectives and contents of this dissertation.

As identified in Section 1.3 there is still a gap between previous research conducted on neural networks learning differential data and practical applications of neural networks learning differential data when neural networks learning differential data are used as learning systems which can incorporate constraints on differential data in learning as described in Section 1.1. The main gap is the algorithm and the implementation of neural networks learning differential data of arbitrary order. Therefore the objectives of this dissertation are set to filling this gap and conducting analyses and experiments on neural networks learning differential data using the implementation in order to show the possibilities of practical applications.

On consideration of the above objectives, we have identified the domain of research as follows:

- Algorithm for neural networks to learn differential data of arbitrary order
- Implementation of the algorithm
- Experiments to show that it really converges
- Analyses on neural networks learning differential data

- Application of the algorithm and the implementation

We have derived an algorithm for neural networks to learn differential data of arbitrary order. This algorithm has features as follows:

- The algorithm can use differential data of arbitrary order.
- The algorithm needs an auxiliary network to propagate backward the error originating from differential data.
- The algorithm is completely local as the standard back propagation algorithm [34]. This is a very important trait of the algorithm since this allows the algorithm deal with neural networks of general structure. For example, the algorithm applies to neural networks without any layer structure or fully connected.
- The algorithm allows each unit has any sigmoid function.

Above neural networks and the algorithm learning differential data are given substances as computer programs. It took a long time (actually many years) to implement them right. We could not realize it during the implementation, but now with hindsight we know that they are very complex and difficult to implement. We believe there is no other implementation of this kind elsewhere.

Then we have conducted the experiments to see the neural networks really converge for second and third order differential data. It sounds simple enough. However it has been a very difficult task since no one has ever tried it yet. When the neural networks do not seem to converge, it is very difficult to tell what is wrong. There are possibilities of mistakes in the algorithm, the programs, or the parameters. Maybe the neural network takes much longer time than expected to converge or the network simply does not converge on the particular differential data. Therefore we believe it is a very important step toward the practical applications of neural networks learning differential data that we have shown the neural networks have actually converged on differential data in this paper.

We have also conducted following analyses on neural networks learning differential data mainly in the first order cases to reveal their characteristics.

- Comparison with the standard back propagation algorithm with more data
- How learnings work
- Sample complexity
- Effect of irrelevant features
- Noise robustness

As practical applications of neural networks learning differential data, we have applied them to solving differential equations in reinforcement learning framework. There are already several works [17, 4, 39, 16] on the application of neural networks to solving differential equations.

Lee and Kang [17] solve certain types of first order ordinary differential equations by using Hopfield-type neural networks to minimize the finite difference equations.

In [4, 39, 16], differential equations along with boundary conditions are turned into minimization problems. Those minimization problems are solved using neural networks with global minimization procedures such as quasi Newton gradient descent algorithm.

We use essentially the same framework as one in [4, 39, 16], of mapping differential equations, boundary conditions, and any other conditions into minimization problems by neural networks. But we employ our implementation of neural networks learning differential data for the minimization procedure different from the previous researches.

Our algorithm is completely localized to each unit (neuron) and it leaves possibilities of parallel implementations for efficient executions, while global minimization procedures are very difficult to implement in parallel ways. It is also very difficult to apply global minimization procedures for adaptive problems. In many learning problems in robotics and others, what has to be learned, changes dynamically. In the problem of continuous action generation in reinforcement learning described in Chapter 6, the probability distribution given in the output space changes as the learning proceeds and so does the differential equation for the neural network to satisfy. In such adaptive cases, our algorithm provides more gradual way of learning than global minimization procedures.

The difficulty with applications of neural networks learning differential data to differential equations is that of moving targets. To minimize the square error of differential equations, neural networks need to propagate backward the errors, which depend on the outputs of the neural network. That is so even for linear differential equations. That means neural networks need to learn or to adapt to something like moving targets. This is totally different situation from the standard back propagation algorithm where the training data is fixed. Therefore there is difficulty of moving targets added to learning differential data.

In the application of neural networks learning differential data to differential equations that appeared in reinforcement learning (See [37] for reinforcement learning), we use neural networks to create random vector generators that realize continuous action generation for any probability distribution. The neural networks have learned differential data derived from the differential equation quite successfully and learned to satisfy the differential equation approximately.

In addition to the above application to the problem in reinforcement learning, two other possible applications of neural networks learning differential data are proposed. Those are differential equations and simulation of human arm. For differential equations, we propose a very general framework that unifies differential equations, boundary conditions, and other constraints. For the simulation, we propose a natural implementation of the minimum-torque-change model.

Finally there are items explored in this dissertation on radial basis function (RBF) networks with utilizing differential data in mind. Those items include the followings:

- Minimizing solutions for the cases with differential error terms
- Best approximation property of the above solutions
- C^l denseness of RBF networks

1.5 Structure of this Dissertation

This section provides the structure of this dissertation.

Chapter 2 reviews the theoretical framework for neural networks mainly about multilayer perceptrons and their learning algorithm, back propagation.

Chapter 3 describes the structure and the algorithm for the neural networks learning differential data.

Chapter 4 gives the outline of implementation and results of experiments on the neural networks learning differential data. Results of many illustrative experiments along with experiments with up to third order differential data, and experiments on a two-dimensional function are reported.

Chapter 5 presents analyses of neural networks learning differential data mainly in the first order cases. Analyses include comparison with extra pattern scheme, how learnings work, sample complexity, effect of irrelevant features, and noise robustness.

Chapter 6 describes an application of neural networks learning differential data to continuous action generation in reinforcement learning. The chapter describes the problem of continuous action generation in reinforcement learning, gives formalization for the problem, and illustrates the results of the experiments.

Chapter 7 proposes two other possible applications of neural networks learning differential data, differential equations and simulation of human arm.

Chapter 8 describes higher order extensions to radial basis function (RBF). Items explored in the chapter include minimizing solutions for the cases with differential error terms, best approximation property of the above solutions, and C^l denseness of RBF networks.

Chapter 9 concludes this dissertation with a summary of the contributions, a discussion of the limitations, and suggestions for future work.

Chapter 2

Multilayer Perceptron

In this chapter, we introduce briefly neural networks in general, describe the multilayer perceptrons, and explain their learning method, back propagation.

2.1 Neural Networks

In this section, we give a very brief introduction of neural networks in general. The content of this section is largely based on [1].

Neural networks can be very loosely defined as follows (This is an adaptation of the definition of neural network information processing in [1]).

> Inspired by neural networks of higher animals, a neural network is a network capable of information processing, in which a large number of relatively simple information processing units are connected together and in which these units communicate with each other by relatively simple signals.

Therefore the important components of neural networks are "unit" and "connection." [1] The neural networks can be categorized in two ways: how the units are connected and the type of information processing in the unit.

From the point of how the units are connected, the neural networks are categorized as follows: [2]

- Layered network
- Mutually connected network

A layered network is a network, which has a layered structure of units with layers ordered from the input layer to the output layer. A Unit in a layer is only connected to the units in the next higher layer. Radial Basis Function (RBF) networks investigated in Chapter 8 are layered networks with three layers. Multilayer perceptrons in general fall in this category. [3]

[1] Rumelhart et al. [33] give more detailed, eight major aspects of parallel distributed processing model.

[2] Though this distinction is useful, the distinction has been blurred recently. We include the categorization for general understanding.

[3] We give a more general definition for multilayer perceptrons in Section 2.2.

A mutually connected network is a network, which allows connections between any two units for both directions. Hopfield networks and Boltzmann machines explained in the following use networks of this type.

From the point of the type of information processing in the unit, the neural networks are categorized as follows: [4]

- Hopfield networks
- Boltzmann machine
- Back propagation

A Hopfield network [12] use a mutually connect network with symmetrical weights. Hopfield networks are used for associative memories and solving optimization problems.

A "Boltzmann machine" [10] is essentially a stochastic version of Hopfield network. A Boltzmann machine can also learn the probabilities of states of the environment and can simulate the environment later.

Back propagation is a learning algorithm (procedure) proposed for multilayer networks. We explain the algorithm in Section 2.3.

After this section, we use the word, "neural network" to mean a multilayer perceptron without any confusion since we do not handle the other types of the networks hereafter.

2.2 Structure of Multilayer Perceptrons

In this section, we define and describe the structure of multilayer perceptrons.

First, we define a "unit," which is also called a "neuron."

Definition 2.1 *A unit has more than or equal to one inputs and a single output. The weighted sum of the inputs is combined with a bias and then is operated on by a sigmoid function to produce the output. Let n be the number of inputs to the unit, o be the output, x_i's be the inputs, w_i's be the weights, b be the bias, and σ be the sigmoid function. The output of the unit is given as follows.*

$$o = \sigma\left(\sum_{i=1}^{n} w_i\, x_i + b\right) \tag{2.1}$$

Here n is a positive integer, o, x_i's, w_i's, and b are real numbers, and the sigmoid function, σ is a one-dimensional non-linear monotonic differentiable function.

Even though any function can serve as a sigmoid function for a unit as long as it is one-dimensional, non-linear, monotonic, and differentiable, the following function given by Equation (2.1) is used in our implementation of neural networks and throughout the experiments described in this dissertation.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

The graph of this function is given by Figure 2.1.

[4] Only major categories are listed.

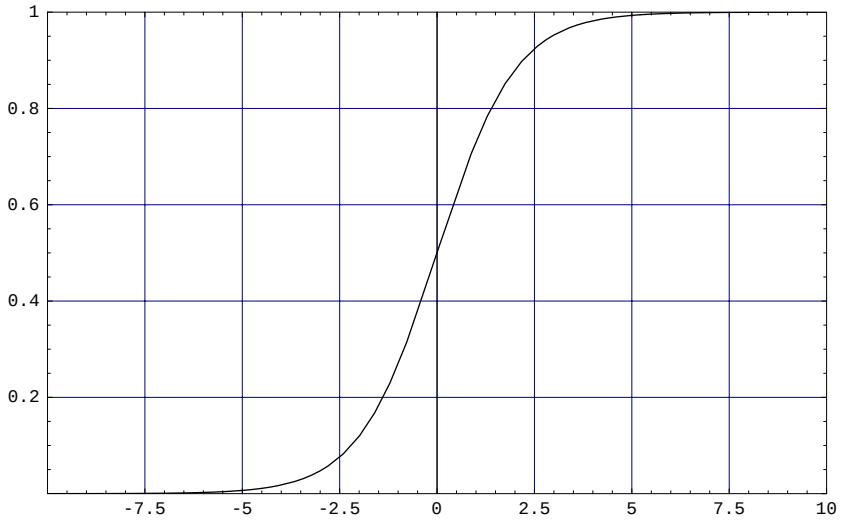

Figure 2.1: Sigmoid Function: The figure shows the sigmoid function, $\sigma(x) = 1/(1+e^{-x})$, used in the implementation and the experiments.

Sigma-Pi Unit There is another type of units called "Sigma-Pi units." (See [33] and [34].) Even though units of this type do not constitute multilayer perceptrons, we use those units in the construction of our δ nets. (See Section 3.5.2) The output of a unit of this type is given as follows and the units are named after this form, where $\{\{i_k|k=1,...,l_i\}|i=1,...,m\}$ gives a partitioning of n inputs.

$$o = \sigma(\sum_{i=1}^{m} w_i \prod_{k}^{l_i} x_{i_k}) \tag{2.3}$$

We give our definition of a multilayer perceptron as follows.

Definition 2.2 *A multilayer perceptron is a network of units defined in Definition 2.1, where a unit receives outputs of other units or the input from the environment as its inputs, and where a unit outputs to other units or to the environment.*

A unit, which receives the input from the environment, is called an "input unit." A unit, which outputs to the environment, is called an "output unit." A multilayer perceptron must have at least one input unit and one output unit.

Here the "environment" means the outside of the network. We decided to allow any connection in the network. Only restriction is that the network is connected to the environment (i.e. outside) in both ways. We do not assume any layer structure for a multilayer perceptron. [5] Units in the network can have different

[5]We understand that this is a little misleading. But we think that using a different name is more confusing.

sigmoid functions. Back propagation is applicable to multilayer perceptrons of this definition and so is the algorithm for neural networks learning differential data proposed in Chapter 3.

Though we have given a very general definition of a multilayer perceptron, many of typical multilayer networks and t he networks used in the implementation and the experiments have the structure like the one depicted in Figure 2.2. This kind of the networks is actually a layered network. (See Section 2.1) The input layer is the layer, which consists entirely of the input units. The output layer is the layer, which consists entirely of the output units. Hidden layers are the layers, which consists entirely of the units without connections to the environment. A hidden layer is sometimes called a middle layer.

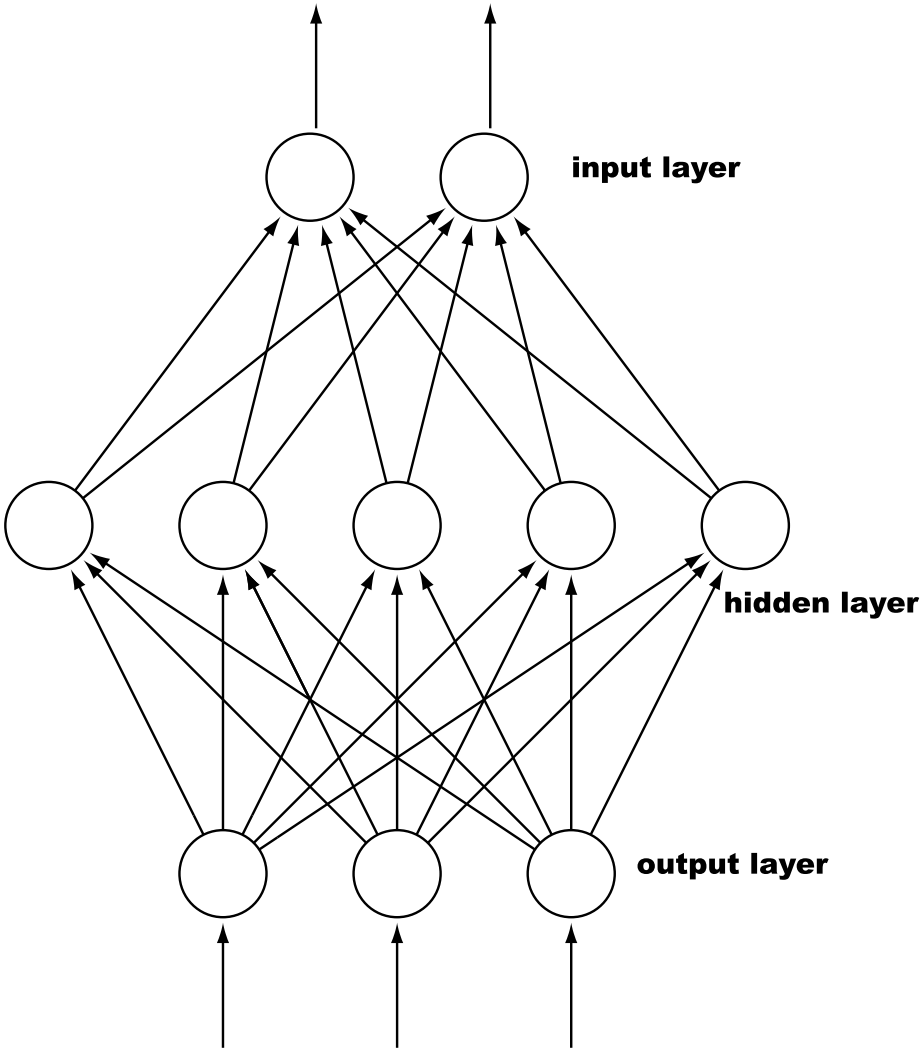

Figure 2.2: Structure of Multilayer Perceptron: The structure of a typical multilayer perceptron with three layers is depicted.

2.3 Back Propagation for Multilayer Perceptron

In this section, we explain briefly back propagation, a learning algorithm for multilayer perceptrons. See [34] for more details.

There are two kinds of information processing performed in multilayer perceptrons. First one is the forward propagation as defined in Definition 2.1, of the input by the environment through the network from the input units to the output units. The other one is the learning algorithm, which consists of the back propagation of the errors by the environment through the network from the output units to the input units, and weight and bias updates.

The latter is called "back propagation" as a whole. The purpose of back propagation is to adjust the internal state (weights and biases) of the multilayer perceptron so that the multilayer perceptron produces the desired output for the specified input.

In order to realize this, the following error function (, which is sometimes called "energy function,") E is defined for desired input and output pairs $\{\{Ip, O_p\}_p\}$, [6] where n is the function defined by the multilayer perceptron.

$$E = \frac{1}{2}\sum_p (O_p - n(I_p))^2 \tag{2.4}$$

For each pair, the multilayer perceptron is made to propagate the input I_p forward, then the squared distance between the output of the network $n(I_p)$ and the desired output is calculated. The squared distances are summed for all the pairs and divided by 2 to produce the error function. If the error function is 0, it means that the multilayer network produces exactly the desired output for each input.

The back propagation algorithm is essentially a gradient descent procedure with respect to this error function. The weights (and the biases) are therefore updated as follows, where α is some positive constant called the "learning constant."

$$\Delta w = -\alpha \frac{\partial E}{\partial w} \tag{2.5}$$

Furthermore a momentum term by the past weight update value $\tilde{\Delta} w$ is added as follows to avoid oscillation for a practical purpose of making rapid learning possible [34],

$$\Delta w = -\alpha \frac{\partial E}{\partial w} + \beta \tilde{\Delta} w \tag{2.6}$$

where β is a positive constant less than 1.

The back propagation algorithm is used for calculation of δ [7], the value assigned to each unit. In back propagation, this δ is propagated backward from the unit to those units, which output to that unit. Actually back propagation is embodiment of repeated applications of the chain rule for partial derivatives.

[6] A desired input and output pair, an input, and an output are sometimes called a pattern, an input pattern, and an output pattern.

[7] This is why back propagation is sometimes called generalized delta rule. The delta rule itself dates back to the learning algorithm for Perceptron [40].

The details of the algorithm and its derivation is omitted here since the algorithm is the restriction on the value net of the algorithm for neural network learning differential data proposed in Chapter 3.

What is important is that this algorithm is completely localized to each unit. A weight update can be calculated from δ and the output of the unit involved. This is the reason why back propagation is applicable not only to single-layer, but also to multilayer perceptrons and such networks even without layer structures.

Chapter 3

Algorithm for Learning Differential Data

In this chapter, we describe the architecture of an extended multilayer perceptron and its algorithm to learn differential data. We also give the correspondence between the proposed algorithm and tangent prop in first order case at the end of this chapter.

3.1 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.

$$\begin{aligned} &\delta_1 = (a_1, ..., a_n) > \delta_2 = (b_1, ..., b_n) \\ &\leftrightarrow \\ &\exists i \quad a_i > b_i \ \wedge \ \forall i \quad a_i \geq b_i \end{aligned}$$

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

$$\frac{\partial^{N(\delta)}}{\partial^{\delta} x} = \frac{\partial^{N(\delta)}}{\partial^{a_1} x_1 \ ... \ \partial^{a_n} x_n} \tag{3.1}$$

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

$$x^{\delta} = x_1^{a_1} \times \cdots \times x_n^{a_n} \tag{3.2}$$

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

$$N(\Delta) \quad = \quad m \tag{3.3}$$

$$T(\Delta) = \sum_{i=1}^{m} c_i \tag{3.4}$$

$$Sum(\Delta) = \sum_{i=1}^{m} c_i \times \delta_i \tag{3.5}$$

$$\Delta(f) = \prod_{(c,\delta)\in\Delta} \left(\frac{\partial^{N(\delta)}}{\partial^{\delta} x} \right)^{c} \tag{3.6}$$

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

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.

$$\frac{\partial^{N(\delta)}}{\partial^{\delta} x}\, e^{f} = \sum_{(d_{\delta,\Delta},\Delta)\, \in\, V(\delta)} d_{\delta,\Delta}\ \Delta(f)\ e^{f} \tag{3.7}$$

$p_{l,i}^{\Delta} = \prod_{(c,\delta')\in\Delta} \left(y_{l,i}^{\delta'} \right)^{c}$ **:**
This is defined by Equation (3.15) again.

3.2 Framework of the Problem

In this section, we give the framework of the problem. n-dimensional input space, m-dimensional output space [1] 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 [2] 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.

3.3 Network Structure

Figure 3.1 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 [36]. On the left-hand side of Figure 3.1 there are the units in the multilayer perceptron part that we call the value net. There is corresponding δ net for each differential operator δ. Figure 3.1 shows one of those δ nets on the right side.

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

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

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 3.1 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.

$$x^\delta_{l,i} = \frac{\partial^{N(\delta)} x^0_{l,i}}{\partial^\delta x_I} \tag{3.8}$$

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.

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 $b_{\{l,i\}}$.

3.4 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 limit to the cases of differentiation along axes, since the general cases need excessively 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 (3.8). 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.

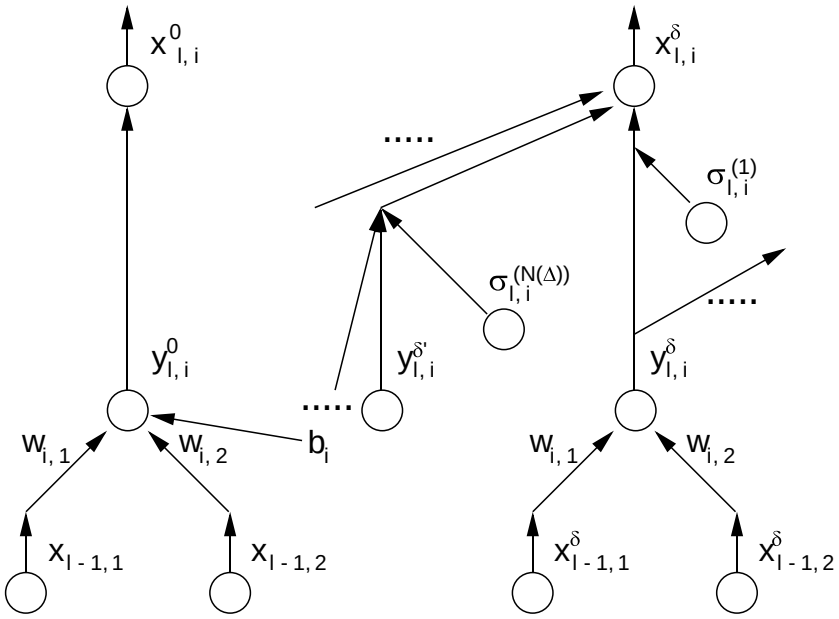

Figure 3.1: *Network structure for learning differential data:* On the left-hand side of the figure there are the units in the multilayer perceptron part, which we call the value net. There is corresponding δ net for each differential operator δ. This figure shows one of those δ nets on the right side. 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 3.1 does not show. Connection weights $w_{i,j}$ are same in the value net and δ nets.

3.5 Forward Propagation

First, we give the algorithm for forward propagation. We show how the input $I = (I^0, ..., I^\delta, ...)$ is 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.

3.5.1 Units 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 sigmoid 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 sigmoid functions.

$$net \triangleright x^0_{l,i} = y^0_{l,i} = net \triangleright y^0_{l,i} = b_i + \sum_{j \in P(i)} w_{ij} x_{l-1,j} \tag{3.9}$$

$$x^0_{l,i} = x_{l,i} = \sigma(net \triangleright x^0_{l,i}) = \sigma(y^0_{l,i}) \tag{3.10}$$

3.5.2 Units 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.

$$net \triangleright \sigma^{(m)}_{l,i} = y^0_{l,i} \tag{3.11}$$

$$\sigma^{(m)}_{l,i} = \sigma^{(m)}(y^0_{l,i}) \tag{3.12}$$

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

$$y^\delta_{l,i} = net \triangleright y^\delta_{l,i} = \sum_j w_{i,j} x^\delta_{l-1,j} \tag{3.13}$$

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 [34].

$$\begin{aligned} x^\delta_{l,i} &= net \triangleright x^\delta_{l,i} = \frac{\partial^{N(\delta)}}{\partial^\delta x}(x^0_{l,i}) \\ &= \sum_{(d_{\delta,\Delta}, \Delta) \in V(\delta)} d_{\delta,\Delta} \left\{ \sigma^{(T(\Delta))}_{l,i} \prod_{(c,\delta') \in \Delta} \left(y^{\delta'}_{l,i} \right)^c \right\} \end{aligned} \tag{3.14}$$

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

$$p^\Delta_{l,i} = \prod_{(c,\delta') \in \Delta} \left(y^{\delta'}_{l,i} \right)^c \tag{3.15}$$

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

proof of the last equation in (3.14) 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 (3.14), 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 (3.14) and (3.7) by the correspondence of the σ function in (3.14) to the exponential function e in (3.7). Therefore $d_{\delta,\Delta}$ in Equation (3.14) is exactly the same as the one in Equation (3.7). □

3.6 Error Function

We define the error function by the following equation.

$$E = \sum_{\delta \geq 0} \alpha^{\delta} E^{\delta} \tag{3.16}$$

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, which is sometimes called teacher signal, E^{δ} is given by the following equation. [3]

$$E^{\delta} = \frac{1}{2} \sum_{p} \left(O^{\delta}_p - x^{\delta}_p \right)^2 \tag{3.17}$$

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 [34]. 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 [4], update values for each δ have to be managed and more memory space is needed. 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.

3.7 Back Propagation

Using E defined in Section 3.6, $w_{i,j}$ is updated as follows.

$$\Delta w_{i,j} \quad = \quad -\frac{\partial E}{\partial w_{i,j}} \tag{3.18}$$

First, we are going to show the back propagation algorithm of ϵ for each type of units. Then we 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

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

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

error function with respect to the input to the unit." We denote that value by $\epsilon \triangleright u$ for the unit u. Therefore the definition of $\epsilon \triangleright u$ is given by the following equation for the error function E.

$$\epsilon \triangleright u = -\frac{\partial E}{\partial net \triangleright u} \tag{3.19}$$

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.

3.7.1 Units in the Output Layer (BP)

ϵ's for the units x^δ's in the output layer [5] are given as follows. (In the following equations subscripts for the layer and the unit are omitted to avoid too complex equations.)

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

$$\begin{aligned} \epsilon \triangleright x^0 &= -\frac{\partial E}{\partial net \triangleright x^0} = -\alpha^0 \frac{\partial E^0}{\partial net \triangleright x^0} \\ &= -\alpha^0 \frac{\partial E^0}{\partial x^0} \frac{\partial x^0}{\partial net \triangleright x^0} = \alpha^0 \left(O_p^0 - x_p^0\right) \sigma^{(1)}(net) \end{aligned} \tag{3.20}$$

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

$$\begin{aligned} \epsilon \triangleright x^\delta &= -\frac{\partial E}{\partial net \triangleright x^\delta} = -\alpha^\delta \frac{\partial E^\delta}{\partial net \triangleright x^\delta} \\ &= -\alpha^\delta \frac{\partial E^\delta}{\partial x^\delta} = \alpha^\delta \left(O_p^\delta - x_p^\delta\right) \end{aligned} \tag{3.21}$$

3.7.2 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.

$$\begin{aligned} \epsilon \triangleright x_{l-1,i}^0 &= -\frac{\partial E}{\partial net \triangleright x_{l-1,i}^0} \\ &= \sum_j -\frac{\partial E}{\partial net \triangleright y_{l,j}^0} \frac{\partial net \triangleright y_{l,j}^0}{\partial net \triangleright x_{l-1,i}^0} \\ &= \sum_j \epsilon \triangleright y_{l,j}^0 \; w_{j,i} \; \sigma_{l-1,i}^{(1)} \\ \epsilon \triangleright y_{l-1,i}^0 &= -\frac{\partial E}{\partial net \triangleright y_{l-1,i}^0} \end{aligned} \tag{3.22}$$

[5] 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.

$$
\begin{aligned}
&= \sum_j -\frac{\partial E}{\partial net \triangleright x^0_{l,j}} \frac{\partial net \triangleright x^0_{l,j}}{\partial net \triangleright y^0_{l-1,i}} + \sum_{m\geq 1} -\frac{\partial E}{\partial net \triangleright \sigma^{(m)}_{l-1,i}} \frac{\partial net \triangleright \sigma^{(m)}_{l-1,i}}{\partial net \triangleright y_{l-1,i}} \\
&= \epsilon \triangleright x^0_{l-1,i} + \sum_{m\geq 1} \epsilon \triangleright \sigma^{(m)}_{l-1,i} \qquad (3.23)
\end{aligned}
$$

Even though $net \triangleright x^0_{l-1,i} = net \triangleright y^0_{l-1,i} = net_{l-1,i}$ (c.f. Equation (3.9)), $x^0_{l-1,i}$ and $y^0_{l-1,i}$ units need different back propagation algorithms. This is because their outputs are connected to the different units. $x^0_{l-1,i}$ unit's output is connected to $y^0_{l,j}$ unit and $y^0_{l-1,i}$ unit's output is connected to $x^0_{l-1,i}$ units and σ units.

3.7.3 Units in the δ Net (BP)

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

$$
\begin{aligned}
\epsilon \triangleright x^\delta_{l-1,i} &= -\frac{\partial E}{\partial net^\delta_{l-1,i}} = \sum_j -\frac{\partial E}{\partial net \triangleright y^\delta_{l,j}} \frac{\partial net \triangleright y^\delta_{l,j}}{\partial net^\delta_{l-1,i}} \\
&= \sum_j \epsilon \triangleright y^\delta_{l,j} \frac{\partial y^\delta_{l,j}}{\partial x^\delta_{l-1,i}} \\
&= \sum_j \epsilon \triangleright y^\delta_{l,j}\ w_{j,i} \qquad (3.24)
\end{aligned}
$$

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

$$
\begin{aligned}
\epsilon \triangleright \sigma^{(m)}_{l,i} &= -\frac{\partial E}{\partial net \triangleright \sigma^{(m)}_{l,i}} = -\frac{\partial E}{\partial \sigma^{(m)}_{l,i}} \frac{\partial \sigma^{(m)}_{l,i}}{\partial net_{l,i}} \\
&= \sum_{\delta: N(\delta)\geq m} -\frac{\partial E}{\partial x^\delta_{l,i}} \frac{\partial x^\delta_{l,i}}{\partial \sigma^{(m)}_{l,i}} \sigma^{(m+1)}_{l,i} = \sum_{\delta: N(\delta)\geq m} \epsilon \triangleright x^\delta_{l,i} \frac{\partial x^\delta_{l,i}}{\partial \sigma^{(m)}_{l,i}} \sigma^{(m+1)}_{l,i} \\
&= \sum_{\delta: N(\delta)\geq m} \epsilon \triangleright x^\delta_{l,i} \\
&\quad \times \left(\sum_{\Delta: T(\Delta)=m \wedge Sum(\Delta)=\delta} d_{\delta,\Delta} \prod_{(c,\delta')\in\Delta} \left(y^{\delta'}_{l,i}\right)^c \right) \sigma^{(m+1)}_{l,i} \\
&= \sum_{\delta: N(\delta)\geq m} \epsilon \triangleright x^\delta_{l,i} \\
&\quad \times \left(\sum_{\Delta: T(\Delta)=m \wedge Sum(\Delta)=\delta} d_{\delta,\Delta}\ \ p^\Delta_{l,i} \right) \sigma^{(m+1)}_{l,i} \qquad (3.25)
\end{aligned}
$$

For the $y^\delta_{l,i}$ units in δ net, the back propagation algorithm will be given by the following equation. (Note that $\partial net^{\delta'}_{l,i} / \partial net \triangleright y^\delta_{l,i} = \partial x^{\delta'}_{l,i} / \partial y^\delta_{l,i}$.)

$$
\epsilon \triangleright y^\delta_{l,i} = -\frac{\partial E}{\partial net \triangleright y^\delta_{l,i}}
$$

$$
\begin{aligned}
&= \sum_{\delta':\delta'\geq\delta} -\frac{\partial E}{\partial net^{\delta'}_{l,i}} \frac{\partial net^{\delta'}_{l,i}}{\partial net \triangleright y^{\delta}_{l,i}} \\
&= \sum_{\delta':\delta'\geq\delta} \epsilon \triangleright x^{\delta'}_{l,i} \\
&\quad \times \sum_{\Delta:Sum(\Delta)=\delta' \wedge \exists c\,(c,\delta)\in\Delta} d_{\delta',\Delta} \sigma^{(T(\Delta))}_{l,i}\; c\; (y^{\delta}_{l,i})^{c-1} \\
&\quad \times \left(\prod_{(c'',\delta'')\in(\Delta-\{(c,\delta)\})} (y^{\delta''}_{l,i})^{c''} \right) \qquad (3.26)
\end{aligned}
$$

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

$$
\epsilon \triangleright x^{\delta}_{l-1,i} = \sum_{j} \epsilon \triangleright y^{\delta}_{l,j}\; w_{j,i} \qquad (3.27)
$$

$$
\begin{aligned}
\epsilon \triangleright \sigma^{(m)}_{l,i} &= \sum_{\delta:N(\delta)\geq m} \epsilon \triangleright x^{\delta}_{l,i} \\
&\quad \times \left(\sum_{\Delta:T(\Delta)=m \,\wedge\, Sum(\Delta)=\delta} d_{\delta,\Delta}\quad p^{\Delta}_{l,i} \right) \sigma^{(m+1)}_{l,i} \qquad (3.28)
\end{aligned}
$$

$$
\begin{aligned}
\epsilon \triangleright y^{\delta}_{l,i} &= \sum_{\delta':\delta'\geq\delta} \epsilon \triangleright x^{\delta'}_{l,i} \\
&\quad \times \sum_{\Delta:Sum(\Delta)=\delta' \wedge \exists c\,(c,\delta)\in\Delta} d_{\delta',\Delta}\; \sigma^{(T(\Delta))}_{l,i}\; c\; (y^{\delta}_{l,i})^{c-1} \\
&\quad \times \left(\prod_{(c'',\delta'')\in(\Delta-\{(c,\delta)\})} (y^{\delta''}_{l,i})^{c''} \right) \qquad (3.29)
\end{aligned}
$$

3.8 Update Rules for Weights

In this section, we 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 3.1. 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 3.7, the update rule for the weight $w_{i,j}$ is as follows.

$$
\begin{aligned}
\Delta w_{i,j} &= -\frac{\partial E}{\partial w_{i,j}} = -\sum_{\delta\geq 0} \frac{\partial E}{\partial net \triangleright y^{\delta}_{l,i}} \frac{\partial net \triangleright y^{\delta}_{l,i}}{\partial w_{i,j}} \\
&= \sum_{\delta\geq 0} \epsilon \triangleright y^{\delta}_{l,i}\; x^{\delta}_{l-1,j} \qquad (3.30)
\end{aligned}
$$

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

Thresholds b_i's [6] are special kinds of weights, which appear only in the value net. Therefore the update rule for the threshold is as follows.

[6] Thresholds are sometimes called biases.

$$\begin{aligned}\Delta b_i &= -\frac{\partial E}{\partial b_i} = -\frac{\partial E}{\partial net \triangleright y^0_{l,i}} \frac{\partial net \triangleright y^0_{l,i}}{\partial b_i} \\ &= \epsilon \triangleright y^0_{l,i} \end{aligned} \tag{3.31}$$

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

In cases we use momentum (c.f. [34]) 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.)

$$\Delta w_{i,j} = \sum_{\delta \geq 0} \epsilon \triangleright y^\delta_{l,i}\, x^\delta_{l-1,j} + \beta \tilde{\Delta} w_{i,j} \tag{3.32}$$

$$\Delta b_i = \epsilon \triangleright y^0_{l,i} + \beta \tilde{\Delta} b_i \tag{3.33}$$

3.9 Correspondence to Tangent Prop in the First Order Case

Up to here, Section 3 has given an algorithm for learning differential data of arbitrary order.

In this section, we give an algorithm for learning differential data of first order as a special case and establish the equivalence between tangent prop as described in [36] and the algorithm proposed here.

The following correspondence establishes the equivalence. First, the value net in this paper corresponds to the Network in [36] and the δ net of the first order corresponds to the Jacobian network in [36]. The symbol correspondences are given in the following, where $\delta = (0, ..., 0, 1, 0, ..., 0)$. The symbols on the left-hand side are those of [36] and the symbols on the right-hand side are those of this paper.

- $a^l_i \rightarrow y^0_{l,i} = net \triangleright x^0_{l,i}$
- $x^l_i \rightarrow x^0_{l,i}$
- $b^l_i \rightarrow -\frac{\partial E}{\partial x^0_{l,i}}$
- $y^l_i \rightarrow \epsilon \triangleright y^0_{l,i}$
- $\alpha^l_i \rightarrow y^\delta_{l,i}$
- $\xi^l_i \rightarrow x^\delta_{l,i}$
- $\beta^l_i \rightarrow \epsilon \triangleright x^\delta_{l,i}$
- $\psi^l_i \rightarrow \epsilon \triangleright y^\delta_{l,i}$
- $\sigma'(a^l_i) \rightarrow \sigma^{(1)}_{l,i}$
- $\sigma''(a^l_i) \rightarrow \sigma^{(2)}_{l,i}$

Since the network has no δ net higher than first order, the followings hold for $\delta = (0, ..., 0, 1, 0, ..., 0)$.

- $V(\delta) = \{(1, (1, \delta))\}$
- $T((1, \delta) = 1$

The algorithms are given as follows by the symbols of this paper. The forward propagation algorithms for the value net and the δ nets are given by Equations (3.9) and (3.10), and Equations (3.12), (3.13), and (3.14) respectively. Except for the last one, they are not any simpler for the first order case. Equation (3.14) takes the following form for the first order case.

$$x^{\delta}_{l,i} = \sigma^{(1)}_{l,i} y^{\delta}_{l,i} \tag{3.34}$$

The backward propagation algorithms for the value net and the δ nets are given by Equations (3.22) and (3.23), and Equations (3.27), (3.28), and (3.29) respectively. Those equations take the following forms for the first order case.

$$\epsilon \triangleright x^{0}_{l-1,i} = \sum_{j} \epsilon \triangleright y^{0}_{l,j} \; w_{j,i} \; \sigma^{(1)}_{l-1,i} \tag{3.35}$$

$$\epsilon \triangleright y^{0}_{l-1,i} = \epsilon \triangleright x^{0}_{l-1,i} + \epsilon \triangleright \sigma^{(1)}_{l-1,i} \tag{3.36}$$

$$\epsilon \triangleright x^{\delta}_{l-1,i} = \sum_{j} \epsilon \triangleright y^{\delta}_{l,j} \; w_{j,i} \tag{3.37}$$

$$\epsilon \triangleright \sigma^{(1)}_{l,i} = \epsilon \triangleright x^{\delta}_{l,i} \; y^{\delta}_{l,i} \; \sigma^{(2)}_{l,i} \tag{3.38}$$

$$\epsilon \triangleright y^{\delta}_{l,i} = \epsilon \triangleright x^{\delta}_{l,i} \; \sigma^{(1)}_{l,i} \tag{3.39}$$

The weight update rules given by Equations (3.32) and (3.33) are the same for the first order case as for the general case.

Along with the correspondence provided above between the symbols, these equations are equivalent to the equations from (3) to (6) in [36].

Chapter 4

Implementation and Experiments

In this chapter, we describe the implementation of the proposed architecture and algorithm for neural networks learning differential data. Then we illustrate the results of rather simple experiments by the above implementation.

4.1 Outline of the Program

The architecture and the algorithm of neural networks learning differential data as described in Chapter 3 have been implemented. The main module of the program is written in C++ while auxiliary modules are written in Yacc, Lex, Perl and Java. The total size is about 5,000 lines.

The overall structure of the program is given in Figure 4.1.

The parser module written in Yacc and Lex of the program reads a configuration file with C++-like syntax as in Figure 4.3.

The program creates the structure of Figure 3.1 up to the necessary differential order. Then it propagates the input forward and the errors backward through the network to learn the given data. During the learning, the main module communicates with the graphical user interface (GUI) module (see Figure 4.2) written in Java through CORBA (Common Object Request Broker Architecture) [9] to display the status of the neural network and let the user change the parameters for the learning.

Though a layered structure of any number of layers and a fixed sigmoid function for all units are being assumed, these restrictions can be easily removed since the program is implemented in object oriented way.

This program produces a log file while it is running. Log files include information on the errors and the values of weights. Perl scripts parse the log files, to produce the graphs to show the changes of errors, and to extract the weights of the network for use in the next session by the program or in Mathematica programs.

The sigmoid function, σ, used for all the units, [1] is given by Equation (4.1). $\sigma(x) = 0.5$ and $\sigma'(x) = 0.25$ when $x = 0$. See Figure 2.1.

[1] Since the sigmoid function is implemented as an object, it is easy to replace this function with the other function.

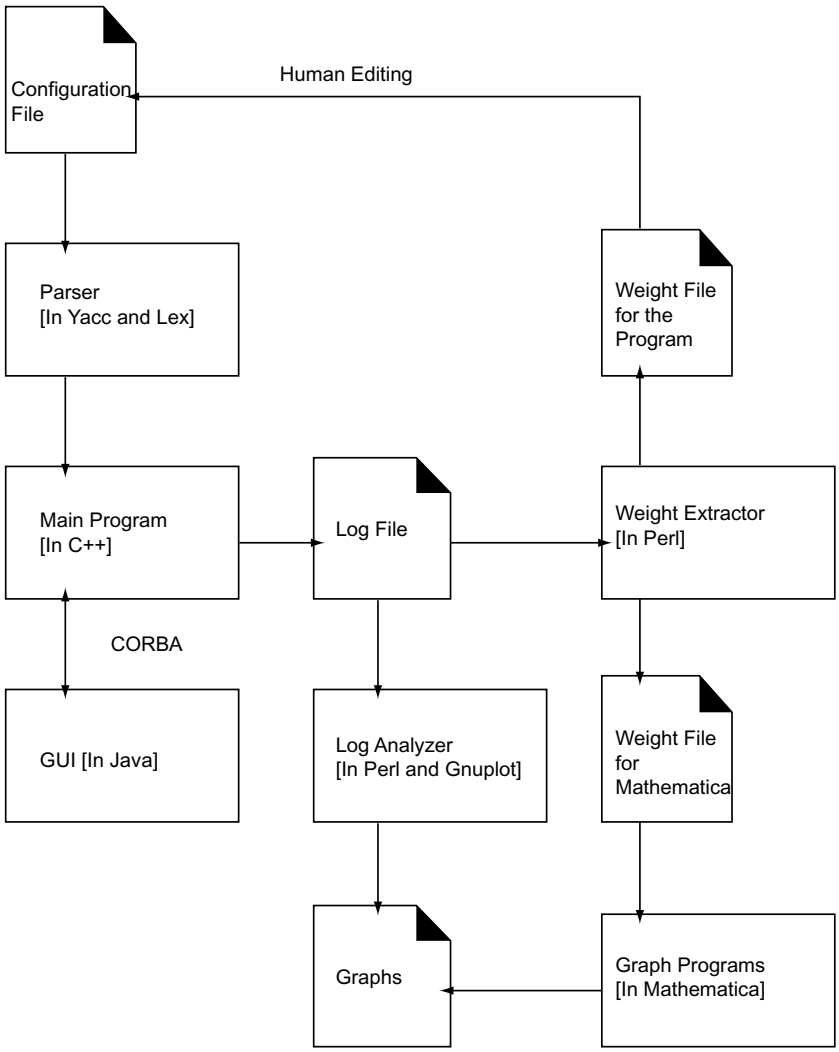

Figure 4.1: Program Structure: The figure shows the overall structure of the program. Details are given in the text.

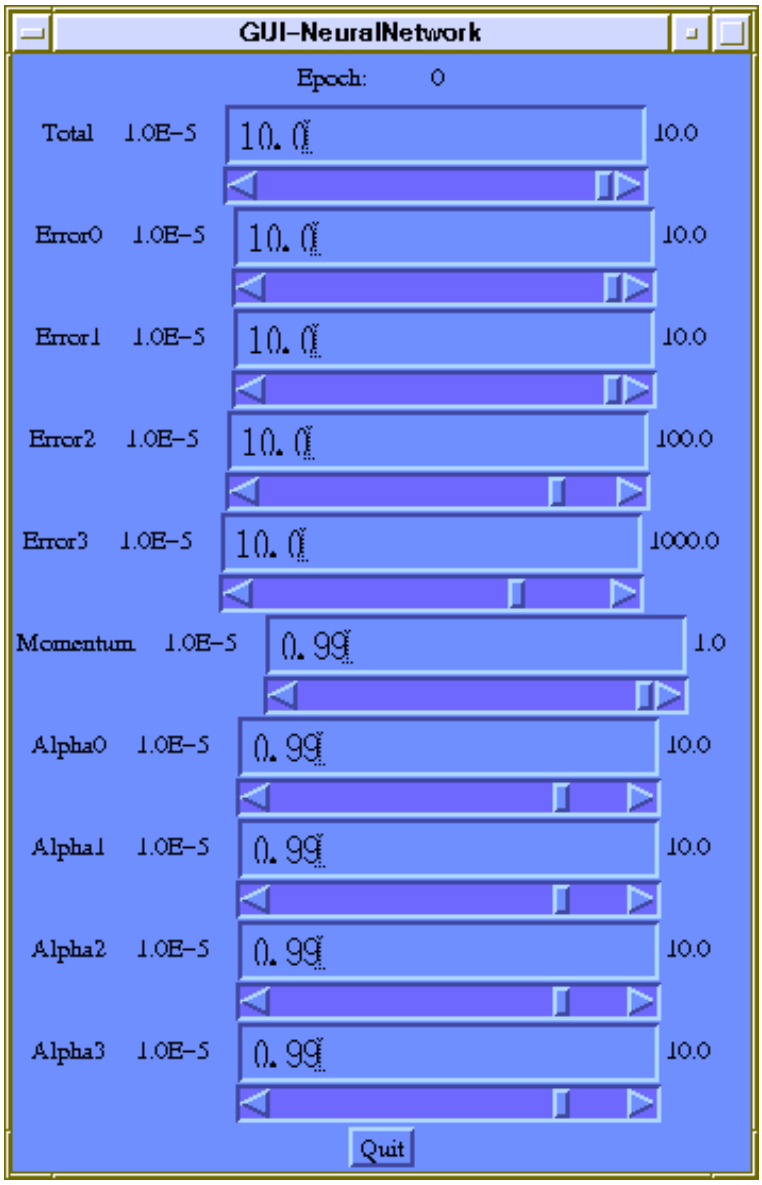

Figure 4.2: Java GUI: The main module of the program communicates with the graphical user interface (GUI) written in Java through CORBA. Users can see the status of the neural network such as the current errors. They can also set the parameters such as learning constants and momentum during learning dynamically by adjusting the slide bar or by typing the number in directly.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

```
//
// e01.cfg
// This is a configuration file for an experiment.
//

number_of_layers = 3 ;

number_of_units = {1,6,1} ;

number_of_differentials = 1;
differential_list = {1};

create_network();

set_alphas({0.40, 0.25});

add_data({0.1, 1.0,     0.5, 2.35619});
add_data({0.5, 1.0,     0.5, -2.35619});
add_data({0.9, 1.0,     0.5, 2.35619});

epoch = 20000;
learn();

quit();
```

Figure 4.3: Example Configuration File

4.1.1 Correspondence between the Algorithm and the Program

This section gives the correspondence between the algorithm described in Section 3 and the program. The information given in this section is utilized when one needs to see the program and modify it.

δ in Section 3.1 corresponds to the object, "Element" in the program. An "Element" has non-negative integer a_i for its i-th slot and this corresponds to the differential operator, $\partial^{a_i}/\partial^{a_i}x_i$. As a whole, an "Element" with a_i in its i-th slot respectively corresponds to the following differential operators.

$$\frac{\partial^{N(\delta)}}{\partial^{\delta}x} = \frac{\partial^{N(\delta)}}{\partial^{a_1}x_1 ... \partial^{a_n}x_n} \tag{4.2}$$

Δ in Section 3.1 corresponds to the object, "Term" in the program. $N(\Delta)$ corresponds to "term.num_elements" in the program. c_i corresponds to "term.powers[i]" for an "Element" δ_i. $T(\Delta)$ corresponds to "term.sigma_diff".

An "Element" (which corresponds to δ) of the program represents the tree defined by the differential operator, $\partial^{N(\delta)}/\partial^{\delta}x$ in Figure 3.1.

A "Term" (which corresponds to Δ) of the program represents the node where the product of outputs of y' units and a σ unit is calculated. [2]

Then as in Equation (3.14), the products are multiplied by the coefficient, $d_{\delta,\Delta}$ and summed. The object "Expression" of the program corresponds to this calculation. The relationship between an "Expression" and an "Element is a one-to-one correspondence. They are separated for the sake of clarity of programming.

$d_{\delta,\Delta}$ in Section 3.1 corresponds to "expression.coefficients[i]", which is the coefficient for the Term, "expression.terms[i]"

In Equation (3.28), the inner summation is taken over the Elements (δ's) that have "total_order" more than m. The outer summation of Equation (3.28) is taken over the Terms (Δ's), which belong to the Expression, which in turn corresponds to the Element δ. [3]

In Equation (3.29), the inner summation is taken over the Elements (δ''s) which have "total_order" more than $N(\delta)$. The outer summation of Equation (3.29) is taken over the Terms (Δ) that belong to the Expression, which in turn corresponds to the Element δ.

4.1.2 Fighting with Bugs

In implementing neural networks as computer programs, it often happens that the program with a few bugs often works and sometimes converges on the training data. Therefore, writing a correct program and checking that the program is working really right is a harder task for neural network implementation than usual programming.

Furthermore it took more than two thousand lines of code to implement the core architecture and algorithm of neural networks learning differential data, while only several tens of lines of code are enough for implementing the core of standard neural network architecture and algorithm. The architecture and the algorithm are much more complex than those of the standard neural networks.

In order to avoid the difficulties and to create a correct program, we take the following measures.

First of all, we used C++ to write the core. Even though, this is the first program for me to write with C++ [4], I believe the choice is the right one. C++ is a strongly typed object oriented programming language. C++ compiler gives us warnings for unmatched types and others and many bugs have been prevented in advance. It would have taken much longer time to implement with C programming language.

The C++ compiler helped us considerably, but the compiler alone does not solve all the problems. The characteristics of neural network implementations described previously prevents from determining the implementation is right.

In order to deal with the problem, we use a neural network model built by Mathematica [41]. The C++ implementation outputs weight values in the log file. The information is used with the Mathematica model and the C++ implementation is checked against the Mathematica model in many aspects such

[2] The node corresponds to the content of the large curly bracket of Equation (3.14)

[3] If the Term belongs to the Expression δ, then $\cup\Delta = \delta$ is guaranteed.

[4] I have written a standard neural network with C.

as forward propagation, error back propagation, and weight and bias updates. This kind of examinations has been conducted many times to remove the bugs.

4.2 Framework of Experiments

In this section, we describe the framework of experiments before we begin to explain the results of experiments. We give mainly the common settings for the experiments.

There is a distinction between batch learning and online learning.

Batch learning is usually employed in a situation where the set of the patterns that the network has to learn is fixed. Forward and backward propagation is executed for each pattern and weight update values are accumulated during the process. The weights are updated only after all the patterns in the set are processed.

Online learning is usually employed in a situation where the pattern comes one after another. Forward and backward propagation is executed for each new pattern and weights are updated each time.

We use *batch learning* with the fixed set of patterns. An *epoch* of batch learning means a whole procedure of forward and backward propagations for patterns in the set and one set of weight updates. *Epoch* also indicates the number of the above procedure executed in the learning up to then.

Important parameters are *learning constants* and *momentums*. We refer to α^{δ} 's in Equation (3.16) as *learning constants*. These parameters determine the size of steps in the gradient descent process. We refer to β 's in Equations 3.32 and 3.33 as *momentums*. The momentum parameter determines the effect of past weight changes on the current direction of movement in weight space. As mentioned in Section 2.3, the term with this parameter is used to avoid oscillation for a practical purpose of making rapid learning possible [34].

4.3 Simple Illustrative Examples

In this section, we give simple and illustrative examples and show the qualitative characteristics of neural networks learning differential data.

We show the effect of differential data for the learning problem by taking the following sine function as a target function.

$$y = 0.3\sin(2\pi(x - 0.1)/0.8) + 0.5 \tag{4.3}$$

In this case, both the input and the output are one dimension. The graph is given by Figure 4.4.

We give three pairs (x, y) of the input and the output, $(0.1, 0.5)$, $(0.5, 0.5)$, and $(0.9, 0.5)$ as training data for a three-layer perceptron with standard back propagation learning algorithm. The three-layer perceptron is with one unit in the input layer, six units in the middle (hidden) layer, and one unit in the output layer.

The output of the consequent neural network after the 20,000 epochs of learning with learning constant $\alpha^0 = 0.4$ (see Equation (3.16)) is given by Figure 4.5. The horizontal line almost like $y = 0.5$ is the output of the neural network.

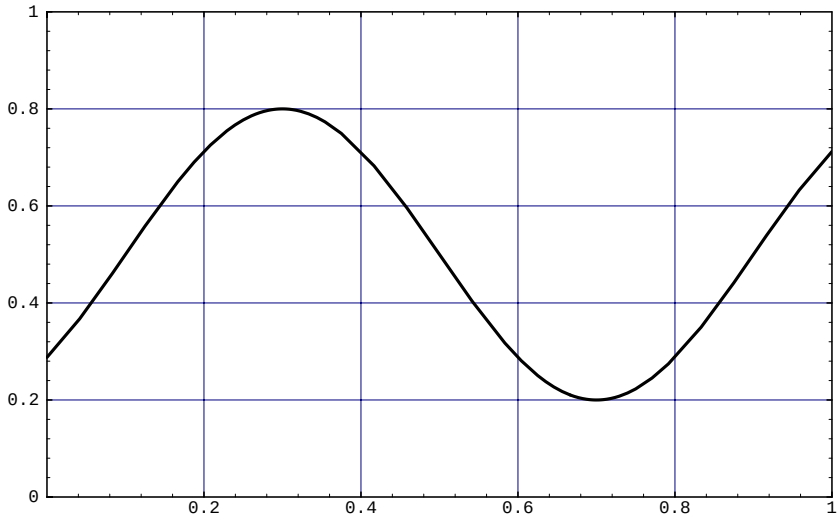

Figure 4.4: Target Sine Function as a Simple Example: $y = 0.3\sin(2\pi(x - 0.1)/0.8) + 0.5$

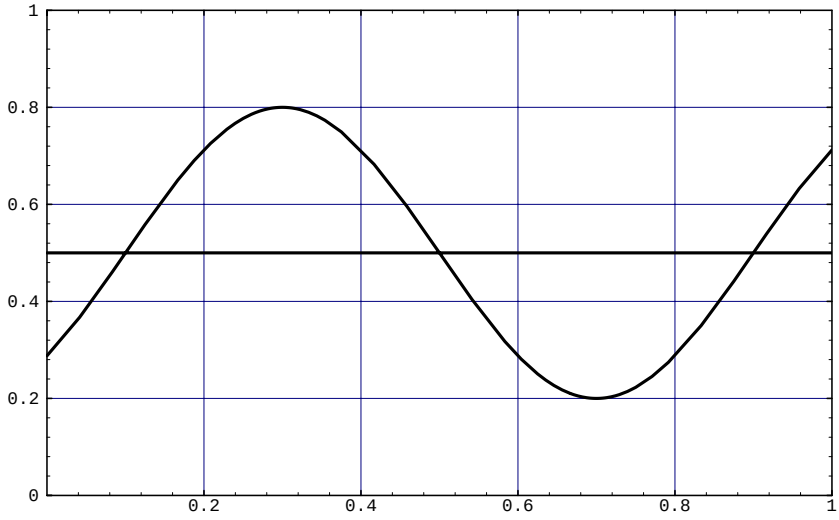

Figure 4.5: Function Learned with Standard Back Propagation: The graph of the consequent neural networks after learning with standard back propagation.

Then we let the neural network of the same architecture learn also the first order differential data on the same points in the input space. We give three tuples of (x, y, y'), $(0.1, 0.5, 2.35619)$, $(0.5, 0.5, -2.35619)$, and $(0.9, 0.5, 2.35619)$ as training data.

The output of the consequent neural network after the 20,000 epochs of learning with learning constants $\alpha^0 = 0.4$ and $\alpha^1 = 0.25$ is given by Figure 4.6. It gives a much better approximation to the target function than the previous result.

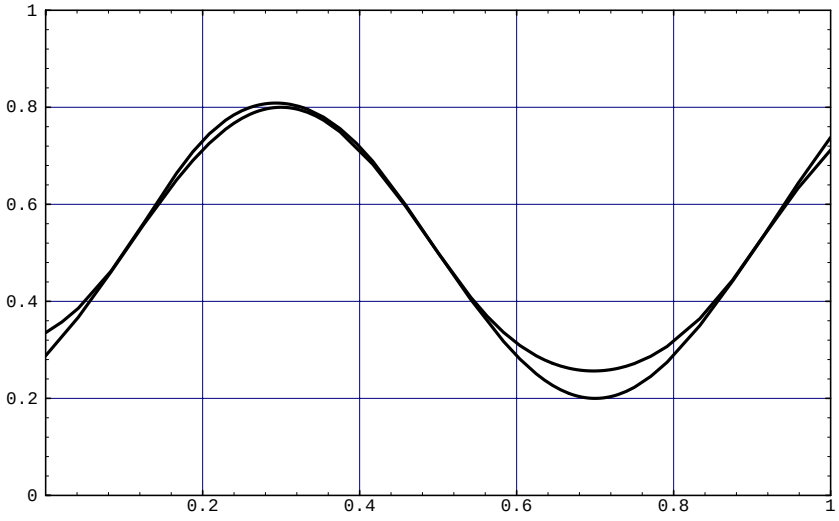

Figure 4.6: Function Learned with First Order Differential Data: The graph of the consequent neural networks after learning with first order differential data.

Though the number of points taken in the input space is the same (three) in both cases, but more data is given in the latter case for the neural network to learn. It is natural that the neural network performs better in the latter case. However it is sometimes in the real problems very difficult or costly to have more observation points to obtain training data. If that is the case and if the differential data can be obtained along with the differential data, then the learning algorithm, which can utilize the differential data, is very essential.

We show the effect of *higher order* differential data for the learning problem by taking the following quadratic function as a target function.

$$y = (x - 0.5)^2 + 0.4 \tag{4.4}$$

Also in this case, both the input and the output are one dimension. The graph of the target function is given by Figure 4.7.

We take $x = 0.5$ as a training data point. We use a three-layer perceptron with one unit in the input layer, four units in the middle (hidden) layer, and one unit in the output layer.

We give a tuple of (x, y, y'), $(0.5, 0.4, 0)$ as training data. Even with this first order differential data for use in learning, the graph of the consequent neural network is given in Figure 4.8 after 10,000 epochs with learning constants $\alpha^0 = 0.4$ and $\alpha^1 = 0.25$. The graph is overlapping with the grid line, $y = 0.4$.

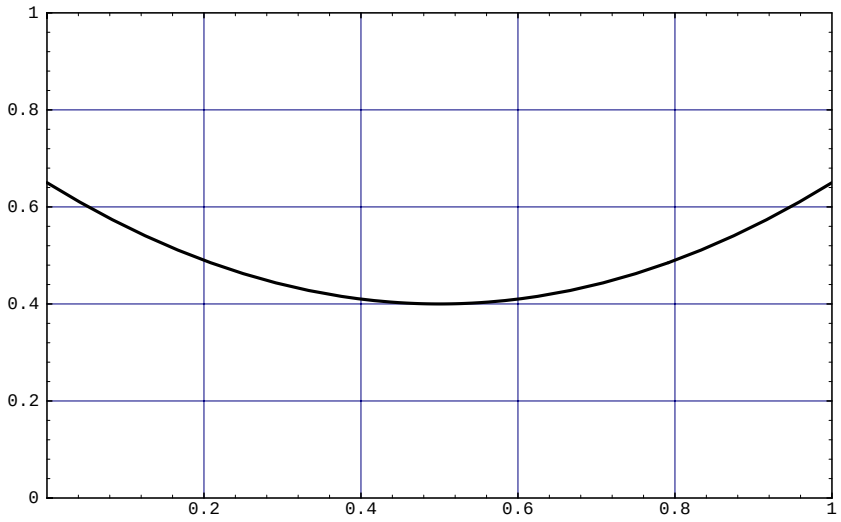

Figure 4.7: Target Function 2

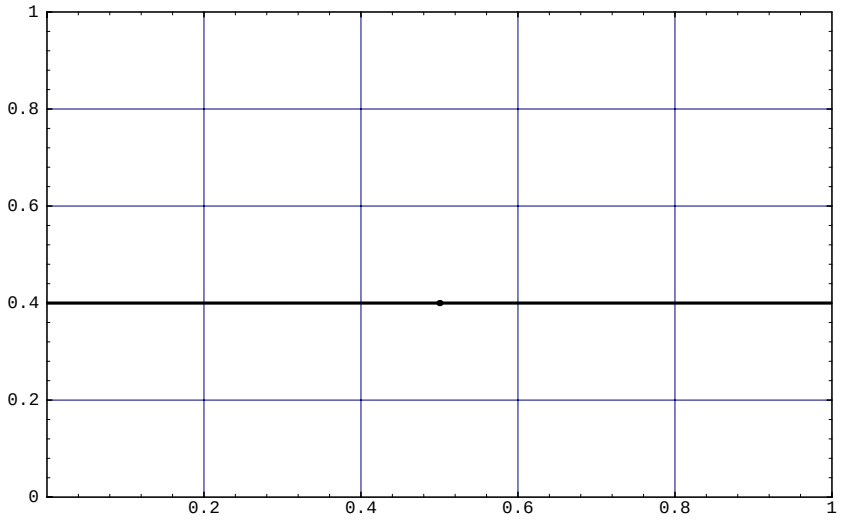

Figure 4.8: Function Learned with First Order Differential Data

But if we also give second order differential data at the training point (a tuple of (x, y, y', y''), $(0.5, 0.4, 0, 2.0)$) the consequent neural network approximates the quadratic function better and the graph is given by Figure 4.9 after 10,000 epochs with learning constants $\alpha^0 = 0.4$, $\alpha^1 = 0.5$, and $\alpha^2 = 0.25$.

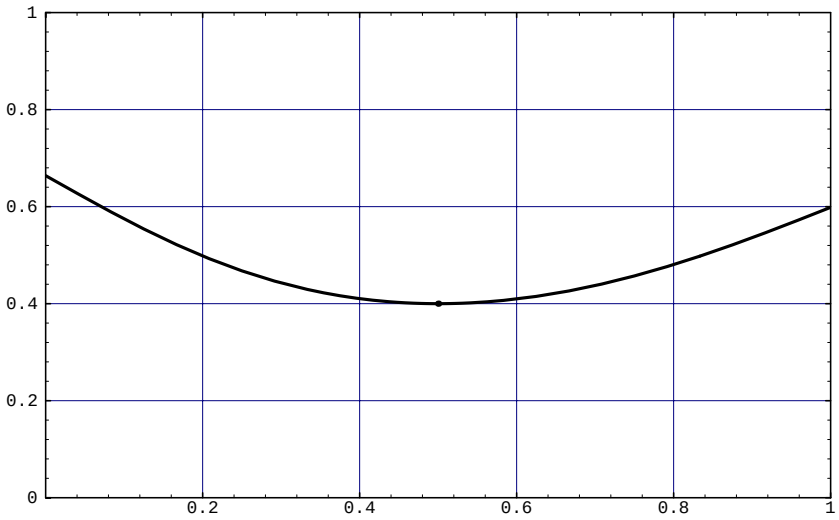

Figure 4.9: Function Learned with Second Order Differential Data

Figure 4.10 gives the graph of the neural networks learning the following target function from differential data of a tuple (x, y, y', y'', y'''), $(0.5, 0, 0, 0, 6.0)$ after 10,000 epochs with learning constants $\alpha^0 = 0.4$, $\alpha^1 = 0.5$, $\alpha^2 = 0.25$, and $\alpha^3 = 0.1$.

$$y = 2(x - 0.5)^3 + 0.5 \tag{4.5}$$

At $x = 0.5$, all the derivatives up to third order are identical between the target function and the neural network. But at the both ends, the neural network deviates from the target function. Though deviations depend on each case, we need to give some qualitative estimates on the deviations and how much data points are needed for good learning results. In this paper, we give results on preliminary experiments in Section 5.3.

4.4 Experiments on Polynomials

In this section, we apply neural networks learning differential data to learning polynomials, report results of experiments, and show some of qualitative characteristics of neural networks learning differential data.

4.4.1 Experiments on a Linear Function

In this section, we apply neural networks learning differential data to learning a linear function. The target function (Equation 4.6) is created as follows. The function is made to output 0.5 at a point randomly chosen from the interval

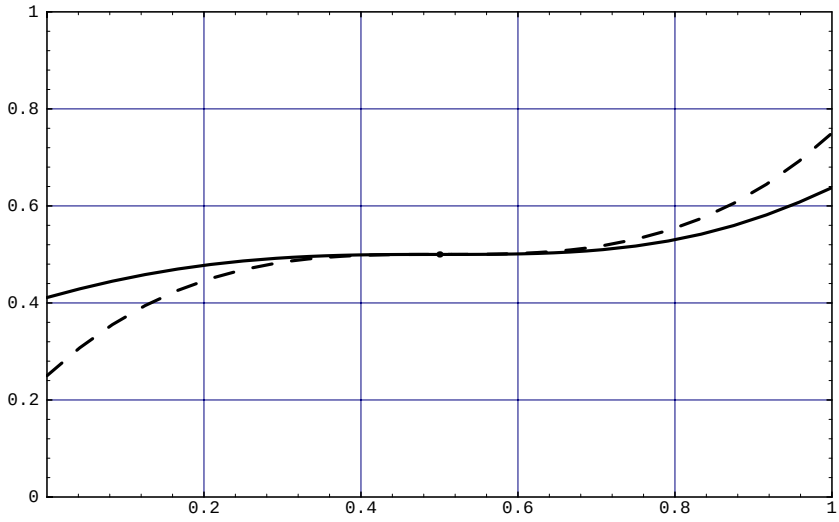

Figure 4.10: Function Learned by Third Order Differential Data: The broken line is the graph of the target function. The solid line is the graph of the consequent neural network after learning.

$[0, 1]$. Then the gradient of the function is adjusted to make the range of the function over the $[0, 1]$ to be included in $[0.2, 0.8]$. The latter step was taken to make use of more sensitive part of neural networks.

$$f(x) = 0.456113\ (x - 0.342268) + 0.5 \tag{4.6}$$

We use a three-layer perceptron with one unit in the input layer, four units in the middle (hidden) layer, and one unit in the output layer. Learning constants are set to $\{0.40, 0.50, 0.25\}$ and the momentum is set to 0.1 for all the cases in this section.

The first training point $(x, y, y', y'') = (0.748002, 0.685061, 0.456113, 0.0)$ is chosen randomly from the interval $[0, 1]$. The graphs of the neural networks with value data only, up to first order, and up to second order differential data after 10,000 epochs, are given by Figures 4.11, 4.12, and 4.13 respectively. At least in the neighborhood of the training point, the neural network with up to second order differential data performs better than other neural networks.

In addition to the first training point, the second training point $(x, y, y', y'') = (0.0593173, 0.370943, 0.456113, 0.0)$ is chosen randomly from the interval $[0, 1]$. The graphs of the neural networks with value data only, up to first order, and up to second order differential data after 10,000 epochs, are given by Figures 4.14, 4.15, and 4.16 respectively.

We cannot tell much difference this time, but all the neural networks with value data only, up to first order, and up to second order differential data have converged on the given training data.

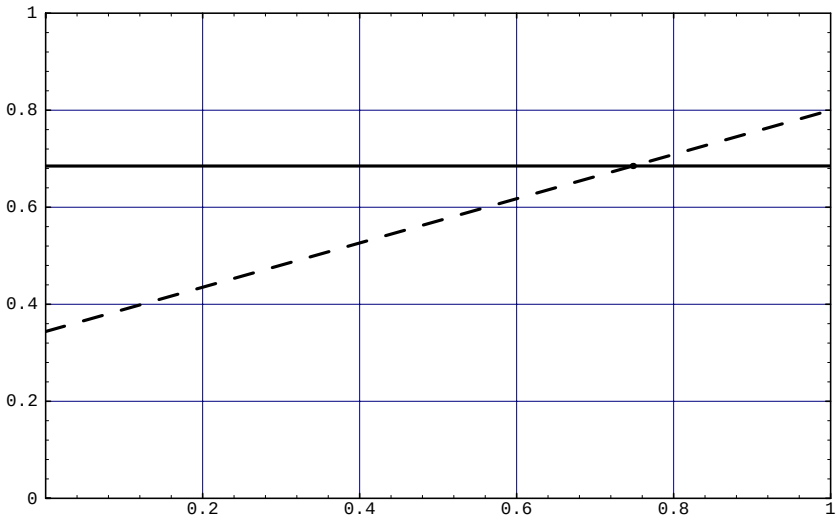

Figure 4.11: Linear Function Learned (One Point, Standard BP): A linear function, $f(x) = 0.456113\ (x - 0.342268) + 0.5$ is learned by a neural network through standard back propagation with training value data from one point. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dot represents the training point.

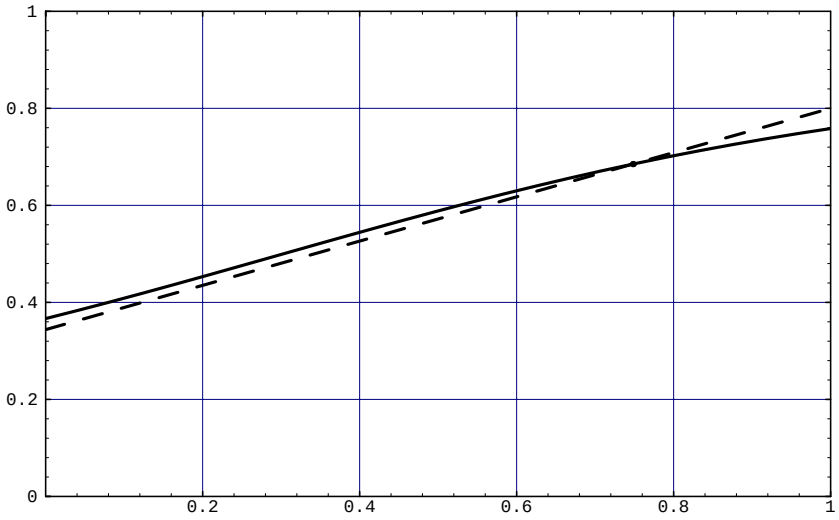

Figure 4.12: Linear Function Learned (One Point, Up to First Order): A linear function, $f(x) = 0.456113\ (x - 0.342268) + 0.5$ is learned by a neural network with training data up to first order differential data from one point. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dot represents the training point.

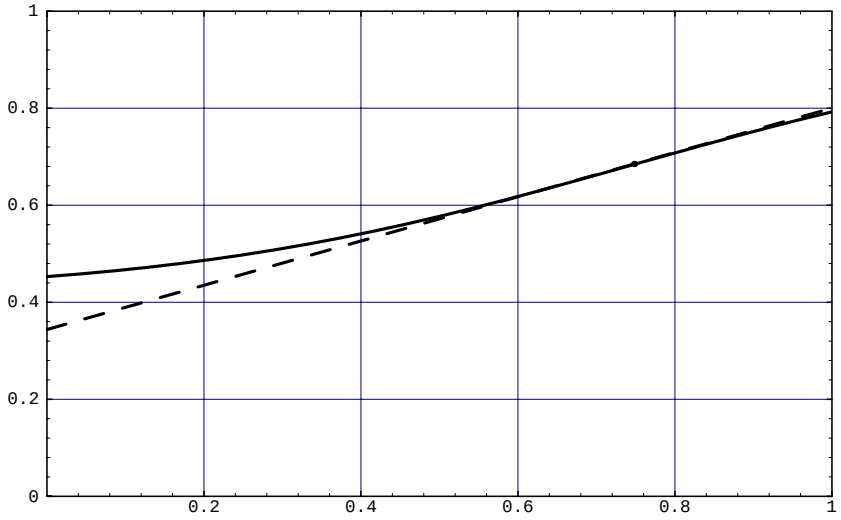

Figure 4.13: Linear Function Learned (One Point, Up to Second Order): A linear function, $f(x) = 0.456113\ (x - 0.342268) + 0.5$ is learned by a neural network with training data up to second order differential data from one point. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dot represents the training point.

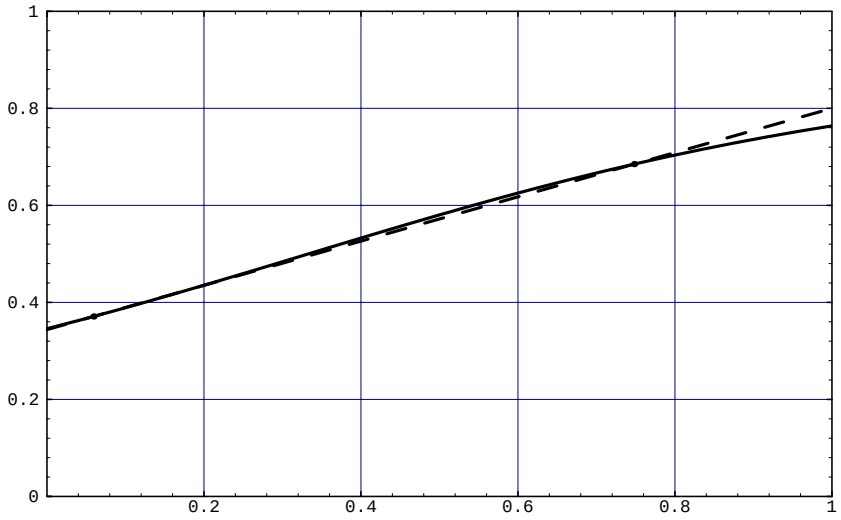

Figure 4.14: Linear Function Learned (Two Points, Standard BP): A linear function, $f(x) = 0.456113\ (x - 0.342268) + 0.5$ is learned by a neural network through standard back propagation with training value data from two points. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dots represent the training points.

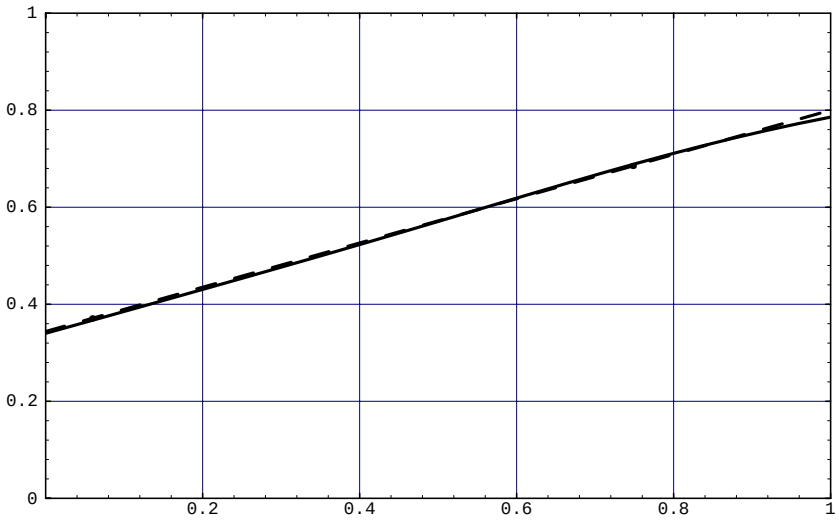

Figure 4.15: Linear Function Learned (Two Points, Up to First Order): A linear function, $f(x) = 0.456113\ (x - 0.342268) + 0.5$ is learned by a neural network with training data up to first order differential data from two points. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dots represent the training points.

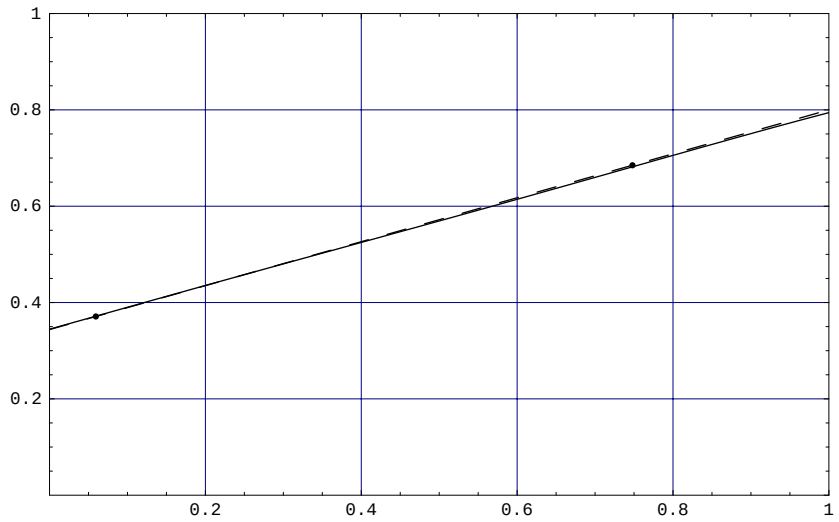

Figure 4.16: Linear Function Learned (Two Points, Up to Second Order): A linear function, $f(x) = 0.456113\ (x - 0.342268) + 0.5$ is learned by a neural network with training data up to second order differential data from two points. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dots represent the training points.

4.4.2 Experiments on a Quadratic Function

In this section, we apply neural networks learning differential data to learning a quadratic function. The target function (Equation 4.7) is created as follows. First, a function is made to output 0 at two points $x = 0.532805$, $x = 0.860492$ randomly chosen from the interval $[0, 1]$. Then other coefficient are adjusted to make the range of the function over the $[0, 1]$ to be $[0.2, 0.8]$. The latter step was taken to make use of more sensitive part of neural networks.

$$f(x) = 1.2363\ (x - 0.532805)\ (x - 0.860492) + 0.233188) \tag{4.7}$$

We use the same settings as in as in Section 4.4.1. We use three-layer perceptron with one unit in the input layer, four units in the middle (hidden) layer, and one unit in the output layer. Learning constants are set to $\{0.40, 0.50, 0.25\}$ and the momentum is set to 0.1 for all the cases in this section.

The training point $(x, y, y', y'') = (0.434126, 0.285203, -0.649113, 2.4726)$ is chosen randomly from the interval $[0, 1]$. The graphs of the neural networks with value data only, up to first order, and up to second order differential data after 10,000 epochs, are given by Figures 4.17, 4.18, and 4.19 respectively.

Also in this case, the neural network with up to second order differential data performs better than other neural networks in the neighborhood of the training point.

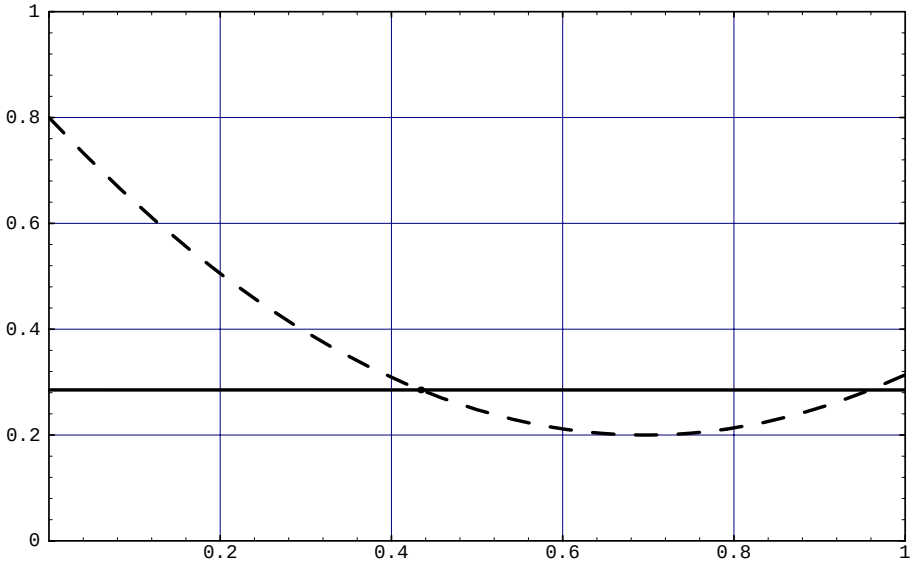

Figure 4.17: Quadratic Function Learned (One Point, Standard BP): A Quadratic function, $f(x) = 1.2363\ (x - 0.532805)\ (x - 0.860492) + 0.233188$ is learned by a neural network through standard back propagation with training value data from one point. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dot represents the training point.

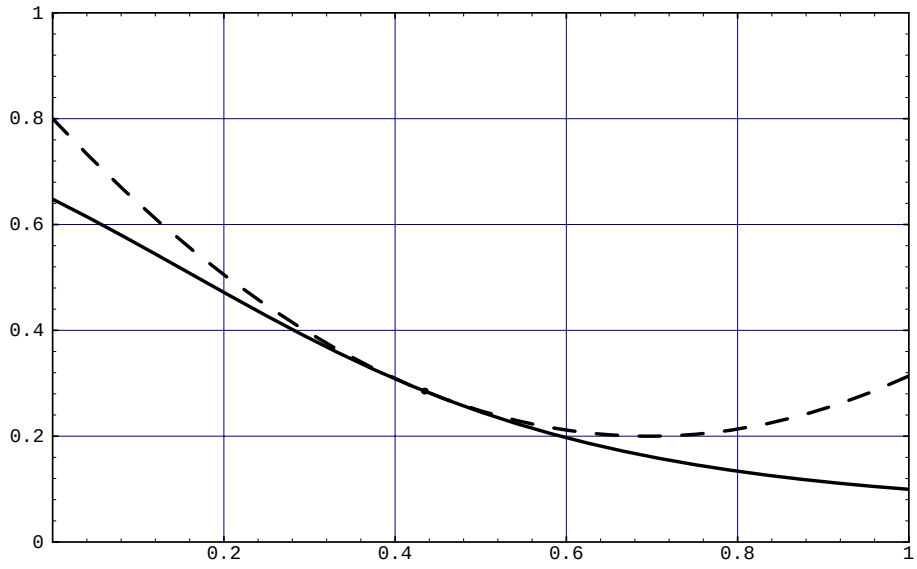

Figure 4.18: Quadratic Function Learned (One Point, Up to First Order): A Quadratic function, $f(x) = 1.2363\ (x - 0.532805)\ (x - 0.860492) + 0.233188$ is learned by a neural network with training data up to first order differential data from one point. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dot represents the training point.

4.5 Experiments on a Sine Function

In this section, we apply neural networks learning differential data to learning a sine function, report results of experiments, and show some of qualitative characteristics and quantitative data of neural networks learning differential data.

The task is essentially more difficult than the task in Section 4.4 since any order derivative of the function is non-constant. This section also provides results on neural networks learning up to third order differential data.

We use the following sine function as a target function, which ranges $[0.1,\ 0.9]$ for the domain of $[0.0,\ 1.0]$. (See figure 4.20.)

$$f(x) \quad = \quad 0.4\ \sin(2\pi x) + 0.5 \tag{4.8}$$

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.

Case 1 (Standard BP with three data points): We trained the network with the standard back propagation algorithm. Learning constants is 1.0 for

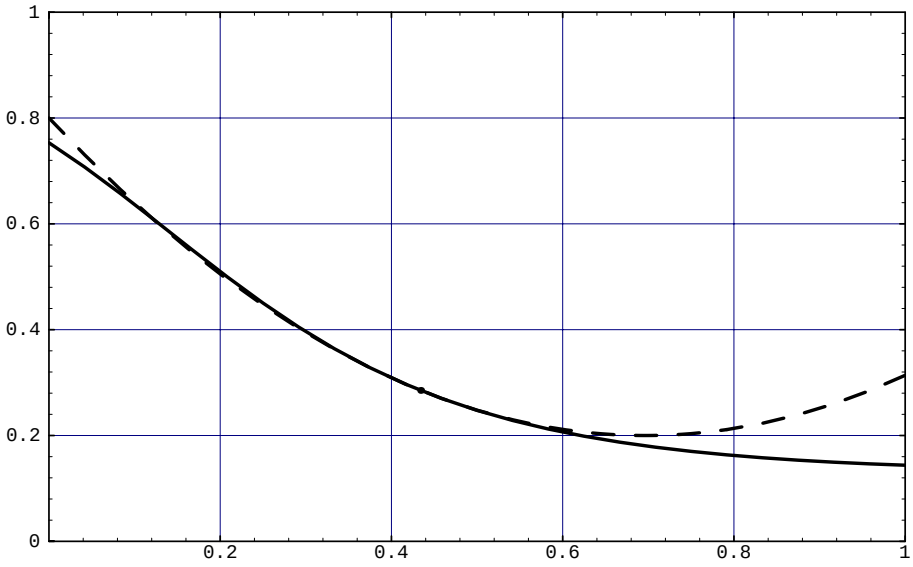

Figure 4.19: Quadratic Function Learned (One Point, Up to Second Order): A Quadratic function, $f(x) = 1.2363\ (x - 0.532805)\ (x - 0.860492) + 0.233188$ is learned by a neural network with training data up to second order differential data from one point. The broken line is the graph of the target function. The solid line is the graph of the neural network. The dot represents the training point.

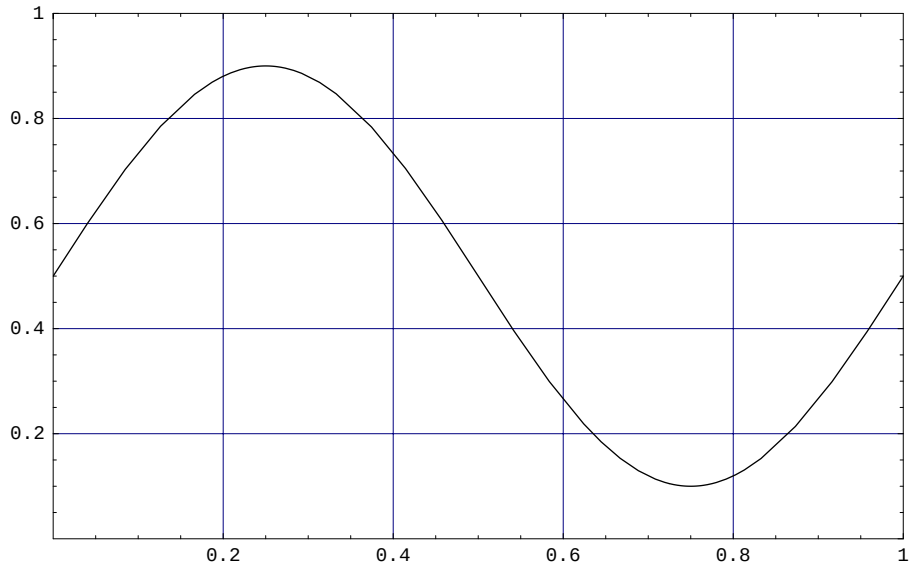

Figure 4.20: Target sine function: $f(x) = 0.4\ \sin(2\pi x) + 0.5$

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.

Case 2 (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 10,000 learning epochs on the value data from the nine training data points.

Case 3 (Up to first 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 and 0.01 for value data and first order differential data, respectively. Three training data points were randomly selected from the interval $[0, 1]$. The neural networks are trained for the 200,000 learning epochs on the training data up to first order from the three training data points.

Case 4 (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.

Case 5 (Up to third 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, 0.001, and 0.0000001 for value data, first order differential data, second order differential data and third order differential data, respectively. Three training data points were randomly selected from the interval $[0, 1]$. The neural networks are trained for the 3,000,000 learning epochs on the training data up to third order from the three training data points.

We carry out the experiment in case 1 to see how well the standard back propagation algorithm performs if the number of data points is the same as in case 3.

We carry out the experiment in case 2 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 4. 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 the case 4 than the standard back propagation cases. Therefore nine data points are selected for the case 2.

Figures 4.21, 4.22, 4.23, 4.24, and 4.25 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 cases 1 than cases 2 and case 3 on the training data points.

The table 4.1 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

	$\| \|_{2,0}$	$\| \|_{2,1}$	$\| \|_{2,2}$	$\| \|_{2,3}$
Case 1	0.1714	1.473	8.780	74.25
Case 2	0.04478	1.007	14.59	260.6
Case 3	0.1085	0.9613	8.712	130.3
Case 4	0.07305	0.8281	12.45	293.3
Case 5	0.04232	0.4039	3.575	74.88

Table 4.1: $\| \|_{2,0}$, $\| \|_{2,1}$, $\| \|_{2,2}$, and $\| \|_{2,3}$ distances: $\| \|_{2,0}$, $\| \|_{2,1}$, $\| \|_{2,2}$, and $\| \|_{2,2}$ distances on the interval $[0, 1]$ between the target function and the trained neural network are given for each case.

the table for each case. These distances are given by the following equations where $n(x)$ is the function defined by the neural network.

$$\|f - n\|_{2,l} = \left\{ \int_0^1 \left(\frac{d^l f(x)}{dx^l} - \frac{d^l n(x)}{dx^l} \right)^2 \right\}^{1/2} \tag{4.9}$$

Figure 4.26 gives learning curves for all the cases. The graphs show the changes of errors for cases 1, 2, 3, 4, and 5 from the top respectively.

We describe several observations obtained from those experiments.

First of all, the neural networks in case 3, 4, and 5 all 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 Table 4.1, the $\| \|_{2,0}$ distance from the target function of the neural network in case 4 happens to be smaller than the one in case 1. 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.

On the other hand, the $\| \|_{2,2}$ distance from the target function of the neural network in case 4 happens to be larger than the one in case 1. This was quite unexpected, but the situation might change when we add more data points.

Even though there are such several reversals in Table 4.1, overall performance gets better with higher order differential data.

Another unexpected observation is that it took fairly large learning epochs for case 2, which are comparable to case 4. We expected to have large learning epochs for case 4 since learning differential data will need subtle adjustments, but we did not expect for case 2 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. [5]

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

[5]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.

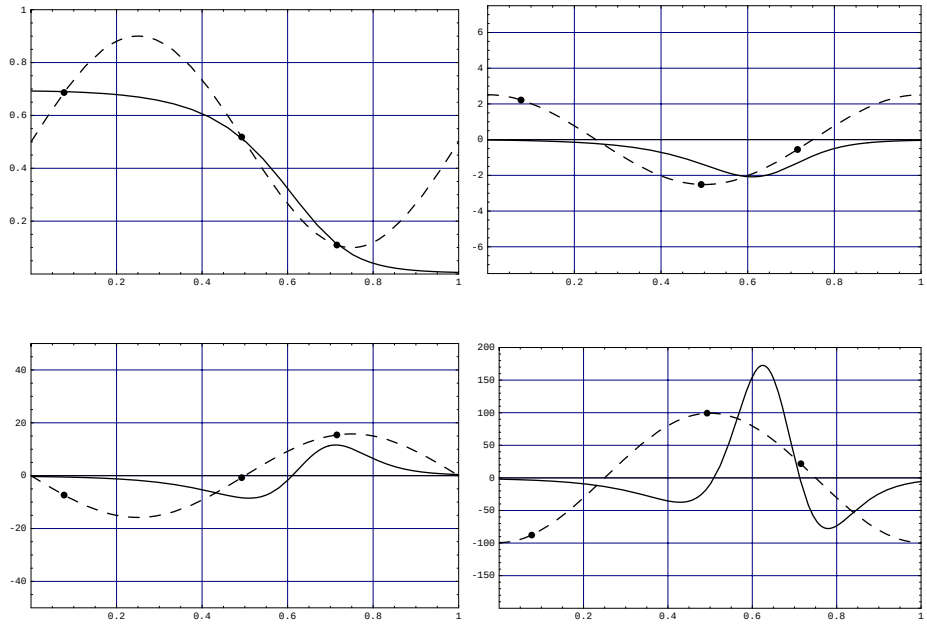

Figure 4.21: [Case 1] *Outputs of the trained network:* The graphs show the value, first order, second order, and third order outputs of the neural network in case 1 from the top respectively. The solid lines show the outputs of the trained neural network in case 1 after 10,000 learning epochs. The broken line is the output of the target function. The dots represent the three training data points.

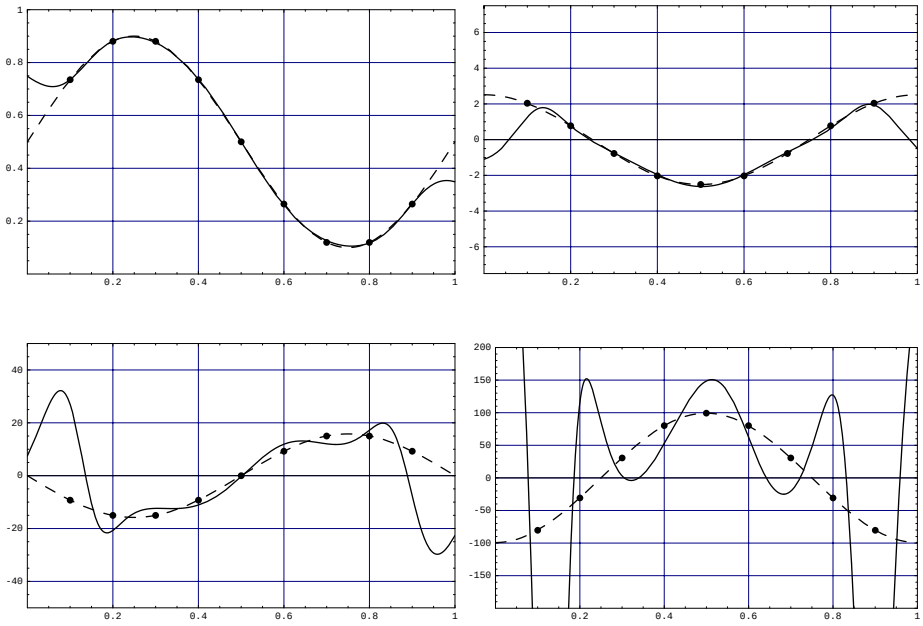

Figure 4.22: [Case 2] *Outputs of the trained network:* The graphs show the value, first order, second order, and third order outputs of the neural network in case 2 from the top respectively . The solid lines show the outputs of the trained neural network in case 2 after 1,000,000 learning epochs. The broken line is the output of the target function. The dots represent the nine training data points.

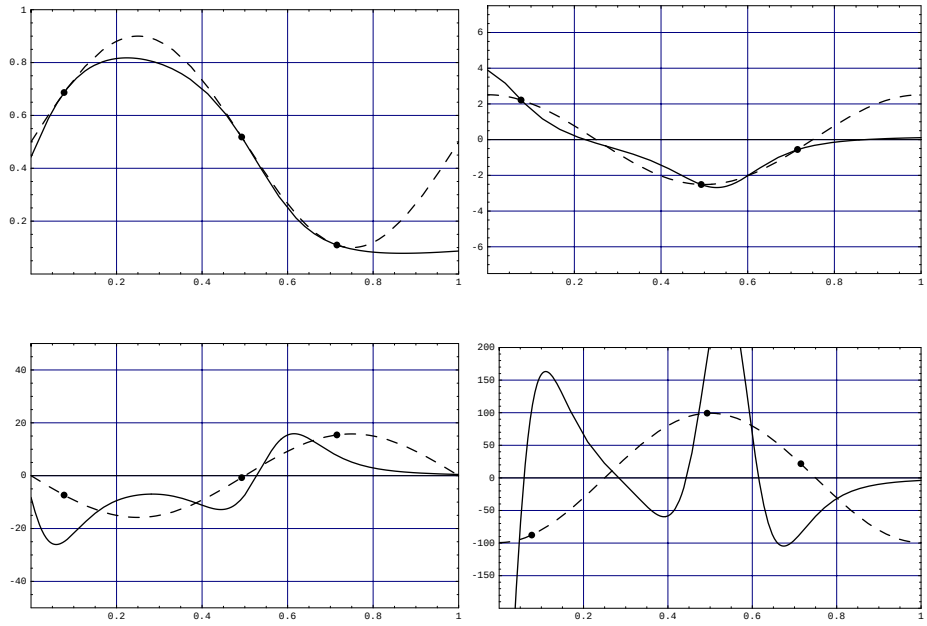

Figure 4.23: [Case 3] *Outputs of the trained network:* The graphs show the value, first order, second order, and third order outputs of the neural network in case 3 from the top respectively . The solid lines show the outputs of the trained neural network in case 3 after 200,000 learning epochs. The broken line is the output of the target function. The dots represent the three training data points.

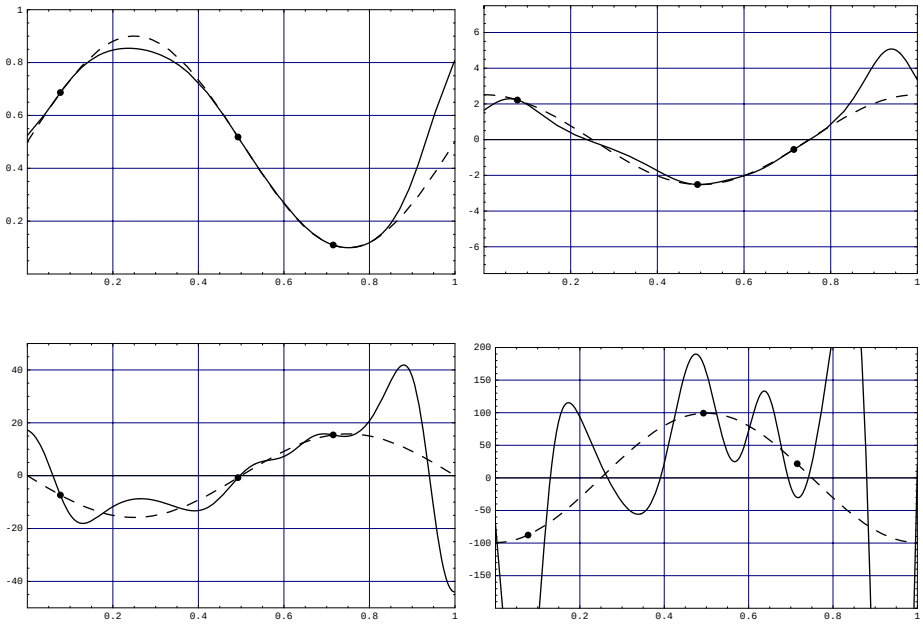

Figure 4.24: [Case 4] *Outputs of the trained network:* The graphs show the value, first order, second order, and third order outputs of the neural network in case 4 from the top respectively . The solid lines show the outputs of the trained neural network in case 4 after 1,000,000 learning epochs. The broken line is the output of the target function. The dots represent the three training data points.

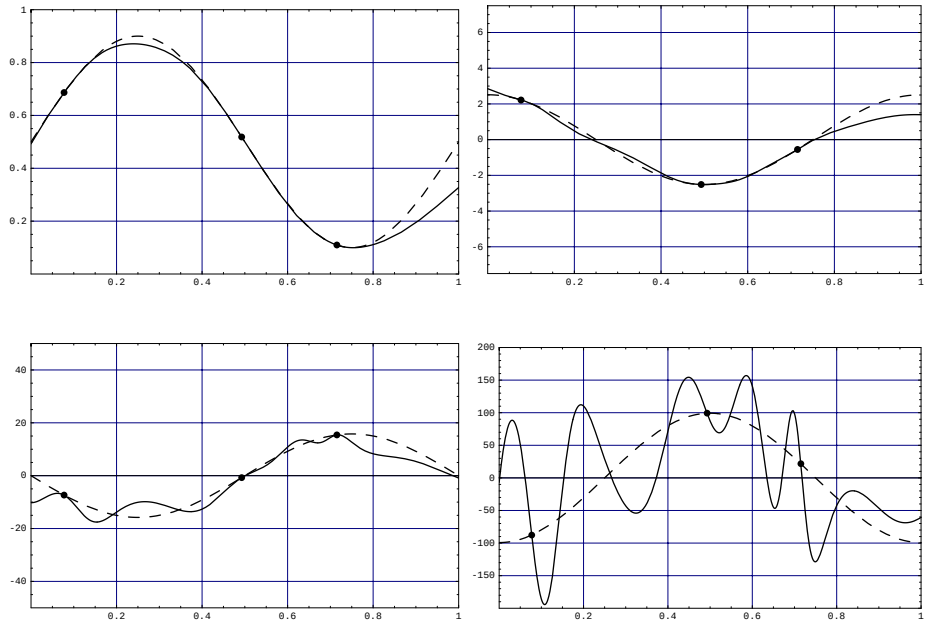

Figure 4.25: [Case 5] *Outputs of the trained network:* The graphs show the value, first order, second order, and third order outputs of the neural network in case 5 from the top respectively . The solid lines show the outputs of the trained neural network in case 5 after 3,000,000 learning epochs. The broken line is the output of the target function. The dots represent the three training data points.

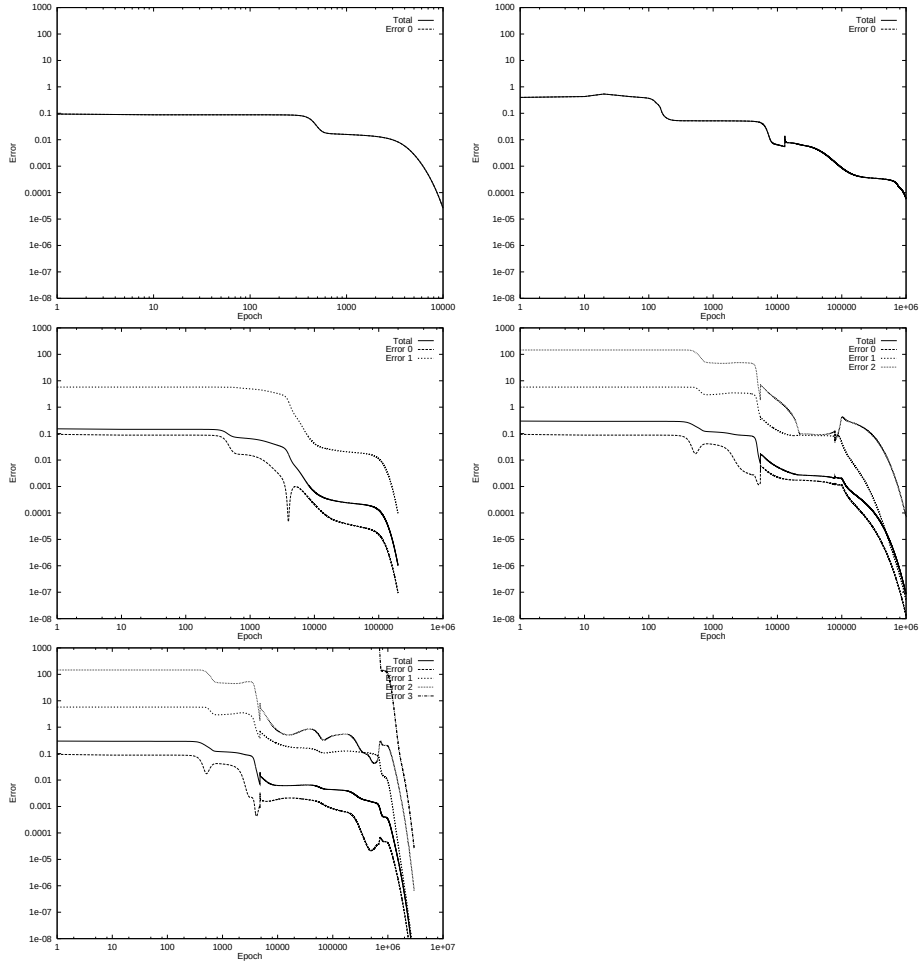

Figure 4.26: *Learning curves:* The graphs show the changes of errors for cases 1, 2, 3, 4, and 5 from the top respectively. The axes of the graphs have logarithmic scaling. "Total", "Error 0", "Error 1", "Error 2" and "Error 3" stand for E in Equation (3.16), value, first order, second order, and third order errors (E^δs in Equation (3.17)) respectively. For cases 1 and 2, "Total" and "Error0" are the same because the learning constant for the cases is 1.0. Case 1, 2, 3, 4, and 5 have learning epochs of 10,000, 1,000,000, 200,000, 1,000,000, and 3,000,000 respectively.

Chapter 5

Analyses

In this chapter, we present analyses of neural networks learning differential data mainly in the first order cases. Analyses include comparison with extra pattern scheme, sample complexity, effect of irrelevant features, and noise robustness.

What is a fair comparison between standard back propagation and learning differential data? There are many cases where one can get differential data along with value data at a very small cost, such as Simard's case of pattern recognition [36]. 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 differential data just additional information? Is differential data not different from, for example, adding new value patterns? In this chapter, we tackle these questions from many viewpoints.

We call the first derivatives of functions *slopes* for short in this chapter. We also call the following type of learning, EBNN learning. In EBNN learning, we let the neural network learn all the axial first derivatives along with the value data of the target function and we use the same learning constant for all the first derivatives. EBNN learning is named after Explanation-Based Neural Network [25] where this type of learning is used. Most of the results in this chapter are on this type of learning.

The experiments for the analyses in Sections 5.2, 5.3, 5.4, and 5.5 are conducted with the implementation specialized for EBNN learning, a prior implementation to the one described in Chapter 4.

5.1 Comparison with Extra Pattern Scheme

In this section, we attempt a comparison between learning differential data and learning extra value patterns. First, we give a mathematical formalization of the problem as a foundation for comparison. Then through the arguments of a simple case and a general case, we show that adding differential error function gives good characteristics to the set of global minima and some advantages over adding additional value error function for new value patterns. [1]

Let $f_t : U \subset R^I \rightarrow R^m$ be the target function that has to be learned. Let the neural network in consideration have s parameters (that are weights and biases).

[1] In this section we 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.

For $w \in R^s = W$, 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 [2]

$$ge(f) = \int_U |f_t(p) - f(p)|^2 \, dp \tag{5.1}$$

The strategy of standard back propagation 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 neural networks learning differential data is to use the value *and differential* data 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:

$$e_{\rm v}(w) = e_{\rm v}(f_n(w)) = \sum_{i=1}^{r} |f_t(p_i) - f_n(w)(p_i)|^2 \tag{5.2}$$

$$e_{\rm s}(w) = e_{\rm s}(f_n(w)) = \sum_{i=1}^{r} \sum_{i=1}^{I} \left| \frac{\partial f_t}{\partial x_i}(p_i) - \frac{\partial f_n(w)}{\partial x_i}(p_i) \right|^2 \tag{5.3}$$

$$Min(ge) = \{w \in W \,|\, ge(w) \text{ is minimum}\} \tag{5.4}$$

$$Min(e_{\rm v}) = \{w \in W \,|\, e_{\rm v}(w) \text{ is minimum}\} \tag{5.5}$$

$$Min(e_{\rm s}) = \{w \in W \,|\, e_{\rm s}(w) \text{ is minimum}\} \tag{5.6}$$

$$Min(e_{\rm v} + e_{\rm s}) = \{w \in W \,|\, e_{\rm v}(w) + e_{\rm s}(w) \text{ is minimum}\} \tag{5.7}$$

where I stands for the number of units in the input layer and subscript s of $e_{\rm s}$ stands for slope (i.e. first derivatives).

If we take value data from new points near the original points, we might 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 neural networks learning differential data is more expensive than that of standard back propagation, it is in a sense fair to compare learning differential data against standard back propagation with more training patterns. We call this method the *extra pattern scheme*, in which we apply the standard back propagation with nearby patterns of the original patterns added

[2] 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 range 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.

to the pattern set. In the following sections we see the qualitative difference between learning differential data 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 back propagation and the same information as learning differential data. 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 applies for comparison between learning differential data and synthesized pattern scheme as one between learning differential data and extra pattern scheme, and we limit our comparison to extra pattern scheme.

5.1.1 A Simple Case

In this section, by using a simple example, we now intuitively demonstrate that learning differential data and the *extra pattern scheme* are qualitatively different.

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 $(w, b) = (1, 0)$. Without loss of generality we pick $x = 0.5$ as a training example. In this case, $e_{\mathrm{v}}(w, b)$ has a surface like one in Figure 5.1. The curve on the surface is the projection of $Min(e_{\mathrm{v}}) = \{(w, b) \,|\, 0.5(w - 1) - b = 0\}$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. In Figure 5.2 we plot the surface of $e_{\mathrm{s}}(w, b)$. 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 $(w, b) = (1, 0)$. With either e_{v} or e_{s} only, there is no reason why the gradient descent method prefers one point to another on the corresponding curve. But if we add e_{v} and e_{s} together, then the surface looks like one in Figure 5.3. The curves on the surface are the projections of $Min(e_{\mathrm{v}})$ and $Min(e_{\mathrm{s}})$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. In this case, $Min(e_{\mathrm{v}} + e_{\mathrm{s}}) = Min(ge) = \{(1, 0)\}$. In order to approximate the function correctly using the value error only, we need two different points as examples. Using $e_{\mathrm{v}} + e_{\mathrm{s}}$ 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 $e_{\mathrm{v}} + e_{\mathrm{s}}$ instead of e_{v}. Let us see what happens if we take e'_{v} with respect to a point in the neighborhood of the point $x = 0.5$ instead of e_{s} and let us see what is the qualitative difference between these two. The surface of $e_{\mathrm{v}} + e'_{\mathrm{v}}$ with e_{v} for $x = 0.5$ and e'_{v} for $x = 0.6$ looks like one in Figure 5.4. Two lines on the surface is the projection of $Min(e_{\mathrm{v}})$ and $Min(e'_{\mathrm{v}})$ on the surface. Two lines are almost identical and the error surface is almost identical to Figure 5.1 except that the scale on Error axis has doubled. The surface of Figure 5.4 is not much different from the surface of Figure 5.1 qualitatively.

On the other hand in Figure 5.3, two curves intersect at a much larger angle, even though two curves are constructed from information on *one* point. And the surface of Figure 5.3 is quite different from Figure 5.1.

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

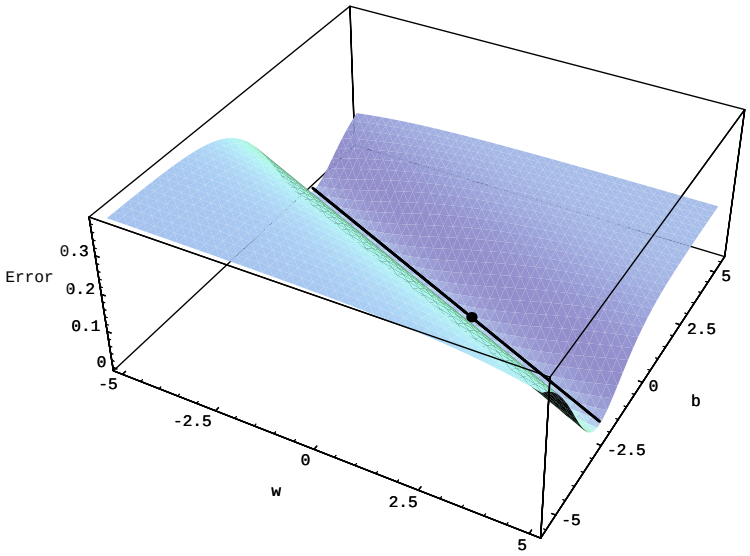

Figure 5.1: **Value error:** $e_{\mathrm{v}}(w,b)$ is plotted. The curve on the surface is the projection of $Min(e_{\mathrm{v}}) = \{(w,b) \,|\, 0.5(w-1)-b=0\}$ onto the surface. The point on the surface is the projection of $(w,b) = (1,0)$.

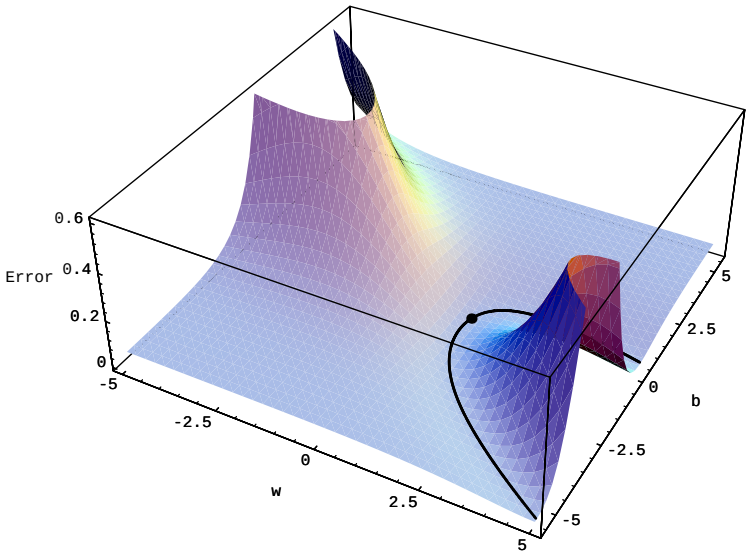

Figure 5.2: **Slope error:** $e_{\mathrm{v}}(w,b)$ is plotted. The curve on the surface is the projection of $Min(e_{\mathrm{s}})$ onto the surface. The point on the surface is the projection of $(w,b) = (1,0)$.

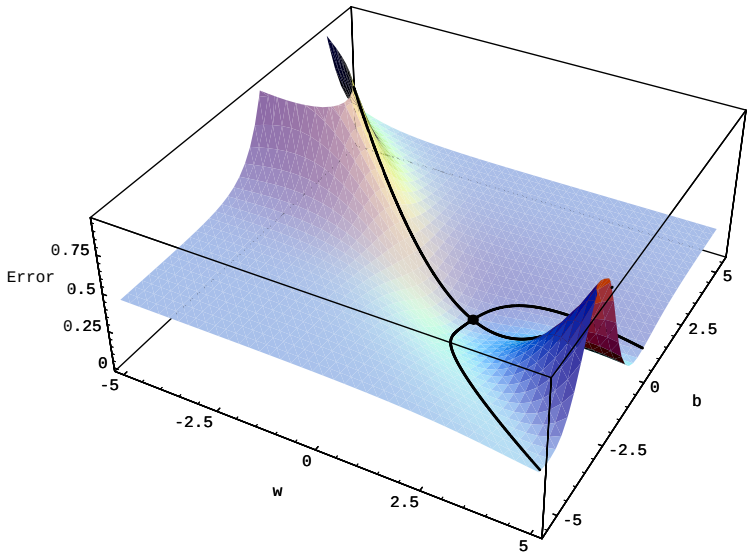

Figure 5.3: **Value error and slope error combined:** $e_{\mathrm{v}} + e_{\mathrm{s}}$ is plotted. The curves on the surface are the projections of $Min(e_{\mathrm{v}})$ and $Min(e_{\mathrm{s}})$ onto the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. Note that two curves intersect at a much larger angle than the curves in Figure 5.4.

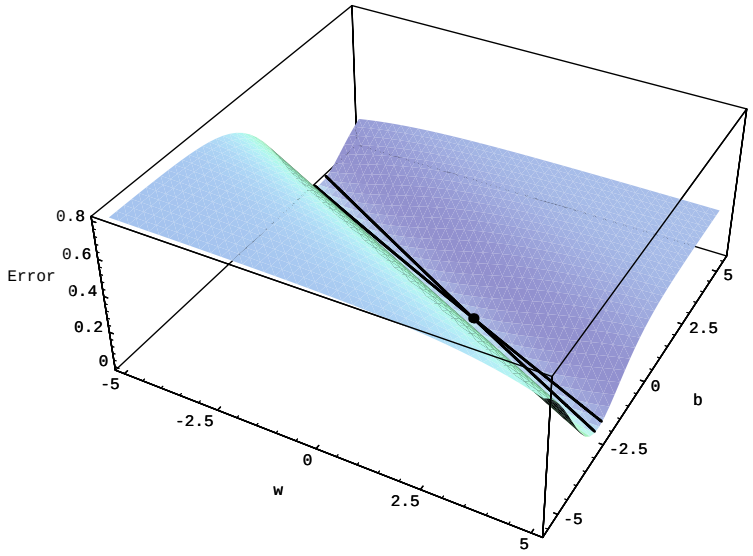

Figure 5.4: **Value errors on two points:** $e_{\mathrm{v}} + e'_{\mathrm{v}}$ is plotted. Two lines on the surface is the projection of $Min(e_{\mathrm{v}})$ and $Min(e'_{\mathrm{v}})$ on the surface. The point on the surface is the projection of $(w, b) = (1, 0)$. Note that these two lines are almost identical.

that learning differential data is much more noise-tolerant than extra pattern scheme.

5.1.2 A General Case

In this section, we extend the analysis in the previous section to a more general case with reasonable conditions. We show even in a general case that the additional value error function of a nearby point of the original point is less noise-tolerant 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 moving p_0 a little. Let us take a very short line $\{p(t) = p_0 + t\ \Delta p = (p^1_0 + t\ \Delta p^1, ..., p^I_0 + t\ \Delta p^I) \mid t \in [0, 1]\}$ in the input space so that $p(0) = p_0$. We notate $\overline{e_{\mathrm{v}}}(t)$ for the value error for the single point $p(t)$.

Here are two assumptions we make.

1. $Min(e^0_{\mathrm{v}})$, $Min(e^0_{\mathrm{s}})$, and $Min(\overline{e_{\mathrm{v}}}(1))$ have at least one point in common. That is:

$$Min(e^0_{\mathrm{v}}) \cap Min(e^0_{\mathrm{s}}) \cap Min(\overline{e_{\mathrm{v}}}(1)) \ni \exists w_0 \tag{5.8}$$

2. $(\nabla_w f_n)(w_0, p_0)$, $(\nabla_w f_n)(w_0, p(1))$, 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.

Assumption 1 is not an unreasonable one since $Min(e_{\rm v}^0)$, $Min(e_{\rm s}^0)$, and $Min(\overline{e_{\rm v}}(1))$ are generally hyperplanes ($n-1$-dimensional manifolds) in the weight space W where n is the dimension of W and where W is high-dimensional enough. If the neural network approximates value and differential data up to the first order at p_0 and value data at $p(1)$ of the target function f_t correctly, then $Min(e_{\rm v}^0) \cap Min(e_{\rm s}^0) \cap Min(\overline{e_{\rm v}}(1))$ is not an empty set.

The normals at w_0 to the surfaces $Min(e_{\rm v}^0)$, $Min(e_{\rm s}^0)$, and $Min(\overline{e_{\rm v}}(1))$ are $(\nabla_w f_n)(w_0, p_0)$, $(\nabla_w f_n)(w_0, p(1))$, and $\left(\nabla_w \frac{\partial f_n}{\partial x_i}\right)(w_0, p_0)$'s respectively.

We see how $Min(\overline{e_{\rm v}}(t))$ changes from $t = 0$ to $t = 1$. In order to do so, we see the difference of the normals to $Min(\overline{e_{\rm v}}(t))$ at $t = 0$ and $t = 1$. By expanding $(\nabla_w f_n)(w_0, p(t))$ around $t = 0$, we obtain the following.

$$\begin{aligned}\frac{\partial}{\partial t}(\nabla_w f_n)(w_0, p(1)) &= \frac{\partial}{\partial t}(\nabla_w f_n)(w_0, p_0) + \sum_i \left(\nabla_w \frac{\partial f_n}{\partial x_i}\right)(w_0, p_0)\, \Delta p^i \\ &\quad + O(|\Delta p|^2) \end{aligned} \tag{5.9}$$

So the difference between the normals to $Min(e_{\rm v}^0)$ and $Min(\overline{e_{\rm v}}(1))$ is approximately a linear combination of normals to $Min(e_{\rm s}^0)$ at w_0. (See Figure 5.5.) That means that $Min(\overline{e_{\rm v}}(1)))$ 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 small amount 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 is efficient. Using $e_{\rm s}^0$ has similar to but more efficient and robust effects on the structure of the minimum set of the error function than using the value errors of points near p_0.

5.2 How Learnings Work

In this section, we pick up one function approximation task and try standard back propagation and EBNN learning to see how learnings work in detail.

We use the following nonlinear function for a learning problem. (See Figure 5.6 for the surface plot of the target function.)

$$g(x, y) = 0.5 e^{-10((x-0.2)^2 + (y-0.5)^2)} + 0.25 \sin(8xy) + 0.25 \tag{5.10}$$

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 $g(x, y)$).

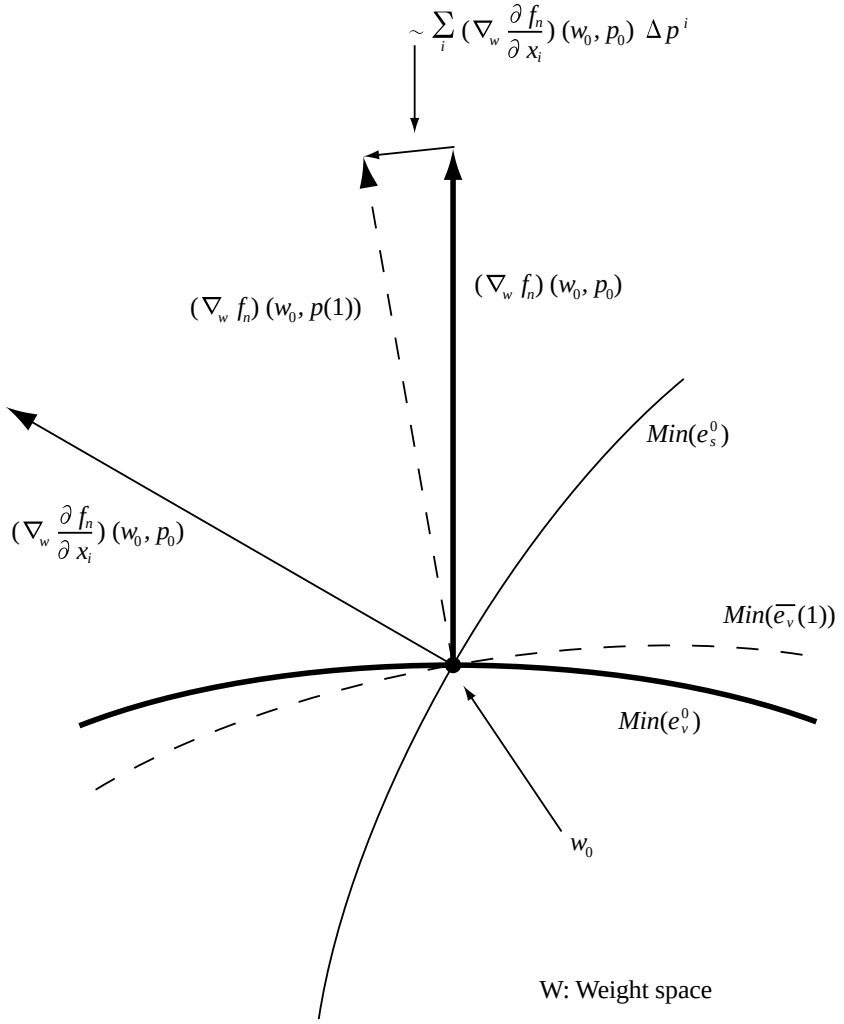

Figure 5.5: **General case:** The figure shows the normal vectors (shown as arrows) to the surfaces $Min(e_{\mathrm{v}}^0)$, $Min(e_{\mathrm{s}}^0)$, and $Min(\overline{e_{\mathrm{v}}}(1))$. The difference between the normal vectors to two surfaces $Min(e_{\mathrm{v}}^0)$ and $Min(\overline{e_{\mathrm{v}}}(1))$ is approximately a linear combination of the normals to $Min(e_{\mathrm{s}}^0)$.

The network is fully connected between layers, and each unit has a bias, except for input units.

One pattern for standard back propagation consists of the input value, x and y, and the desired value output, $g(x, y)$, for the output unit. One pattern for EBNN learning consists of the input value, x and y, the desired value output, $g(x, y)$, for the output unit of the value net and the desired outputs for δ nets, $\frac{\partial g}{\partial x}(x, y)$ and $\frac{\partial g}{\partial y}(x, y)$.

We use the learning constants 0.5 for standard back propagation, and 0.444 for value data, 0.0278 for EBNN error (the sum of squared errors of all axial first derivatives) for EBNN learning.[3] We made the network learn for 10,000 epochs. We use ten patterns. The input value of first pattern is taken to be $(0.5, 0.5)$. The input values of the remaining nine patterns were taken from the uniform distribution on $[0, 1] \times [0, 1]$. In Figure 5.6, the points on the surface are the projections of the input values of the ten patterns onto the surface. The lines coming out from the points are the normals to the surface.[4]

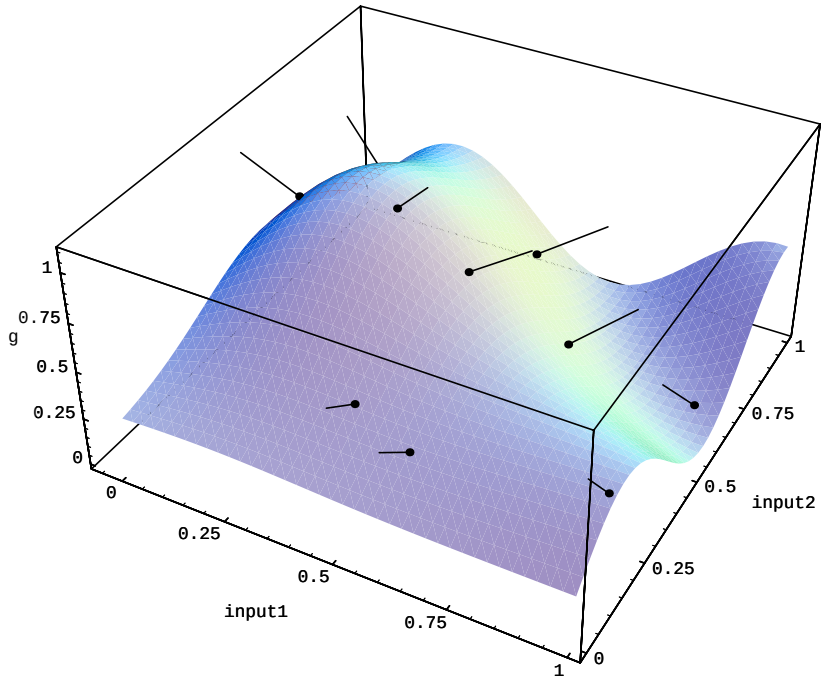

Figure 5.6: **Surface of the target function:** The surface of the target function is shown. The points on the surface are the projections of input values onto the surface. The lines coming out from the points are the unit normals to the surface.

[3]The choice of learning constants for EBNN learning is made empirically. First 0.5 for back propagation is divided into the ratio of $8 : 1$ for the value net and δ nets. Then the learning constant for δ nets is calculated by dividing the quotient by the number of δ nets (in this case 2).

[4]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.

The graph of the value error E^0 (see Equation (3.17)) for standard back propagation is given by Figure 5.7. The surface learned by learning values only is given by Figure 5.8. The broken lines are the normals to the surface of target function and the solid lines are the normals to this surface learned by back propagation. Note that these normals are quite different from each other.

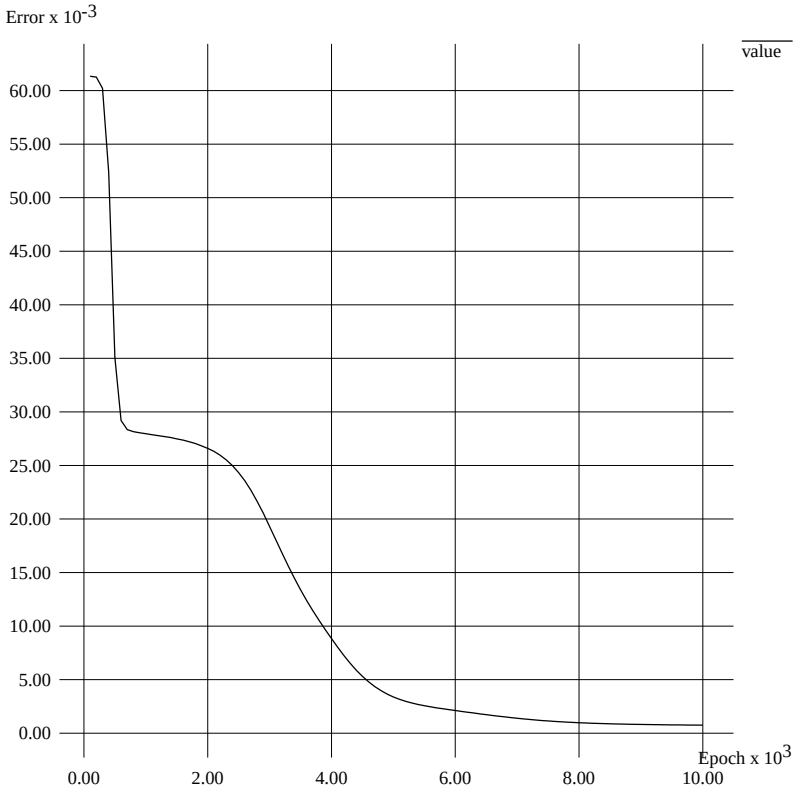

Figure 5.7: **Error curve of back propagation:** The change of the value error E^0 is plotted against the number of epochs.

The graphs of the total error E (, which learning constants are involved in the calculation. See Equation (3.16),) for EBNN learning, its value error part, and its slope error part are given in Figure 5.9. The surface learned by EBNN learning given by Figure 5.10. The broken 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 broken lines are hidden by the 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* after Section 5.1. It is 0.0102 for the function learned by back propagation and it is 0.0063 for

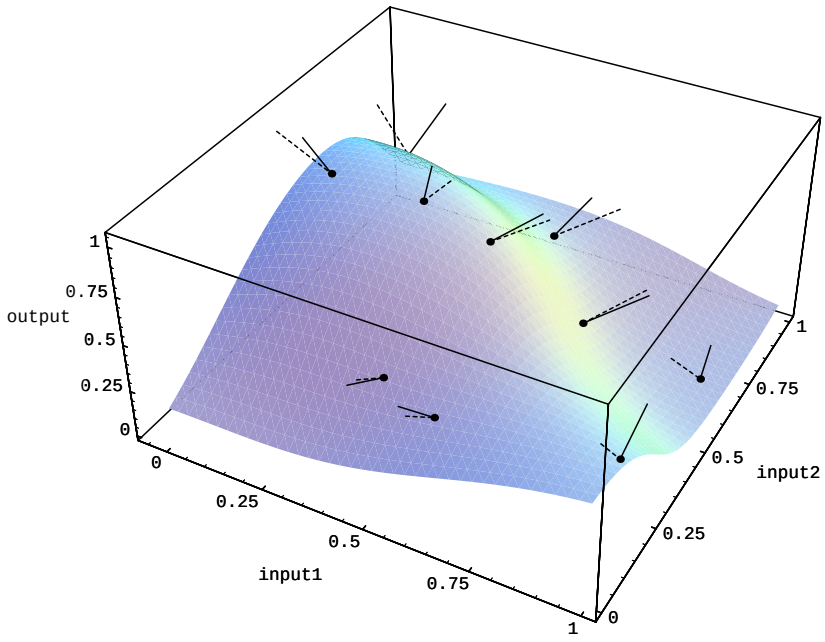

Figure 5.8: **Surface learned by back propagation:** The surface learned by back propagation is shown. The broken lines are the normals to the surface of the target function and the solid lines are the normals to this surface learned by back propagation. Note that these normals are quite different from each other.

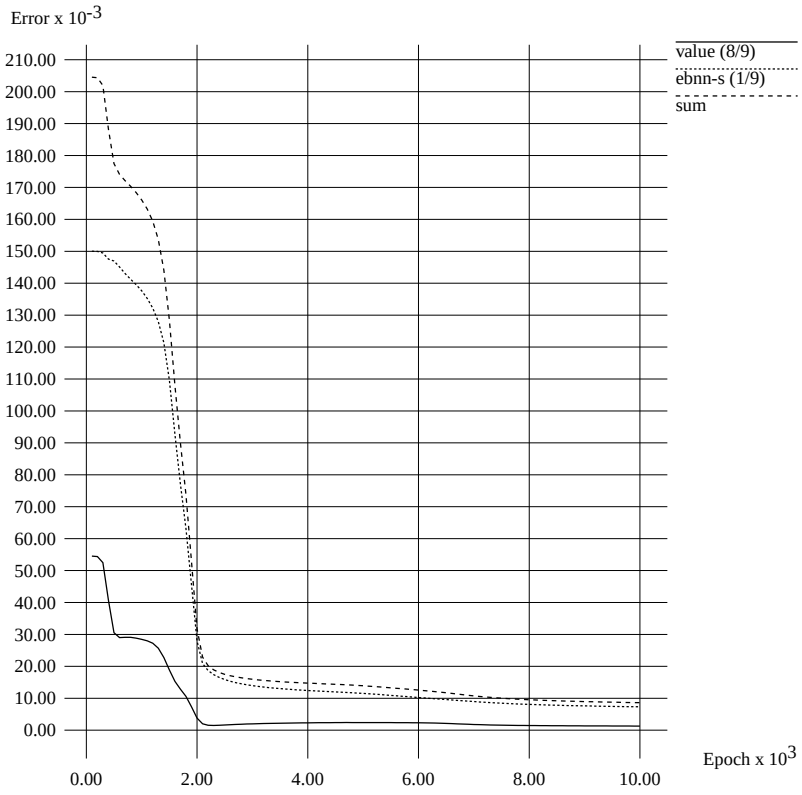

Figure 5.9: **Error curves of EBNN learning:** The graphs of the total error E for EBNN learning (sum), its value error part (value), and its slope error part (ebnn-s) are plotted against the number of epochs for the EBNN learning.

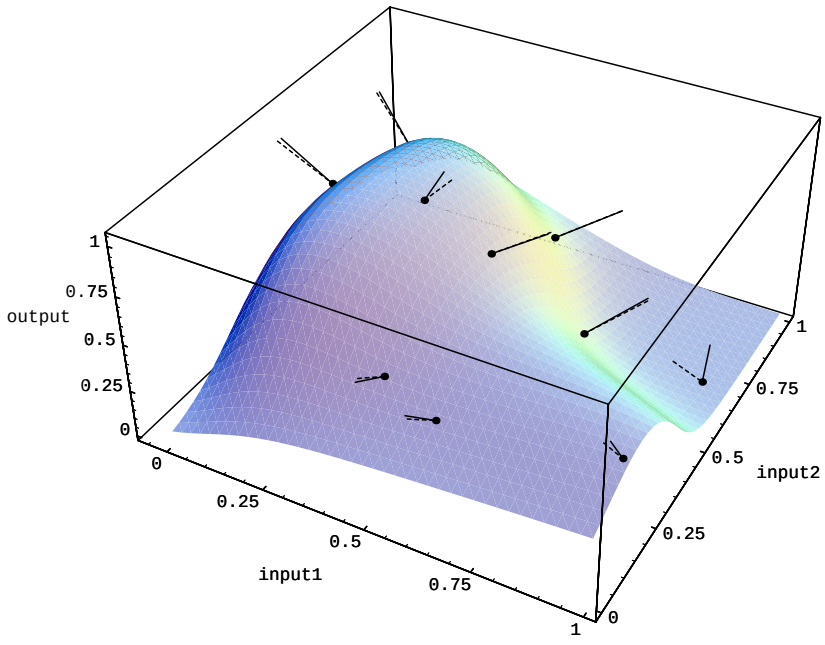

Figure 5.10: **Surface learned by EBNN learning:** The surface learned by EBNN learning is shown. The broken lines are the normals to the surface of target function and the solid lines are the normals to this surface learned by EBNN learning. Note that in this case these two normals for each point are very close except for one point. Because they are so close, some of the broken lines are hidden by the solid lines.

the function learned by EBNN learning. [5] The generalization errors for learning procedures by back propagation and EBNN learning are plotted against the number of epochs in Figure 5.11. The generalization error for EBNN learning is faster to drop and its final value is smaller.

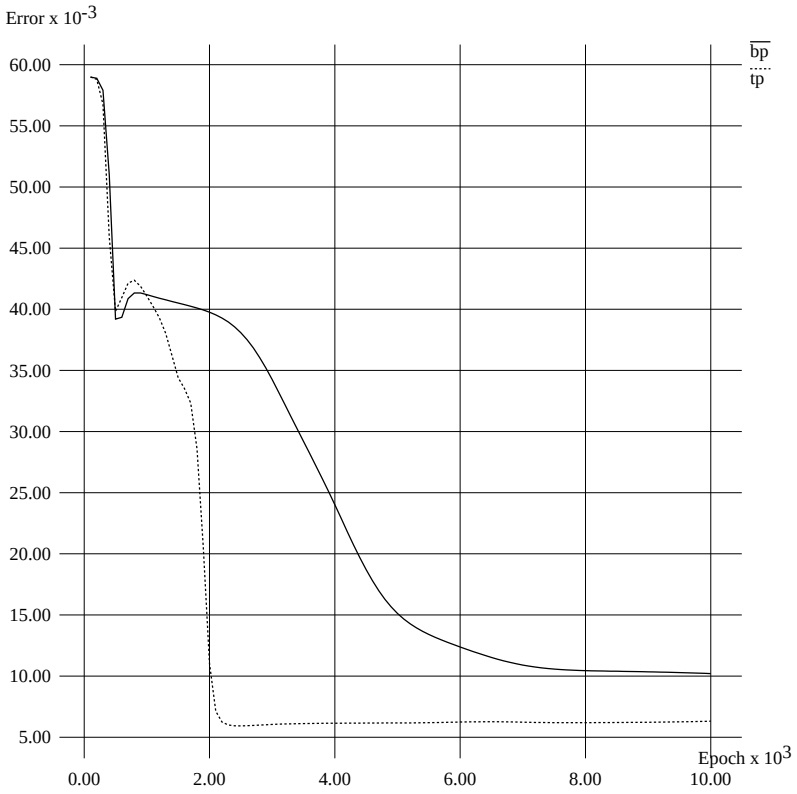

Figure 5.11: **Graphs of generalization errors:** The generalization errors for learning procedures by back propagation and EBNN learning are plotted against the number of epochs. Note that the generalization error for EBNN learning is faster to drop and the final value of it is smaller than that of back propagation.

5.3 Sample Complexity

In this section, we explain the results of the experiments to see the improvement in generalization of EBNN learning compared to back propagation learning in terms of the number of required training patterns.

We use the function g of Equation (5.10) in Section 5.2. We use x and y for two inputs of the neural network. The general settings for the experiments are

[5] We have to remind that it is not always the case that EBNN learning outperforms back propagation. However in the following section, statistical results support that EBNN learning does better than back propagation on average.

the same as those described in the beginning of Section 5.2.

We run the experiments for from 1 to 100 patterns. In each pattern set, the first pattern is taken to be $(0.5, 0.5)$ and the other remaining patterns are taken from the uniform distribution on the input space, $[0, 1] \times [0, 1]$.

We estimate the performance of each learned neural network after 10,000 epochs by the average of squares of value errors from the target function on 26×26 equidistant lattice points. This performance is same as the *generalization error* in Section 5.2 except for the number of lattice points.

We plot the performances of back propagation and EBNN learning against the number of patterns in Figure 5.12. Except for small number of patterns and for more than 50 patterns, EBNN learning outperforms back propagation. By crossing the graphs horizontally, one can see the required number of patterns to achieve certain performance for EBNN learning is 1 to 2 times smaller than that of back propagation in the region where EBNN learning outperforms back propagation.

The neural networks used in both methods are the same. Therefore in each particular trial, back propagation might outperform EBNN learning.[6] By averaging over 10 trials, we can see the apparent tendency that EBNN learning outperforms back propagation as Figure 5.12 shows it.

For first several patterns and for patterns more than 50, EBNN learning is outperformed by back propagation. We assume the reasons are the following. Back propagation approximates only values at points while EBNN learning approximates also slopes at points. While there are a few points, with back propagation the whole surface stays at the level of somewhat around the given values, But in EBNN learning, fitting slopes makes other parts of surface deviate from the target surface. As soon as the points are dense enough, EBNN learning starts to outperform the back propagation.

But then, the performance of back propagation catches up that of EBNN learning at 50 patterns. After examining the data of learning procedure, we observed that EBNN learning stops improving on either the value error or the EBNN slope error in the early stages 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 learning is outperformed by back propagation is because of the above and because we estimate the performance in terms of the generalization error that is essentially the value error.

5.4 Effect of Irrelevant Features

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

We use the function g of Equation (5.10) in Section 5.2. We used 2 inputs for x and y as relevant inputs and other 0, 3, or 8 inputs as irrelevant inputs. (Therefore the numbers 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.

$$\overline{g}(x, y, x_1, ...) = g(x, y) \tag{5.11}$$

[6] Actually in each individual trial, it often happened that back propagation outperforms EBNN learning for certain areas.

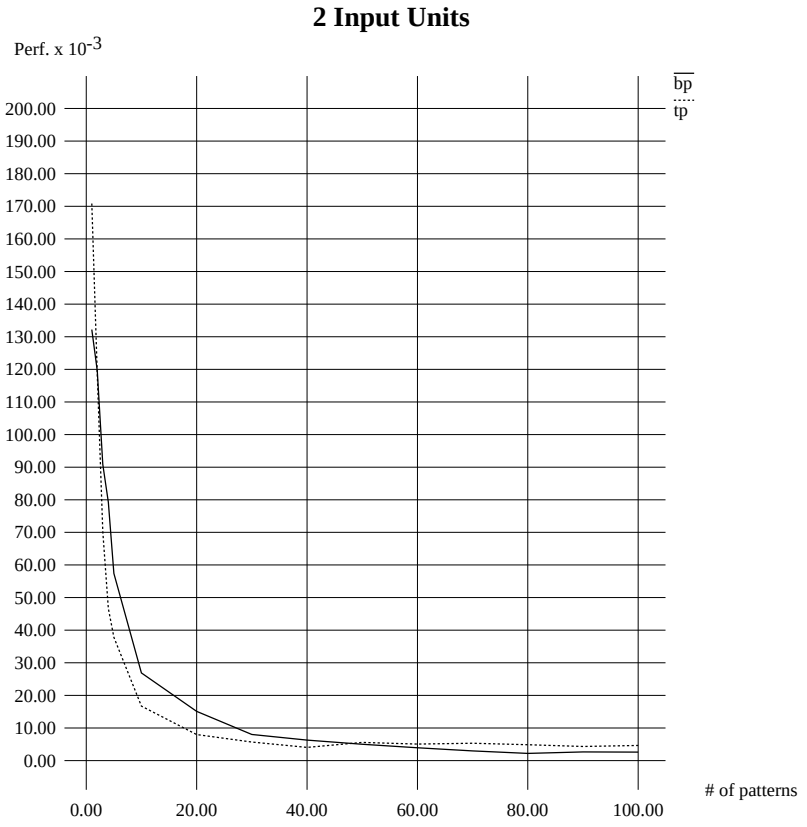

Figure 5.12: **Graphs of sample complexity:** We plot the performances of back propagation (bp) and EBNN learning (tp) against the number of patterns for from 1 to 100 patterns and join them by the lines. Except for small number of patterns and for more than 50 patterns, EBNN learning outperforms back propagation. By crossing the graphs horizontally, one can see the required number of patterns to achieve certain performance for EBNN learning is 1 to 2 times smaller than that of back propagation in the region where EBNN learning outperforms back propagation.

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

We do essentially the same experiment in Section 5.3 for each number of input units. We collected the performance of each method on from 1 to 100 patterns. The performances are plotted against the number of patterns for the networks of 2, 5, and 10 inputs in Figure 5.13. We can see that the performance of back propagation degenerates rapidly as the number of irrelevant input units increases, while EBNN learning stays at the same level.

We use above results to produce Figure 5.14. 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. (This is similar to the mistake bound in [19].) The detailed explanation of what are plotted is the following. In these graphs, we plot 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.)

$$I.A.P. = \sum_{i=1}^{n} \sqrt{\text{the average performance for } i \text{ patterns}} \tag{5.12}$$

Where the average performance is missing, we interpolated its value linearly from the values known. [7]

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

5.5 Noise Robustness

In the frameworks like Explanation-Based Neural Network (EBNN) [25] in which target slopes are constructed from incomplete and/or inaccurate world models, noises in target slopes are inevitable. In this section we explain the results of experiments to see how noises in target values and in target slopes affect back propagation and EBNN learning. The experimental results are adopted from [22].

We use the function g of Equation (5.10) in Section 5.2 as the target function again and used fixed 20 patterns for all the experiments. By numerical integration, the averages of the absolute values of derivatives are given by:

$$\int_0^1 \int_0^1 \left| \frac{dg}{dx} \right| dxdy \quad \simeq \quad 0.893483 \tag{5.13}$$

[7] As Figure 5.14 is produced from the results in Figure 5.13, the graphs in Figure 5.14 give the same conclusion that EBNN learning is more robust with respect to the number of irrelevant input units.

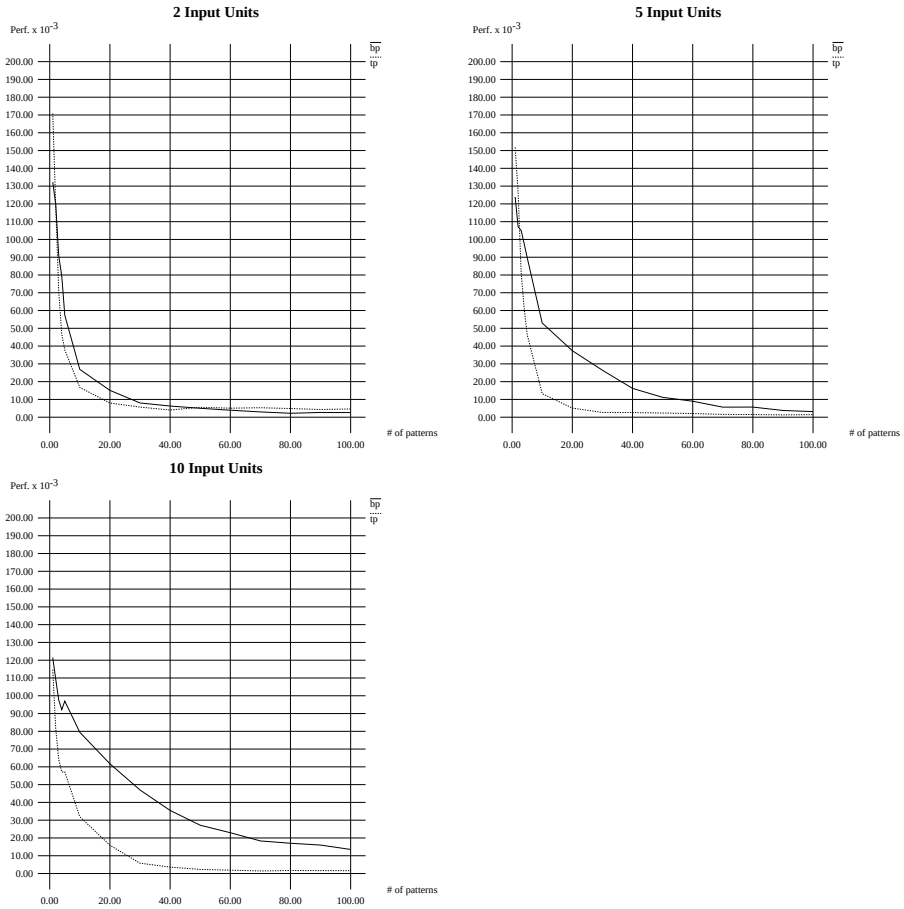

Figure 5.13: **Graphs of effect of irrelevant features:** The performances are plotted against the number of patterns for the networks of 2, 5, and 10 inputs for from 1 to 100 patterns. We can see that the performance of back propagation (bp) degenerates rapidly as the number of irrelevant input units increases, while EBNN learning (tp) stays at the same level.

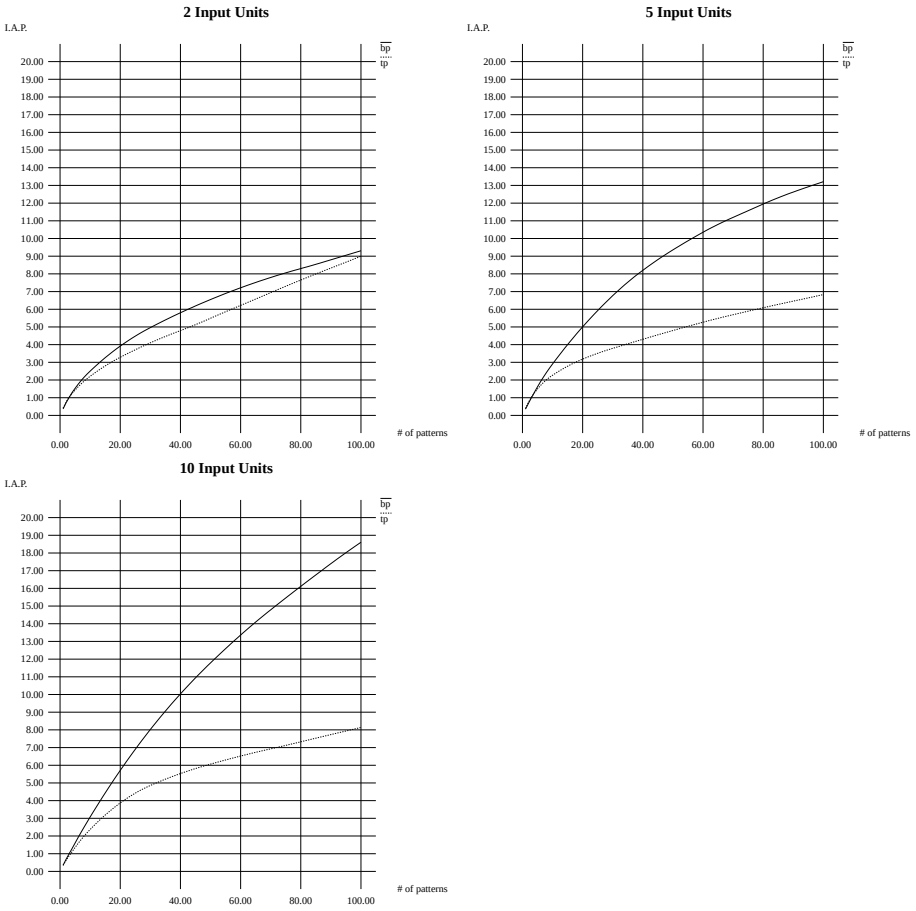

Figure 5.14: **Graphs of mistake bound:** In these graphs, we plot I.A.P., the integrated value of the square root of the average performance up to the number of patterns. These graphs show that EBNN learning is more robust with respect to the number of irrelevant input units.

$$\int_0^1 \int_0^1 \left| \frac{dg}{dy} \right| dxdy \quad \simeq \quad 1.01263 \tag{5.14}$$

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 δ net) output.[8] 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 back propagation, 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 given in Figure 5.15.

We then run EBNN learning, 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 given in Figure 5.16.

We also run EBNN learning, changing both the value and slope noise levels as in the previous two experiments. The result is given in Figure 5.17.

From these experiments, we can conclude the following.

From Figure 5.15 and the cross section of Figure 5.17 for which "*slope noise level* = 0.0", EBNN learning generalizes better than standard back propagation even if value noises are present.

By observing the cross sections of Figure 5.17 for which value noise level is constant, we can conclude that the greater the value noise level, the less the slope noise level matters. Also from the above cross sections and Figure 5.15, we can also conclude the slope noise level for which EBNN learning has the same performance as standard back propagation increases, as the value noise level increases.

In Figure 5.16, the curve of the performance by EBNN learning shows a graceful degeneration as the slope noise level increases. The crossover point of these two curves in Figure 5.16 shows when EBNN learning is degenerated enough so as to have the same performance of back propagation without noise. 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 learning is quite robust against noises.

This results support the observations in Sections 5.1.1 and 5.1.2 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 is robust.

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

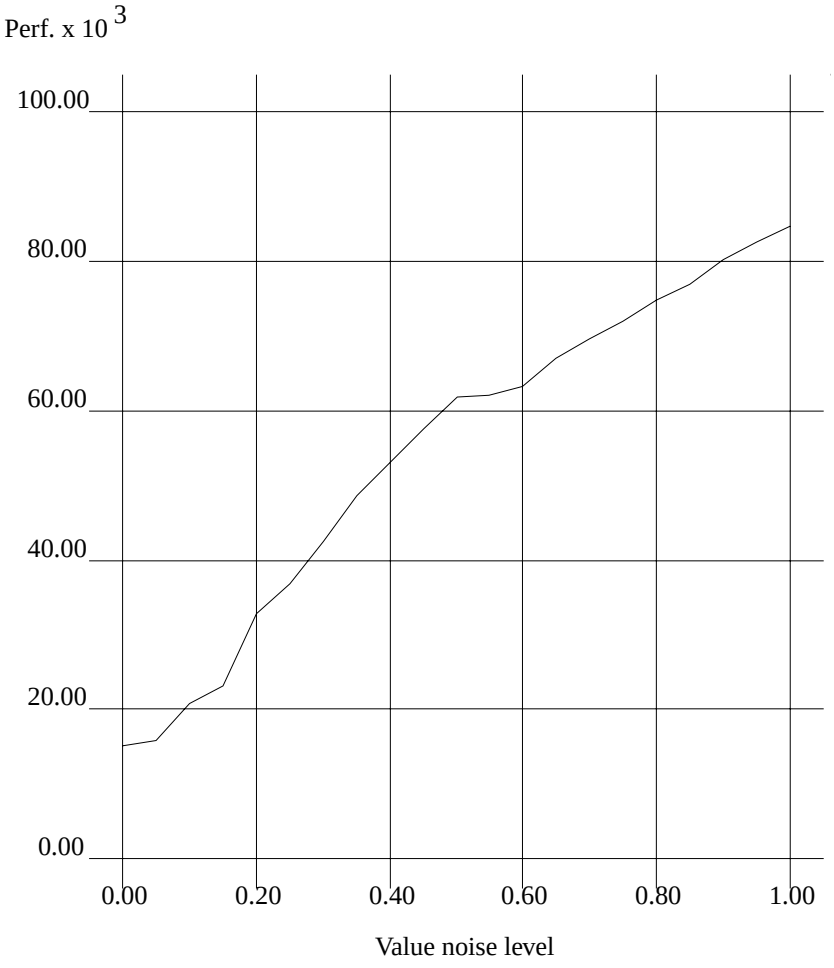

Figure 5.15: **Performance of back propagation:** The performance of standard back propagation against the value noise level is plotted. Noises are added to desired output values. The value noise level is varied from 0.0 to 1.0 in increments of 0.05.

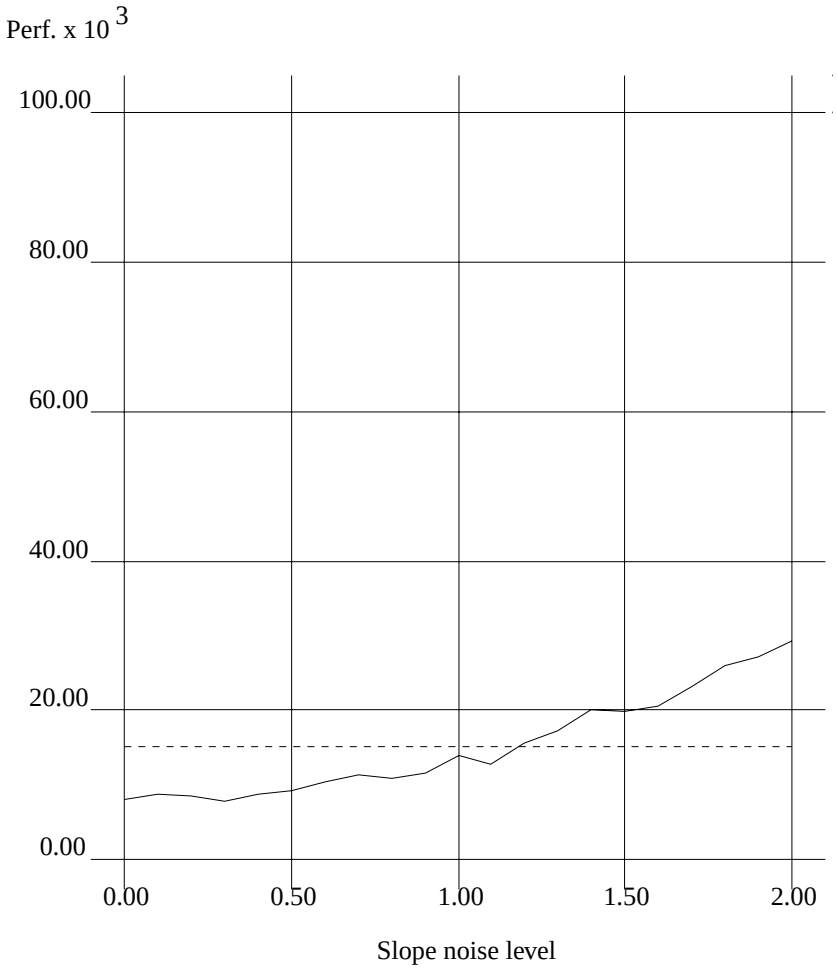

Figure 5.16: **Performance of EBNN learning (1):** The performance of EBNN learning against the slope noise level is plotted as the solid line. Noises are added only to desired outputs for δ nets. The slope noise level is changed from 0.0 to 2.0 with step size 0.1. The broken line is the performance of standard back propagation with no noise in the training patterns. (See Figure 5.15 where the value noise level is 0.0.) Notice that solid line and the broken line cross around where the noise level is 1.2. That is the crossover point where the desired outputs for δ nets are degraded enough to match the performance of standard back propagation.

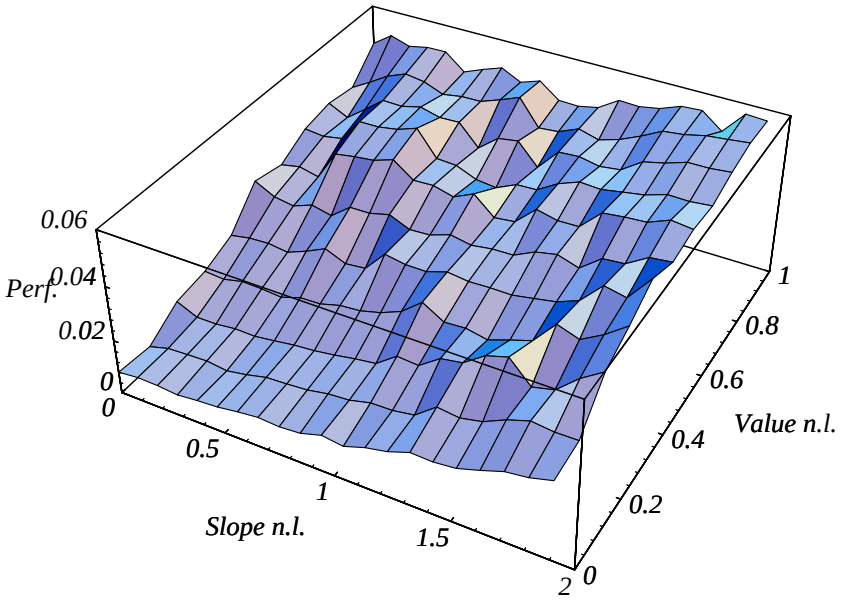

Figure 5.17: **Performance of EBNN learning (2):** The performance of EBNN learning against both the value and slope noise levels is plotted.

Chapter 6

An Application to Continuous Action Generation in Reinforcement Learning

In this chapter, we describe an application of neural networks learning differential data to continuous action generation in reinforcement learning. We describe the problem of continuous action generation in reinforcement learning, give formalization for the problem, and illustrate the results of the experiments.

In order to deal with continuous action generation in reinforcement learning, we need to have a random vector generator for an arbitrary probability distribution. As being explained in Section 6.2, the problem is reduced to solving a non-linear (partial) differential equation. The implementation as described in Chapter 4 of neural networks learning differential data is modified to solve the differential equation and the experiments are conducted.

6.1 Problem of Continuous Action Generation in Reinforcement Learning

In this section, we describe the problem of continuous action generation in reinforcement learning and the necessity of a random vector generator for an arbitrary probability distribution.

In general, it is difficult to handle the continuous action space in robotics and hence in reinforcement learning. For the cases where there are mathematical models of environment, those mathematical models are utilized to deal with continuous action space. But it is ordinarily not possible to have mathematical models of environment for the cases in robotics.

Reinforcement learning can provide the probability distribution over the continuous action space for the controller to produce an action. The problem is with existence of appropriate random vector (action) generators for the given arbitrary probability distribution.

Common techniques used in reinforcement learnings include the followings and the combinations of the followings.

- Quantize continuous action spaces. Then apply discrete methods. (See the beginning of Chapter 4 of [37] and the acrobat problem in [26])
- Use a parameterized probability distribution such as Gaussian distributions for continuous action generation. Then adjust parameters, such as the mean and the deviation in Gaussian distribution case, to fit the given probability distribution. (See Section 4.3 and 4.4 of [14])

Those methods have the following shortcomings.

Quantization of continuous action spaces needs knowledge of characteristics of the problem beforehand. Therefore it is not applicable without introducing arbitrariness of the system integrators. The quantization also introduces so-called "the curse of dimensionality," meaning that its computational requirements grow exponentially with the number of state variables. Since better approximation of a given probability distribution needs finer quantization that requires more variables, it is vulnerable to "the curse of dimensionality."

We want to use continuous action spaces as is, to avoid the above shortcomings. As mentioned above, the problem with this approach is appropriate random vector (action) generators for the given arbitrary probability distribution.

There are several methods known to generate random vectors for the given probability distributions. These include the followings [32].

- Transformation method
- Rejection method

But they are not suitable to use in reinforcement learning for the following reasons. The transformation method that transforms the uniform distribution is only applicable to the probability distributions whose inverse functions of their integrals are known. Since the probability distribution is not known in advance, the transformation method is not directly applicable.

Though the rejection method does not have a preference for the probability distribution, it takes time for generating the random vector because of rejections of candidate vectors.

We propose to use neural networks learning differential data for random vector generators as continuous action generators in reinforcement learning. We use learning capability of neural networks learning differential data to realize the inverse functions of the integrals of given probability distributions as in the transformation method. This allows us to have the random vector generators necessary in reinforcement learning, which can adapt to an arbitrary probability distribution and which can generate random vectors fast enough.

6.2 Formalization

In this section, we give formalization for the problem of continuous action generation and show how neural networks learning differential data is applied to the problem.

As described in the introduction of this chapter, the problem of continuous action generation is formalized as follows.

Problem 6.1 *Realize a random vector generator for an arbitrary probability distribution $p(v)dv$ of R^l. dv is Lebesgue measure of R^l and v is an R^l-valued variable.*

Problem 6.1 is reduced to solve the following problem in the one-dimensional case.

Problem 6.2 *Find the inverse function $F^{-1}(u)$ of $F(v)$ that satisfies $F(v) = \int p(v)dv$.*

If this inverse function is obtained, we can realize the desired random vector (number) generator as follows. First, we generate one random value from the uniform distribution du on $[0, 1]$ and then we give $v = F^{-1}(u)$ as the output.

This way, the uniform distribution du on the interval $[0, 1]$ is transformed by $F^{-1}(u)$ into the given probability distribution $p(v)dv$ and the random vector (number) generator for the given probability distribution $p(v)dv$ is realized

This is the essence of the transformation method. The situation is easily understandable by Figure 6.1 adapted from [32].

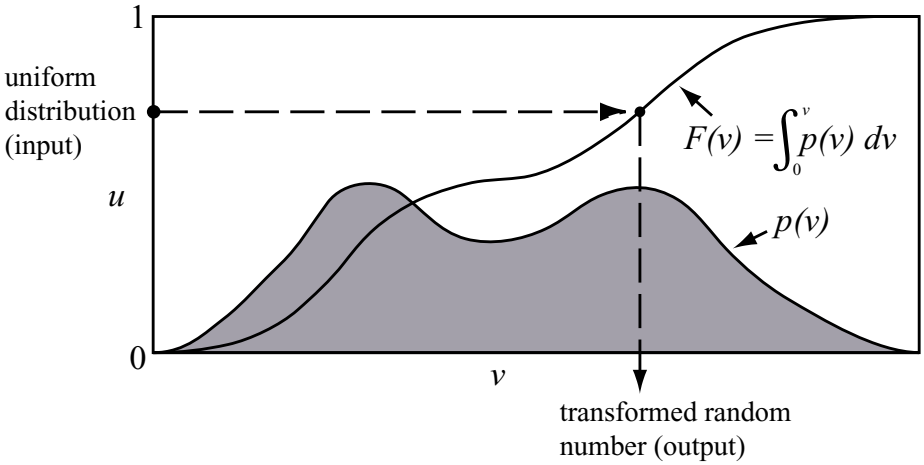

Figure 6.1: Transformation Method: Outline of the transformation method is shown in the figure.

If the probability distribution has the inverse function of its integral known beforehand, the random vector generator is realized this way.

We solve Problem 6.1 by letting neural networks learn the function F^{-1} in Problem 6.2. We formalized the problem as the following including the multi-dimensional cases. We treat the cases of arbitrary dimensions in this section first. Then we derive the specific forms of ϵ's to give to the output layers of the neural network in Sections 6.2.1, 6.2.2, and 6.2.3. for the general case, the case of Fixed Value Outputs, and one-dimensional case respectively.

We define n as the function defined by the neural network in use. Let u and v stand for the variables of the input and output spaces respectively and Let w stand for the weight vector of the neural network.

$$v = n(u) = n(w, u) \tag{6.1}$$

We assume the domain (input space) and range (output space) of the function have the same dimension. [1]

Let $\tilde{p}(u)$ be a probability distribution on the domain. We take the uniform distribution or other distributions with known random generators for $\tilde{p}(x)$. Let $p(v)dv$ another probability distribution given on the range. Then what we want is the neural network n that satisfies the following condition.

$$p(v)dv = \tilde{p}(u) \left| \frac{\partial u}{\partial v} \right| dv \tag{6.2}$$

The condition is reduced to the following differential equation including Jacobian.

$$\left| \frac{\partial v}{\partial u} \right| = \frac{\tilde{p}(u)}{p(v)} \tag{6.3}$$

Replacing $v = n(u)$ gives the following differential equation for n.

$$\left| \frac{\partial n(u)}{\partial u} \right| = \frac{\tilde{p}(u)}{p(n(u))} \tag{6.4}$$

In above two differential equations, $|\ |$ stands for the determinant of the matrix.

If we can give a function $n(u)$ which satisfies Equation (6.4), then we can realize the random vector generator for the given $p(v)dv$. [2]

For the rest of this section, we consider solving the differential equation of Equation (6.4) by neural networks. The structure of neural networks is arbitrary as long as the input space and the output space have the same dimension and they have no loops.

In terms of neural networks learning differential data, solving Equation (6.4) is interpreted as the following error minimizing problem. We take enough points in the domain and find the set of weights of the neural network that minimize the following error function. [3] [4] [5]

$$E_{pd}(w, u) = \frac{1}{2}\alpha \left(\left| \frac{\partial n(w, u)}{\partial u} \right| - \frac{\tilde{p}(u)}{p(n(w, u))} \right)^2 \tag{6.5}$$

[1] If the range has the lower dimension than the dimension of the domain, then by adding extra dimensions with fixed output we can make the range and the domain have the same dimension.

[2] We need to consider relations between the boundaries of the domain and the range in multi-dimensional cases. But this depends on each case and the consideration is omitted.

[3] The subscript pd of E stands for probability distribution.

[4] For the sake of simplicity and without loss of generality, we give the error function and the following equations for the case with one point.

[5] As in Chapter 3, we introduce the learning constant α in the error function itself. We denote this constant simply by α because this α is different from α^δ's dependent on the direction of derivative δ in Chapter 3.

We apply the gradient descent to this error function. Weights are updated by the derivative of the error function with respect to w. So the update rule is given by the following equation.

$$\begin{aligned}\Delta w &= -\frac{\partial}{\partial w} E_{pd}(w,u) \\ &= \alpha \left(\frac{\tilde{p}(u)}{p(n(w,u))} - \left| \frac{\partial n(w,u)}{\partial u} \right| \right) \times \\ &\quad \left(\frac{\partial}{\partial w} \left| \frac{\partial n(w,u)}{\partial u} \right| + \frac{\tilde{p}(u)}{p(n(w,u))^2} p'(n(w,u)) \frac{\partial}{\partial w} n(w,u) \right) \end{aligned} \tag{6.6}$$

What we have to note here is that Equation (6.6) include terms like $p(n(w,u))$ and $p'(n(w,u))$. The neural network propagates backward ϵ's which is determined by the output $n(u)$ of the network.

Note also that $p(v)$ and $p'(v)$ have to be given as functions beforehand in order to calculate the error for the neural network since the error has to be given for arbitrary $v = n(w,u)$.

6.2.1 ϵ's in the General Case

In this section, we derive the specific form of ϵ's in the general case to give to the output layers of the neural network for backward propagation to calculate the values of (6.6) in the framework of Chapter 3.

We denote by $\epsilon^1_{i,k}$, ϵ for the output unit of δ net corresponds to the derivative with respect to u_i input unit of the k-th output unit of the value net. We denote by ϵ^0_k, ϵ for the k-th output unit of the value net. As in Chapter 3.7, these ϵ's are the derivatives of the error function with respect to the input to the unit.

We also denote by $\Delta_{k,i}$, the minor of the matrix, $\partial n(w,u)/\partial u$ removing k-th row and i-th column. [6] Then the following holds.

$$\left| \frac{\partial n(w,u)}{\partial u} \right| = \sum_{i,k} (-1)^{i+k} \Delta_{k,i} \frac{\partial n_k(w,u)}{\partial u_i} \tag{6.7}$$

Equation (6.6) consists of the part for back propagation of the value net and the first order part for back propagation of the δ net.

Let $\sigma_k(net)$ be the sigmoid function for the k-th output unit in the value net. [7] Then ϵ^0_k is given by the following.

$$\epsilon^0_k = \alpha \left(\frac{\tilde{p}(u)}{p(n(u))} - \left| \frac{\partial n(w,u)}{\partial u} \right| \right) \frac{\tilde{p}(u)}{p(n(w,u))^2} \, p'(n(w,u)) \, \sigma^{(1)}_k(net) \tag{6.8}$$

Multiplying by $\sigma^{(1)}_k(net)$ is needed since the definition of ϵ is the derivative of the error function with respect to the input *net* to the unit.

$\epsilon^1_{i,k}$ is given by the following.

$$\epsilon^1_{i,k} = (-1)^{i+k} \alpha \, \Delta_{k,i} \left(\frac{\tilde{p}(u)}{p(n(w,u))} - \left| \frac{\partial n(w,u)}{\partial u} \right| \right) \tag{6.9}$$

In this case, we do not need to multiply it by a derivative of $\sigma_k(net)$ since the units in the δ net has the output same as the input to the unit.

[6] For 1×1 matrix, we let $\Delta_{1,1} = 1$.

[7] *net* is the input to the unit.

6.2.2 ϵ's in the Fixed Value Output Case

In this section, we derive the specific form of ϵ's in the fixed value output case to give to the output layers of the neural network for backward propagation to calculate the values of (6.6) in the framework of Chapter 3.

By a fixed value output case, we mean a case in which the output of the value network of the neural network is fixed for the input point. Such cases are cases where some of input points on the boundary of the domain are mapped into points on the boundary of the range.

In the fixed value output case, ϵ is fixed for those points since the value output of the neural network is fixed. Therefore it is a straightforward application of the framework of Chapter 3.

Let the fixed value output of the neural network be v_f and the learning constant be β. Then the error function for fixing the value output is given as follows. [8]

$$E_{fp}(w,u) = \frac{1}{2}\,\beta\,\left(n(w,u) - v_f\right)^2 \tag{6.10}$$

Since the value output is fixed, the derivation of ϵ's for Equation (6.5) is much simpler. Instead of Equation (6.6), we have the following equation without value parts for the update of the weights.

$$\begin{aligned}\Delta w &= -\frac{\partial}{\partial w}E_{pd}(w,u) \\ &= \alpha\left(\frac{\tilde{p}(u)}{p(n(w,u))} - \left|\frac{\partial n(w,u)}{\partial u}\right|\right) \times \left(\frac{\partial}{\partial w}\left|\frac{\partial n(w,u)}{\partial u}\right|\right)\end{aligned} \tag{6.11}$$

Therefore for the fixed output value case, ϵ_i^0 is derived only from Equation (6.10) and is given by Equation (6.12). $\epsilon_{i,k}^1$ for the fixed output value case is given by Equation (6.13) that stays the same as Equation (6.9).

$$\epsilon_k^0 = \beta\left(v_f - n(w,u)\right) \tag{6.12}$$

$$\epsilon_{i,k}^1 = (-1)^{i+k}\alpha\,\Delta_{k,i}\left(\frac{\tilde{p}(u)}{p(n(w,u))} - \left|\frac{\partial n(w,u)}{\partial u}\right|\right) \tag{6.13}$$

6.2.3 ϵ's in the One-Dimensional Case

In this section, we derive the specific form of ϵ's in the one-dimensional case where the input space and the output space is one-dimensional. In that case, $|\partial n(w,u)/\partial u| = \partial n(w,u)/\partial u$ and we let $\Delta_{1,1} = 1$ for 1×1 matrix. Therefore, $\epsilon^1 = \epsilon_{1,1}^1$ for the δ net of Equation (6.9) is simplified as follows.

$$\epsilon^1 = \alpha\left(\frac{\tilde{p}(u)}{p(n(w,u))} - \frac{\partial n(w,u)}{\partial u}\right) \tag{6.14}$$

With this ϵ^1, ϵ^0 of Equation (6.8) for the value net in the one-dimensional case can be expressed as follows.

[8] The subscript fp of E stands for fixed point.

$$\epsilon^0 = \epsilon^1 \frac{\tilde{p}(u)}{p(n(w,u))^2} p'(n(w,u)) \sigma^{(1)}(net) \tag{6.15}$$

Especially when the probability distribution $\tilde{p}(u)$ given to the input space is uniform (i.e. $\tilde{p}(u) = c$), above Equations 6.15 and 6.14 are simplified as follows.

$$\epsilon^1 = \alpha \left(\frac{c}{p(n(w,u))} - \frac{\partial n(w,u)}{\partial u} \right) \tag{6.16}$$

$$\epsilon^0 = \epsilon^1 \frac{c}{p(n(w,u))^2} p'(n(w,u)) \sigma^{(1)}(net) \tag{6.17}$$

6.3 Experiments on Learning to Be Random Vector Generators

In this section, we report the experiments on neural networks learning to be random vector generator by using the results of Section 6.2.

The implementation as described in Chapter 4 of neural networks learning differential data is modified to realize changing ϵ's of Equations (6.16) and (6.17) in Section 6.2.3.

Here follows common conditions for the experiments.

The input space and the output space are both one-dimensional intervals. The input space is $[0, 1]$ and the output space is $[0.25, 0.75]$. The probability distribution given to the input space is the uniform distribution (i.e. $\tilde{p}(u) = 1$). The value outputs of the end points of the input interval, $u = 0$ and $u = 1$ are fixed to be $n(0) = 0.25$ and $n(1) = 0.75$ respectively. Therefore the results of Section 6.2.2 are used for those end points of the input interval, while the results of Section 6.2.3 are used for the inside points of the input interval.

Fixing the Outputs for the End Points In this paragraph, we show that there is no problem in fixing the outputs for the end points of the input interval.

Let $p(v)$ be the probability distribution on the output interval $[a, b]$. Then the following holds.

$$\int_a^b p(v)dv = 1 \tag{6.18}$$

In the case of the uniform distribution on the input interval $[0, 1]$, for the neural network $n(u)$ that satisfies Equation (6.4), the following holds.

$$\frac{dv}{du} = \frac{1}{p(v)} = \frac{1}{p(n(u))} \tag{6.19}$$

Fixing the output of the end points and learning to satisfy Equation (6.4) are not contradictory as the following equation shows.

$$1 = \int_0^1 du = \int_0^1 \frac{du}{dv} dv = \int_{n(0)}^{n(1)} p(v)dv \tag{6.20}$$

Since $p(v)$ is a probability distribution, $n(0) = a$ and $n(1) = b$ have to hold for the right hand side of the equation to be 1.

6.3.1 Gaussian Distribution Case

In this section, we report the experiments on Gaussian distribution case.

We give the following (part of) Gaussian distribution on the output interval $[0.25, 0.75]$.

$$p(v) = 2.42261\ e^{-10(v-0.5)^2} \tag{6.21}$$

The constant 2.42261 is used to make the integral of $p(v)$ on $[0.25, 0.75]$ to be 1. The graph of $p(v)$ is given by Figure 6.2.

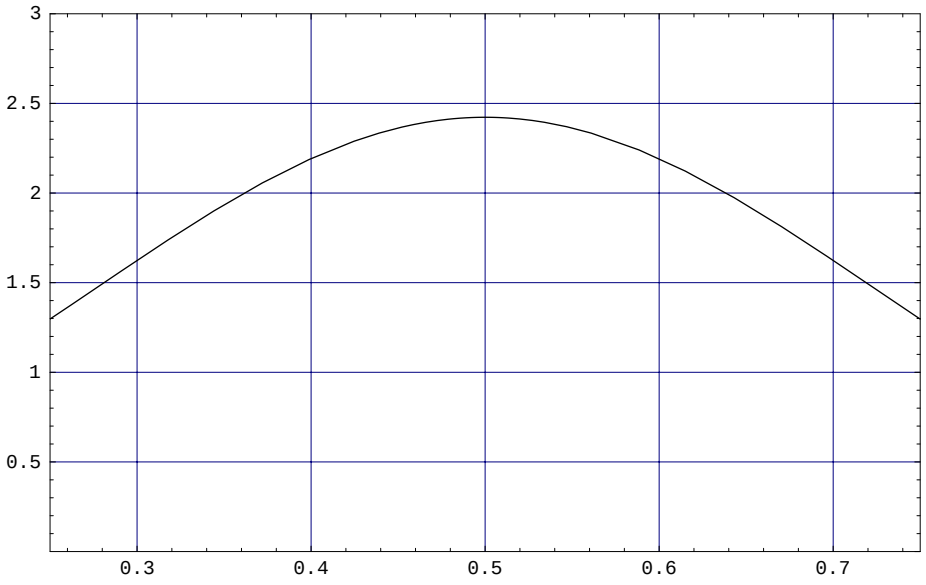

Figure 6.2: Gaussian Distribution: The graph of $p(v) = 2.42261\ e^{-10(v-0.5)^2}$ on the interval $[0.25, 0.75]$ is given.

We use three-layer perceptron with one unit in the input layer, sixteen units in the middle (hidden) layer, and one unit in the output layer and we make the neural network to learn to be a random vector (number) generator for this $p(v)$ following Section 6.2.

Learning constants are set such that $\alpha = 0.1$, $\beta = 0.175$, and the momentum is set to 0.2. We take eleven training points including the end points of interval, each separated for 0.1 on the interval $[0, 1]$.

The results are given in Figures 6.3, 6.4, and 6.5 after 100,000 epochs of learning.

The total sum of the value error (Equation (6.10)) for the two end points is 9.72601×10^{-7} and the total sum of the first order error (Equation (6.5)) for all eleven points is 5.20717×10^{-4}.

The results indicate that the neural network learns to be a random vector (number) generator quite successfully.

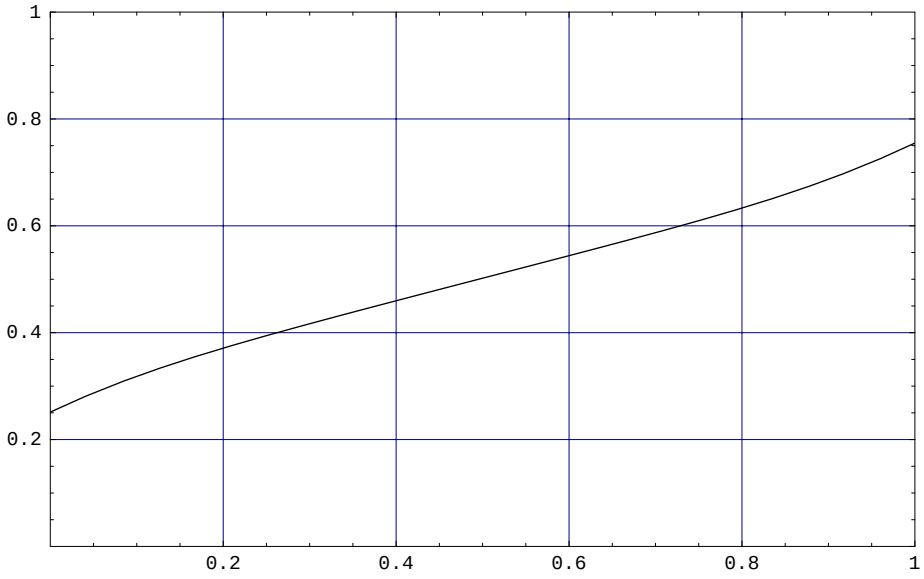

Figure 6.3: Value Output (Gaussian Distribution Case): We use three-layer perceptron with one unit in the input layer, sixteen units in the middle (hidden) layer, and one unit in the output layer to make the neural network to learn to be a random vector (number) generator for the Gaussian distribution. Learning constants are set such that $\alpha = 0.1$, $\beta = 0.175$, and the momentum is set to 0.2. We take eleven training points including the end points of interval, each separated for 0.1 on the interval $[0, 1]$. The graph shows the value output of the neural network after 100,000 epochs of learning. The graph corresponds to the graph $u = F(v)$ in Figure 6.1 with u-axis and v-axis reversed.

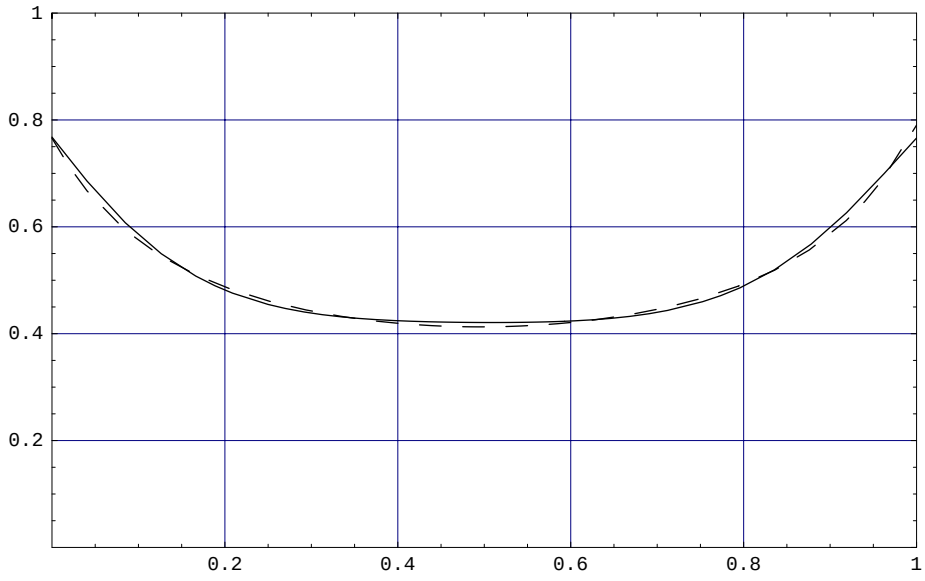

Figure 6.4: First Order Output (Gaussian Distribution Case): We use three-layer perceptron with one unit in the input layer, sixteen units in the middle (hidden) layer, and one unit in the output layer to make the neural network to learn to be a random vector (number) generator for the Gaussian distribution. Learning constants are set such that $\alpha = 0.1$, $\beta = 0.175$, and the momentum is set to 0.2. We take eleven training points including the end points of interval, each separated for 0.1 on the interval $[0, 1]$. The graph by the solid line shows the first order output of the neural network after 100,000 epochs of learning. The graph by the broken line shows the inverse of the Gaussian distribution density at the point of the value output of the neural network. (i.e. $1/p(n(w, n))$

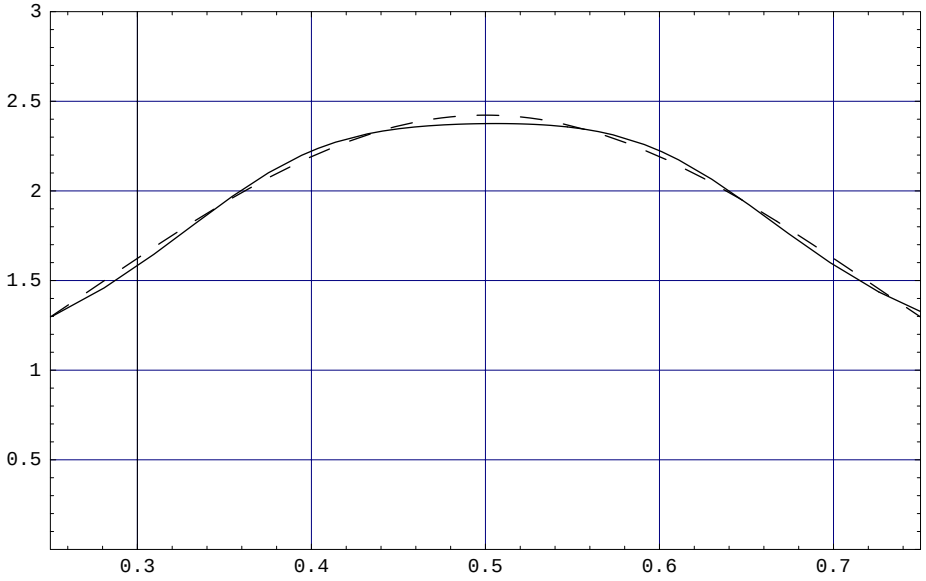

Figure 6.5: Probability Distribution Learned (Gaussian Distribution Case): The graph of the probability distribution learned by the neural network after 100,000 epochs is given by the solid line. The graph is the parametric plot of $\{(n(u), 1/(\partial n(u)/\partial u) \mid 0 \leq u \leq 1\}$. The graph by the broken line is of the target probability distribution and is same as the one in Figure 6.2.

6.3.2 Two-Peak Distribution Case

In this section, we report the experiments on two-peak probability distribution to see how well the neural network performs for a more complex case than one of Section 6.3.1.

We give the following two-peak distribution on the output interval $[0.25, 0.75]$.

$$\begin{aligned} p(v) &= 2.81763 \left(e^{-40(v-0.425)^2} + 0.65\, e^{-200(v-0.675)^2} \right) \\ &= 2.81763\, e^{-40(v-0.425)^2} + 1.83146\, e^{-200(v-0.675)^2} \end{aligned} \tag{6.22}$$

The constant 2.81763 is used to make the integral of $p(v)$ on $[0.25, 0.75]$ to be 1. The graph of $p(v)$ is given by Figure 6.6.

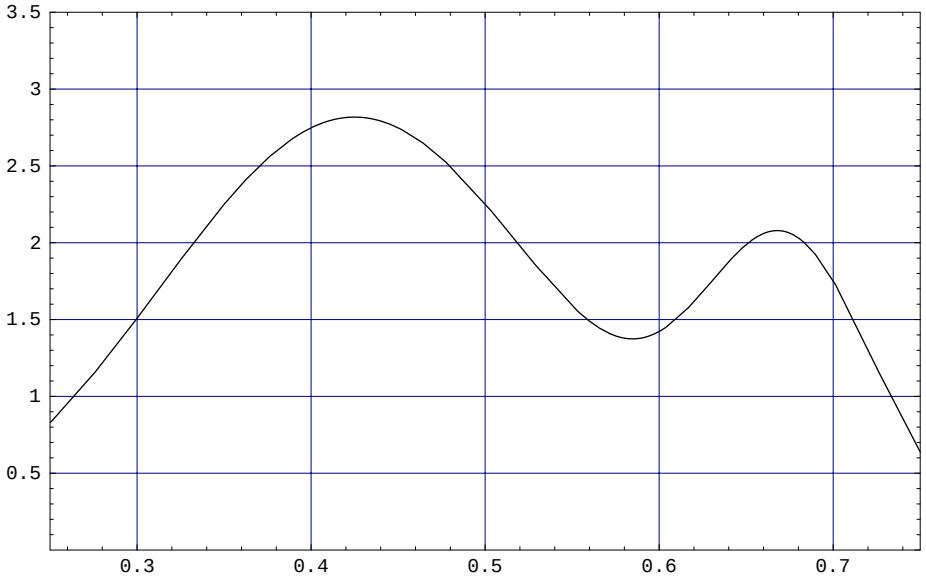

Figure 6.6: Two-Peak Distribution: The graph of $p(v) = 2.81763\ (e^{-40(v-0.425)^2} + 0.65\ e^{-200(v-0.675)^2})$ on the interval $[0.25, 0.75]$ is given.

We use three-layer perceptron with one unit in the input layer, sixteen units in the middle (hidden) layer, and one unit in the output layer and we make the neural network to learn to be a random vector (number) generator for this $p(v)$ following Section 6.2.

Learning constants are set such that $\alpha = 0.1$, $\beta = 0.005$, and the momentum is set to 0.2. We take eleven training points including the end points of interval, each separated for 0.1 on the interval $[0, 1]$.

The results are given in Figures 6.7, 6.8, and 6.9 after 10,000,000 epochs of learning.

The total sum of the value error (Equation (6.10)) for the two end points is 4.59249×10^{-7} and the total sum of the first order error (Equation (6.5)) for all eleven points is 6.07166×10^{-3}.

Even though a large number of learning epochs are needed and the graph still deviates from the ideal, the results indicate that the neural network can learn to be a random vector (number) generator even for a complex probability distribution.

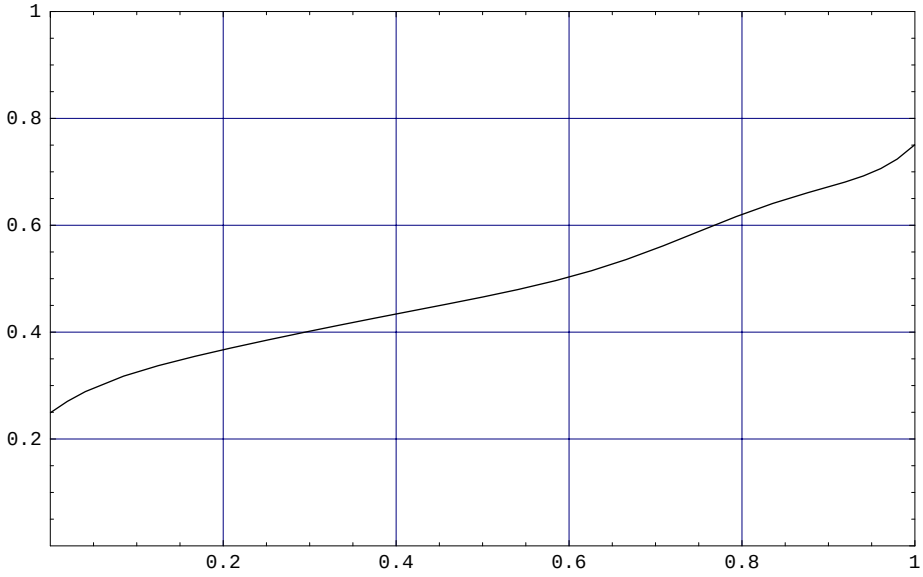

Figure 6.7: Value Output (Two-Peak Distribution Case): We use three-layer perceptron with one unit in the input layer, sixteen units in the middle (hidden) layer, and one unit in the output layer to make the neural network to learn to be a random vector (number) generator for the two-peak distribution. Learning constants are set such that $\alpha = 0.1$, $\beta = 0.005$, and the momentum is set to 0.2. We take eleven training points including the end points of interval, each separated for 0.1 on the interval $[0, 1]$. The graph shows the value output of the neural network after 10,000,000 epochs of learning. The graph corresponds to the graph $u = F(v)$ in Figure 6.1 with u-axis and v-axis reversed.

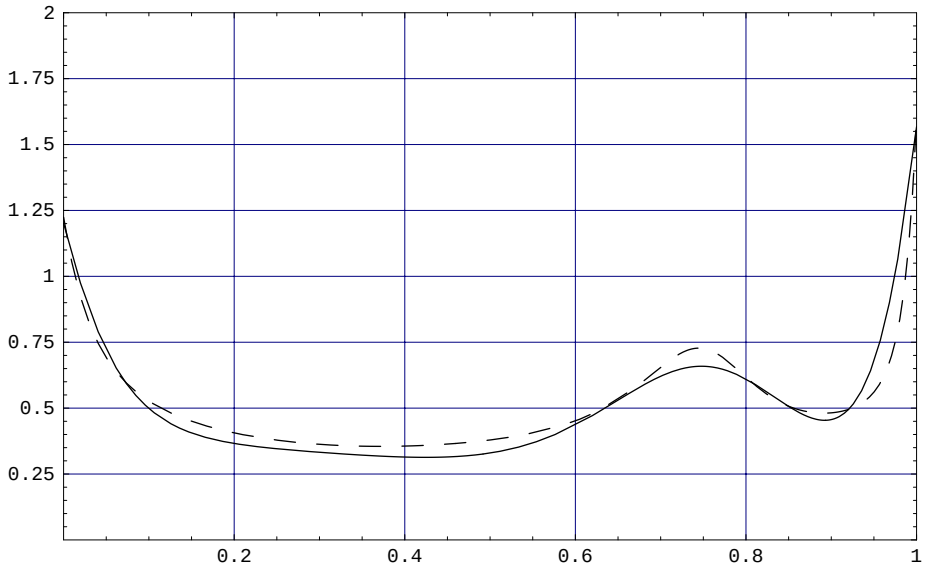

Figure 6.8: First Order Output (Two-Peak Distribution Case): We use three-layer perceptron with one unit in the input layer, sixteen units in the middle (hidden) layer, and one unit in the output layer to make the neural network to learn to be a random vector (number) generator for the Gaussian distribution. Learning constants are set such that $\alpha = 0.1$, $\beta = 0.005$, and the momentum is set to 0.2. We take eleven training points including the end points of interval, each separated for 0.1 on the interval $[0, 1]$. The graph by the solid line shows the first order output of the neural network after 10,000,000 epochs of learning. The graph by the broken line shows the inverse of the two-peak distribution density at the point of the value output of the neural network. (i.e. $1/p(n(w, n))$

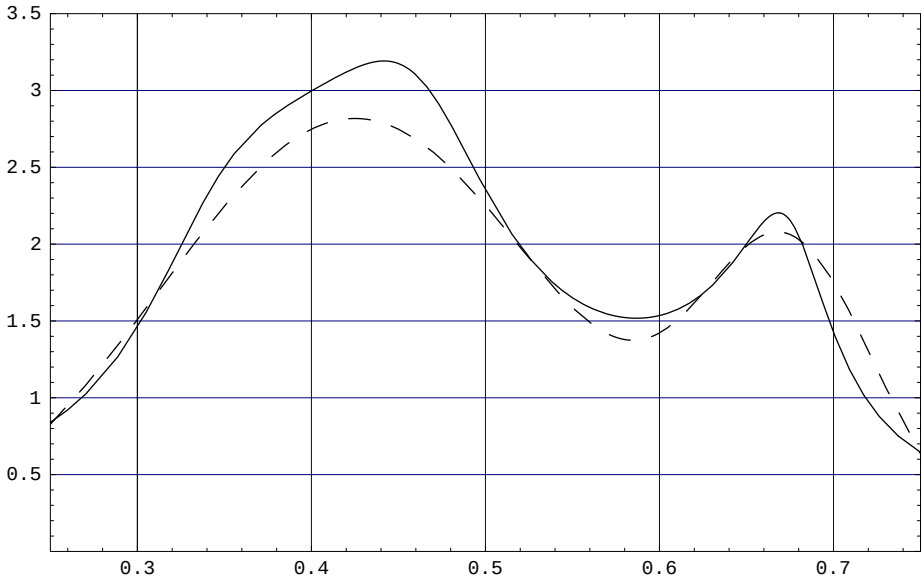

Figure 6.9: Probability Distribution Learned (Two-Peak Distribution Case): The graph of the probability distribution learned by the neural network after 10,000,000 epochs is given by the solid line. The graph is the parametric plot of $\{(n(u), 1/(\partial n(u)/\partial u) \mid 0 \leq u \leq 1\}$. The graph by the broken line is of the target probability distribution and is same as the one in Figure 6.6.

Chapter 7

Other Possible Applications

In this chapter, we describe two more possible applications of neural networks learning differential data, solving differential equations and simulation of human arm movement. For those examples, neural networks with capabilities to learn differential data can take advantage of knowledge in forms of constraints on differential data and easily incorporate such constraints into the learning of training value data.

7.1 Differential Equations

In this section, we describe an application of neural networks learning differential data, solving differential equations.

As practical applications of neural networks learning differential data, neural networks learning differential have been applied to solving differential equations [17, 4, 39, 16].

Lee and Kang [17] solve certain types of first order ordinary differential equations by using Hopfield-type neural networks to minimize the finite difference equations.

In [4, 39, 16], differential equations along with boundary conditions are turned into minimization problems. Those minimization problems are solved using neural networks with global minimization procedures such as quasi Newton gradient descent algorithm.

We use essentially the same framework of mapping differential equations, boundary conditions, and any other conditions into minimization problems by neural networks in Chapter 6. But we employ our implementation of neural networks learning differential data for the minimization procedure different from the previous works.

Our algorithm is completely localized to each neurons and it leaves possibility of parallel implementations for efficient executions, while global minimization procedures are very difficult to implement in parallel ways. It is also very difficult to apply global minimization procedures for adaptive problems. In many learning problems in robotics and others, what has to be learned, changes dynamically. In the problem of continuous action generation in reinforcement learning described in Chapter 6, the probability distribution given in the output space changes as the learning proceeds and so does the differential equation for

the neural network to satisfy. In such adaptive cases, our algorithm provides more gradual way of learning than global minimization procedures.

The difficulty with applications of neural networks learning differential data to differential equations is that of "moving targets." To minimize the square error of differential equations, neural networks need to propagate backward the errors, which depend on the outputs of the neural network. That is so even for linear differential equations. That means neural networks need to learn or to adapt to something like moving targets. This is totally different situation from the standard back propagation algorithm where the training data is fixed. Therefore there is difficulty of moving targets added to learning differential data.

In the following two sections, we give formalization for an application of neural networks to solving differential equations and applications from meteorology.

7.1.1 Formalization

We give substantially wider and more general formalization than one in [4]. With this formalization, we can have a unified and continuous view.

Let n be the function defined by a neural network, D^i be any linear or non-linear differential operator D^i and Ω_i be any set of points in the input space of the neural network. We consider the following set of constraints on the neural network.

$$D^i n(x) = 0 \quad \text{for } x \in \Omega_i,\ i = 1, ..., m \tag{7.1}$$

We do not make any distinction between differential equations, boundary conditions, and any other conditions as long as they are expressible in the above forms. We give several examples that can be expressed in the above forms.

For example, a Poisson equation on a domain Ω can be expressed as follows, where f is a function given on Ω.

$$\Delta n(x) - f(x) = 0 \quad \text{for } x \in \Omega \tag{7.2}$$

A boundary condition on the boundary $\partial\Omega$ of a domain Ω can be expressed as follows, where B is a differential operator and g is a function given on $\partial\Omega$.

$$Bn(x) - g(x) = 0 \quad \text{for } x \in \partial\Omega \tag{7.3}$$

In the case where Ω_i is a set of finite points and where D^i has the following form, then it is reduced to the constraints given in the standard back propagation for the target function f on the points in Ω_i.

$$n(x) - f(x) = 0 \quad \text{for } x \in \Omega_i = \{p_1, ..., p_n\} \tag{7.4}$$

In the case where Ω_i is a set of finite points as above and where D^i has the following form, then it is reduced to the constraints for neural networks learning differential data considered in Chapter 3 for the target function f on the points in Ω_i.

$$\frac{\partial^{N(\delta)} n}{\partial^\delta x}(x) - \frac{\partial^{N(\delta)} f}{\partial^\delta x}(x) = 0 \quad \text{for } x \in \Omega_i = \{p_1, ..., p_n\} \tag{7.5}$$

Learning algorithm of neural networks for constraints Equation (7.1) works as follows.

First, we take finite points $\{p_{i,1}, ...p_{i,l_i}\}$ from the each set Ω_i. Those points should be taken to represent the set Ω_i appropriately according to some measure. With learning constants α_i [1], we define the energy function from Equation (7.1) for the neural network as follows. [2]

$$E = \sum_{i=1}^{m} \frac{1}{2}\ \alpha_i\ \sum_{j=1}^{l_i}\ \left(D^i n(x)|_{x=p_{i,j}}\right)^2 \tag{7.6}$$

Let $D^i_{k,\delta}$ be the derivative of D^i with respect to the k-th unit in the δ net (the value net when $\delta = 0$.).

We update each weight w of the neural network with the update value Δw according to the gradient direction of the energy function (Equation (7.6)) with respect to the weight as in the following equation.

$$\Delta w = -\frac{\partial E}{\partial w} \tag{7.7}$$

Without any loss of generality, we can assume there are only one i and one j in Equation (7.6). Then Equation (7.7) can be modified as follows. (We omit $|_{x=p_{i,j}}$ in the equation for the readability.)

$$\begin{aligned} \Delta w &= -\frac{\partial E}{\partial w} = -\alpha_i\ D^i n(x)\ \frac{\partial\left(D^i n(x)\right)}{\partial w} \\ &= -\alpha_i \sum_{\delta \geq 0}\ D^i n(x)\ \frac{\partial\left(D^i n(x)\right)}{\partial x^\delta}\ \frac{\partial x^\delta}{\partial w} \end{aligned} \tag{7.8}$$

In order to calculate these update values for weights, we create value net and δ nets (Section 3.3) up to the highest differential order used in Equation (7.1).

Since the back propagation algorithms given in Section 3.7 are linear with respect to ϵ's given to the output layers, we use the same algorithm to calculate the weight updates (Equation 7.8) just by replacing the ϵ's for the output layers.

While the ϵ's for the output layers of standard neural networks learning differential data are given in Section 3.7.1), the ϵ's for the output layers are given in general differential cases are given as follows.

First, we give ϵ's for the value net. Let σ_k be the sigmoid function of the k-th unit in the output layer of the value net. Then $\epsilon^0_{k,p_{i,j}}$ for the unit is given by the following equation.

$$\epsilon^0_{k,p_{i,j}} = \alpha_i\ D^i n(x)|_{x=p_{i,j}}\ \frac{\partial D^i n(x)}{\partial x^0_k}|_{x=p_{i,j}}\ \sigma'_k \tag{7.9}$$

$\epsilon^\delta_{k,p_{i,j}}$ for the k-th unit in the δ net ($\delta \neq 0$) is given by the following equation.

$$\epsilon^\delta_{k,p_{i,j}} = \alpha_i\ D^i n(x)|_{x=p_{i,j}}\ \frac{\partial D^i n(x)}{\partial x^\delta_k}|_{x=p_{i,j}} \tag{7.10}$$

Then we let the value net and the δ networks of the neural network propagate forward at each point $p_{i,j}$ of the set $\{p_{i,j} \mid i = 1, ..., m,\ j = 1, ...l_i\}$.

[1] It is possible to change α_i for each $p_{i,j}$ with different j.

[2] The power 2 can be changed to any positive real number. It is even possible to use a function in place of x^2 as long as the function is differentiable and monotonically increasing.

These two equations, Equations 7.9 and 7.10, constitute the main part of the formalization. Weight updates are calculated by backward propagation of those ϵ's.

If we have further special conditions such as the situation in Section 6.2.2 where the value output is fixed and where the coefficients of differential equations depend on the value output, those can be incorporated into considerations as Section 6.2.2.

As one can see from Equations 7.9 and 7.10, ϵ's depend on the outputs of the value net and the δ nets even for linear differential equations. This is the difficulty, which is pointed out as "moving targets."

7.1.2 Applications from Meteorology

In this section, we give two possible applications from meteorology. We believe the problems from meteorology have a good mixture of differential equations and value data available for neural networks learning differential data.

The first application is about the temperature and the wind velocity. The temperature T and the wind velocity u satisfy the following PDE, where t stands for the time.

$$\frac{\partial T}{\partial t} + (u \cdot \nabla)T = 0 \tag{7.11}$$

It is very costly to add observation points for the temperature and the wind velocity. For this problem, we can use a neural network 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 partial differential equation, Equation (7.11), for the neural network to learn.

Another application is about the wind field. In meteorology, a method called MASCON (Mass-Consistent) is used to obtain the global information of the wind field from rather few (from several to several hundreds) observation points [35, 3]. Even though it is necessary to solve a complete fluid equation in theory, this method makes it possible to solve the problem only with observations of wind field at observations points and the constraints of mass consistency given by the following equation.

$$\nabla f(x) = 0 \tag{7.12}$$

This will be one of perfect applications of neural networks to differential equations as described in Section 7.1.1 and we consider to apply it in the near future.

7.2 Simulation of Human Arm Movements by the Minimum-Torque-Change Model

In this section, we describe a possible application of neural networks learning differential data to simulation of human arm movements by the minimum-torque-change model (see [38]). This is one of applications that require differential data of order higher than or equal to third.

The objective of such models is to explain for trajectory formation in human arm movements. Uno and Kawato [38] proposed a model that the trajectory gives minimum of the following value, where m stands for the number of joints involved in the motion and τ_i stands for the torque generated at the i-th joint.

$$C_\tau = \frac{1}{2}\int_0^T \sum_{i=1}^{m} \left(\frac{d\tau_i}{dt}\right)^2 dt \tag{7.13}$$

In [20], 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 a third-order differential constraint on the neural network. If we consider the 2-links in a plane, τ_1 and τ_2 and are given by the following equations [38], where θ_i is the position (angle) of i-th arm and M_i, L_i, S_i, I_i, B_i, and l_i are physical constants of i-th arm.

$$\begin{aligned}
\tau_1 &= (I_1 + I_2 + 2M_2L_1S_2\cos\theta_2 + M_2(l_1)^2)\ \theta_1'' \\
&\quad +(I_2 + M_2L_1S_2\cos\theta_2)\ \theta_2'' \\
&\quad -M_2L_1S_2(2\theta_1' + \theta_2')\ \theta_2'\sin\theta_2 + B_1\theta_1' \qquad (7.14)\\
\tau_2 &= (I_2 + M_2L_1S_2\cos\theta_2)\ \theta_1'' + I_2\theta_2'' \\
&\quad +M_2L_1S_2(\theta_1')^2\sin\theta_2 + B_2\theta_2' \qquad (7.15)
\end{aligned}$$

Using these equations, we can calculate the integrand of Equation (7.13) from the arm positions. We take enough points uniformly distributed along the time line that is the input space of the neural network. The same formalization proposed in Section 7.1 is used to minimize an approximation of Equation (7.13) by the finite sum of the integrand on the chosen points. If there are other constraints such as one that the arm has to be in a certain position at a certain time, these constraints can also be incorporated by the formalization proposed in Section 7.1.

In this way, the neural network used is much simpler and the constraint of minimum-torque-change can be implemented more naturally by the neural networks learning differential data.

Chapter 8

Higher Order Extensions to Radial Basis Function (RBF)

In this chapter, first, we summarize the formulae of Fourier transformation and introduce the related parts of the treatment of RBF by Poggio and Girosi with annotations necessary for our extension. Then we extend their results to the cases with differential error terms. The extended results include minimizing solutions, their best approximation properties, and C^l denseness of the RBFs with Green's functions.

8.1 Formulae

Here we summarize the formulae of Fourier transformation and other used in this chapter.

Fourier transformation $\mathcal{F}(f)$ of the function f is given as follows:

$$\mathcal{F}(f) = \frac{1}{(\sqrt{2\pi})^n} \int e^{-ix\omega} f(x) dx \tag{8.1}$$

Inverse Fourier transformation $\bar{\mathcal{F}}(f)$ of the function f is given as follows:

$$\bar{\mathcal{F}}(g) = \frac{1}{(\sqrt{2\pi})^n} \int e^{ix\omega} g(\omega) d\omega \tag{8.2}$$

The integration of Gaussian functions from $-\infty$ to ∞ is given as follows:

$$\int_{-\infty}^{\infty} e^{-\frac{\sigma^2 x^2}{2}} dx = \frac{\sqrt{2\pi}}{\sigma} \tag{8.3}$$

8.2 Treatment of RBF by Poggio and Girosi

In [31], Poggio and Girosi give relationships of Radial Basis Function with regularization theory. We summarize the related parts in this section.

Poggio and Girosi give detailed treatment of Radial Basis Function (RBF) in [31] and [8]. In [31], Poggio and Girosi describe relationships between regularization theory and Radial Basis Functions. Namely they show that minimizing solutions for squared value errors and a smoothing term have the form of RBFs.

In [8] Girosi and Poggio define the concept of best approximation property as "an approximation scheme has the best approximation property if in the set A of approximating functions there is one that has minimum distance from any given function of a larger set Φ." They showed that radial basis functions have the best approximation property since they are a linear sum of finite functions. They also showed that multilayer networks with sigmoid functions do have the best approximation property.

In the same paper [8] they showed the linear combinations of Gaussian functions are C^0 dense in continuous functions on a compact metric space.

In the rest of this section, we describe their derivation of the minimizing solutions. They give minimizing solutions for the following.

$$H[f] = \sum_i \{(y_i - f(x_i))^2 + \lambda ||Pf||^2 \tag{8.4}$$

Here P is a constraint functional, which is usually a differential operator and *lambda* is the regularization parameter.

The minimizing solution is given by Equation (5) in Section III of [31] as follows.

$$f(x) = \frac{1}{\lambda} \sum_{i=1}^{N} (y_i - f(x_i)) G(x; x_i) \tag{8.5}$$

Here $G(x; y)$ is the Green's function of $\hat{P}P$ where $\hat{P}$ is the adjoint of the differential operator P. Therefore $G(x; y)$ satisfies the following equation.

$$\hat{P}PG(x; y) = \delta(x - y) \tag{8.6}$$

The derivation of the above equation (8.5) is given as follows. By a variational method applied to Equation (8.4), minimizing solutions must satisfy the following.

$$\hat{P}Pf(x) = \frac{1}{\lambda} \sum_{i=1}^{N} (y_i - f(x)) \delta(x - x_i) \tag{8.7}$$

The integral transformation with the kernel as the Green function $G(x; y)$ produces Equation 8.5.

If P is a translation- and rotation-invariant operator, then $G(x; y) = G(||x - y||)$. Therefore minimizing solutions (Equation (8.5)) are linear combinations of radial basis functions.

Especially if P is chosen as follows,

$$\hat{P}P = \sum_{m=0}^{\infty} (-1)^m \frac{\sigma^{2m}}{m! 2^m} \nabla^{2m} \tag{8.8}$$

then the solution for the following equation (8.9) is given by Equation 8.10 where n is the dimension of the input space. We define $G(x; y) = G(x - y)$ using G.

$$\hat{P}PG(x) = \sum_{m=0}^{\infty} (-1)^m \frac{\sigma^{2m}}{m! 2^m} \nabla^{2m} G(x) = \delta(x) \tag{8.9}$$

$$G(x) = \sigma^n e^{-(x^2/2\sigma^2)} \tag{8.10}$$

The above is derived as follows. The Fourier transformation applied to the both sides of Equation (8.9) will give the following.

$$\begin{aligned}
\mathcal{F}(\sum_{n=0}^{\infty}(-1)^n \frac{\sigma^{2n}}{n!2^n}\nabla^{2n}G(x)) &= \sum_{n=0}^{\infty}(-1)^n \frac{\sigma^{2n}}{n!2^n}(-x^2)^n \mathcal{F}(G(x)) \\
&= e^{\frac{x^2\sigma^2}{2}}\mathcal{F}(G(x)) \qquad (8.11)\\
\mathcal{F}(\delta(x)) &= e^{i(x,0)} = 1 \qquad (8.12)
\end{aligned}$$

Therefore the next equation follows from Equation (8.9)).

$$\mathcal{F}(G(x)) = e^{-\frac{x^2\sigma^2}{2}} \tag{8.13}$$

Lastly, by applying the inverse Fourier transformation, $G(x)$ is obtained as follows.

$$\begin{aligned}
G(x) = \bar{\mathcal{F}}\mathcal{F}(G(x)) &= \bar{\mathcal{F}}(e^{-\frac{x^2\sigma^2}{2}}) \\
&= \frac{1}{(\sqrt{2\pi})^n}\int e^{i(x,w)}e^{-\frac{w^2\sigma^2}{2}}dw \\
&= \frac{1}{(\sqrt{2\pi})^n}e^{-(x^2/2\sigma^2)}\int e^{-\frac{\sigma^2}{2}(w-\frac{ix}{\sigma^2})^2}dw \\
&= \sigma^n e^{-(x^2/2\sigma^2)} \qquad (8.14)
\end{aligned}$$

8.3 Minimizing Solutions for the Cases with Differential Error Terms

We extend the results of [31] on RBF to the cases with differential error terms. We give the minimizing solution for the following functional H with the regularization parameter λ. [1]

$$H(f) = \sum_i \{(y_i - f(x_i))^2 + (\vec{y'}_i - \nabla f(x_i))^2\} + \lambda||Pf||^2 \tag{8.15}$$

Here $\vec{y'}_i = \{y'^1_i, ..., y'^n_i\}$, $\nabla f = \{\partial f/\partial x^1, ..., \partial f/\partial x^n\}$, and n is the dimension of the input space.

This equation (8.15) can be rewritten as follows.

$$\begin{aligned}
H(f) = \int \sum_i \Big\{ &\{(y_i - f(x))\}^2\delta(x - x_i) + \{(\vec{y'}_i - \nabla f(x))\}^2\delta(x - x_i)\Big\} \\
&+\lambda|Pf|^2 dx
\end{aligned} \tag{8.16}$$

[1] Minimizing solutions can be obtained the same way for the cases with the higher order differentials.

We apply the variational method to the equation.

$$\begin{aligned}\delta H(f) &= \int\Big\{\sum_i -\{(y_i - f(x))\delta(x-x_i)\}\,\delta f \\ &\qquad -\Big\{(\vec{y'}_i - \nabla f(x))\delta(x-x_i)\Big\}\nabla\delta f \\ &\qquad +\lambda < Pf, P\delta f > \Big\}dx \\ &= \int\Big\{\sum_i -\{(y_i - f(x))\delta(x-x_i)\} + \nabla\Big\{(\vec{y'}_i - \nabla f(x))\delta(x-x_i)\Big\} \\ &\qquad +\lambda(\hat{P}Pf)\Big\}\delta f dx\end{aligned} \tag{8.17}$$

$< , >$ stands for an inner product of a vector space.

Therefore the minimizing solution f must satisfy the following equation.

$$\hat{P}Pf = \frac{1}{\lambda}\sum_i\Big\{\{(y_i - f(x))\delta(x-x_i)\} - \nabla\Big\{(\vec{y'}_i - \nabla f(x))\delta(x-x_i)\Big\}\Big\} \tag{8.18}$$

As explained in the previous section 8.2, we apply the integral transformation with the kernel of Green's function satisfying Equation (8.6) to the both sides of Equation (8.18). Then we obtain the minimizing solution as follows.

$$f(x) = \frac{1}{\lambda}\sum_i\Big\{(y_i - f(x_i))G(x;x_i) + (\vec{y'}_i - \nabla f(x_i))\nabla G(x;x_i)\Big\} \tag{8.19}$$

As in the previous section 8.2, if we choose P as the operator satisfying Equation (8.8) , the Green's function $G(x;y)$ for $\hat{P}P$ is given by Equation (8.10). In this case, $\nabla G(x;y)$ is given by the following equation.

$$\nabla G(x;y) = -\frac{(x-y)}{\sigma^2}\sigma^n e^{-(x-y)^2/2\sigma^2} \tag{8.20}$$

Therefore Equation (8.19) for the minimizing solution f can be rewritten as follows for this case.

$$\begin{aligned}f(x) &= \frac{1}{\lambda}\sum_i\Big\{(y_i - f(x_i))\sigma^n e^{-\frac{(x-x_i)^2}{2\sigma^2}} \\ &\qquad -\sigma^{n-2}(\vec{y'}_i - \nabla f(x_i))(x-x_i)e^{-\frac{(x-x_i)^2}{2\sigma^2}}\Big\} \\ &= \frac{1}{\lambda}\sum_i\Big\{(y_i - f(x_i)) - \frac{1}{\sigma^2}(\vec{y'}_i - \nabla f(x_i))(x-x_i)\Big\} \\ &\qquad \times\sigma^n e^{-\frac{(x-x_i)^2}{2\sigma^2}}\end{aligned} \tag{8.21}$$

By determining the values for $\{f(x_i)\}_i$ and $\{f'(x_i)\}_i$ to satisfy the above equation on the points $\{x_i\}_i$, the function f is fixed.

It is essentially the same even when the differential error terms are of orders higher than one. For a general P, there are differentials of the Green's function

corresponding the error terms. For the P satisfying Equation (8.8), the minimizing function is the additions of the product of $(x - x_i)^\delta$ and the difference between the teacher signal and the value of the function at the point. [2]

8.4 Best Approximation Property

By fixing the number of sampling (data) points, the set of functions of the form given by Equation (8.21) has the best approximation property since the function is represented by the linear combination of finite functions.

8.5 C^l Denseness

Section 8.3 gives the minimizing solution of the form by the linear combination of the products of first-degree polynomials and Gaussian functions. Therefore we predicted that the set of linear combinations of the products of first or general degree polynomials and Gaussian functions is C^1 dense in the C^1 functions on a compact set. But we found out that a stronger fact holds. The set of linear combinations of Gaussian functions with a fixed variance is C^l $(l \geq 1)$ dense in the C^l functions on a compact set. This section proves the fact.

First, we extend the result in [8] by Poggio and Girosi that "the set of linear combinations of Gaussian functions with arbitrary centers and arbitrary variance." We show that "the set of linear combinations of Gaussian functions with arbitrary centers and a *fixed* variance." This result goes in accordance with the regularization theory. As we have seen in Section 8.2, the minimizing solution for the operator P satisfying Equation (8.8) is a linear combination of Gaussian functions with a *fixed* variance, which is determined by P.

In order to prove that linear combinations of Gaussian functions with a fixed variation are C^0 dense in C^0 functions, we need to show the following for a compact set $K \subset R$.

$$[\{e^{-(x-b)^2/2} \mid b \in R\}] = C^0(K) \tag{8.22}$$

Here $[F]$ is the closure of F. This can be proved by the Stone's theorem as follows.

We have the following two equations.

$$\begin{aligned} e^{-(x-b)^2/2} + e^{-(x+b)^2/2} &= e^{-\frac{1}{2}(x^2-2bx+b^2)} + e^{-\frac{1}{2}(x^2+2bx+b^2)} \\ &= e^{-\frac{b^2}{2}} e^{-\frac{x^2}{2}} (e^{bx} + e^{-bx}) \\ &= e^{-\frac{b^2}{2}} e^{-\frac{x^2}{2}} 2\cosh(bx) \end{aligned} \tag{8.23}$$

$$e^{-(x-b)^2/2} - e^{-(x+b)^2/2} = e^{-\frac{b^2}{2}} e^{-\frac{x^2}{2}} 2\sinh(bx) \tag{8.24}$$

Therefore proving C^0 denseness turns into proving $f(x)e^{\frac{x^2}{2}}$ is included in $[\{\cosh(bx) \mid b \in R\} \cup \{\sinh(bx) \mid b \in R\}]$ for $\forall f$.

[2] See the definition (Equation 3.2). δ is the one corresponding to the differential operator of the error term.

We use the Stone's theorem (c.f. [8] Theorem B.1) for the set $[\{\cosh(bx) \mid b \in R\} \cup \{\sinh(bx) \mid b \in R\}]$ above. The constant function is included as the case of $b = 0$. It is also obvious that there exists a function for any two distinct points such that the function can distinguish these points. The next equation and likes show that the set is an algebra,

$$\cosh bx \cosh b'x = \frac{1}{2}[\cosh(b + b')x + \cosh(b - b')x] \tag{8.25}$$

The Stone's theorem can be applied and the C^0 denseness is proved.

Next we prove that linear combinations of Gaussian functions with a fixed variation are C^1 dense in C^1 functions, As we have proved $[\{e^{-(x-b)^2/2} | b \in R\}] = C^0(K)$ and $e^{-(x-b)^2/2}$ can be approximated $[\{(x - b)e^{-(x-b)^2/2} \mid b \in R\}]$ in $C^0(K)$.

By absorbing the increments to x by b, finite sum approximation of the integration can be rewritten as follows.

$$\begin{aligned} e^{-(x-b)^2/2} &= -\int_{-\infty}^{x} (x - b)e^{-(x-b)^2/2} dx \\ &\simeq -\int_{-M}^{x} (x - b)e^{-(x-b)^2/2} dx \\ &\simeq \sum_{i=1}^{N} \alpha_i (x - b_i)e^{-(x-b_i)^2/2} \end{aligned} \tag{8.26}$$

This leads the inclusion in the center of the next relation.

$$C^0(K) = [\{e^{-(x-b)^2/2} \mid b \in R\}] \subset [\{(x - b)e^{-(x-b)^2/2} \mid b \in R\}] \subset C^0(K) \tag{8.27}$$

The equation on the left is from Equation (8.22) and the inclusion on the right is apparent.

Hence,

$$[\{(x - b)e^{-(x-b)^2/2} \mid b \in R\}] = C^0(K) \tag{8.28}$$

When $f \in C^1(K)$ is given, there is $g \in [\{(x - b)e^{-(x-b)^2/2} | b \in R\}]$ which is close to f' in C^0. Since $h(x) = \int g(x)dx \in [\{e^{-(x-b)^2/2} | b \in R\}]$ and since K is compact, h is close to f in C^1 if we take g close enough to f'. Therefore $[\{e^{-(x-b)^2/2} | b \in R\}] = C^1(K)$ is shown. For the cases of $l > 1$, the repetitions of the same arguments show $[\{e^{-(x-b)^2/2} | b \in R\}] = C^l(K)$.

In case of multi-dimensional input spaces, it can be proved similarly that $\{e^{-(\vec{x}-\vec{b})^2/2} \mid \vec{b} \in R^n\}$ is C^0-dense. Then by showing $\{x_1 ... x_n e^{-(\vec{x}-\vec{b})^2/2} \mid \vec{b} \in R^n\}$ is C^0-dense and by approximating the following integration with a finite sum

$$f(x, y) = \int \frac{\partial^n f}{\partial x_1 .. \partial x_n} dx_1 ... dx_n \tag{8.29}$$

C^1-denseness can be proved.

Similarly C^l-denseness ($l > 1$) for one- and multi-dimensional cases can be proved.

Chapter 9

Conclusion

In this chapter, we present a summary of contributions, a discussion of limitations, and suggestions for future works.

9.1 Contributions

In this dissertation, we investigated the framework to introduce constraints on differential data into neural networks as learning systems.

First, we introduced architecture and an algorithm for neural networks learning differential data of arbitrary order after a summary account of multilayer perceptrons. This completely localized algorithm enable multilayer perceptrons learn differential data of orders not only first but also higher than first. Such algorithms were previously non-existent.

Secondly, we described an implementation of the architecture and the algorithm above as computer programs and the preliminary experiments on neural networks learning differential data using the implementation. In the experiments, we showed that neural networks actually converge on differential data up to the third order for a non-trivial target function. Since no one has ever shown that the algorithm truly converges on differential data of orders higher than first, establishing the fact should help promote the application of neural networks learning differential data greatly.

Then we analyzed neural networks learning differential data. We reported the analyses such as comparison with extra pattern scheme, how learnings work, sample complexity, effects of irrelevant features, and noise robustness. These analyses should help judge when to use the neural networks learning differential data instead of other options.

We also described an application of neural networks learning differential data to continuous action generation in reinforcement learning. This application is actually an application of solving differential equations. The details of how neural networks are applied to solving differential equations and the results of experiments should help encourage the applications of neural networks learning differential data to other kinds of differential equations.

Thereupon we identified two more possible applications differential equations and simulation of human arm movements.

Finally, we described higher order extensions to radial basis function (RBF)

networks, another type of neural networks other than multilayer perceptrons. We gave minimizing solutions for the cases with differential error terms, best approximation property of the above solutions, and a proof of C^l denseness of RBF networks.

This dissertation as a whole lay the foundations for applications of neural networks learning differential data as learning systems. We gave detailed accounts of their architecture, algorithm, implementations, and applications in order to help promote their applications. The analyses provided should also help judge when to use the neural networks learning differential data instead of other options. The completely localized algorithm and its implementations will open up the whole range of possibilities of applications with their efficiency and their ability to handle differential data of arbitrary order.

9.2 Limitations

In this section, we describe the limitations of the results presented in this dissertation.

The limitations are followings:

- Analyses of neural networks learning differential data of orders higher than first are unexplored.
- Comparisons with other algorithms and methods that can incorporate knowledge in the forms of constraints on differential data are unexplored.
- The choices of parameters such as numbers of units in hidden layers, learning constants, and momentum are arbitrary, mostly by trial and error.
- The implementation is limited to layered neural networks with full connections between layers, while the algorithm itself is not limited to the specific architecture of neural network.
- Statistical aspects are largely unexplored in Chapter 4

9.3 Future Work

In this section, we provide suggestions for future work in the research area of neural networks learning differential data.

Even though we have shown the possibilities of practical applications for neural networks learning differential data, there are still items to be explored to make them really useful tools. These items include the followings:

- Development and implementation of algorithms or methods that guarantee (to some degree) the stable convergence, especially development of appropriate choosing methods of parameters.
- Comparisons with other algorithms and methods that can incorporate knowledge in the forms of constraints on differential data.
- More analyses, especially on learning with differential data of higher orders and statistical aspects of the learning

- Framework and implementations of neural networks to handle general differential equations
- More applications of neural networks learning differential data such as those described in Chapter 7

Bibliography

[1] Hideki Asou. *Neural Network Information Processing (In Japanese)*. Sangyo-Tosho, March 1988.

[2] Tianping Chen and Hong Chen. Approximation capability to functions of several variables, nonlinear functionals, and operators by radial basis function neural networks. *IEEE Transaction on Neural Networks*, 6(4):904–910, july 1995.

[3] M. H. Dickerson. Mascon-mass consistent atmospheric flux model for regions with complex terrain. *J. Appl. Meteor.*, 17:241–253, 1978.

[4] M. W. M. G. Dissanayake and N. Phan-Thien. Neural-network-based approximations for solving partial differential equations. *Communications in Numerical Methods in Engineering*, 10:195 – 201, 1994.

[5] Ken-ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.

[6] A. Ronald Gallant and Halbert L. White Jr. On learning the derivatives of an unknown mapping with multilayer feedforward networks. *Neural Networks*, 5:129–138, 1992.

[7] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–742, 1984.

[8] Federico Girosi and Tomaso Poggio. Networks and the best approximation property. *Biological Cybernetics*, 63:169–176, 1990.

[9] OMG (Object Management Group). Corba. *http://www.omg.org/*.

[10] G. E. Hinton and T. J. Sejnowski. Learning and relearning in boltzmann machines. In *Parallel Distributed Processing*, volume I, chapter 7. The MIT Press, 1986.

[11] Neville Hogan. An organizing principle for a class of voluntary movements. *The Journal of Neuroscience*, 4(11):2745 – 2754, 1984.

[12] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA*, 79:2554–2558, April 1982.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3:551–560, 1990.

[14] Hajime Kimura, Kazuteru Miyazaki, and Shigenobu Kobayashi. A guideline for designing reinforcement learning systems. *Journal of SICE*, 38(10):618–623, October 1999.

[15] S. Kirkpatrik, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598), May 1983.

[16] Isaac Elias Lagaris, Aristidis Likas, and Dimitrios I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987 – 1000, September 1998.

[17] Hyuk Lee and In Seok Kang. Neural algorithm for solving differential equations. *Journal of Computational Physics*, 91:110 – 131, 1990.

[18] Xin Li. On simultaneous approximations by radial basis function neural networks. *Applied Mathematics and Computation*, 95:75–89, 1998.

[19] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1989.

[20] 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*, MBE 88-169:79 – 84, 1989.

[21] R. Masuoka, N. Watanabe, A. Kawamura, Y. Owada, and K. Asakawa. Neurofuzzy system – fuzzy inference using a structured neural network. *Proceedings of the International Conference on Fuzzy Logic & Neural Networks Iizuka, Japan, July*20 – 24, 1990, pages 173–177, july 1990.

[22] Ryusuke Masuoka. Noise robustness of ebnn learning. In *Proceedings of 1993 International Joint Conference on Neural Networks*, volume 2, pages 1665–1668. IEEE, 1993.

[23] Ryusuke Masuoka. Neural networks learning differential data. *IEICE Transactions*, to appear.

[24] Ryusuke Masuoka and Michio Yamada. Neural networks which learn from differential data. *IEICE Technical Report*, NC94-66:49–56, 1995.

[25] 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.

[26] Kazuteru Miyazaki, Hajime Kimura, and Shigenobu Kobayashi. Theory and applications of reinforcement learning based on profit sharing. *Journal of JSAI*, 14(5):40–47, September 1999.

[27] A. Okada, R. Masuoka, and A. Kawamura. Knowledge-based neural network — using fuzzy logic to initialize a multilayered neural network and interpret postlearnig results. *Fujitsu Scientific and Technical Journal*, 29(3):217–226, September 1993.

[28] H. Okada, R. Masuoka, N. Watanabe, and A. Kawamura. Fuzzy-logic-based neural networks – initializing and explaining multilayered neural networks with fuzzy logic –. *submitted to Trans. on Neural Networks, Special Volume (Neural Networks Research in Japan) of the Progress in Neural Networks Series*, pages –.

[29] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3:246–257, 1991.

[30] J. Park and I. W. Sandberg. Approximation and radial-basis-function networks. *Neural Computation*, 5:305–316, 1993.

[31] Tomaso Poggio and Federico Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497, september 1990.

[32] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1988.

[33] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland. A general framework for parallel distributed processing. In *Parallel Distributed Processing*, volume I, chapter 2. The MIT Press, 1986.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume I, chapter 8. The MIT Press, 1986.

[35] Y. K. Sasaki. *Lecture notes on variational methods for environmental analysis and prediction problems, Severe Storm Research Notes.1*. Disaster Prevention Research Institute, Kyoto University, 1979.

[36] 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.

[37] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. The MIT Press, 1998.

[38] Y. Uno and M. Kawato. Dynamic performance indices for trajectory formation in human arm movements. *IEICE Technical Report*, NC 94-28:33 – 40, 1994.

[39] B. Ph. van Milligen, V. Tribaldos, and J. A. Jiménez. Neural network differential equation and plasma equilibrium solver. *Physical Review Letters*, 75(20):3594 – 3597, November 1995.

[40] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. *1960 IRE WESCON Convention Record*, pages 96 – 104, 1960.

[41] Stephen Wolfram. *The Mathematica Book, Fourth Edition*. Cambridge University Press, 1999.