01 May 1990

Input Data Pattern Encoding for Neural Net Algorithms

Hyeoncheol Kim

George Winston Zobrist
Missouri University of Science and Technology

INPUT DATA PATTERN ENCODING FOR NEURAL NET ALGORITHMS

*H. Kim and G. W. Zobrist

CSc-90-3

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401 (314)341-4491

*This report is substantially the M.S. thesis of the first author, completed May 1990.

ABSTRACT

First, a brief overview of neural networks and their applications are described, including the BAM (Bidirectional Associative Memory) model.

A bucket-weight-matrix scheme is proposed, which is a data pattern encoding method that is necessary to transform a set of real-world numbers into neural network state numbers without losing the pattern property the set has. The scheme is designed as a neural net so that it can be combined with other data processing neural nets. The net itself can be used as a bucket-sorting net also. This shows that traditional data structure problems can be an area that neural networks may conquer, too.

A simulation of the net combined with the BAM model on a digital computer is done to show performance of the proposed data encoding method with both non-numerical image pattern and numerical data pattern examples.

Table of Contents

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

LIST OF TABLE viii

SECTION

I Artificial Neural Networks **1**

A Fundamentals 1

1 Neurons 1

2 Network operation 2

3 Differences of neural computing from other computing . . . 1

B History, Properties and Their Applications 5

C Bidirectional Associative Memory (BAM) 7

1 Introduction 7

2 Training 8

3 Recalling 10

II Data Encoding **12**

A Input Formats 12

B Encoding schemes 14

1 Binary scheme 16

2 Bucket-Weight Matrix (BWM) scheme 17

III Binary Pattern Net **22**

A Training step 22

B Recalling step (Bucket sorting) 25

C Analysis 28

1 Training time and memory used 28

2 Noise filtering 29

IV Examples **32**

A Example with a numeric data pattern 32

B Example with a non-numeric image pattern 34

V Conclusion and further research **43**

APPENDICES **45**

A Programming of BAM model 45

B Programming of Bucket-weight matrix scheme 52

C Programming of binary pattern net (bucket sorting net) 56

D The result of bucket sorting using bucket sorting net 59

E The data used in examples in chapter IV-A 65

REFERENCES **71**

VITA **72**

List of Figures

1 Biological neuron 1

2 Artificial neuron with activation function 2

3 Topology of a BAM, showing the two fields of neurons connected by synapses 8

4 A taxonomy of nine neural nets 12

5 Backpropagation nets testing with different input formats 14

6 Image recognition 16

7 Training step of binary-pattern net. The matrix C is the bucket-weight matrix. N numbers are encoded into $n * b$ numbers 23

8 Recalling step of a binary-pattern net. This step is used when bucket sorting is required. The weight matrix C is from a bucket-weight matrix. 26

9 The net of the recalling step for a bucket sorting. The link weights are the key of the sorting. 27

10 Bucket sorting time (in milli-seconds) for the binary-pattern net. (a) when the number of data is fixed at 1000 and (b) when the number of buckets is fixed at 100 27

11 Three standard measures of scalability; problem size can be from training patterns, neurons, or synapses 29

12 Trade-off between accuracy of recognition and noise-tolerance by varying bucket sizes 31

13 Matrix A is binary pattern matrix for company A, and matrix B is for company B. 34

14 Graphs of values in table VI and results of recalling with different inputs 35

15 Binary data extracting from a visual image pattern 36

16 Two input patterns 36

17 Two possible noisy patterns of pattern 2 37

18 Input pairs for training 38

19 Recalling of noisy patterns and its incorrect output 39

20 Extracting vector values from image pattern 1 in figure 16 39

21 Recalling of noisy patterns and its correct outputs 11

List of Tables

I Input data and its target output for backpropagation test 13

II Results when 2 different types of inputs are used 15

III 3 different types of inputs and desired outputs for 2 patterns trained by backpropagation 19

IV 3 different types of distorted inputs for 2 patterns 20

V Results of recalling of the trained net 21

VI Weekly closing prices of company A and B 33

VII Noise rates of two methods 41

I Artificial Neural Networks

A Fundamentals

1 Neurons

Artificial Neural Networks are biologically inspired. Artificial Neural networks are composed of elements that perform in a manner that is analogous to the most elementary functions of the biological neuron. Figure 1 shows the corresponding components of the brain which inspired artificial neural networks.

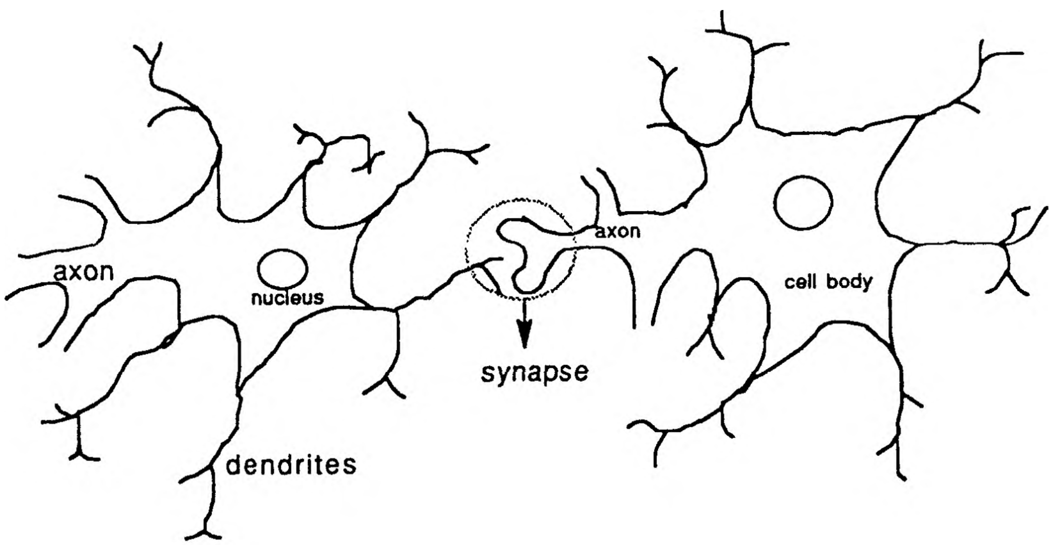


Figure 1: Biological neuron

The *neuron* is the fundamental cellular unit of the nervous system and in particular the brain. Its *nucleus* is a simple processing unit which receives and combines signals from many other neurons through input paths called *dendrites*. If the combined signal is strong enough, it activates the firing of the neuron which produces an output signal; the path of the output signal is called the *axon*.

An estimated 10^{11} neurons participate in 10^{15} interconnections over transmission paths in the human brain. The axon (output paths) of a neuron splits up and connects to dendrites (input paths) of other neurons through a junction referred to as a ***synapse***. The amount of signal transferred depends on the ***synapse strength*** of the junction. This synaptic strength is what is modified when the brain learns, and the synapse can be thus considered the basic memory unit of the brain.

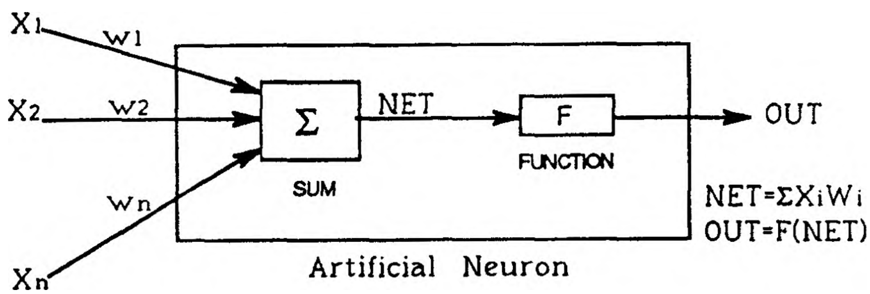


Figure 2: Artificial neuron with activation function

The artificial neuron was designed to mimic the first-order characteristics of the biological neuron. The artificial neuron has many input paths (dendrites) and combines the values on these input paths. The combined input is then modified by an activation function. This transfer function can be a threshold function or a continuous function of the combined input. The value output by the transfer function is generally passed directly to the output path of the neuron connected to input paths of other neurons through connection ***weights*** which correspond to the synaptic strength of neural connections. Figure 2 illustrates the above description.

2 Network operation

There are two main steps in the operation of a network – *Learning* and *Recall*. *Learning* is the process of adapting the connection weights according to a predetermined procedure so that each input vector produces the desired output vector. Learning algorithms are categorized as supervised and unsupervised. *Supervised* *learning* requires the pairing of each input vector with a target vector representing the desired output. For each input vector, a desired output vector is presented to the system, and the network gradually configures itself to achieve that desired input/output mapping. Hopfield nets[10] and perceptrons are trained with supervision. Such learning is generally some variation on one of three types :

1. *Hebbian learning* where a connection weight is incremented if the product of the excitation levels of both the input and desired output are high. That is, a neural pathway is strengthened each time it is used. In symbols:

   $$w_{ij}(n+1) = w_{ij}(n) + \alpha OUT_i OUT_j$$

   where

   $w_{ij}(n) =$ the value of a weight from neuron i to neuron j prior to adjustment

   $w_{ij}(n+1) =$ the value of a weight from neuron i to neuron j after adjustment

   $\alpha =$ the learning rate coefficient

   $OUT_i =$ the output of neuron i and input to neuron j

   $OUT_j =$ the output of neuron j

2. *delta rule learning* based on reducing the error between an input and its desired output. In symbols:

   $$\delta = T - A$$

   $$\Delta_i = \eta \delta x_i$$

   $$w_i(n+1) = w_i(n) + \Delta_i$$

   where

   $\delta =$ the difference between the desired output T and the actual output A

   $\eta =$ learning rate coefficient

   $\Delta_i =$ the correction associated with the *ith* input x_i

$w_i(n+1)$ = the value of weight i after adjustment

$w_i(n)$ = the value of weight i before adjustment

3. *competitive learning* in which processing elements compete among each other and the one which yields the strongest response to a given input modifies itself to become more like that input.

Unsupervised learning requires no target vector for the output. The learning algorithm modifies network weights to produce output vectors that are consistent. Nets trained without supervision, such as the Kohonen's feature-map forming nets[9], are used as vector quantizers or to form clusters. No information concerning the correct class is provided to these nets during training.

Recall refers to how the network globally processes an input vector and creates a response of an output vector.

3 Differences of neural computing from other computing

Artificial neural networks exhibit a surprising number of the brain's characteristics. The following are some of their capabilities and differences from traditional computing.

1. *learning by examples* : neural networks generate their own rules by learning from being shown examples.

2. *distributed associative memory* : the difference is the way to store information. Neural computing memory is both distributed and associative. The connection weights are the memory units of a neural network. Two properties can be obtained here.

   (a) *generalization* : once trained, a network's response can be, to a degree, insensitive to minor variations in its input. This ability to see through noise and distortion to the pattern that lies within is vital to pattern recognition in a real-world environment.

(b) *fault tolerance* : because information is not contained in one place, but is distributed throughout the system. Neural nets can survive the failure of some nodes.

3. *parallelism* : neural architectures provide a natural model for parallelism since each neuron is an independent unit. A massively parallel architecture like the human brain can solve serially operated computer's slowing-down problems in many applications.

4. *pattern recognition* : neural computing systems possess the ability to match large amounts of input information simultaneously and then generate categorical or generalized output as well as the ability to learn and build unique structures for a particular problem.

5. *abstraction* : some artificial neural networks are capable of abstracting the essence of a set of inputs. In one sense, with this ability to extract an ideal from imperfect inputs, it might learn to produce something that it has never seen before.

B History, Properties and Their Applications

When the perceptron was first developed in the 1950s by Frank Rosenblatt, it created a considerable sensation. Artificial Intelligence was also affected, as the perceptron was based on biological and psychological approaches to computing architectures. AI was firmly rooted in logic and rules.

In the middle of 1960s, Minsky and Papert showed conclusively one vital flaw in the perceptron theory. They proved that it was incapable of doing a class of problems known as the exclusive–OR. As a result, artificial neural nets and other biologically-based approaches to computing lapsed for nearly two decades until backpropagation was invented in the early 1980s, providing a systematic means for training multilayer networks, thereby overcoming limitations presented by Minsky. After that, interest in artificial neural networks has grown rapidly over the past few years. Backpropagation has been used in many impressive demonstrations of artificial neural network capabilities such as data compression, signal processing, noisy filtering, etc. Many other network algorithms have been developed that have specific advantages, such as counterpropagation networks, Hopfield nets, associative memory, adaptive resonance theory, Boltzmann machines, cognition and more. In general, those nets belong to three big models according to their properties. A brief description of the three models[5] and their applications are given below :

1. **Associative Memory Model** offers many of the computational capabilities of neural networks. This model is a mapping from data to data, a mathematical abstraction from the familiar associative structure of human and animal learning. This model can be used for simple visual processing. The network can *associate* or map variations of particular patterns. The model gives the network fault tolerance because input patterns are stored in a distributed fashion throughout the network.

2. **Optimization Model** gives solutions to very difficult combinatorial optimization problems, such as the traveling salesman problem which is an NP (nondeterministic polynomial) complete problem. This property can be used to perform many difficult tasks in computer vision, such as computing motion and brightness perception, surface interpolation, and localizing edges. This property is very important for real-time vision systems such as adaptive flight-control systems.

3. **Self-organization** lets neural network based systems adapt to unpredictable changes in their environment and unexpected situations that cannot be described mathematically. Self-organization is effective for dealing with problems that have a complicated or impossible-to-define algorithm, and it can be used for robotic control. Self-organization will enable control with inaccurately known mechanical structures.

Combinations of neural-network properties will become even more powerful. For example, a combination of self-organization and optimization can be useful for robotic-control path minimization and collision avoidance.

Neurocomputing is very powerful in many problems that humans do easily and, seemingly without thinking, such as language processing, data compressing, character recognition, pattern recognition, signal processing such as prediction and system modeling, financial and economic modeling, and optimization problems. Adaptive expert systems can be a good commercial product, which take advantage of the prodigious pattern recognition, self-programming, learning, and fault tolerance capabilities of neural networks by using them as front ends for rule-based and knowledge-based expert systems.[8] As neural networking systems develop, they will play an important role in furthering many intelligent information processing technologies and applications.

C Bidirectional Associative Memory (BAM)

1 Introduction

Human memory is often associative; one thing reminds us of another, and that, of still another. The BAM is a simple nonlinear neural-network associative memory that recalls or content-addresses stored associations (x,y) by minimizing a system energy. The BAM accepts an input vector on one set of neurons and produces a related output vector on another set as the input *rolls* into the nearest energy minimum. Figure 3 shows the basic BAM configuration.[3] The BAM is a two-layer feedback network of interconnected neurons. Patterns are stored in the synapses between the neurons. The state of the neurons represents a short-term memory(STM), as it may be changed quickly by applying another input vector. The values in the weight matrix form a long-term memory(LTM) and are changeable only on a longer time scale, using techniques to be discussed later. The short-term memory (STM) reverberations gradually seep pattern information into long-term memory (LTM), the synapses between the neurons. Associative memories are fundamental computing structures of artificial neural system and can be naturally implemented on neurocomputers.

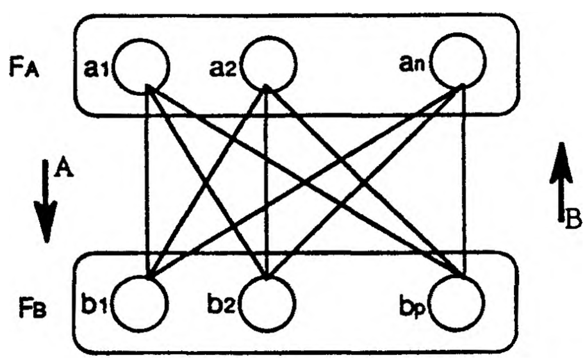


Figure 3: Topology of a BAM, showing the two fields of neurons connected by synapses

2 Training

In a BAM, all synaptic information is contained in an n-by-p connection matrix M which is long-term memory. With matrix M betweenF_A and F_B, all inputs quickly map to a pattern of stable reverberation. The training in BAM is Hebbian learning. A BAM learns a particular set of associations $(A_1, B_1), \ldots, (A_m, B_m)$ by summing bipolar correlation matrices. The learning scheme tends to place distinct associations (A_i, B_i) at or near local energy minima. The number of patterns to be stored and recalled cannot be more that the number n of neurons in F_A or the number p of neurons in F_B. BAM correlation learning improves if bipolar vectors and matrices are used instead of binary vectors and matrices. Bipolar matrices are binary matrices with -1s replacing 0s. The bipolar version of a binary pattern $A_1 = (1\ 0\ 1\ 0\ 1\ 0)$ is $X_1 = (+1\ -1\ +1\ -1\ +1\ -1)$. Assume that X and Y will denote the respective bipolar version of the binary vectors A and B.

The BAM learning scheme converts each binary pattern pair (A_i, B_i) to a bipolar pair (X_i, Y_i), converts them into a matrix $X_i^T Y_i$, and then adds up the matrices $M = X_1^T Y_1 + X_2^T Y_2 + \ldots + X_m^T Y_i$ where the colume vector X_i^T is the vector transpose of the row vector X_i.

Suppose two binary associations are trained in the BAM

$$A_1 = (1\ 0\ 1\ 0\ 1\ 0) \quad B_1 = (1\ 1\ 0\ 0)$$

$$A_2 = (1\ 1\ 1\ 0\ 0\ 0) \quad B_2 = (1\ 0\ 1\ 0).$$

Convert these binary pairs to bipolar pairs:

$$X_1 = \begin{pmatrix} 1 & -1 & 1 & -1 & 1 & -1 \end{pmatrix} \quad Y_1 = \begin{pmatrix} 1 & 1 & -1 & -1 \end{pmatrix}$$

$$X_2 = \begin{pmatrix} 1 & 1 & 1 & -1 & -1 & -1 \end{pmatrix} \quad Y_2 = \begin{pmatrix} 1 & -1 & 1 & -1 \end{pmatrix}.$$

Convert these bipolar vector pairs to two bipolar correlation matrices and compute the weight matrix $M = X_1^T Y_1 + X_2^T Y_2$:

$$\begin{pmatrix} 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & -2 \\ 0 & -2 & 2 & 0 \\ 2 & 0 & 0 & -2 \\ -2 & 0 & 0 & 2 \\ 0 & 2 & -2 & 0 \\ -2 & 0 & 0 & 2 \end{pmatrix}$$

The matrix element m_{ij} indicates the symmetric synapse between neurons a_i and b_i. The synapse is excitatory if $m_{ij} > 0$, inhibitory if $m_{ij} < 0$. An association (A_i, B_i) from M can be erased by adding $-X_i^T Y_i$ to M. The BAM energy E of association or state (A_i, B_i) is $-A_i M B_i^T$. In the example, $E(A_1, B_1) = E(A_2, B_2) = -6$.

3 Recalling

Suppose that an input pattern A is presented to BAM field F_A. The n neurons across F_A have their binary values 1 or 0. Each neuron a_i in F_A fans out its binary value across the p pathways and the synaptic value m_{ij} multiplies the binary value a_i. Each neuron b_j in F_B receives a fan-in of input products $a_i m_{ij}$ from each of its n synaptic connections.

$$o_j = \sum_{i=1}^{n} a_i m_{ij}$$

We compute the output vector, given input vector A_1.

$$\begin{aligned} O = A_1^T M &= \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & -2 \\ 0 & -2 & 2 & 0 \\ 2 & 0 & 0 & -2 \\ -2 & 0 & 0 & 2 \\ 0 & 2 & -2 & 0 \\ -2 & 0 & 0 & 2 \end{pmatrix} \\ &= \begin{pmatrix} 4 & 2 & -2 & -4 \end{pmatrix} \end{aligned}$$

Now threshold this vector by applying the threshold rule :

$$b_i = \begin{cases} 1, & \text{if} o_i > 0, \\ 0, & \text{if} o_i < 0, \\ unchanged, & \text{if} o_i = 0 \end{cases}$$

Then threshold(O) = (1 1 0 0) which is the desired output B_1.

Neuron b_j then fans out its output signal across the n pathways m_{ij} to each neuron a_i in F_A. Each a_i then generates its binary signal from all its summed inputs and sends it back to F_B. And round and round the BAM goes. Each pass around the loop causes the system to descend toward an energy minimum, the location of which is determined by the values of the weights.

The BAM is error-correcting. For example, if an incomplete or partially incorrect vector is applied at A, the network tends to produce the closest memory at B, which may be required, but the network converges to the nearest stored memory. For example, the input A = (0 1 1 0 0 0) is just A_2 perturbed by 1 bit. Then AM = (2 -2 2 -2) $\Longrightarrow$ (1 0 1 0) = B_2, and thus A evokes the resonant pair (A_2, B_2).

II Data Encoding

A Input Formats

A taxonomy of nine important neural nets is presented in figure 4.[4] This taxonomy is first divided between nets with binary and continuous valued inputs. Below this, nets are divided between these trained with and without supervision.

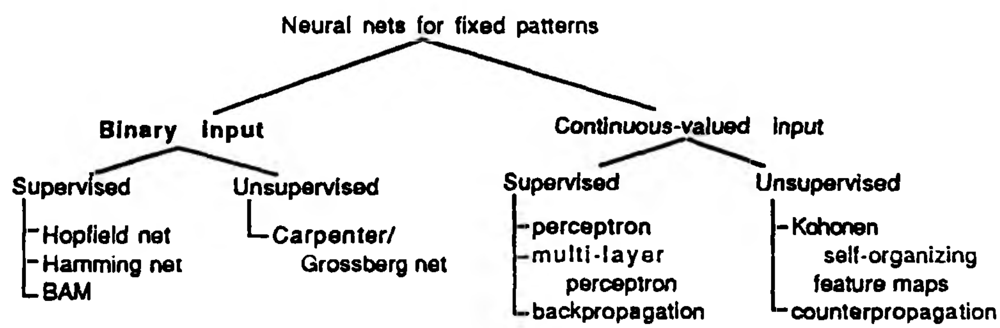


Figure 4: A taxonomy of nine neural nets

Many association memories and classifiers, such as the Hopfield net, Hamming net, bidirectional associative memory and Carpenter/Grossberg net use binary data as their inputs. These nets are most appropriate when exact binary representations are possible as with black and white images where input elements are pixel values, or with ASCII representation of each character. These nets are less appropriate when input values are actually continuous, because a fundamental representation problem must be addressed to convert the analog quantities to binary values.[4]

Although the other nets, such as the perceptron, backpropagation and Kohonen training nets can use continuous input data, they produce better classified results when they use binary inputs. As the data go through hidden layers, they are compressed into the range of 0 $\sim$ 1 by a sigmoid function or hard limit threshold of each neuron. So using binary inputs lets each neuron make a more accurate decision. Three different data patterns are tested on a backpropagation net that can use continuous inputs to see if binary inputs lead the net to a more accurate classification than continuous inputs.

Table I: Input data and its target output for backpropagation test

pattern	continuous-valued input	desired output
pattern A	45 13 32 5 4	1 0 0 0 0
pattern B	50 32 23 45 49	0 0 1 0 0
pattern C	4 19 37 24 11	0 0 0 0 1

Table I shows the continuous-valued input data and desired outputs to be tested on a backpropagation net. To represent the continuous-valued inputs with binary inputs, binary digits are used. That is, 45 is (0 0 0 0 1 0 1 1 0 1) in binary. For each decimal number, 10 digits are used in binary in this example. So 50 neurons are used in an input layer when binary inputs are used, while 5 neurons are used when continuous-valued inputs are used. Besides the input layer, there are 3 more layers, 2 hidden layers which have 4 neurons each and an output layer which has 5 neurons. For both cases, the net is trained 10000 times, respectively. The backpropagation nets are shown in figure 5. The result is shown in table II.

As seen in table II, although pattern C is classified correctly, pattern A and pattern B are not classified when continuous-valued inputs are used in backpropagation. Since pattern A and B produce the same outputs (0.5 0 0.5 0 0) even after 10,000 training steps, there is no hope of convergence into (1 0 0 0 0) and (0 0 1 0 0), respectively. On the contrary, when binary inputs are used, all three

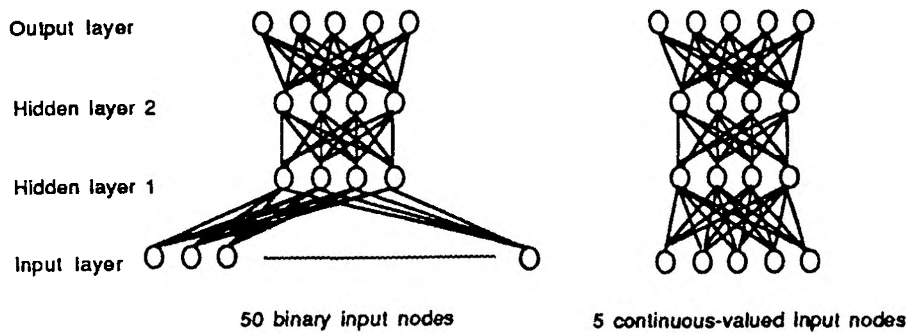


Figure 5: Backpropagation nets testing with different input formats

patterns, A, B, and C are classified correctly and produce almost the same actual outputs as those desired after 10,000 training steps. So using binary inputs helps the net to recognize patterns more accurately than using continuous valued inputs. Binary inputs provide the net a better degree of fault tolerance or robustness than continuous inputs. In the example above, five real-valued numbers are distributed to 50 binary valued neurons. So damage to a few nodes out of the 50 nodes or links need not impair the overall performance significantly. Damage to few nodes out of 5 real valued nodes will cause a great damage to performance of the net.

B Encoding schemes

Many current neural net algorithms are developed for pattern recognition. It is natural to feed the nets a pattern for its input, and the nets will consider a set of input data as one pattern. How to extract data from an input pattern is important, no matter whether it is an image pattern or a numeric valued pattern.

If the input pattern is an image pattern, it is rather easy to get binary data from the image pattern. Figure 6 shows a set of binary inputs for the letter A drawn on a grid.[1] If a line passes through a square, the corresponding neuron's

Table II: Results when 2 different types of inputs are used

inputs	testing patterns	actual outputs				
continuous inputs	pattern A	**0.51**	0.01	**0.49**	0.01	0.01
	pattern B	**0.51**	0.01	**0.49**	0.01	0.01
	pattern C	0.01	0.0	0.0	0.0	**0.99**
binary inputs	pattern A	**0.99**	0.01	0.0	0.01	0.01
	pattern B	0.0	0.0	**0.99**	0.0	0.01
	pattern C	0.01	0.0	0.01	0.0	**0.99**

input is one; otherwise, that neuron's input is zero.

In many real-world problems of practical interest, the patterns to be trained and recalled in a neural net are described by a set of decimal numbers rather than ready-to-use binary numbers from a visual image pattern. For example, suppose that we want to implement a system that analyzes the stock price in a stock market using a neural network. The data to be used in the neural net algorithm is weekly closing stock prices of a company for a certain period, which is an array of decimal numbers, not binary numbers. And the data set has its own unique pattern. It is more natural to tell a neural net algorithm the pattern, not just data itself because the current algorithms are developed to be good at pattern recognition. There could be several different types of patterns according to the properties and purposes of the applications. The pattern used here is one that depends on the trend (up or down) of data value. For example, a set of data { 2, 4, 8, 11 } is real data which has an "ascending order" pattern. { 12, 14, 23, 40 } has a "ascending order" pattern also. But it is a little different from the former one because the ratio of rate is different. Even though there are some applications in which a data

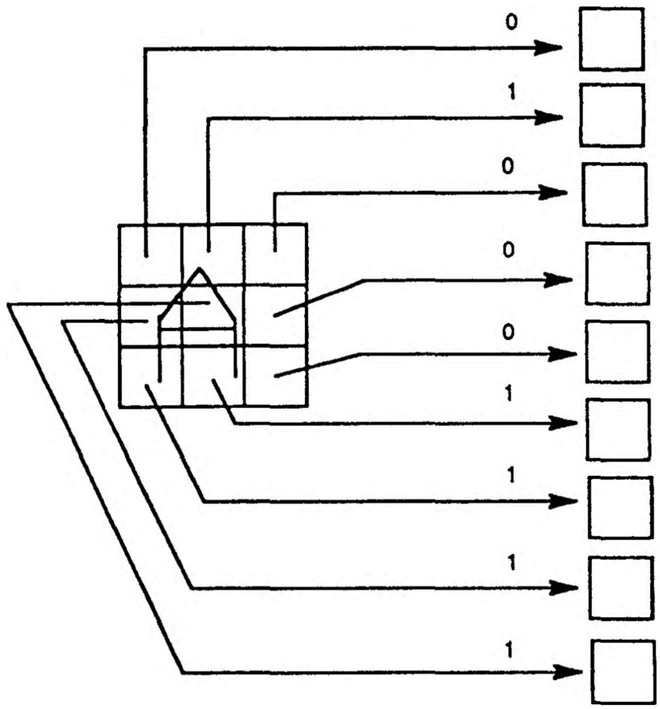


Figure 6: Image recognition

value itself is important, most applications use a pattern of the data, not just the value itself. Therefore, we must have a means to encode real-valued numbers into binary neural state numbers *without* losing its pattern. The converted binary data should have the same pattern that the real data had.

A new scheme is proposed to represent real-valued numbers by neuron state binary numbers, which is essential in solving numerical problems on neural networks. One way of mapping the positive integer space onto the neuron state space will be shown before the new transformation method is considered. Comparisons of the two schemes with examples will be considered also.

1 Binary scheme

The easiest way of converting numbers into binary data is to use binary digits. For example, 9 is expressed by 1001. This scheme is used in examples in the previous section to show the performance of binary inputs over continuous valued inputs.

A number N uses $\lceil \log_2(N+1) \rceil$ bits to express itself in binary. If the number of elements of a data set is D, then $D * \lceil \log_2(N+1) \rceil$ bits are required. Despite the simplicity, this scheme has the following problems when it is used in neural net algorithms.

- **Not noise-tolerant :** Even a small distortion in a value might give rise to a large error in the binary number represented. For example, if a number 7 (0111) decreases by 1 to 6 (0110), only 1 digit is contaminated in the binary number. What if the number 7 increases by 1 to 8? The 0111 becomes 1000. All of the 4 digits are contaminated completely even though the decimal number has just a single digit of noise.

- **Not fault-tolerant :** Even a single failure in a highly significant bit gives rise to a large error in the number represented. For example, if the most significant bit of (1 0 1 1 0 1) which is 45 in decimal is corrupted, it becomes (0 0 1 1 0 1) which is 13.

- **Obscure whole pattern :** It is a set of data, not an element of the set, that has a pattern. Converting each decimal number into a binary number can not represent the pattern that the set has. You can not see the forest for the trees.

2 Bucket-Weight Matrix (BWM) scheme

Each element from a data set with a certain range of values is expressed with a number of buckets. The bucket corresponding to the element's value is set to 1, and the rest are set to 0. The number of buckets is determined by the degree of distortion tolerance to be used in an application. For example, if the input values of a pattern range from 0 to 8 and the number of buckets is 4, then $(\textit{the range of input values})/(\textit{the number of buckets}) = (8-0)/4 = 2$ is the size of each bucket as well as the size of distortion tolerated. The first bucket corresponds to values from 0 to 2, and the second bucket is for values from 2 to 4, and so on. Since there are 4 buckets, each number is represented by 4 digits, and each digit represents each bucket. If the bucket contains the number, it is set to 1, and the other buckets, that is, digits are set to 0. A number 4 is represented by 0100, and 7 is represented by 0001 because the number 4 belongs in second bucket and the number 7 in fourth bucket. So the elements whose values are in the same bucket's range are represented the same. A number 3 is represented by 0100 just like a number 4 because the second bucket represents elements whose values are more than 2 and less than or equal to 4. Noise filtering ability can be obtained in this way.

Doing this for every element makes a so-called bucket-weight matrix for the data set. The matrix has a property of the pattern that the data set had. Besides the pattern keeping ability, this scheme has a good noise-filtering ability and works even with negative numbers and fractional numbers. This scheme will be discussed in the next chapter in more detail.

The following example will show how well the two schemes filter distorted patterns. Table III shows the three different input types from the three different schemes mentioned above for two patterns to be learned by backpropagation. The assumption of input data range and bucket numbers in a BWM scheme representation are the same as the ones mentioned above. These input data will be learned separately with the same desired outputs for each pattern. The nets are trained 10,000 times with the inputs, respectively. Table IV shows the distorted inputs of the data in table III. The bold faced digits are the ones contaminated by a change of 1 in the decimal input value.

As seen in table IV, for a small distortion in a decimal number, that is, 7 $\longrightarrow$ 8, the data using the binary scheme are corrupted completely, that is, 0111 $\longrightarrow$

Table III: 3 different types of inputs and desired outputs for 2 patterns trained by backpropagation

pattern	scheme	inputs			desired outputs	
pattern A	non-binary	7	4	2	1	0
	binary	0111	0100	0010		
	BWM	0001	0100	1000		
pattern B	non-binary	2	5	6	0	1
	binary	0010	0101	0110		
	BWM	1000	0010	0010		

1000, while the data using the BWM scheme are not corrupted. This is the worst case of the binary scheme because a small distortion in non-binary input causes 100% of the binary bits to be corrupted. The worst case in the BWM scheme is when a distorted number is out of its original bucket. In pattern B of table III and table IV, the number 5 belongs to the third bucket, and the distorted number 4 belongs to the second bucket. So the BWM scheme input 0010 becomes 0100, that is, at most 2 bits are changed in the worst case of the BWM scheme no matter how long the inputs are. These three different types of inputs are recalled, and the results are illustrated in table V.

As seen in table V, the binary scheme could not filter the distorted inputs of pattern A while the other two schemes corrected the input errors and produced correct outputs. The distorted input of pattern A using the binary scheme produce wrong outputs which are closer to the desired outputs of pattern B and cause the net to make a wrong decision. From the results of table II and table V, it is shown that the BWM scheme helps a net's accuracy of classification over continuous inputs and helps noisy filtering and fault-tolerance capabilities over a

Table IV: 3 different types of distorted inputs for 2 patterns

pattern	scheme	noisy inputs			desired outputs	
	non-binary	8	4	2		
pattern A	binary	1000	0100	0010	1	0
	BWM	0001	0100	1000		
	non-binary	2	4	6		
pattern B	binary	0010	0100	0110	0	1
	BWM	1000	0100	0010		

binary scheme.

In next chapter, this BWM scheme will be implemented in a neural net called the *Binary Pattern Net*, and a bucket sorting method using the net will be given, too.

Table V: Results of recalling of the trained net

pattern	scheme	actual outputs				desired output	
		w/ correct inputs		w/noisy inputs			
pattern A	non-binary	0.99	0.01	0.99	0.01	1	0
	binary	0.99	0.01	**0.22**	**0.78**		
	BWM	0.99	0.01	0.99	0.01		
pattern B	non-binary	0.01	0.99	0.01	0.99	0	1
	binary	0.01	0.99	0.01	0.99		
	BWM	0.01	0.99	0.04	0.96		

III Binary Pattern Net

The Bucket-weight matrix scheme, which was proposed in the previous section briefly, converts a set of input numbers into a set of neural net state binary numbers without losing its unique pattern. This scheme is implemented in a neural net-like algorithm so that this net can be combined with other nets, which would be nets that have real world data processing ability. This binary pattern net can be used as a bucket sorting net as will be shown later in this chapter.

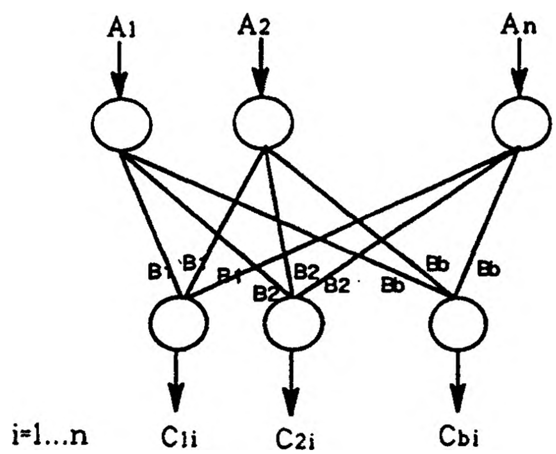
A constant number of buckets are generated according to the range of values in an input data set. Each element goes to its bucket and makes the bucket value 1 and the others 0. A $(number\ of\ buckets) * (number\ of\ elements)$ matrix is obtained after all of the elements in the set are put in their buckets. The pattern in the data set is determined according to how big and where, relatively, in the set each element is. Like other nets, the operation of this net has 2 steps, the training step and the recalling step. Since the bucket-weight matrix is generated in the training step, the bucket-weight matrix scheme is performed in this step. The recalling step is necessary for bucket sorting of the data set.

A Training step

In this step, a set of numbers becomes a set of binary pattern numbers. Figure 7 illustrates the training model of the Binary Pattern Net.

Going through this net, n numbers become $n * b$ binary numbers that are the bucket-weight matrix. The neurons in the first layer have input numbers, and the connection weights from the first layer to the second layer are initialized with B_i that is the upper limits of buckets. The summation function of each neuron in the second layer is $Net_{ij} = A_j - B_i$, and the transfer function is a threshold function that is shown in figure 7. The Net_{ij} value represents the distance between an element, A_j, and a bucket boundary, B_i. The transfer function in the C layer is a

A(i) = Input data element
n = the number of elements in the input data
max = the maximum value of the input data range
min = the minimum value of the input data range
b = the number of buckets
B(i) = ((max-min+1)/b) * i



$NET_{ij} = A_j - B_i$

$OUT_{ij} = \begin{cases} -\infty, & \text{if } NET_{ij} > 0 \\ NET_{ij}, & \text{otherwise} \end{cases}$

Winner-takes-all

$C_{ji} = \begin{cases} 1, & \text{winner for each } i \\ 0, & \text{others} \end{cases}$

Figure 7: Training step of binary-pattern net. The matrix C is the bucket-weight matrix. N numbers are encoded into $n * b$ numbers

winner-take-all style, that is, C_{ij} is set to 1 for the winner of all the elements of j, 0 for the rest. In this way, the $n * b$ bucket-weight matrix has n 1s and $(n * b - n)$ 0s, and there is only one 1 in each row.

Example We have a set of data which has 5 elements { 7, 3, 11, 5, 1 }. Suppose the range of input data values is 0 ~ 12, and the number of buckets is 4.

We can show the process with matrix manipulation.

A(i; i=1,5) = 7, 3, 11, 5, 1

B(i; i=1,4) = $\frac{12}{4}i$ = 3, 6, 9, 12

Definition of $\ominus$: Net(i,j) = A(i) $\ominus$ B(j) is defined as $Net_{ij} = A_i - B_j$.

$$
\begin{aligned}
Net_{ij} &= \begin{pmatrix} 7 \\ 3 \\ 11 \\ 5 \\ 1 \end{pmatrix} \ominus \begin{pmatrix} 3 & 6 & 9 & 12 \end{pmatrix} \\
&= \begin{pmatrix} 4 & 1 & -2 & -5 \\ 0 & -3 & -6 & -9 \\ 8 & 5 & 2 & -1 \\ 2 & -1 & -4 & -7 \\ -2 & -5 & -8 & -11 \end{pmatrix} \\
Out_{ij} &= \begin{cases} -\infty, & \text{if } Net_{ij} > 0, \\ Net_{ij}, & \text{otherwise.} \end{cases} \\
&= \begin{pmatrix} -\infty & -\infty & -2 & -5 \\ 0 & -3 & -6 & -9 \\ -\infty & -\infty & -\infty & -1 \\ -\infty & -1 & -4 & -7 \\ -2 & -5 & -8 & -11 \end{pmatrix} \\
C_{ij} &= \begin{cases} 1, & \text{if the largest in i}^{\text{th}} \text{ row,} \\ 0, & \text{otherwise.} \end{cases} \\
&= \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}
\end{aligned}
$$

The final matrix C is the bucket-weight matrix. The binary digits of each row represent each number of the data set. A_1, 7, is represented by 0010, $C_{1j,\ j=1\ldots4}$

that is first row of matrix C; A_2, 3, is transformed to 1000, $C_{2j,\ j=1\ldots4}$ and so on. Each column represents a bucket. The binary matrix C has the pattern that the continuous valued data set A had.

B Recalling step (Bucket sorting)

In general, a bucket sort has three phases, which we may call ***distribution***, ***sorting buckets***, and ***combining buckets***.[11] Suppose there are k buckets. During the distribution phase, each key is examined. Then it does some work to indicate in which bucket the key belongs. In the second phase, an algorithm is used to sort buckets by a comparison of keys. The third phase requires that the keys be copied from the buckets into one file. If the distribution of the keys is known in advance, the range of keys to go into each bucket can be adjusted so that all buckets receive an approximately equal number of keys.

In this recalling step, the unsorted numbers in the input set of the training step are distributed into buckets by the values. The bucket size can be adjusted. If this recalling step is recursively used to create smaller and smaller buckets or if a large number of buckets is used, then all of the keys can be sorted completely in the first phase, that is, this recalling step, even though this seems inefficient in terms of the amount of space needed.

Figure 8 illustrates the recalling model of a binary pattern net. The input numbers are the numbers in set A, and the weight matrix used is the bucket-weight matrix from the training step of this net.

Example This example is the same as the one in the training step.

Definition of $\otimes$: S(i,j) = A(i) $\otimes$ C(i,j) is defined as $S_{ij} = A_i * C_{ij}$.

$$\text{Sorted matrix } S_{ij} = \begin{pmatrix} 7 & 3 & 11 & 5 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

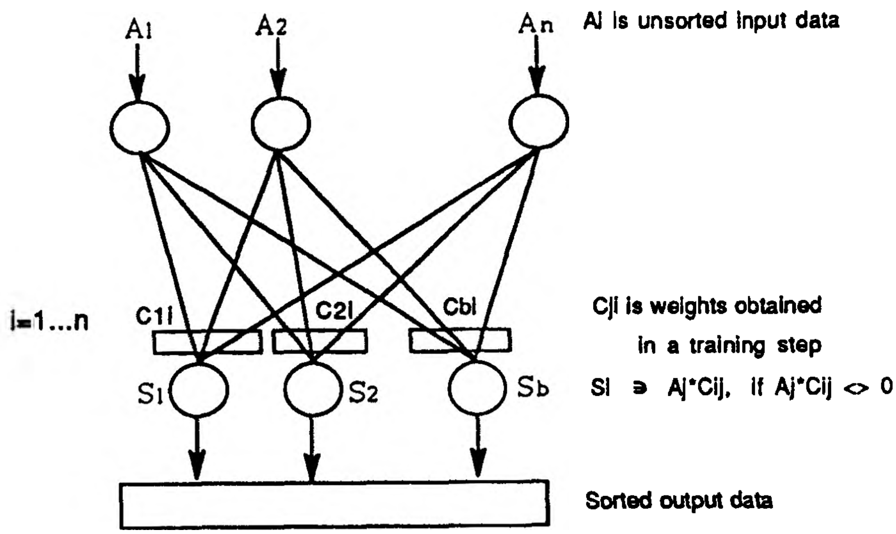


Figure 8: Recalling step of a binary-pattern net. This step is used when bucket sorting is required. The weight matrix C is from a bucket-weight matrix.

$$= \begin{pmatrix} 0 & 0 & 7 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 \\ 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{array}{cccc} \downarrow & \downarrow & \searrow & \searrow \\ S_1 & S_2 & S_3 & S_4 \end{array}$$

Each column represents a bucket. The first bucket contains 1 and 3 ,the second bucket has 5, and so on. In this way the input data are bucket-sorted in this recalling step. Figure 9 shows the nets of this example.

The 1,000 numbers generated by a random number generator are distributed into buckets by this net on a serially operated computer, and the results are in the appendix D. Figure 10 illustrates the graph of running time of bucket sorting in this net.

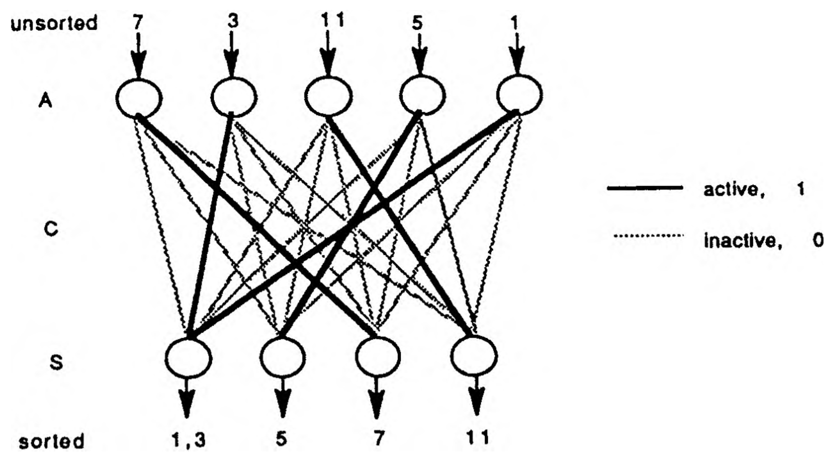


Figure 9: The net of the recalling step for a bucket sorting. The link weights are the key of the sorting.

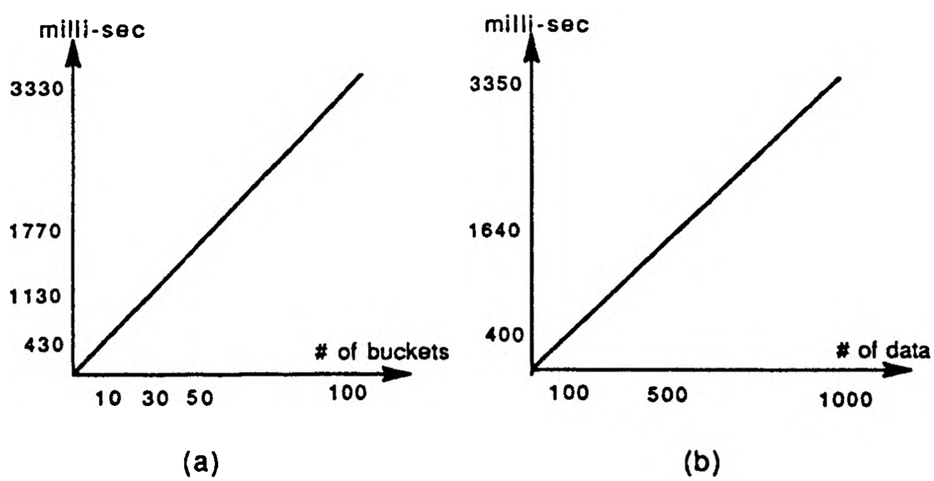


Figure 10: Bucket sorting time (in milli-seconds) for the binary-pattern net. (a) when the number of data is fixed at 1000 and (b) when the number of buckets is fixed at 100

C Analysis

The performance of the net will be discussed in this section. The following are the properties of this net.

1. Encoding a set of real world numbers into a set of neural net state binary numbers without losing its unique pattern.

2. Good noise-filtering ability.

3. Linear complexity, Scalability.

1 Training time and memory used

Figure 10 shows that training time increases linearly with the number of elements or the number of buckets; that means that the computational-complexity function of a bucket-weight matrix net is linear. In the training step, suppose n is the number of elements and b is the number of buckets. $A(n) \ominus B(b)$ becomes an $n * b$ matrix, and it requires $n * b$ multiplications and an average of $1/2 * n * b$ comparisons. In general, since the number of buckets, b, is constant, the training time is linear of n. So the complexity of the net when it is simulated on a serially operated computer is $\bigcirc(n)$, which means this net is scalable. *Scalable*, in this case, refers to the ability of a neural network developed on a digital computer to be enlarged easily to perform larger real-world tasks. When we want to enlarge a small experimental neural network into a real-world application, scalability becomes important. The graph in figure 11 compares three scalability standards.[7] If you improve a training algorithm from exponential to polynomial scalability, you will significantly increase the number of patterns you can train. The bucket-weight matrix net has linear scalability, which is an improvement over polynomial scalability.

The number of bits or neurons required in the BWM scheme depends on the number of buckets (b) and the number of elements in the set (n). It requires $b * n$

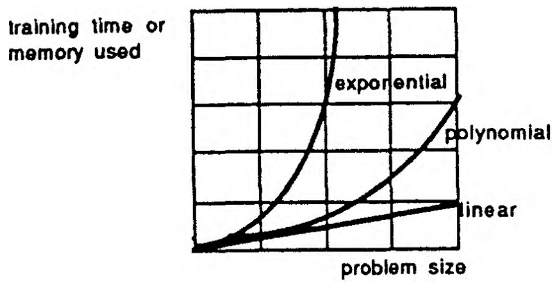


Figure 11: Three standard measures of scalability; problem size can be from training patterns, neurons, or synapses

bits to express the set of data. Each element requires b bits regardless whether it is a large number or not. Generally, the memory used is linear in the problem size if b is constant. The numbers of neurons increases $b * 100$ (%) because n continuous valued numbers become $b * n$ binary numbers. This might limit the possibility of covering large numbers of elements using a small number of neurons, but for a neural computer it is not a fatal disadvantage because the use of ample neurons with much redundancy is the key to improving its computational capability and system stability[6].

2 Noise filtering

This binary weight matrix net has a noise-filtering ability. The elements whose values belong to the same bucket are represented by same binary bits in the BWM scheme. So a little change of a value within the bucket size does not affect its binary result. The bucket size determines the noise-tolerance range and the bucket size is determined by the number of buckets. For example, consider the example data in the previous section, which is { 7, 3, 11, 5, 1 } and a bucket size of 3. If the first element 7 is contaminated by two units, it becomes 9. The resulting bucket weight matrix is the same because both 7 or 9 belong to the third bucket. But if the 7 is changed to 6, a different binary matrix is obtained because 6 belongs to the second bucket, not to the third bucket. So the number which is on the border of each bucket has a half chance to be filtered according to its noise direction. Figure 12 shows a graph of the trade-offs between noise tolerance and accuracy of pattern recognition with the data used in the example of the training step. Different bucket-weight matrices for different bucket numbers or bucket sizes are shown in the figure also. In figure 12, by connecting the 1s in the matrices, the similarity of input data pattern A and converted binary data C can be found.

Though the noise filtering ability is decreasing as the bucket size gets very small, the number of affected bits in the resulting binary matrix is 2 at most. Compared to the number of total bits, 2 bits are very few. These few distortions can surely be filtered out in the main neural net algorithms that are going to use the binary matrix as its input data pattern data. Since most neural net algorithms have noise filtering ability, this net will increase the ability a lot when this net is combined with those nets.

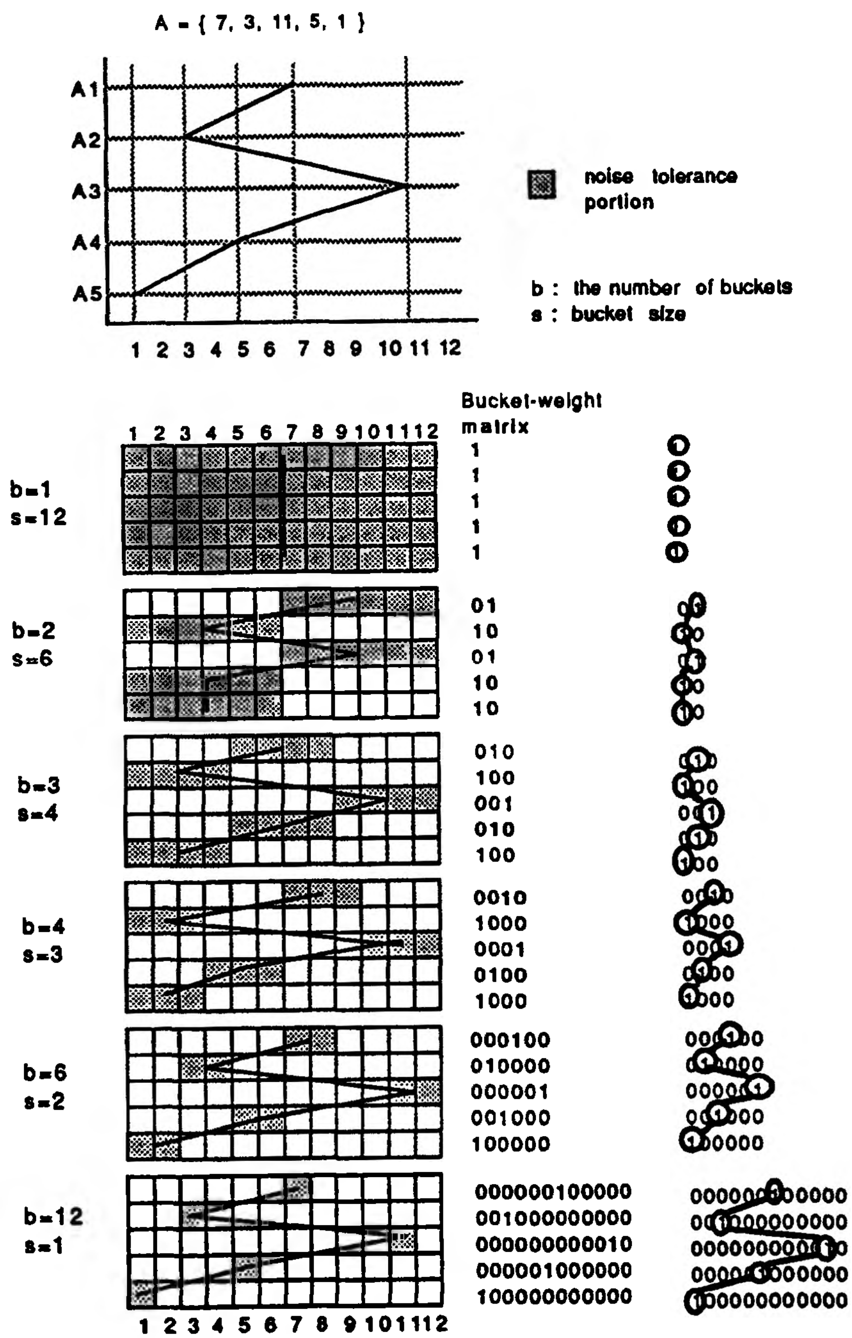


Figure 12: Trade-off between accuracy of recognition and noise-tolerance by varying bucket sizes

IV Examples

The bucket-weight scheme connected to the BAM model is simulated. The scheme is implemented in the training step of a binary pattern net. Two examples will be shown here; one is a pattern recognition with a non-visual numeric-valued pattern, and the other is with a visual image pattern.

A Example with a numeric data pattern

Recognition of some patterns of stock prices in a stock market is performed here. The data collected are weekly closing prices of 2 companies, A and B, for 12 weeks. The 12 week stock prices of each company have their own pattern of the trend of prices, and we want to classify the patterns with the BAM model. Table VI shows the raw data of 2 companies.

The BAM model uses bipolar numbers for its inputs, but the data we have in this example is not bipolar. So we need a scheme to convert the raw data into bipolar data. The bucket-weight matrix scheme will be used for this example. A bucket-weight matrix will be generated in a training step of the binary pattern net. The matrix is used as an input set in the BAM. In a binary pattern net, suppose that 16 buckets are used so that the bucket size is $(323.98 - 177.35)/16 = 9.16$ because the raw data is in the range 177.35 to 323.98 as seen in table VI. With this bucket size, the BWM scheme will have the noise-tolerance rate of $9.164/(323.98 - 177.35) * 100 = 6.25\%$. The final binary data, that is, bucket-weight matrix obtained in a training step of the binary pattern net are in figure 13. These data are trained in the BAM model with auto-associations of the two patterns.

Results

Figure 14 shows a graph of the trend (up and down) of the values in table VI and graphs of results of recalling with some incomplete inputs. The association model produces the complete output, the following stock prices, given the partial inputs,

Table VI: Weekly closing prices of company A and B

	Company A	Company B
1^{st} week	177.35	265.84
2^{nd} week	198.25	264.38
3^{rd} week	221.89	257.72
4^{th} week	249.01	249.01
5^{th} week	271.47	239.56
6^{th} week	260.91	231.62
7^{th} week	262.13	229.54
8^{th} week	295.73	229.97
9^{th} week	310.07	241.03
10^{th} week	316.61	260.90
11^{th} week	300.01	300.01
12^{th} week	251.43	323.98

the first 3 weeks' prices as seen in graph (a). Graph (b) shows the correct output, given noisy inputs. The noise in graph (c) is filtered in the binary-pattern net already before it goes to the BAM net. Since the BAM net has noise filtering property, a double noise filtering system can be obtained by using the two nets together. Another noisy input is tested in graph (d), which is a mixed input of pattern A and B. Appendix E has all of the input data and output data used in these tests. The program codes used in these tests are in appendix A and B.

$$A = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

$$B = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

Figure 13: Matrix A is binary pattern matrix for company A, and matrix B is for company B.

B Example with a non-numeric image pattern

In many image recognition problems, the most common way to extract binary image numbers is illustrated in figure 15 which shows a set of inputs for a shape drawn on a grid. If a line passes through a square, the corresponding neuron's input is one; otherwise, that neuron's input is zero.

Even though this method is simple to get binary input numbers from an image pattern, it has unexpected side effects in terms of noise problems. This will be discussed by testing two patterns with the BAM model. Two different patterns in figure 16 will be trained and tested to be classified in the BAM model.

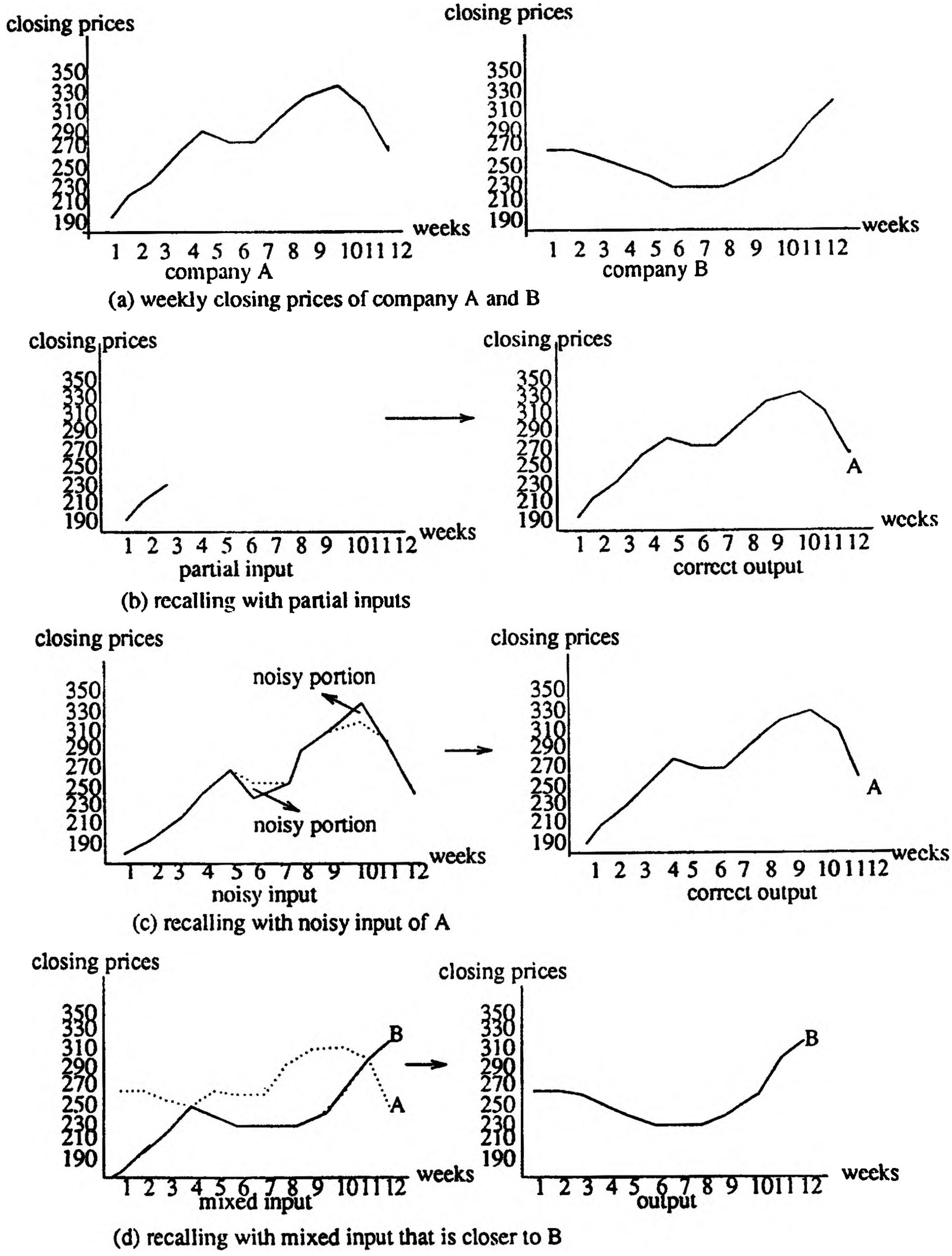


Figure 14: Graphs of values in table VI and results of recalling with different inputs

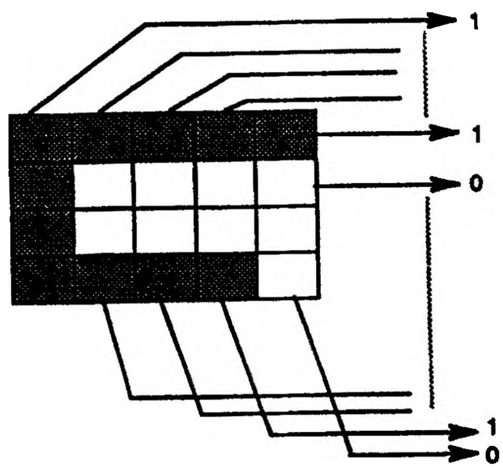


Figure 15: Binary data extracting from a visual image pattern

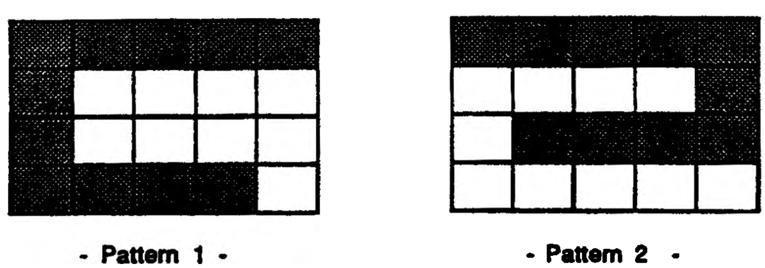


Figure 16: Two input patterns

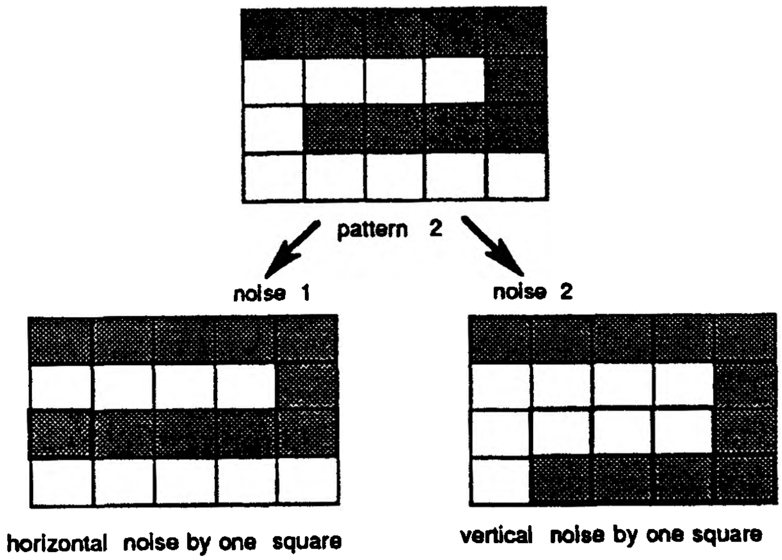
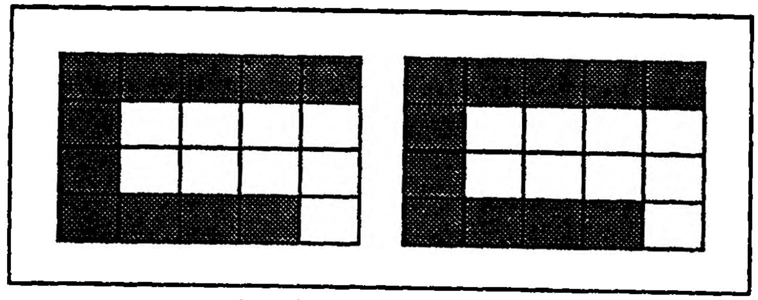


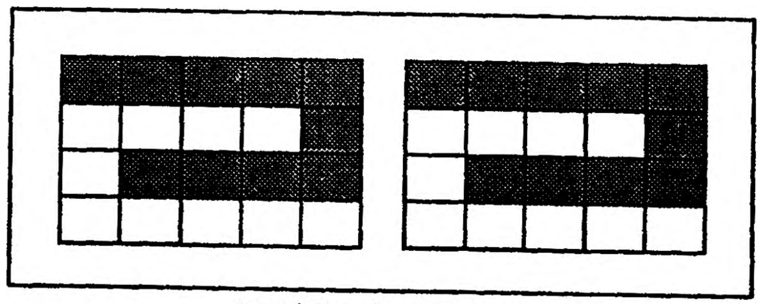
Figure 17: Two possible noisy patterns of pattern 2

Say that pattern 2 is corrupted by one square horizontally and vertically as seen in figure 17. The horizontal noise of pattern 2 causes just one square to be changed while vertical noise causes seven squares to be changed even though both of them make just one square of noise. Figure 17 explains this. This side-effect will result in a wrong pattern classification. The two patterns in figure 16 are trained in an auto-association manner like figure 18. After that, the two patterns distorted by one square in height are tested, and totally wrong outputs are obtained. Figure 19 shows the output results. The noisy input of pattern 1 has 13 squares identical to pattern 1 while it has 16 squares identical to pattern 2; that is, the input pattern is 15% more closely matched to pattern 2 than pattern 1. So wrong classification is performed. The same goes for the result of noisy input of pattern 2 and its wrong output. So another data extracting method is necessary to overcome this unexpected side effect. Vector values to represent the patterns can be used in this case.

Figure 20 illustrates one particular example of how to extract vector values



Association of pattern1



Association of pattern2

Figure 18: Input pairs for training

that have directions and scalar values from pattern 1. Suppose that the scalar value represents the number of squares. As seen in figure 18, the two patterns are composed of three lines each. The differences of the two patterns are the heights of the patterns and which side is open. Suppose that the starting point to extract vector values of lines is the end point of the upper line. Consecutive vectors begin at the end of previous vectors. The vector data obtained in this way is $(-5,\ -4,\ +4)$ for pattern 1 and $(+5,\ -3,\ -4)$ for pattern 2. Since the data is non-binary numerical, the scheme proposed in chapter III-A is used to convert the data into binary format. Suppose 5 buckets are used in this case. Then the bucket size is 2 since the maximum is +5, minimum is -5, and there are 5 buckets

So the upper limits of the buckets B = {-3, -1, 1, 3, 5}. The two bucket-weight matrices from the two vector data are

$$pattern1 = (-5,\ -4,\ +4)$$

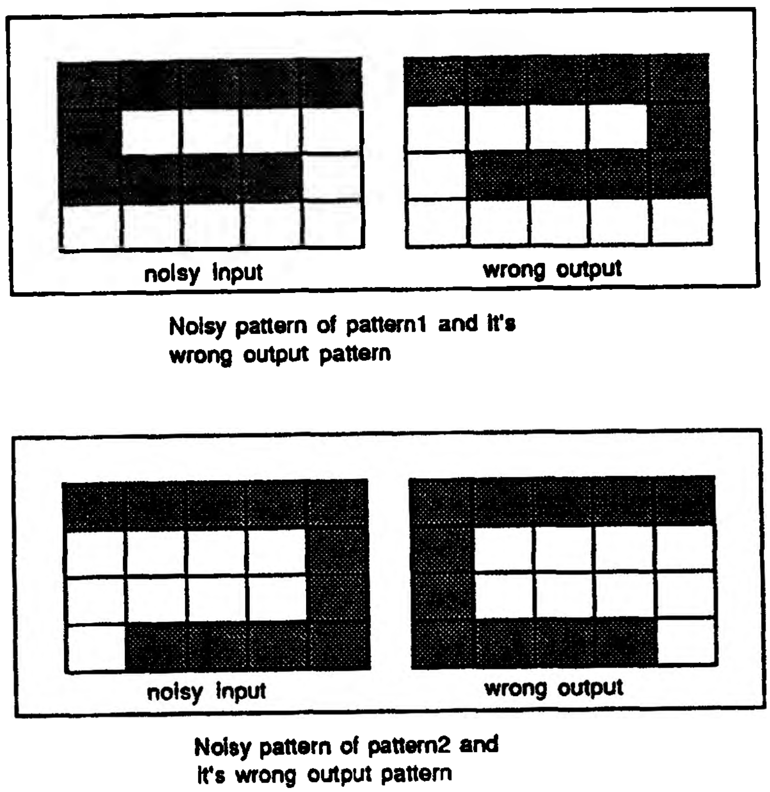


Figure 19: Recalling of noisy patterns and its incorrect output

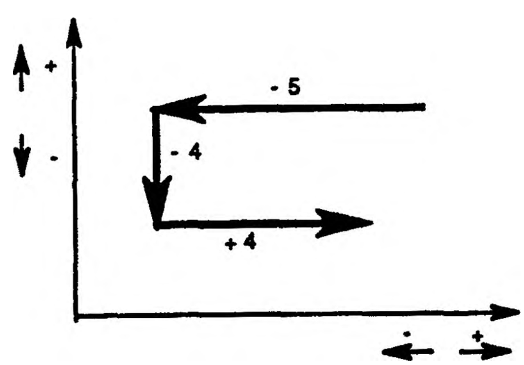


Figure 20: Extracting vector values from image pattern 1 in figure 16

$$
\begin{aligned}
Net_{ij} &= \begin{pmatrix} -5 \\ -4 \\ +4 \end{pmatrix} \ominus \begin{pmatrix} -3 & -1 & 1 & 3 & 5 \end{pmatrix} \\
&= \begin{pmatrix} -2 & -4 & -6 & -8 & -10 \\ -1 & -3 & -5 & -7 & -9 \\ 7 & 5 & 3 & 1 & -1 \end{pmatrix} \\
Out_{ij} &= \begin{pmatrix} -2 & -4 & -6 & -8 & -10 \\ -1 & -3 & -5 & -7 & -9 \\ -\infty & -\infty & -\infty & -\infty & -1 \end{pmatrix} \\
bucket\ weight\ matrix\ for\ pattern1 &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

The bucket weight matrix for pattern 2 can be obtained in the same way as above.

$$
\begin{aligned}
pattern2 &= (+5,\ \ -3,\ \ -4) \\
bucket\ weight\ matrix\ for\ pattern2 &= \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}
\end{aligned}
$$

The vector values of the two noisy input patterns in figure 19 are $(-5,\ -3,\ +4)$ and $(+5,\ -4,\ -4)$, respectively. And their bucket-weight matrices are exactly same as the ones above. This means the noise is filtered in a bucket-weight matrix scheme before going through the BAM model. Even if there are many buckets and each bucket size is small, the number of input elements contaminated by a pattern noise is two at most. Two elements of noise in the matrix is small enough that the BAM model can surely filter it because the BAM model has a noise filtering ability.[3] The results in this way are shown in figure 21. This method is closer to the human's pattern recognition method as we classify the two patterns by their rough shape, that is, whether the right side is open or the left side is open, not by their precise lengths and angles. In this way, we can recognize scaled patterns and rotated patterns easily while the first method cannot help it.

The two methods mentioned above can be compared in terms of noisy rate of input binary data. Suppose that an input pattern is distorted by one unit and

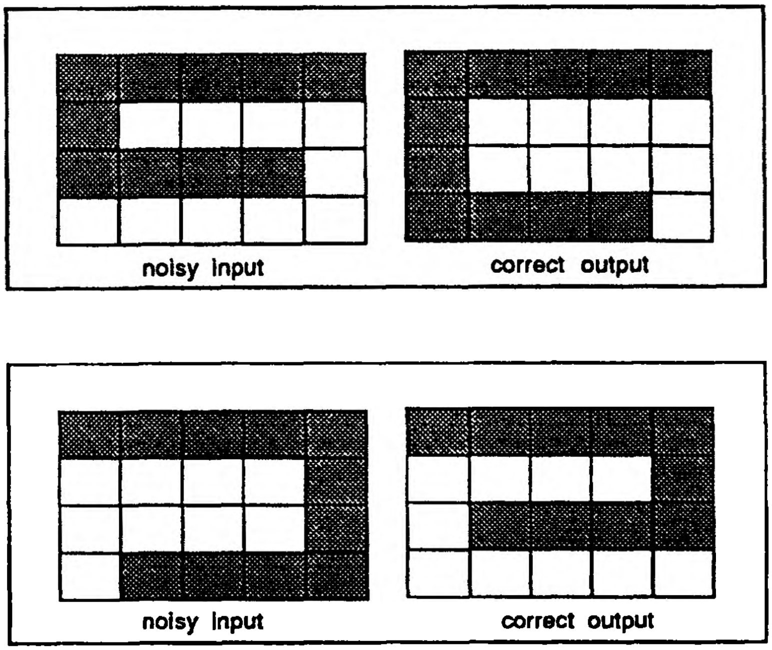


Figure 21: Recalling of noisy patterns and its correct outputs

the binary input data from the pattern has $n * n$ elements. Table VII compares the rates of corruption of the converted binary inputs caused by the distortion of the original input pattern.

Table VII: Noise rates of two methods

method	best case	worst case
first method	$1/(n * n) * 100(\%)$	$2/n * 100(\%)$
second method	$0(\%)$	$2/(n * n) * 100(\%)$

The best case in the first method is the case of noise 1 in figure 17, and the worst case is the noise 2 case in the figure. In the case of the second method that uses vector inputs, the BWM scheme filters most of the small errors if the errors are in the noise-tolerance range. At worst, 2 bits in the bucket-weight matrix are changed regardless the size of the matrix.

V Conclusion and further research

Most current neural network algorithms have considerable power in pattern recognition. A method that extracts a particular pattern from real world data as closely as possible to the human's pattern extracting method surely increases the neural networks' pattern recognition ability. In this thesis, one of the input data encoding methods, the bucket-weight matrix scheme, is discussed and tested with both a visual image pattern and a non-visual numeric pattern. The results show that this scheme is a very encouraging and effective data extracting method. This scheme provides a means to make neural net state input data from a numerical data pattern, avoids unexpected side effects that might happen in a data encoding procedure ,and has a good error-correcting property.

The binary-pattern net that implements the scheme can be used as a bucket-sorting net also. The procedure and results of a bucket-sorting net are shown in this report, and this gives us a positive possibility of another application of neural networks. That is, neural computing can contribute to many traditional data structure problems such as sorting, searching, indexing and the hashing function as well as pattern recognition problems. Since the basic structure and procedure of neural networks is parallel and distributed, some problems of data structures such as timing and fault tolerance can be solved with these neural networks properties. In implementing the algorithms on a parallel machine, there are some problems to be solved such as communication time problems. The under-construction neurocomputers that are electrical or optical implementations of neural networks will surely solve these problems and open a new area of the computer world.

The adaptive expert system is a very encouraging application of neural networks. The expert system is the most successful one in terms of the practice of artificial intelligence. The strong point of the expert system is its *inference* ability while the neural network has a powerful *recognition* ability. So the combining of the two abilities, recognition and inference, will approach a human's brain capability. There is no doubt that this neural network will contribute to humankind's life as well as many artificial intelligence application areas.

Appendix A
Programming of the BAM model

B. Kosko's BAM model[3] is programmed in C language here.

```
/*
*************************************************************
*************                                    ************
*************   BIDIRECTION ASSOCIATIVE MEMORY   ************
*************                                    ************
*************                  By Hyeoncheol Kim ************
*************                               1990 ************
*************                                    ************
*************************************************************
******************************************************************/
#include <stdio.h>
#include <math.h>

#define  Number  2       /* The number of the patterns*/
#define  In_element 20   /* The number of the input elements */
#define  Out_element 20  /* The number of the output elements */

/****************************************************************
    DEFINING THE GLOBAL VARIABLES AND ARRAYS FOR MAIN PROGRAM
******************************************************************/

float M[In_element][Out_element];     /* The trained connections */

main(){

/*****************************************************************
   DEFINING THE LOCAL VARIABLES AND ARRAYS FOR MAIN PROGRAM
******************************************************************/

Π char ans,fn1[10], fn2[10];       /* The name of the inputs */
Π int  flg,i, j, n;                /* Integer Variables */
Π float input[Number][In_element], output[Number][Out_element];
ΠΠΠΠ       /* Variables for input and output*/
         float M1[In_element][Out_element];
                                         /* Variable for connection */


Π FILE  *in_file, *out_file, *fopen(), *fclose();


/*****************************************************************
 READ IN THE INPUT DATA AND THE NAME OF THE TWO FILES;
 INPUT AND OUT PUT
 ****************************************************************/
```

```
Π for(i=0;i<In_element;i++){
Π for(j=0;j<Out_element;j++){
Π M[i][j] = 0.0;
Π }
Π }
Π /* The initial values of the connection */

Π for(n=0;n<Number;n++){
  Π
Π printf("Enter the file name of input data\n");
Π gets(fn1);
Π in_file = fopen(fn1,"r");
Π if(in_file == NULL){
Π printf("Error Opening\n");
Π exit(1);
Π }
Π /*  Read in the first data for the associative memory */

Π printf("Enter the file name of target output data\n");
Π gets(fn2);
Π out_file = fopen(fn2,"r");
Π if(out_file == NULL){
Π printf("Error Opening\n");
Π exit(2);
Π }

         /* Read in the second data for the associative menory */

Π for(i=0;i<In_element;i++){
Π fscanf(in_file,"%f",&input[n][i]);
Π if(input[n][i] == 0.0){
Π input[n][i] = -1.0;
ΠΠΠ   }
          else{
Π  input[n][i] = 1.0;
Π      }
ΠΠΠΠ  }

         /* With hard limit threshold, transform the data
            into the bipolar */


         for(i=0;i<Out_element;i++){
Π fscanf(out_file,"%f",&output[n][i]);
Π if(output[n][i] == 0.0){
Π output[n][i] = -1.0;
```

```
ΠΠΠ     }
         else{
Π output[n][i] = 1.0;
Π     }
ΠΠΠΠ   }

         /* With hard limit threshold, transform the data
            into the bipolar */


Π for(i=0;i<In_element;i++){
Π for(j=0;j<Out_element;j++){
Π M1[i][j] = input[n][i]*output[n][j];
Π }
Π }
Π /* Constructing the connection for the hetero associative memory */

Π for(i=0;i<In_element;i++){
Π for(j=0;j<Out_element;j++){
Π M[i][j] = M[i][j] +M1[i][j];
Π }
Π }
Π } /* The N patterns have been learned */

printf("*** End of Training Step ! ***\n");

do{
flg=0;
 RECALL(); /* procedure of recalling the data with arbitrary input */
printf("Another test?(y/n)\n"); gets(ans); if (ans=='y') flg=1;
}while(flg=1);

Π } /* This is the end of the main function */




/*******************************************************************
THIS IS A PROCEDURE OF RECALLING THE OUTPUT FROM THE TRAINED NETWORK
*******************************************************************/

RECALL()      /* The Name of the recalling procedure */
{
/*****************************************************************
     DEFINING THE VARIABLES FOR THE RECALL
```

```
*****************************************************************/

char fn1[10], fn2[10];
int  i, j, k, l;
float input[In_element], output[Out_element];
float buffer, check[In_element];

FILE  *in_file, *out_file, *fopen(), *fclose();

/*******************************************************************
П      READ IN THE NAME OF THE DATA FILE
*******************************************************************/

printf("Enter the Name of the input file for recalling\n");
gets(fn1);
in_file = fopen(fn1,"r");
if(in_file == NULL){
printf("Opening Error\n");
exit(4);
}

printf("Enter the Name of the out file for recalling\n");
gets(fn2);
out_file = fopen(fn2,"r");
if(out_file == NULL){
printf("Opening Error\n");
exit(4);
}


/****************************************************************
CHANGING THE ELEMENTS INTO BIN.
BY LETTING THEM GO THROUGH THE THRESHOLD
*****************************************************************/

for(i=0;i<In_element;i++){
fscanf(in_file,"%f",&input[i]);     /* initial value of input[] */
ПППП }

for(i=0;i<Out_element;i++){
fscanf(out_file,"%f\n",&output[i]); /* initial value of output[] */
ПППП }

/******************************************************************
TAKE THIS PROCEDURE UNTIL THE OUTPUT DOES NOT CHANGE
BY FALLING INTO A HOLE
```

```
****************************************************************/

do{

for(i=0;i<In_element;i++){
check[i] = input[i];
ΠΠΠ  }
ΠΠΠ  /* Initializing the checking variables */

buffer = 0.0;
for(i=0;i<Out_element;i++){
buffer = 0.0;
for(j=0;j<In_element;j++){
buffer = buffer + input[j]*M[j][i];
                          }
if( buffer > 0.0 ){
output[i] = 1.0;
ΠΠ    }
if( buffer < 0.0 ){
output[i] = 0.0;
ΠΠ    }       /* if buffer = 0.0, output[] is not changed */
                          }
ΠΠΠ /* The output at the first layer */


for(i=0;i<In_element;i++){
buffer = 0.0;
for(j=0;j<Out_element;j++){
buffer = buffer + M[i][j]*output[j];
ΠΠΠ  }
if( buffer > 0.0 ){
input[i] = 1.0;
ΠΠ    }
if( buffer < 0.0 ){
input[i] = 0.0;  /* If buffer = 0.0, input[] is not changed */
ΠΠ    }
ΠΠΠ  }
ΠΠΠ  /* The output at the second layer */

buffer = 0.0;
for(i=0;i<In_element;i++){
if(check[i] != input[i]){
buffer = buffer +1.0;
                         }
ΠΠΠ}
}while(buffer !=0.0);
```

```
printf("output: \n");
for(i=0;i<Out_element;i++){
if ((i%5)==0) printf("\n");
printf("%.0f ",output[i]);
}
printf("\n");
/* THIS IS THE END OF THE RECALLING PROCEDURE */
}
```

Appendix B

Programming of Bucket-weight matrix scheme

Bucket-weight matrix scheme in chapter II-B is programmed in C language.

```
/*****************************************************
*                                                    *
*            Making Bucket-Weight Matrix             *
*                                                    *
*                     By Hyeoncheol Kim              *
*                                 1990               *
*                                                    *
*****************************************************/

#include <stdio.h>
#include <math.h>

#define max_sor 100     /* Number of input elements */
#define max_bk 1000     /* Number of groups (buckets) */

main(){

   char irum[10];
   int i,j,bk,flg,no_element;
   float flt,ind_keys[max_bk],in_data[max_sor],
         train[max_sor][max_bk];
   double mx,mn,scale,offset;

   FILE *in_file, *out_file, *ifp, *ofp, *fopen(), *fclose();

   printf("Enter the input file name :\n"); /* raw input data */
   gets(irum);
   ifp = fopen(irum, "r");
   ofp = fopen("conv.o", "w");

   if (ifp == NULL | ofp == NULL){
   printf("Error in opening file\n");
   exit(1);
   }

   /* convert raw data to scaled data */

   no_element = 0;
   mx = 0.0; mn = 9999999.0;
   while ( fscanf(ifp,"%f",&flt) == 1 ) {
     no_element = no_element + 1;
     if (flt < mn) mn = flt;
     if (flt > mx) mx = flt;
   }
```

```
scale = 8 / (mx-mn);
offset = 1 -scale*mn;

printf("scale = %.4f, offset = %.4f\n", scale, offset);

fseek(ifp, 01, 0);
while (fscanf(ifp,"%f",&flt) == 1){
    fprintf(ofp, "%.3f\n", flt*scale+offset);
}

fclose(ifp); fclose(ofp);

bk = 16;   /* number of buckets */
ind_keys[0] = 1.5;
for (i=1;i<bk;i++) {
ind_keys[i] = ind_keys[i-1] + 0.5;
                   }

in_file = fopen("conv.o","r");
if(in_file == NULL){
printf("Error in opening file\n");
exit();
}
for (i=0;i<no_element;i++){
fscanf(in_file,"%f\n",&in_data[i]);
printf(" %.0f ",in_data[i]);
                           }
printf("\n");

printf("Enter the file name for a pattern table
      (exept conv.i, conv.o) :\n");
gets(irum);
out_file = fopen(irum,"w");
if(out_file == NULL){
printf("Error in opening file\n");
exit();
}

/* making pattern table of binary data from the scaled data */

for (i=0;i<no_element;i++){
flg=0;
for (j=0;j<bk;j++){
    if (flg==1) {train[i][j]=0.0;}
    else {
```

```
        train[i][j]=in_data[i]-ind_keys[j];
        if (train[i][j]>0.0) {train[i][j]=0.0;}
        else {train[i][j]=1.0; flg=1;}
           }
                   }
                           }


    for (i=0;i<no_element; i++){
    for (j=0;j<bk; j++){
    fprintf(out_file,"%.0f ",train[i][j]);
                            }
    fprintf(out_file,"\n");
                            }

    for (i=0;i<no_element; i++){
    for (j=0;j<bk; j++){
    printf(" %.0f ",train[i][j]);
                            }
    printf("\n");
                            }


} /* end of program */
```

Appendix C

Programming of binary pattern net (bucket sorting net)

The binary pattern net in chapter III-B is programmed in C language for bucket sorting.

```
/*********************************************************
*                                                       *
*           BUCKET SORTING USING NEURAL NET             *
*                                                       *
*                                  By Hyeoncheol Kim    *
*                                               1990    *
*                                                       *
**********************************************************/

#include <stdio.h>
#include <math.h>
#include "types.h"
#include "macros.h"
#include "clocks.c"

#define max_sor 1000    /* Number of elements to be sorted */
#define max_gv 100      /* Number of groups */

main(){

   char fn[10];
   int i,j,ttime,flg,Sor_element;
   float ind_keys[max_gv],sor_data[max_sor],
         train[max_sor][max_gv];
   float gv,mx;

   FILE *in_file, *fopen();

   printf("Enter the # of data to be sorted:\n");
   gets(c);
   Sor_element=atoi(c);     /* number of data to be sorted */

   printf("Enter the maximum value:\n");  /* suppose minimum is 0 */
   gets(c);
   mx=atoi(c);

   printf("Enter the 'G' value (# of buckets):\n");
   gets(c);
   gv=atoi(c);

   for (i=0;i<gv;i++) {
   ind_keys[i] = (mx*(i+1))/gv;
                    }
```

```
    printf("Enter the file name to be searched :\n");
    gets(fn);
    in_file = fopen(fn,"r");
    if(in_file == NULL){
    printf("Error in opening file\n");
    exit();
    }
    for (i=0;i<Sor_element;i++){
    fscanf(in_file,"%f\n",&sor_data[i]);
    printf(" %.0f ",sor_data[i]);
    if((i%13) == 0) printf("\n");
                            }
    printf(" will be sorted! \n");

clock_init();
CLOCK_START(TOTAL_TIME);
                      /* training step and making sorted table */
    for (i=0;i<Sor_element;i++){
    flg=0;
    for (j=0;j<gv;j++){
        if (flg==1) {train[i][j]=0.0;}
        else {
          train[i][j]=sor_data[i]-ind_keys[j];
          if (train[i][j]>0.0) {train[i][j]=0.0;}
          else {train[i][j]=sor_data[i]; flg=1;}
             }
                      }
                               }

ttime=clock_val(TOTAL_TIME);
printf("\n Time is : %d \n",ttime);

                       /* printing sorted data from the table */
    for (i=0;i<gv; i++){
    printf("\nbucket %d :",i);
    for (j=0;j<Sor_element; j++){
        if (train[j][i]!=0.0) printf(" %.0f ",train[j][i]);
                                }
                                }

} /* end of program */
```

Appendix D

The result of bucket sorting using sorting net

The results of bucket sorting using bucket sorting net in chapter III-B is shown here. 1,000 random numbers are bucket-sorted.

```
Enter the # of data to be sorted:
1000
Enter the maximum value:
1000
Enter the 'G' value (# of buckets):
100
Enter the file name to be searched :
sordata
 176  309  535  948  172  702  226  495  125  84  390  277  368
 983  535  766  646  767  780  823  152  625  315  347  917  520
 401  607  785  932  870  867  675  758  582  389  356  200  827
 416  464  979  126  213  958  737  409  780  758  957  28  319
 757  243  590  43  956  319  59  442  915  572  119  570  252
 496  237  477  406  873  427  358  382  43  161  522  697  97
 401  773  245  343  230  298  305  887  37  651  399  676  733
 938  233  838  967  779  432  674  809  159  280  135  864  750
 208  140  295  803  219  563  716  198  990  250  431  755  861
 895  978  395  432  127  458  238  986  653  604  242  455  790
 79  476  153  246  945  614  988  477  800  744  381  480  527
 98  594  347  143  780  711  446  705  95  963  551  740  579
 638  782  188  302  283  684  293  565  418  307  445  566  488
 607  416  130  256  36  977  115  378  647  350  553  358  565
 476  164  615  172  555  292  872  835  845  896  595  541  168
 655  691  264  107  815  191  423  352  839  137  263  177  480
 380  505  503  352  526  121  520  607  733  557  344  802  591
 267  671  552  789  888  890  68  801  907  644  165  301  166
 285  842  536  36  207  21  358  621  520  546  154  823  33
 26  378  616  20  627  915  375  729  396  982  597  112  222
 799  871  738  14  740  418  362  204  183  76  116  159  788
 40  791  599  403  229  183  614  332  605  964  378  184  300
 514 54  144  10  885  958  626  956  631  39  351  146  106  197
 84  27  946  920  908  866  149  172  68  651  737  102  160
 94  122  25  762  957  28  647  108  428  310  19  885  758
 510  166  763  881  500  875  735  235  52  605  876  504  678
 989  605  496  590  895  45  883  108  520  579  10  387  477
 193  508  775  354  698  913  671  706  427  21  213  948  503
 194  645  128  265  336  704  38  954  755  874  634  244  636
 850  237  721  339  50  485  897  242  528  494  855  346  124
 216  115  363  204  436  828  510  820  411  871  713  644  581
 953  461  521  359  326  9  978  432  176  159  534  578  314
 342  158  437  243  201  720  220  195  423  774  831  245  5
 514  346  82  705  260  351  536  869  304  79  454  377  465
 829  24  904  198  633  129  236  600  647  840  843  157  214
 624  435  569  90  381  724  511  795  883  101  660  549  728
```

```
451 841 774 386 833 627 620 440 225 246 496 623 73
133 62 720 851 973 659 957 351 577 641 957 927 435
587 851 408 294 844 650 898 595 389 470 190 126 468
693 992 726 980 669 719 377 85 49 27 552 986 341
844 131 381 789 95 756 522 154 853 954 375 514 121
869 842 652 978 967 504 144 297 529 869 734 761 300
588 88 390 122 597 518 303 908 675 786 926 788 366
560 380 778 749 629 247 814 16 468 448 498 326 249
447 391 914 881 503 209 914 190 179 753 860 820 838
928 355 950 339 687 855 424 655 248 42 394 625 423
764 52 567 411 3 558 969 890 837 764 968 758 872
805 849 360 112 178 276 555 725 377 469 454 695 700
58 188 798 307 819 531 909 127 24 741 72 714 609
449 325 632 898 944 742 474 77 702 660 113 165 497
557 258 673 934 558 933 692 767 31 953 616 820 241
14 224 869 538 454 332 334 37 327 753 108 271 832
757 382 779 742 769 804 89 878 523 814 88 262 376
498 224 18 19 814 521 624 291 518 876 610 321 655
599 285 714 933 677 609 1 844 713 773 74 874 31
612 814 628 895 420 650 237 326 275 251 880 475 465
70 755 28 104 978 367 608 538 678 448 644 627 142
380 675 511 780 444 560 934 555 749 559 735 771 184
758 596 752 230 802 637 123 991 408 957 924 266 627
708 262 24 326 804 730 122 845 853 312 597 322 408
978 399 629 127 715 882 409 943 182 921 167 834 730
981 894 279 999 994 190 13 802 378 347 149 535 47
375 290 808 760 433 693 531 640 622 110 177 219 635
375 420 850 204 343 892 823 838 277 260 37 797 264
353 568 797 768 544 978 383 905 127 703 476 542 883
379 24 793 572 426 262 42 564 761 517 48 327 959
56 251 986 723 920 866 687 998 237 857 964 578 957
379 849 541 503 800 919 367 290 877 31 78 585 189
930 240 678 421 471 174 835 593 671 224 998 819 296
415 696 709 221 357 153 983 268 730 37 410 745 198
844 659 406 94 367 155 52 718 227 534 482 600 932
790 31 557 438 714 392 908 673 710 959 666 996 583
402 72 66 403 685 311 78 27 268 736 842 373 684
469 675 48 234 708 571 424 999 583 618 739 904 913
132 338 42 701 480 638 983 713 50 909 281 988 352
465 904 610 708 367 799 611 499 221 320 317 444 169
727 5 883 672 479 207 511 166 550 503 568
will be sorted!

 Time is : 3391 ms

Result of bucket-sorting :
```

```
bucket 0 : 10  10  9  5  3  1  5
bucket 1 : 20  14  19  16  14  18  19  13
bucket 2 : 28  21  26  27  25  28  21  24  27  24  28  24  24  27
bucket 3 : 37  36  36  33  40  39  38  31  37  31  37  31  37  31
bucket 4 : 43  43  45  50  49  42  47  42  48  48  42  50
bucket 5 : 59  54  52  52  58  56  52
bucket 6 : 68  68  62  70  66
bucket 7 : 79  76  79  73  72  77  74  78  72  78
bucket 8 : 84  84  82  90  85  88  89  88
bucket 9 : 97  98  95  94  95  94
bucket 10 : 107  106  102  108  108  101  108  104  110
bucket 11 : 119  115  112  116  115  112  113
bucket 12 : 125  126  127  130  121  122  128  124  129  126  121
            122  127  123  122  127  127
bucket 13 : 135  140  137  133  131  132
bucket 14 : 143  144  146  149  144  142  149
bucket 15 : 152  159  153  154  159  160  159  158  157  154  153
            155
bucket 16 : 161  164  168  165  166  166  165  167  169  166
bucket 17 : 176  172  172  177  172  176  179  178  177  174
bucket 18 : 188  183  183  184  190  190  188  184  182  190  189
bucket 19 : 200  198  191  197  193  194  195  198  198
bucket 20 : 208  207  204  204  201  209  204  207
bucket 21 : 213  219  213  216  220  214  219
bucket 22 : 226  230  222  229  225  224  224  230  224  221  227
            221
bucket 23 : 237  233  238  235  237  236  237  237  240  234
bucket 24 : 243  245  250  242  246  244  242  243  245  246  247
            249  248  241
bucket 25 : 252  256  260  258  251  260  251
bucket 26 : 264  263  267  265  262  266  262  264  262  268  268
bucket 27 : 277  280  276  271  275  279  277
bucket 28 : 283  285  285  290  290  281
bucket 29 : 298  295  293  292  300  294  297  300  291  296
bucket 30 : 309  305  302  307  301  310  304  303  307
bucket 31 : 315  319  319  314  312  311  320  317
bucket 32 : 326  326  325  327  321  326  326  322  327
bucket 33 : 332  336  339  339  332  334  338
bucket 34 : 347  343  347  350  344  346  342  346  341  347  343
bucket 35 : 356  358  358  352  352  358  351  354  359  351  351
            355  360  353  357  352
bucket 36 : 368  362  363  366  367  367  367  367
bucket 37 : 378  380  378  375  378  377  377  375  380  377  376
            380  378  375  375  379  379  373
bucket 38 : 390  389  382  381  387  381  386  389  381  390  382
```

```
              383
bucket 39 :  399  395  396  391  394  399  392
bucket 40 :  401  409  406  401  403  408  408  408  409  410  406
              402  403
bucket 41 :  416  418  416  418  411  411  420  420  415
bucket 42 :  427  423  428  427  423  424  423  426  421  424
bucket 43 :  432  431  432  436  432  437  435  440  435  433  438
bucket 44 :  442  446  445  448  447  449  448  444  444
bucket 45 :  458  455  454  451  454  454
bucket 46 :  464  461  465  470  468  468  469  465  469  465
bucket 47 :  477  476  477  480  476  480  477  474  475  476  471
              480  479
bucket 48 :  488  485  482
bucket 49 :  495  496  500  496  494  496  498  497  498  499
bucket 50 :  505  503  510  504  508  503  510  504  503  503  503
bucket 51 :  514  520  520  520  520  514  511  514  518  518  511
              517  511
bucket 52 :  522  527  526  528  521  522  529  523  521
bucket 53 :  535  535  536  534  536  531  538  538  535  531  534
bucket 54 :  541  546  549  544  542  541  550
bucket 55 :  551  553  555  557  552  552  560  558  555  557  558
              560  555  559  557
bucket 56 :  570  563  565  566  565  569  567  568  564  568
bucket 57 :  572  579  579  578  577  572  578  571
bucket 58 :  582  590  590  581  587  588  585  583  583
bucket 59 :  594  595  591  597  599  600  595  597  599  596  597
              593  600
bucket 60 :  607  604  607  607  605  605  605  609  610  609  608
              610
bucket 61 :  614  615  616  614  620  616  612  618  611
bucket 62 :  625  621  627  626  624  627  623  629  625  624  628
              627  627  629  622
bucket 63 :  638  631  634  636  633  632  637  640  635  638
bucket 64 :  646  647  644  647  645  644  647  641  650  650  644
bucket 65 :  651  653  655  651  660  659  652  655  660  655  659
bucket 66 :  669  666
bucket 67 :  675  676  674  671  678  671  675  673  677  678  675
              678  671  673  675  672
bucket 68 :  684  687  687  685  684
bucket 69 :  697  691  698  693  695  700  692  693  696
bucket 70 :  702  705  706  704  705  702  708  703  709  710  708
              701  708
bucket 71 :  716  711  713  720  720  719  714  714  713  715  718
              714  713
bucket 72 :  729  721  724  728  726  725  730  730  723  730  727
bucket 73 :  737  733  740  733  738  740  737  735  734  735  736
```

```
             739
bucket 74 :  750  744  749  741  742  742  749  745
bucket 75 :  758  758  757  755  758  755  756  753  758  753  757
             755  758  752  760
bucket 76 :  766  767  762  763  761  764  764  767  769  768  761
bucket 77 :  780  780  773  779  780  775  774  774  778  779  773
             780  771
bucket 78 :  785  790  782  789  788  789  786  788  790
bucket 79 :  800  799  791  795  798  797  797  793  800  799
bucket 80 :  809  803  802  801  805  804  802  804  802  808
bucket 81 :  815  820  814  820  819  820  814  814  814  819
bucket 82 :  823  827  823  828  829  823
bucket 83 :  838  835  839  831  840  833  838  837  832  834  838
             835
bucket 84 :  845  842  850  843  841  844  844  842  849  844  845
             850  849  844  842
bucket 85 :  855  851  851  853  860  855  853  857
bucket 86 :  870  867  864  861  866  869  869  869  869  866
bucket 87 :  873  872  871  875  876  874  871  872  878  876  874
             880  877
bucket 88 :  887  888  890  885  885  881  883  883  881  890  882
             883  883
bucket 89 :  895  896  895  897  898  898  895  894  892
bucket 90 :  907  908  904  908  909  905  908  904  909  904
bucket 91 :  917  915  915  920  913  914  914  920  919  913
bucket 92 :  927  926  928  924  921  930
bucket 93 :  932  938  934  933  933  934  932
bucket 94 :  948  945  946  948  950  944  943
bucket 95 :  958  957  956  958  956  957  954  953  957  957  954
             953  957  959  957  959
bucket 96 :  967  963  964  967  969  968  964
bucket 97 :  979  978  977  978  973  980  978  978  978  978
bucket 98 :  983  990  986  988  982  989  986  981  986  983  983
             988
bucket 99 :  992  991  999  994  998  998  996  999

 sorted!!!
```

Appendix E

The data used in example in chapter IV-A

All of the input data and outputs used for the tests in chapter IV-A are shown here; the raw data that is weekly closing stock prices of company A and B, corresponding bucket weight matrices obtained by the binary pattern net program in appendix 2, and the output obtained by the BAM program in appendix 1.

(1) Company A

Weekly closing stock prices for company A :

177.35
198.25
221.89
249.01
271.47
260.91
262.13
295.73
310.07
316.61
300.01
251.43

Binary input data obtained by Binary Pattern Net for company A :

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

(2) Company B

Weekly closing stock prices for company B :

265.84
264.38
257.72
249.01

239.56
231.62
229.54
229.97
241.03
260.90
300.01
323.98

Binary input data obtained by Binary Pattern Net for company B :

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

(3) Partial Input for company A

Weekly closing prices for the first three weeks for company A :

177.35
198.25
221.89
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0

Binary input data obtained by Binary Pattern Net for partial input :

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

```
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Output obtained by BAM for partial input :

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
```

(4) Noisy input for company A

Weekly closing prices with noise for company A :

```
177.35
198.25
221.89
249.01
271.47
251.00  -> noise
262.13
295.73
310.07
320.61  -> noise
300.01
251.43
```

Binary input data obtained by Binary Pattern Net for noisy input :

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
```

Output obtained by BAM for noisy input :

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
```

(5) Mixed input of company A and company B

Mixed prices for company A and B :

```
177.35
198.25
221.89
249.01
239.56
231.62
229.54
229.97
241.03
260.90
300.01
323.98
```

Binary input data obtained by Binary Pattern Net for mixed input :

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Output obtained by BAM for mixed input :

0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

REFERENCES

1. Wasserman, P.D., *Neural Computing: Theory and Practice*, Van Nostrand Reinhold, 1989.

2. Klimasauskas, C.C., *NeuralWorks:An Introduction to Neural Computing*, NeuralWare Inc., 1988.

3. Kosko, B., *Constructing an Associative Memory*, Byte, September, 1989, 137–144.

4. Lippmann, R.P., *An Introduction to Computing with Neural Nets*, IEEE ASSP Magazine, April, 1987.

5. Josin, G., *Neural-Network Heuristics*, Byte, October, 1987, 183–192.

6. Takeda, M., and Goodman, J.W., *Neural Networks for computation:number representations and programming complexity*, Applied Optics, Vol.25, No.18, (15 September, 1986).

7. Morse, K.G., *In an Upscale World*, Byte, August, 1989, 222–223.

8. Rosenfeld, E., *Neurocomputing - A New Industry*, IEEE proceedings , IV 831-838.

9. Kohonen, T., *Self-Organization and Associative Memory*, Springer-Verlag, Berlin,1984.

10. Hopfield, J.J., *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, Proc. Natl. Acad. Sci. USA, Vol.79, 2554-2558, April, 1982.

11. Baase, S., *Computer Algorithms*, Addison-Wesley Publishing, 1978.