# An Enhanced Learning for Restricted Hopfield Networks

Faezeh Halabian

Supervisors:

Dr. Iluju Kiringa

Dr. Tet Yeap

A thesis submitted to the University of Ottawa

in partial fulfillment of the requirements for the

M.A.Sc. degree in Electrical and Computer Engineering

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

# Table of Contents

**Table of Figures**

**List of Tables**

*To my loving husband, wonderful mother, and dear family*

# Acknowledgments

I would like to express my sincere gratitude to my supervisors, Professor Tet Yeap and Professor Iluju Kiringa for their great support and guidance during my M.A.Sc studies. The completion of this study would not have been possible without their consistent help and guidance. I also would like to thank my committee members, Professor Qi-Jun Zhang and professor Azzedine Boukerche for their brilliant comments and suggestions. Last but not the least, I would like to express my special thanks to my dear husband for his invaluable support which made my M.A.Sc career a memorable phase of my life, and also to my family for their constant encouragement and support.

**Abstract**

This research investigates developing a training method for Restricted Hopfield Network (RHN) which is a subcategory of Hopfield Networks. Hopfield Networks are recurrent neural networks proposed in 1982 by John Hopfield. They are useful for different applications such as pattern restoration, pattern completion/generalization, and pattern association.

In this study, we propose an enhanced training method for RHN which not only improves the convergence of the training sub-routine, but also is shown to enhance the learning capability of the network. Particularly, after describing the architecture/components of the model, we propose a modified variant of SPSA which in conjunction with back-propagation over time result in a training algorithm with an enhanced convergence for RHN. The trained network is also shown to achieve a better memory recall in the presence of noisy/distorted input. We perform several experiments, using various datasets, to verify the convergence of the training sub-routine, evaluate the impact of different parameters of the model, and compare the performance of the trained RHN in recreating distorted input patterns compared to conventional RBM and Hopfield network and other training methods.

# Chapter 1

# Introduction

In this thesis, we study the performance improvement and convergence of the training sub-routine for Restricted Hopfield Networks (RHN). We propose certain methods that result in a training algorithm with an enhanced convergence for RHN. The trained network is shown to achieve a better memory recall when the input is noisy/distorted. Before delving into details of the proposed solution, we first present an overview of RHN, its potential applications, and the deficiency of the existing training methods. Then we describe the problem for training RHN and the approach that we follow in this thesis.

## 1-1 Motivation

The field of machine learning has been attracting lots of attention, with the rise of Neural Networks. Neural Networks are computational processing systems that are inspired by the operation of the biological nervous system (such as the human brain) [1]. A deep

neural network combines multiple nonlinear processing layers, using simple elements operating in parallel and inspired by biological nervous systems. A neural network in general consists of an input layer, several hidden layers, and an output layer. The layers are interconnected via nodes, or neurons, where each intermediate (i.e., hidden) layer uses the output of the previous layer as its input. They collectively learn from the input in order to optimize the network final output. The input will be loaded, in the form of a multidimensional vector to the input layer which will calculate some features and distribute them to the hidden layers. The hidden layers will then make decisions from the previous layer, extracting new features. In the process of learning, it is evaluated how a change in the input weights determents or improves the final output. That is, the weights at each layer are determined so as to optimize the final output.

Neural Networks cover a wide variety of applications in different areas such as Industrial Control Systems, Electromagnetics, Digital Communications, Manufacturing, and on top of all pattern recognition. The field of pattern recognition is concerned with the automatic discovery of regularities in data using computer algorithms. Capturing such regularities, neural networks can take actions such as classifying the data into different categories [2]. There are different applications for pattern recognition in different areas such as statistical data analysis, signal processing, bio-informatics, image processing, information retrieval, data compression, computer graphics, and machine learning.

One of the use-cases which involves pattern recognition using recurrent neural networks is Associative Memory. They are also known as Content-Addressable-Memory (CAM), which are of two types, auto-associative and hetero-associative memories. Auto-association memory or an auto association network is any type of memory that can retrieve a previously stored pattern that resembles the current pattern. Hopfield has empirically demonstrated that the associative memory as a network is very attractive for many applications [3] and [4].

The network proposed by Hopfield is a cyclic neural network with feedback connections from outputs to the inputs. After being fed with an input, the state of the neurons will continue to change continuously until they converge or periodically oscillate. Hopfield formulated an Energy function for the network using the Lyapunov Direct Method showing that the network converges to a stable state if it has symmetric weights and each network node does not have self-feedback [5].

Hopfield neural networks are in two forms of discrete and analog. The main difference between these two forms is in the activation-function/mapping that is used at each node. In either form of Hopfield networks, the network can only be programmed to memorize patterns using the Hebbian Rule which has a limited memory capacity of storing 0.14N patterns, where N is the number of nodes in the network [5]. Since then, there have been so many efforts for improvement of the memory capacity and trainability of the network [6], [7]. For example, Gardiner [8], [9], [10] improved the network performance as an associative memory by passing the training patterns to the network iteratively. He used the perceptron convergence procedure to train each node to generate the correct state using the states of all the other nodes for a particular training vector.

Recently, an analog restricted Hopfield network [5] is proposed by Yeap to solve the problem of memory capacity and the trainability issue of the Hopfield network. The architecture of the RHN is similar to a Restricted Boltzmann Machine (RBM) [11], [12], [13]. The network is a fully-connected bipartite graph that consists of two layers of nodes, visible and hidden nodes, connected to each other via bidirectional weighted paths. The main difference of RHN architecture compared to the conventional Hopfield Network is that there is no intralayer connection in RHN. The input is presented at visible nodes which manage the flow of information to/from the hidden nodes. When an input vector is received by the RHN, the network iterates, sending signals back and forth between the two layers until all the nodes reach to an equilibrium state. Based on an

appropriate weight matrix which is set via a training sub-routine, the network is expected to generate some desired output vectors [5].

Since the output of the hidden nodes in RHN are not known, and given that the conventional Hebbian rule cannot handle analog quantities in general, the Hebbian rule is not a good choice for training RHN. Moreover, as argued in [5], applying the backward-error-propagation method may not be straightforward for RHN because of complications involved with the calculation of the gradient with respect to all the weights [5]. Therefore, the simultaneous perturbation stochastic approximation (SPSA), which was originally introduced by Spall in 1996 [14], [15], has been proposed in [5] as a simple method to train the RHN. SPSA uses gradient approximation that requires only 2 objective function measurements over each iteration (regardless of the dimension of the optimization problem) [11], [12]. The algorithm is simple to implement because the gradient of the objective function (known as loss function) can be estimated using only two function value measurements.

In this study, we propose a new training method for RHNs which shows a significant improvement in terms of convergence compared to the SPSA method. The proposed method not only enhances the convergence of the training sub-routine, but also is shown to enhance the learning capability of the network.

## 1-2 Problem statement

In order to exploit the potential advantages of the RHN, it is crucially important to develop an efficient training method. In this thesis, we strive to provide an enhanced training method for the RHNs, which then will be evaluated (in terms of convergence and performance) against other existing solutions. As already discussed, the conventional back-propagation technique is not readily applicable to recursive neural networks, such as RHN. So, we first strive to analytically calculate the gradient of network objective function (also known as loss function) for RHN using the back-

propagation technique. Then, we employ the gradient decent algorithm to iteratively update the weight matrix of the network. The gradient-decent algorithm, however, may end up with a local optimum solution. To address this issue, we propose a new variant of SPSA that is used to get out of a local optimum solution by updating the weight matrix in a random direction. The complexity of random search in the proposed scheme is reduced by confining the search space (only to components that have the greatest contribution to the loss function). The proposed methods not only enhance the convergence of the training algorithm for a RHN, but also are shown to enhance the learning capability of the network, especially in the presence of distortions at the input.

## 1-3 Methodology

Training a neural network such as the RHN involves finding a plurality of weights which optimize (i.e., minimize) an objective function. The quality of training (in terms of accuracy and complexity) depends on the proper definition of the objective/loss function and the way the loss function is optimized. By considering an appropriate loss function and using different optimization techniques (back-propagation, gradient decent algorithm, etc.) we strive to enhance the learning procedure for the RHN. Moreover, since the gradient decent method may get stuck at local optimum solutions, we propose a modified variant of SPSA with a reduced complexity to get out of the local optimum solution. We then implement the proposed scheme, by programming in Python, to evaluate the performance of the whole network, verify the convergence of the training sub-routine, evaluate the impact of different parameters of the model, and compare the performance of the trained network in recreating distorted input patterns compared to conventional RBM and Hopfield network and other training methods. Besides synthetic data, we use more extensive and practical datasets (such as Tensorflow.keras) to verify the convergence of the training sub-routine and evaluate the performance of the network, which demonstrate the robustness of the RHN against the distortions in practice.

## 1-4 Thesis organization

This thesis has been organized as follows. Chapter 2 presents the necessary background on neural networks, associative memories, and Hopfield networks. We also discuss the process of learning in neural Networks and provide further details about the existing training methods. Finally, this chapter presents the deficiencies of Hopfield Networks and describes why it is worthwhile to implement RHN. Chapter 3 presents the architecture of RHN and our proposed new training method. In this chapter, we present the implementation of the back-propagation technique for the RHN, as well as a new variant of SPSA algorithms. In chapter 4, we present the experimental results using the new training method and comparing that against existing solutions. Chapter 5 concludes the thesis by providing a summary and some potential directions for future work.

# Chapter 2

# Background and related works

## 2-1 An overview of the chapter

In this chapter, we present the necessary background on neural networks, as well as the state of the art on Hopfield networks, which we particularly investigate in this thesis. Specifically, we first provide an overview of different kinds of neural networks (such as feed-forward, and recurrent) and compare them in terms of training and their capability of learning for different application scenarios. We also review the back-propagation scheme which is a common training method for various neural networks and serves as a basis for our developments in Chapter 3. Then, we study the Hopfield network and other related work, address different applications of such recurrent neural networks, and also discuss custom training methods for this class of networks.

## 2-1 A brief review on Neural Networks

The field of machine learning has been attracting lots of attention, with the rise of the Artificial Neural Network (ANN). Artificial Neural Networks (ANNs) are

computational processing systems which try to mimic the operation of biological nervous systems (such as the human brain) [2]. A deep neural network combines multiple nonlinear processing layers, an input layer, several hidden layers, and an output layer. The layers are interconnected via weights and each hidden layer uses the output of the previous layer as its input. They collectively learn from the input in order to optimize the network final output. The basic structure of an ANN can be modeled as shown in Figure 2-1.

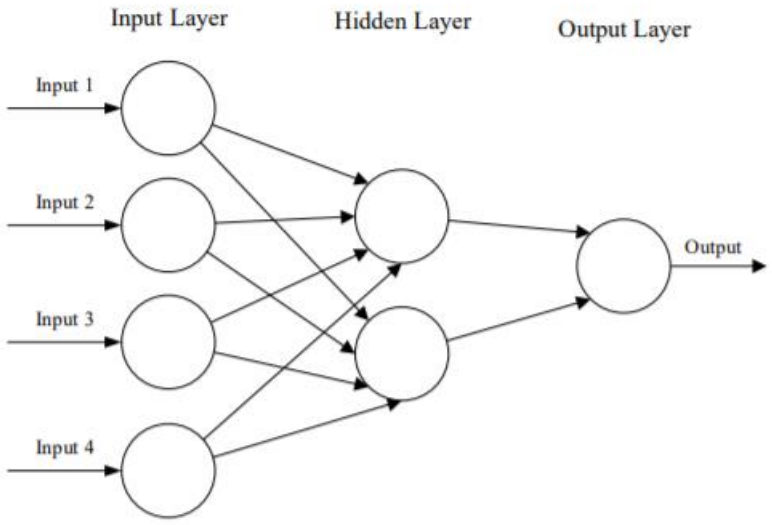


**Figure 2-1: A simple neural network with three layers, comprising an input layer, a hidden layer, and an output layer. This structure is the basis of common neural network architectures, such as Feed-forward Neural Networks (FNN) and Recurrent Neural Networks (RNNs) [2].**

There are two key learning paradigms to train the network in Machine Learning applications; supervised and unsupervised learning. Supervised learning is learning through pre-labeled inputs, which act as targets. For each training example, there will be a set of input values (vectors) and one or more associated designated output values. The goal of this form of training is to reduce the model's overall classification error, through the correct calculation of the output value for training examples. Unsupervised learning differs in that the training set does not include any labels [2]. Success is usually

determined by whether the network is able to reduce or increase an associated cost function. However, it is important to note that most image-focused pattern-recognition tasks usually depend on classification using supervised learning.

### 2-1-1 Feed-forward Neural Networks

Feed-forward Neural Networks are the first and simplest type of ANNs [16]. They are called feed-forward because the flow of information travels only in the forward direction from the input layer to the hidden layer and finally to the output layer [17]. Also, the connections between nodes should not form a cycle [18][19].

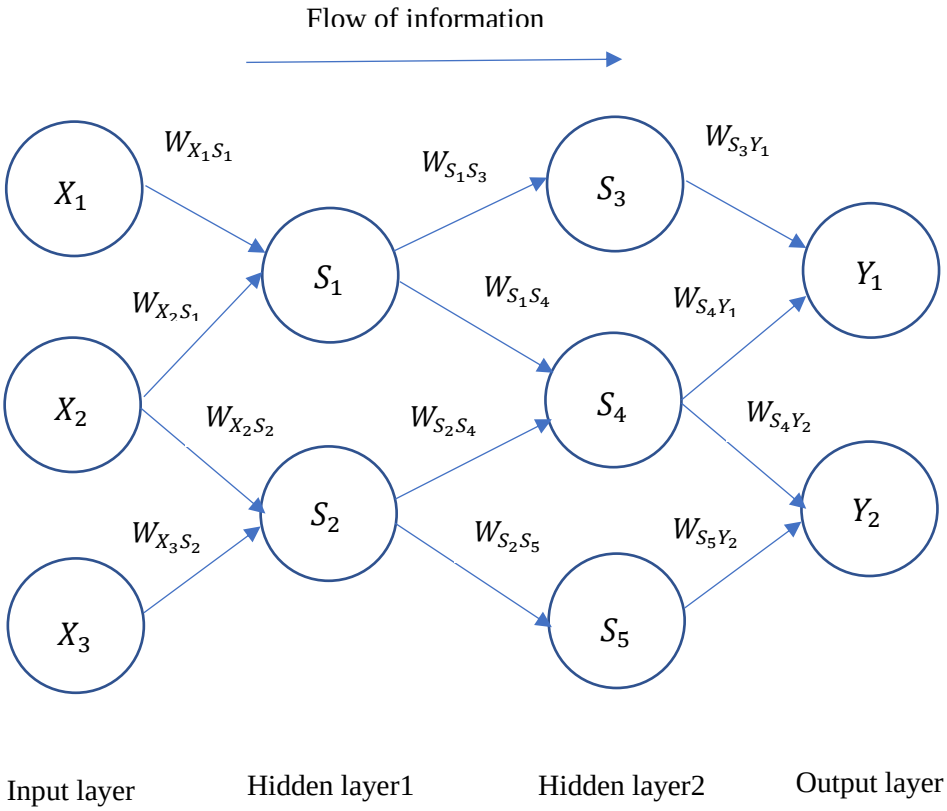


**Figure 2-2: The general architecture and flow of information in a feed-forward network**

FNNs are mostly used for supervised learning where we already know the result/output of the network and there is no sequential or time-dependent data. The main target in

these networks is to approximate function $f$ on fixed-size input x in a way that $f(x) \approx y$ for training pairs (x, y) [20].

- *Single-layer perceptron*

Single-layer perceptron is the simplest form of a FNN. A perceptron is a unit in the neural network that does simple calculations to detect certain features of the input data [21]. It enables the network to learn the weights for the inputs and draws a linear decision boundary [22]. A perceptron has only one input layer and one output layer, and no hidden layer. The output units are computed directly based on the weighted summation of the input units plus some *bias* values.

The perceptron's output is typically a binary value. In this case, the weighted summation is passed into a *step function* which maps the output values to either 0 or 1 [23]. If $\mathbf{x}$ denotes the vector of input units, and $W = [w_{i,j}]$ the weight matrix for connection from every input unit $j$ to output unit $i$, $\boldsymbol{z} = W\mathbf{x} + \boldsymbol{b}$ denotes the input to the step function. The output of each unit then is $v_i = 1$ if $z_i \geq 0$, and $v_i = 0$, otherwise.

Alternatively (or, in the case of perceptron with analog output), the weighted summation of the inputs may pass through a non-linear mapping (which is the so-called *activation* function) before passing into the step function. The following two functions are typically used as activation functions [24][25]:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2-1}$$

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2-2}$$

The first one, called the sigmoid function maps every real-valued input to the range of (0,1), whereas the second one, so-called (hyperbolic) tangent function maps the inputs to the range of $(-1,1)$. In any case, a single-layer perceptron is only a linear classifier.

That is, they may not even implement simple non-linear classifications, such as an XOR operation. For this sort of non-linear mappings, one needs to employ a multi-layer perceptron.

- *Multi-layer perceptron*

A more generic form of FNN is ***multi-layer perceptron*** (MLP) which was proposed by Rosenblatt in 1950 [26]. They consist of more than one layer of the perceptron, cascaded along with each other, so that they are capable to implement *non-linearly separable mappings* [27] (such as an XOR operation, for instance [28], [29]). Despite single-layer perceptron, MLPs have at least one hidden layer each composed of multiple perceptrons [30] (Figure 2-3).

Since MLPs are classified as a FNN, there are no loops in their computation graph. That is, their connection graph is a *directed acyclic graph* (DAG) where no layer's output depends on itself. Due to this particular structure, MLPs, and FNN in general, have usually a rather simple learning algorithm compared to *recurrent neural networks*, which have cycles in their dependency graphs. We present an overview of back-propagation, which is the most common method for training FNN [31] in Section 2-2, after completing our survey on recurrent neural networks.

Figure 2-3 shows an MLP with two hidden layers that computes a one-dimensional output and an n-dimensional input $x = \{x_1, \dots, x_n\}$.

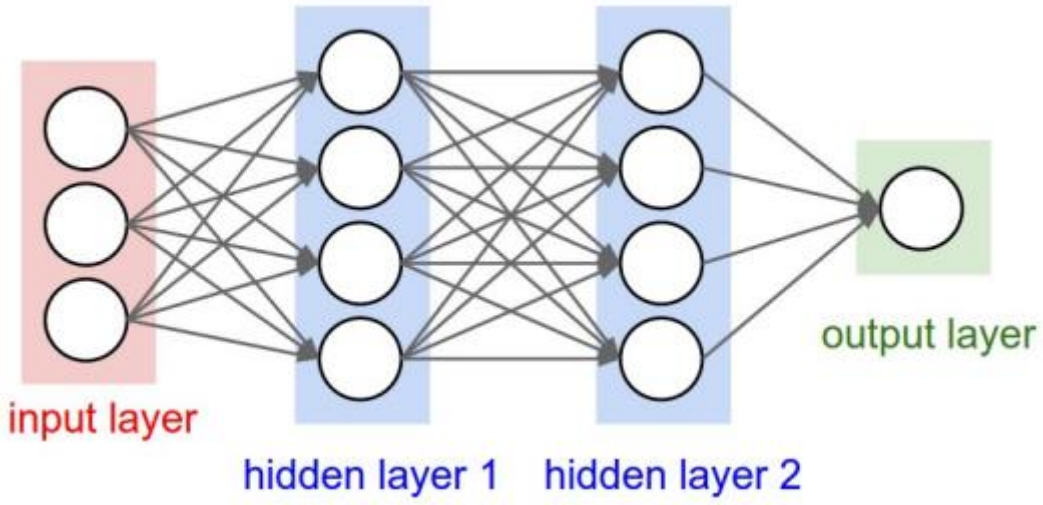


**Figure 2-3: A feed-forward network with multi-layers of perceptrons [32]**

As you can see from Figure 2-3, every perceptron in layer $l$ is connected to every perceptron of layer $l-1$. That is, every perceptron output depends on the output of all perceptrons in its previous layer [33]. Hence, the output of a feed-forward neural network can be found through the following steps:

1) Feed the input layer ($l_0$)
2) Calculate the weighted summation and outputs of each hidden layer from $l_1$ to $l_{m-1}$
3) Find the output **v** at the very last layer $l_m$

### 2-1-2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of ANNs which can work with sequential data or time-series data. They are suitable for problems with an ordinal or temporal nature, such as natural language processing [34], moving object detection, pattern recognition, and speech recognition. RNNs are designed to take a series of inputs with no predetermined limit on size. The output of these networks depends on the prior elements within the sequence, as opposed to traditional deep neural networks where it is assumed that inputs and outputs are independent of each other [35], [36].

Figure 2-4 shows how the output and state of the network evolve over time for a simple RNN. It is observed how the current and future outputs are affected by the previous input of the network. The role of the hidden node(s) here is to capture the relationship between neighbour inputs which may exist in a serial input. The input might be also changing in every step so that every input undergoes a different transition depending on the state of the network.

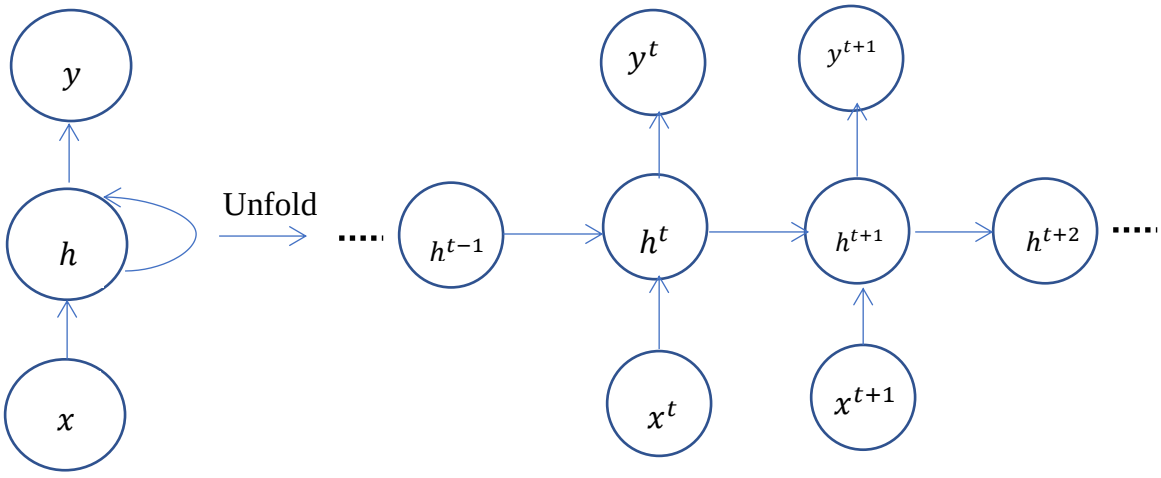


**Figure 2-4: Unfolding the operation of a simple recurrent neural network over tine [34]**

Figure 2-5 shows a more generic, but yet simple architecture for a Recurrent Neural Network. Another characteristic of recurrent neural networks, which is distinguished from FNNs, is parameter sharing [37]. In feed-forward networks, separate weight matrices are used at each layer, while in recurrent networks the same weights will be shared across different layers (corresponding to different iterations) of the network.

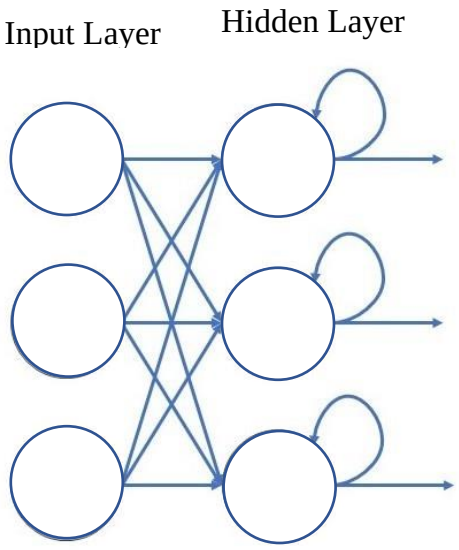


**Figure 2-5: The general architecture for a recurrent neural network**

## 2-2 Back-propagation

Back-Propagation (BP) is a training method that applies to a broad family of Artificial Neural Networks (ANN), whose architecture consists of multiple interconnected layers [38]. Particularly, to train an ANN to learn a desired input-output mapping (such as a non-linear function, y = f (x)), we usually define a loss function, $J(\boldsymbol{W})$, which captures a measure of the error at the output [39]. The output of the network is a function of the weights across the network, and so is the loss function. The loss function usually has a lower bound of zero that is attained when the error approaches zero. Hence, the objective is to find the weights, $\boldsymbol{W}$, such that the loss function is minimized. Gradient-decent (or steepest-decent) algorithm is the most common method to iteratively solve such a minimization problem when the objective function is well-defined (i.e., has a derivative with respect to optimization parameters). According to the gradient-descent scheme, one can iteratively reduce the loss/cost function by updating the optimization parameters in the opposite direction of the gradient vector [40]. In order to employ the gradient-descent method to solve the training optimization problem for an ANN, one needs to find the derivative of the loss function with respect to every element of a given weight matrix. In

the general, one needs to apply the chain rule to find the derivative of the loss function with respect to every element of the weight matrix. This calculation however could be complicated due to the layered architecture of the network. Backpropagation is a systematic method which exploits the layered architecture of the network to find the derivative of the loss function with respect to the weights across different layers of the network.

Backpropagation is generally applicable to every network with a loss function which is a function of the network output and can be written as a summation of components for different input patterns (i.e., training examples). In this method, every input pattern is first fed to the network (in the usual forward direction) to find the output at each layer of the network. Then, starting with the very last layer, one may find the error at the output layer (for a given training example) which is subsequently back-propagated to find the error at the preceding layers. In this way, one can find the gradient for every component of the loss function corresponding to each training example, and then find the overall value of the gradient vector.

Concretely, let $W^{(l)} = \left[w_{i,j}^{(l)}\right]$ denote the weight matrix at the $l^{th}$ layer of the network, where $w_{i,j}^{(l)}$ is the weight of the edge connecting node $j$ at layer $l-1$ (where $l=0$ is the input) to node $i$ at layer $l$. The input to each layer, which has a preceding layer, is the weighted summation of the output of the previous layer, $\boldsymbol{z}^{(l+1)} = W^{(l)}\mathbf{v}^{(l)}$, where $\mathbf{v}^{(1)} = \mathbf{x}$ is the input. The output of layer $l+1$ is found bypassing $\boldsymbol{z}^{(l+1)}$ through an activation function, $\mathbf{v}^{(l+1)} = g(\boldsymbol{z}^{(l+1)})$. As an example, we can consider the logarithmic loss function [41]:

$$J(W) = -\frac{1}{P}\sum_{p=1}^{P}\sum_{l=1}^{L}\left[y_i^p\, log\left(v_i^p\right) + \left(1-y_i^p\right) log\left(1-v_i^p\right)\right], \tag{2-3}$$

or the mean-square loss function,

$$J(W) = -\frac{1}{P}\sum_{p=1}^{P}\sum_{l=1}^{L}\left[y_i^p - v_i^p\right]^2, \tag{2-4}$$

where $\mathbf{y}^p = [y_i^p]$ is the desired output for the training example with index $p$. Indeed, $J(W)$ measures how different is the current output $\mathbf{v}^p$ compared to the desired $\mathbf{y}^p$. The output for each training example, $\mathbf{v}^p(W)$ is a function of the weights that are used across the network, and so is the $J(W)$. The goal is to find the derivative of $\partial J/\partial w_{i,j}^{(l)}$. According to the chain rule

$$\frac{\partial J}{\partial w_{i,j}^{(l)}} = \frac{\partial J}{\partial z_i^{(l)}}\frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial J}{\partial z_i^{(l)}}\boldsymbol{v}_j^{(l-1)} \tag{2-5}$$

where the first term $\delta_i^{(l)} \coloneqq \partial J/\partial z_i^{(l)}$ is defined as local gradient/error for node $i$ at layer $l$. Starting with the last layer, $\delta_i^{(L)} = y_i^p - v_i^p$, it is shown that local error at preceding layers can be found by back-propagating the error according to the following formula

$$\boldsymbol{\delta}^{(l)} = \left(\boldsymbol{W}^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)} .* \boldsymbol{g}'\left(\boldsymbol{z}^{(l)}\right), \tag{2-6}$$

where ".∗" is element-wise multiplication. For the sigmoid activation function, it can be shown that $\boldsymbol{g}'\left(\boldsymbol{z}^{(l)}\right) = \mathbf{v}^{(l)} .* [\mathbf{1} - \mathbf{v}^{(l)}]$.

In summary, by feeding the network with different training examples, one can find the output at each layer. Then, starting with the very last layer, the local error is found at the output $\boldsymbol{\delta}^L$. By back-propagating the error according to (2-6), the local error can be subsequently calculated at the preceding layers. According to (2-5), the derivative of the loss function with respect to $w_{i,j}^{(l)}$, corresponding to the edge connecting node $j$ at layer

$l-1$ to node $i$ at layer $l$, is given by output of node $j$, $v_j^{(l-1)}$, times the error at node $i$, $\delta_i^{(l)}$.

Now, given the gradient, we can use the gradient descent method to update $W$ in the opposite direction of the gradient.

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \alpha \frac{\partial J(W)}{\partial w_{i,j}^{(l)}} \tag{2-7}$$

This process decreases the value of the loss function repeatedly until it converges to a local minimum. The initial values of the weight matrix are set randomly at the beginning. In the next chapter, we discuss, how to apply this method to restricted Hopfield networks which are from the class of recurrent neural networks. We also present an innovative method to get out of local optimum points, to achieve a global optimum solution using the back-propagation scheme.

## 2-3 Related work

In this section, we cover the related work on using recurrent neural networks as an associative memory; discuss different kinds of neural network architectures, and the existing training methods. Particularly, we first study the related work on Hopfield network and bidirectional associative memories and discuss the custom training methods that are proposed for such recurrent networks. Then we present the architecture of the restricted Hopfield network, which is recently proposed in [5] to address the limited memory capacity of Hopfield networks. We also review the existing work on Boltzmann machines as a class of *stochastic neural networks* that have a similar architecture to the RHN, but with application to stochastic pattern analysis.

### 2-3-1 Associative memory

An associative memory (AM) is a special kind of neural network that are designed to map the inputs to a certain set of output patterns where the inputs might be corrupted by noise or other sorts of distortions. There are two kinds of associative memory: Auto-associative and hetero-associative memories [42].

- *Auto-associative versus hetero-associative memory*

Auto-associative memory is indeed a content addressable memory which is able to resemble the current pattern by retrieving a previously stored pattern which is the most similar. They are designed to filter the noise/distortion of the input or recreate a pattern when only part of it is available at the input. Hetero-associative memory, on the other hand, is another type of associative memory which is designed to recognize the set of paired patterns for which the recalled patterns are different from triggering patterns.

Figure 2-6 shows the architecture of the simplest form of an associative memory which is called linear associator. It is a 2-layer fully connected network for which the output is generated according to a single feed-forward computation. As in FNNs, the output nodes are connected to the input units via the edges which are weighted by $W = [w_{ij}]$. The weight matrix, $W$, is set such that the network can store different input-output pattern pairs, $\{(x_k, y_k) \mid k = 1, 2, \ldots, N\}$.

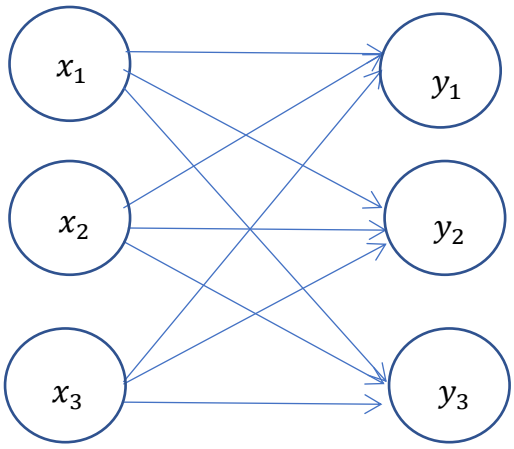


**Figure 2-6: The architecture of a single-layer auto-associative memory**

Indeed, the goal is to find the weight matrix W such that when an input pattern is presented, the stored pattern which is the mapping of the input pattern is retrieved. As in other feed-forward neural networks, the weight matrix can be found using back-propagation or other optimization-based training methods. In the next sub-sections, we discuss some recurrent neural networks which are proposed as associative memory.

### 2-3-2 Hopfield Network

In 1982, John Hopfield [1], [2], [3] proposed an influential recurrent neural network that was able to store and retrieve memory like the human brain. The architecture he proposed can be used for many potential applications in different areas such as optimization engine for the traveling-salesman problem, solving other combinatorial optimization problems, using as an associative memory, and also for pattern recognition.

To be used as an associative memory, a Hopfield network should be initially trained to store several patterns. Then it would be able to recognize any of those learned patterns, being feed by even some corrupted/distorted inputs.

- *The architecture of Hopfield Network*

A Hopfield network is a single-layer recurrent neural network that consists of fully connected nodes, i.e., every node is connected to every other node (Figure 2-7). Each node can act as input or output and the connection weights between the nodes are symmetric with no self-connectivity. That is:

$$w_{i,j} = w_{j,i}, \qquad \forall i, j, \tag{2-8}$$

$$w_{i,i} = 0, \qquad \forall i. \tag{2-9}$$

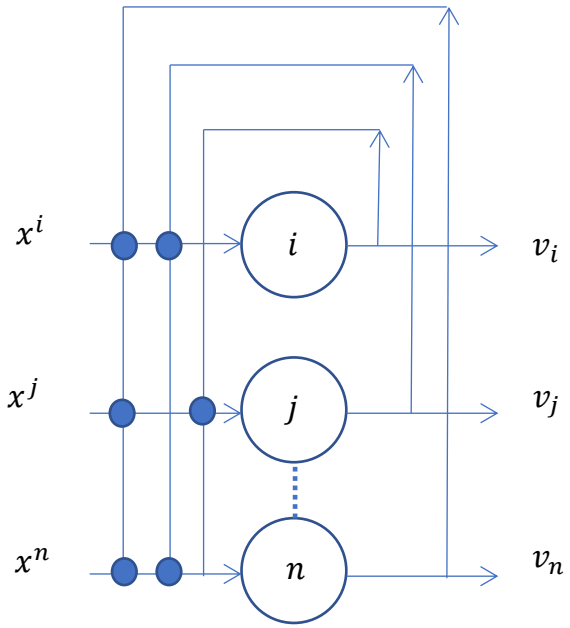


**Figure 2-7: The general architecture of a Hopfield network**

The dynamics of the network can be described by the following equation:

$$V_i = g\left(\sum_{i \neq j} w_{i,j} V_j + x_i\right), \tag{2-10}$$

where the activation function $g(x)$, can be either sigmoid or hyperbolic tangent. The state of each node/neuron in this network depends on its own input as well as the state of other neurons in the network. Being a recurrent neural network, the neurons repeatedly exchange their state variables until the network eventually settles down and returns a stable pattern, which is supposed to be the closest or the best guess for the desired pattern. The network is indeed designed in a way to mimic the human brain [4].

Following the Lyapunov Direct Method, it is shown that the network has an energy function, which is minimized (under certain conditions) as the network state evolves [4]:

$$E = -\frac{1}{2}\sum_{i,j=1}^{N} w_{i,j} x_i \, x_j. \tag{2-11}$$

Hence, it is shown that the network converges to a stable state provided that it has symmetric weights and each network node does not have self-feedback [4]. Hopfield network can be in either forms of analog or discrete. In both formats, it can only be programmed to memorize patterns using the Hebbian Rule.

- *Training of Hopfield Network with Hebbian rule*

Hopfield networks can be programmed to memorize patterns using the Hebbian Rule. Hebbian rule has been proposed by Donald Hebb in 1949 [45]. It is a common learning rule in the area of Neural Networks which is inspired by simplified physiological models to mimic the activity-dependent features of synaptic plasticity.

In case the outputs are updated using a sigmoid activation function $g(.)$ as in (2-1), then the output vector is in the range (0, 1). Given the binary input vectors $(S(p), p = 1..P)$, the weights for training the network, in this case, are calculated according to the following formula:

$$W_{i,j} = \sum_{p=1}^{P} (2S_i(p) - 1)(2S_j(p) - 1). \quad (2\text{-}12)$$

When a hyperbolic tangent activation function is used, the output vector is in the range $(-1, 1)$ and the weight matrix is set according to the following formula:

$$W_{i,j} = \sum_{p=1}^{P} S_i(p) S_j(p). \quad (2\text{-}13)$$

Intuitively, the Hebbian rule sets the weights according to a correlation metric across different patterns. Accordingly, the trainability of the network with the Hebbian rule depends on cross-correlation of the patterns. Indeed, correct recalling of all the patterns cannot be guaranteed unless the patterns are orthogonal [47].

- *Information capacity of Hopfield network*

For random input patterns, the information capacity of the network is shown to be at most $0.14N$, while accepting a small error in re-creating the patterns, where $N$ is the number of nodes in the network [48]. Without accepting any error at the output, the capacity of the Hopfield network is asymptotically (i.e., as $N$ goes to infinity) upper bounded by $N/4 \ln N$ [48]. It means that the larger is the number of nodes, the worse is the *utility factor* (defined as the ratio of the capacity to the number of processing nodes in the network [43]). Moreover, the capacity is achieved for *"orthogonal patterns"*. The capacity of the network would be far less when the patterns are correlated [47].

### 2-3-3 Bidirectional Associative Memory

Bidirectional associative memory (BAM) is a recurrent neural network which consists of two layers of nodes [49][50]. They are originally proposed to operate as a *hetero-associative memory*. A hetero-associative memory is particularly designed to recognize

the set of paired patterns for which the recalled patterns are different from triggering patterns.

Figure 2-8 shows the basic architecture of a BAM, a recurrent neural network comprising two layers of nodes, wherein the input, denoted by $\mathbf{x} = [x_1, x_2, \dots, x_N]$, is presented at the first layer, and the output, denoted by $\mathbf{y} = [y_1, y_2, \dots, y_M]$ is generated at the second layer. The input nodes are connected to the output via bidirectional links taking symmetric weights. The nodes at each layer, however, are not inter-connected. When a (noisy) pattern is presented at the input, the signal is iteratively sent from the input to the output and from the output back to the input layer, until a stable state is established at the output. The recurrent nature of the network is expected to help to filter out the input noise as the signal is successively updated [49].

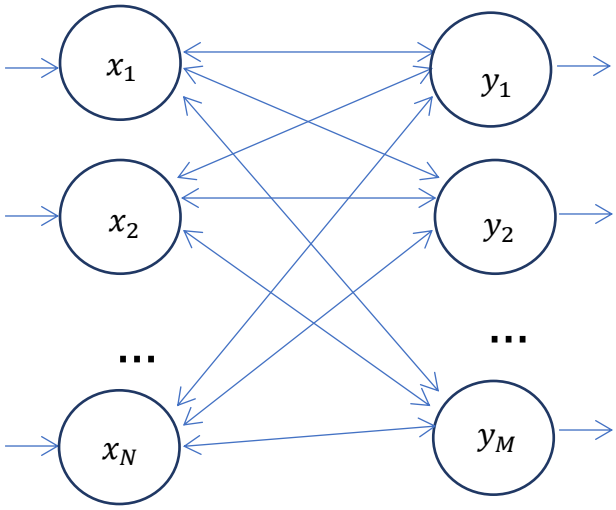


**Figure 2-8: The architecture of a bidirectional associative memory**

The weight matrix is to be determined so that the network can memorize the pattern-pairs at their stable state. Let $W = [w_{i,j}]$ denote the weight matrix, where $w_{i,j}$ is the weight of the (symmetric) link between node $j$ at the input layer, and node $i$ at the output. Given the specific architecture of the network (ruling out inter-layer connections), it is shown in [49] that the network always converges to a stable state for every choice of the

weight matrix. Moreover, an extension of the Hebbian rule is proposed in [49] to set the weight matrix for a given set of associated pattern pairs, $\{(\mathbf{x}^{(k)}, \mathbf{y}^{(k)}) \mid k = 1, 2, \ldots, P\}$,

$$w_{ij}(k) = \beta \sum_{k=1}^{P} x_j{}^{(k)} y_i{}^{(k)}, \tag{2-14}$$

where $\beta$ is normalizing constant which is typically $\beta = 1/N$. Such a weight matrix indeed superimposes the information of different pattern pairs. So unless the training patterns are orthogonal, the superimposition may introduce distortion which makes it impossible to perfectly recreate all the patterns [43]. A relaxation-based training method (which is discussed in further detail in the next sub-section) is proposed in [43] to improve the trainability of BAM. Either of Hebbian rule and or relaxation-based methods, however, can be used to learn the desired pattern with binary values. In [44] an optimization-based learning method, called SPSA, has been proposed for training BAM which can handle both binary and analog values. We will study this method in the next chapter, where we discuss its application for training RHNs.

### 2-3-4 Studies on training BAM and Hopfield networks

As already discussed, both BAM and Hopfield networks can be trained by the *Hebbian rule* using a correlation matrix which superimposes the information of different training patterns. This method, however, results in a poor information capacity for the network, especially when the training patterns are correlated or linearly dependent [47]. There are a considerable number of studies which investigate alternative training methods to enhance the trainability of this sort of recurrent neural networks.

For example, Gardiner [8], [9], [10] proposed a method to improve the performance of the Hopfield network as an associative memory by presenting the training patterns repeatedly and using the perceptron convergence procedure to train each node.

Particularly, it generates the correct state given the states of all the other nodes for each training vector, rather than trying to memorize the patterns in one presentation cycle.

A more generic training method (that applies to both BAM and Hopfield network) is proposed in [43] which avoids the complexities of solving an optimization problem (such as finding the gradient, choosing the step-size, and getting stuck in a local minimum). Towards this, the training problem for a broad class of binary recurrent neural networks is formulated as solving a system of linear inequalities. Then, a relaxation-based method is applied where the weight matrix is iteratively updated by considering the training examples one by one while relaxing the training constraints for other patterns at each time.

To gain more intuition, consider the training of a BAM with binary desired output patterns. The problem is to find a weight matrix which maps every input pattern $\mathbf{x}^{(k)}$ to the desired pattern $\mathbf{y}^{(k)}$. Hence, the following system of linear inequalities should be satisfied for all nodes and for every pattern $k = 1, 2, \dots, P$ [43].

$$\left(\sum_{j=1}^{N} w_{i,j} {x_j}^{(k)} + \theta_j\right) y_i^{(k)} > 0, \qquad for\ i = 1, 2, \dots, M \tag{2-15}$$

$$\left(\sum_{i=1}^{M} w_{i,j} {y_i}^{(k)} + \theta_i\right) x_j^{(k)} > 0, \qquad for\ j = 1, 2, \dots, N \tag{2-16}$$

The relaxation technique is an iterative method for solving the above system of linear inequalities. Each linear inequality describes a half-space. By examining one-equality at a time, the corresponding vector (i.e., row/column of the weight matrix) is updated by moving in the normal direction to the half-space boundary. The step-size is determined based on the distance from the boundary so that the considered inequality is satisfied after the update. One can evaluate the inequalities one-by-one in an arbitrary order, or

choosing the inequality with the maximum distance/violation in each iteration [51]. Alternatively, one can cycle through different inequalities in an arbitrary order, but updating those for which the distance/violation is greater than a pre-determined threshold. In this way, one can reduce the violation for inequalities one-by-one. The whole procedure is then shown to converge to a feasible solution (if there exists any) to the system of linear inequalities [43].

A similar method is used in [52] to solve the training problem for a broader class of BAM (so-called *dynamic associative memory*) wherein asymmetric weights are assumed in the forward and reverse directions. In this case, the weights for each layer are iteratively updated, while assuming a stable output for the other layer.

This training method applies to a wide variety of (recurrent) neural networks (such as Hopfield, BAM, DAM, and the like) for which the training problem can be described by a system of linear-inequalities. A major limitation, however, is that it's only applicable for desired outputs of binary values. Also, it's only applicable to the network architectures where the desired output of each node is known in advance, and not to the architectures with hidden nodes (such as the RHN model that is presented in the next subsection). In the next chapter, we present some optimization-based techniques which can handle analog quantities as well.

### 2-3-5 Restricted Hopfield network

As shown in Figure 2-9, the architecture of the Restricted Hopfield network [5] consists of two layers of nodes, $N$ visible and $M$ hidden nodes. They are connected to each other by directional weighted edges in a way that forms a fully-connected bipartite graph with no intralayer connection. The visible nodes perform both as input and output nodes managing the flow of information. Particularly, they get the inputs as their initial state and generate the output after passing the information through the hidden nodes.

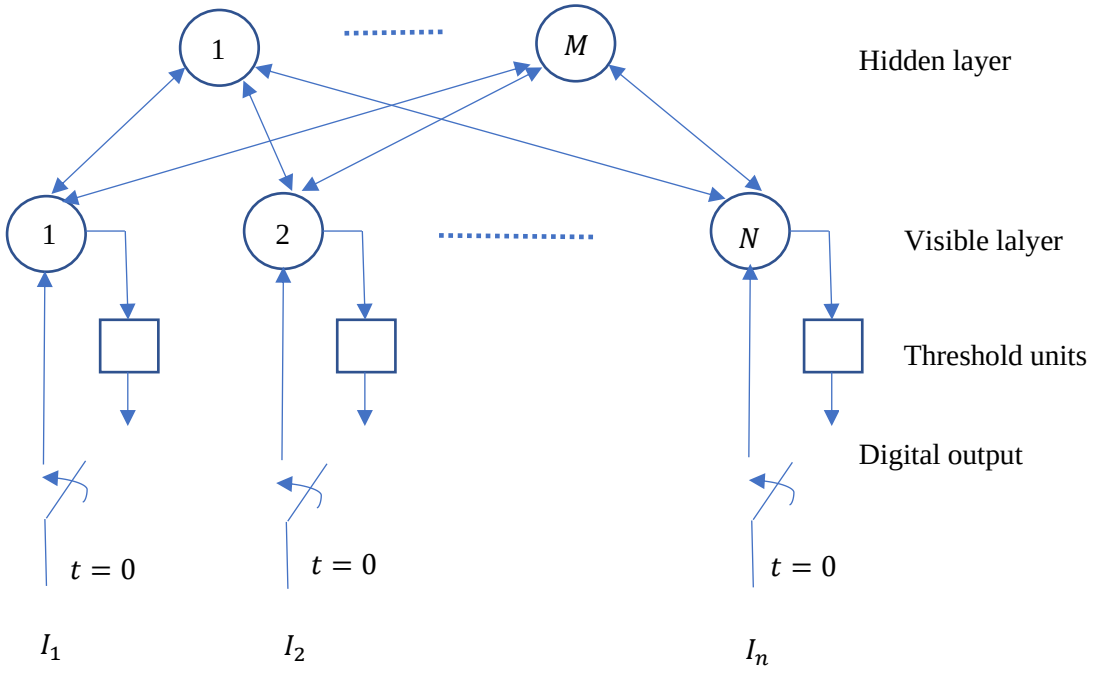


**Figure 2-9: The general architecture for restricted Hopfield network**

Specifically, let $V^V = [V_j^V], j = 1, 2, \dots, N$, denote the vector of visible node outputs, and $V^H = [V_i^H], i = 1, 2, \dots, M$, denote the outputs of the hidden nodes. The network starts running from an initial state condition:

$$V_j^V\ (0) = I_j, \tag{2-17}$$

$$V_i^H(0)\ =\ 0, \tag{2-18}$$

where $I_j$, the initial state of visible node $j$, can be considered as the input to the network. The dynamics of the network then is described by the following differential equations. Particularly, in the forward path:

$$\frac{du_i^H}{dt} = \sum_{j=1}^{N} w_{ij}^H V_j^V + \theta_i^H, \tag{2-19}$$

$$V_i^H = g(u_i^H). \tag{2-20}$$

In the above equations:

- $u_i^H$ is the input to hidden node $i$, which is updated proportionally to the weighted summation of the outputs from the visible nodes.
- $w_{ij}^H$ is the weight for the edge connecting the output of visible node $j$ to the input of hidden node $i$.
- $\theta_i^H$ is a bias that is introduced at the input of hidden node $i$.
- $g(.)$ is the activation/output function (such as *sigmoid*) for different nodes.

In the backward direction, the output of visible nodes is updated in a similar fashion:

$$\frac{du_i^V}{dt} = \sum_{j=1}^{M} w_{ij}^V V_j^H + \theta_i^V, \tag{2-21}$$

$$V_i^V = g(u_i^V), \tag{2-22}$$

where

- $u_i^V$, is the input to visible nodes, which is updated proportionally to the weighted summation of the outputs from hidden nodes.
- $w_{ij}^V$ is the weight for the edge connecting the output of hidden node $j$ to the input of visible node $i$.
- $\theta_i^V$ is a bias that is introduced at the input of visible node $i$.

Like the Hopfield Networks, an RHN can get either digital or analog values as the input. For the output/activation function $g(.)$, either a sigmoid or a hyperbolic tangent function can be used for all the nodes. Accordingly, the output of all the nodes takes a value between 0 and 1 when using the sigmoid function and takes a value between -1 and 1 if

using the hyperbolic tangent output function. To obtain digital outputs, threshold units can be deployed as shown in Figure 2-9.

Lyapunov Direct Method can be used to show the convergence of the RHN which described by the above system of differential equations. Particularly, it can be shown that the following equation is the energy or a Lyapunov function for the RHN if and only if all the weights are *symmetric*.

$$E = -\frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{N} w_{ij}^{H} V_i^{H} V_j^{V} - \frac{1}{2}\sum_{j=1}^{N}\sum_{i=1}^{M} w_{ji}^{V} V_j^{H} V_i^{V} - \sum_{i=1}^{M} V_i^{H}\theta_i^{H} - \sum_{i=1}^{N} V_i^{V}\theta_i^{V} \tag{2-23}$$

According to the Lyapunov direct method, the system of differential equations implemented by the RHN increment the energy function, which has an upper bound, provided that the weight matrix is *symmetric*. Hence, the network needs to converge. Indeed, when an input vector is presented to the network, the proposed network iterates, sending signals back and forth between the two layers until all its nodes reach an equilibrium wherein the energy function is maximized. In summary, the weight matrix is assumed to be *symmetric* hereinafter, so that the network always converges. We discuss the SPSA scheme as well as a new training method for RHN in Chapter 3. We conclude this chapter by presenting another class of recurrent neural networks with an architecture similar to RHN, which can be viewed as a stochastic counterpart for RHN.

### 2-3-6 Restricted Boltzmann machine

Boltzmann machine is a stochastic recurrent neural network consisting of two layers of visible and hidden nodes connected via symmetrically weighted connections where each node makes probabilistic decisions to be on or off. The whole network then can be programmed to learn stochastic patterns over its set of inputs [53]. Implementing a

generative model, Boltzmann machines have been used for various applications, such as dimensionality reduction (by extracting *latent factors*) [54], collaborative filtering/prediction [56], and feature learning which lets them serve as a building block to form a "*deep belief neural network"* [57], [58].

A restricted Boltzmann machine (RBM) comprises two layers of nodes which form a bipartite graph, wherein there is no intra-layer connection between any of visible (or hidden) nodes [59], [60]. This is in contrast to the "unrestricted Boltzmann machine" where there could be intralayer connections between hidden nodes [53]. The restricted version, however, is shown to result in more efficient training algorithms, such as a gradient-based contrastive divergence algorithm [61], [62]. Indeed, a Boltzmann machine is to be trained in a way that hidden nodes contain latent factors of the input data, so that the input can be regenerated based on the state of the hidden nodes. This process is so-called factor analysis. An RBM indeed implements a binary version of factor analysis [54].

To gain some intuition, assume that each user of a video streaming service has certain preferences over a set of specific movies. Based on this information (whether or not each user likes each of the movies), one can derive certain *latent factors* of the user preferences. For example, a user who likes movies such as "Star Wars" and "Lord of the Rings" may have a strong association with a latent factor of fantasy and science fiction. Given the user preferences for different movies, one can associate each user with various latent factors such as action, drama, tragedy, fantasy and science fiction, and the like. An RBM can be trained in a way that when the user preferences are set at the visible nodes, the state of the hidden nodes represents the latent factors (e.g., different types of movies) that the user is *most probably* associated with. Conversely, given the latent factors that a user is associated with, the network can suggest the list of movies that are probably preferred by the user.

- *Implementation of the RBM*

Assume a network of $M$ hidden nodes ($N$ visible nodes) that are trained to extract a certain number of $M$ latent factors based on the binary observation vectors of size $N$ (as shown in Figure 2-10). For each observation vector, the network calculates a weighted summation of the input units to find the *correlation* of the input with each of the latent factors. Then, the weighted summation over different input units, $a_i = \sum_j w_{i,j} x_j$ (which is referred to as *"activation energy"*), is passed to an activation function at each hidden node $i$ to find the probability that node $i$ is triggered to an active state [56].

For instance, assuming a sigmoid $g(a) = 1/(1 + e^{-a})$ activation function, $\sigma_i \coloneqq g(a_i)$ is interpreted as the *probability* for node $i$ to be in an active state. Hence, the state of node $i$ is set to 1 with a probability of $\sigma_i$ and to 0 with a probability of $1 - \sigma_i$. Note that $\sigma_i$ is close to 1 (close to 0, respectively) for a positive (negative) activation energy with a sufficiently large absolute value.

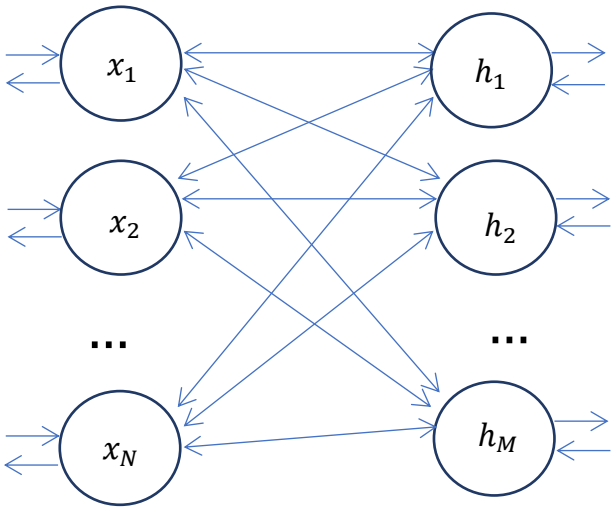


**Figure 2-10: The architecture of a restricted Boltzmann machine with stochastic neurons. In the forward direction the network extracts the latent factors of the input, whereas in the reverse direction it can regenerate the original data based on the latent factors.**

Given the state of the hidden nodes, the network finds the activation energy for the input units in the reverse direction, and then stochastically updates the input units' state according to the probabilities suggested by the activation function. Indeed, a trained RBM network can also be used in the reverse direction to regenerate the original data when the latent factors are known (see Figure 2-10). For example, for a user with known preferences over different types of movies, the network can suggest a list of movies that the user most probably likes.

In another application, the RBM can be used to filter distorted or noisy inputs or reconstruct a pattern when only part of it is known at the input. In this case, the network may operate in a recurrent fashion, where each iteration helps to better filter out the noise of the input and then regenerate the output based on more accurately estimated latent factors [56].

- *Training the RBM*

Suppose that a set of $P$ training examples (i.e., binary input vectors of size $N$) is given to train an RBM with $M$ hidden nodes and $N$ visible units. Each training example specifies the input to the visible units, though the desired values of hidden nodes (and hence the latent factors) are not known in advance. Assuming an initial random weight matrix, one can find the state of the hidden nodes for each given input pattern. According to the units' update rule, the units which are connected with a positive weight (negative weight, respectively) try to make each other to be in the same (different) states.

Starting with one training example $p$ at the input, let $e_{i,j}^{+}(p) = h_i x_j$ denote the correlation of hidden node $i$ and visible node $j$ in the forward direction, and $e_{i,j}^{-}(p) = h_i \hat{x}_j$ denote the correlation in the reverse direction, where $\hat{x}_j$ is the value of visible unit $j$ of pattern $p$ that is estimated by the network. To make the network estimate the true state for each input unit, one can iteratively update the weight matrix $W = [w_{i,j}]$ as [62]:

$$w_{i,j} \leftarrow w_{i,j} + \alpha\left[e_{i,j}^{+}(p) - e_{i,j}^{-}(p)\right], \tag{2-24}$$

where $\alpha$ is a small step size (so-called the learning rate). Then one should move on to the next pattern, and repeat this procedure for all patterns until converging to a stable weight matrix where the error between the input and regenerated patterns approaches zeros (or becomes less than a pre-determined threshold). In [56] it is shown that the moving direction in (2-24) closely approximates the gradient of an objective function that is defined in terms of an energy function. This iterative technique, which is so-called *contrastive divergence*, is shown to properly converge in many practical applications [56].

## 2-4 Summary

Table 1 summarizes the main features and capabilities of various neural network architectures that we studied in this chapter. Particularly, various architectures are compared in terms of memory capacity, complexity of training, and their performance to regenerate heavily distorted input patterns, when the network is trained as an associative memory.

As we already discussed, Hopfield network can be programmed to store only a few patterns using Hebbian rule. Hence, it shows a poor memory capacity and a poor performance to regenerate distorted and noisy patterns. An MLP network can be used to recognize the desired patterns from distorted inputs and then regenerate them. But it shows an extremely inferior performance compared to RBM and RHN which have a dynamic recursive architecture [5]. Besides their superior performance, the recursive neural networks (such as RBM and RHN) can be potentially trained with a lower complexity owing to their simple architecture where the same weight matrix is repeatedly used across different layers. In Chapter 4, we compare the performance of RHN and RBM for regenerating distorted and noisy patterns. It is observed that RHN

results in a superior performance compared to RBM, especially for regenerating input patterns which are distorted with an analog noise. The reader may refer to Chapter 4 for further discussions on this matter.

**Table 1: A summary of features and capabilities of various neural network architectures**

| Network Architecture | Stochastic vs Deterministic | Recursive vs Feed-Forward | Memory Capacity | Complexity of Training | Performance to Regenerate Heavily Distorted Inputs | |
|---|---|---|---|---|---|---|
| | | | | | Analog Noise | Bit-wise Distortion |
| Hopfield | Deterministic | Recursive | Limited (Not scalable) | Low | Poor | Poor |
| RHN | Deterministic | Recursive | Scalable | Moderate | Good | Good |
| RBM | Stochastic | Recursive | Scalable | Moderate | Fair | Good |
| MLP | Deterministic | FF | Scalable | Heavy | Fair | Fair |

# Chapter 3

# Restricted Hopfield Network

## 3-1- Motivation

The conventional Hopfield network can be programmed in one shot to memorize patterns using the Hebbian Rule. Using a correlation matrix to capture the information of different training examples, however, results in a limited memory capacity especially when the patterns are correlated. Using even the best training methods, Hopfield networks are able to store at most 0.14N patterns (where N is the number of nodes in the network) while yet tolerating a small error at the output [48].

There have been many efforts for improving the information capacity and solving the trainability issue of the Hopfield network [3], [7], [43]. The most recent effort for solving the problem of memory capacity and the trainability issue of the Hopfield Network has been made in [5] by proposing a new architecture that is the so-called Restricted Hopfield Network (RHN). However, training of the network in [5] is based on the Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm, which is hard to converge. To exploit the potential advantages of this new architecture, in this chapter

we propose a new training method for RHNs which is shown to improve the performance of the network.

Particularly, we propose to use back-propagation overtime to find a decent direction for iteratively updating the weight matrix. In this way, we can ensure that the training error is reduced in each iteration. Moreover, we propose a modified variant of the SPSA algorithm which is jointly used with backpropagation to ensure that the algorithm does not get stuck in local optimum points. Our proposed method here not only improves the convergence of the training algorithm but also is shown to improve the robustness of the network against the distortions of the inputs. Finally, we argue that some of the presented ideas (such as the modified variant of the SPSA) can also be useful in other contexts such as hyper-parameter optimization.

- **Organization**: To present our proposed training method, first we present the architecture of RHN and the way it can be implemented. We also review the SPSA training algorithm and discuss why it may not be much efficient. Then we present the details of our proposed techniques which are to improve the training (both in terms of convergence and performance) for this class of networks. We conclude this chapter by presenting some possible extensions.

## 3-2 A discrete-time model for RHN

To deploy the RHN as described in Section 2-3-5, one needs to implement a dynamic system, described with a set of differential equations, which is computationally demanding. To address this issue, it is possible to implement a discrete-time version of the RHN which approximates the model of Section 2-3-5 when the input to the network is varying at a low pace. Particularly, as in [5], we deploy RHN as a recurrent neural network that follows a simple updating rule assuming that the input to the network is fixed in each iteration.

Specifically, an input pattern (which can be possibly distorted) is fed to the network as the initial value for visible nodes. Then the output of visible nodes is passed to the hidden nodes, using the weight matrix $\boldsymbol{W} = [w_{i,j}]_{M\times L}$, which gives the input, $\boldsymbol{U^H} = \boldsymbol{WV^V} + \boldsymbol{\theta^H}$, to the hidden nodes, where $\boldsymbol{\theta^V} = [\theta_i^{\boldsymbol{V}}]$ is the bias vector for visible nodes. Passing $\boldsymbol{U}^H$ through the activation (e.g., sigmoid) function, gives the output for hidden nodes. In the same way, the output of hidden nodes is passed to visible nodes using the same weight on each edge in the reverse direction. Hence, the input to visible nodes, and the resulting output is given by $\boldsymbol{U}^V = \boldsymbol{W}^T\boldsymbol{V}^H + \boldsymbol{\theta}^V$, and $\boldsymbol{V}^V = g(\boldsymbol{U}^V)$, respectively, where $\boldsymbol{W^T} = [w_{j,i}]_{L\times M}$ is transposition of $\boldsymbol{W}$.

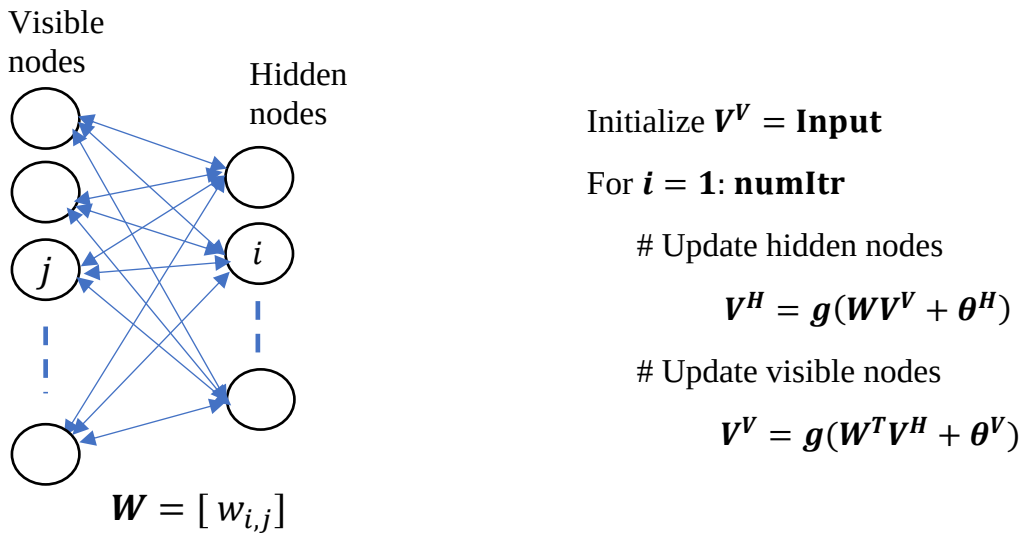


**Figure 3-1: The discrete-time model for the RHN**

This procedure is repeated a certain number of times, as indicated by **numItr**, in the pseudo-code of Figure 3-1. Repeating this procedure over time is like going through different layers of a multi-layer neural network, as shown in Figure 3-2 for the case that $\mathbf{numItr} = 2$.

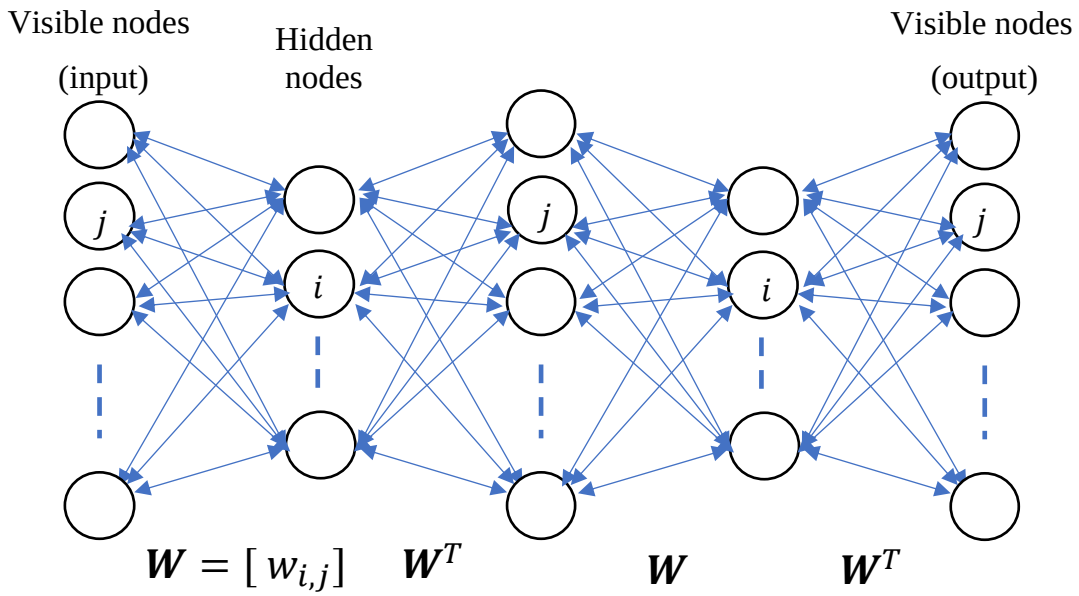


**Figure 3-2: The recurrent implementation of neural network over time. In this example the network makes two iterations.**

## 3-3 Training the RHN: Existing methods

Since the desired outputs for hidden nodes are not known, the conventional Hebbian rule is not applicable to train RHN. However, we may know the desired output of the network (i.e., the output of visible nodes) in most of the applications. Using this information, we can evaluate how well the network is trained to memorize certain patterns. Particularly, we can define a loss function in terms of the network response to different patterns (e.g., the mean square error of the output). Then, one needs to find the weight matrix $\boldsymbol{W}$ in a way that the loss function is minimized.

In conventional neural networks, the back-propagation method is commonly used to find the gradient of the loss function with respect to each element of the weight matrix. The gradient is an appropriate direction to update the weight matrix since it guarantees the steepest descent in the loss function when iteratively updating the weights [46]. However, as argued in [5], it may not be straightforward to apply back-propagation

overtime for a recurrent neural network to calculate the gradient with respect to all elements of the weight matrix.

To come up with a simple solution, it is proposed [5][41] to use the simultaneous perturbation stochastic approximation (SPSA) algorithm which uses an approximation of the gradient while it requires only 2 loss function measurements in each iteration regardless of the dimension of the optimization problem [9, 10]. Hence, it might be suitable for a high-dimensional optimization problem wherein an objective function is to be minimized for which the gradient may not be calculated analytically (e.g., due to complexity of calculations).

Particularly, let $J(W)$ denote a loss function defined in terms of the weight matrix. According to the SPSA scheme, a simultaneous perturbation matrix, $\Delta(k)$, is generated in each iteration $k$ to update the elements of the training matrix. Each element $\Delta_{i,j}(k)$ of the perturbation matrix, is generated with a probability of 0.5 being either +1 or -1. The rate of change in the loss function then is used to decide whether move along or in the opposite direction of the perturbation matrix:

$$\Delta w_{i,j}(k) = \frac{J\big(W(k) + c(k)\Delta(k)\big) - J\big(W(k) - c(k)\Delta(k)\big)}{2c(k)\Delta_{i,j}(k)}, \tag{3-1}$$

where $c(k)$ is a scalar serving as the step size to approximate the rate of change in the loss function. Finally, $\Delta w_{i,j}$ is used to update each element of the weight matrix:

$$w_{i,j}(k+1) = w_{i,j}(k) - a(k)\Delta w_{i,j}(k), \tag{3-2}$$

where $a(k)$ is the step size to update $\boldsymbol{W}$.

## 3-4 Training the RHN: Our proposed techniques

As already discussed, the SPSA algorithm only finds a crude estimate of the gradient of the loss function. So, it may not necessarily reduce the loss function by updating the weight matrix according to a randomly selected perturbation matrix. In other word, the updating direction is not necessarily a decent direction, and therefore there is no guarantee for the SPSA algorithm to converge.

To enhance the performance of the training algorithm for the RHNs, we propose to use back-propagation to find the actual gradient of the loss function for iteratively updating the weight matrix. Since RHN is a recursive neural network, we implement back-propagation over time using the discrete-time model of Section 3-2 so that we can calculate the gradient of the loss function analytically. In this way, we can ensure that the training loss is reduced in each iteration.

The iterative gradient-descent algorithm, however, may end up with a local optimum solution. To address this issue, we propose a modified and enhanced version of the SPSA algorithm, which is jointly used with back-propagation to ensure that the algorithm does not get stuck in local optimum solutions. In the following, first, we investigate the implementation of back-propagation overtime for the RHN. Then, we present our proposed enhancement for the SPSA scheme. Finally, we present an overview of the whole procedure that is proposed for training the RHN.

### 3-4-1- Back-propagation over time for the RHN

To implement back-propagation over time, consider one iteration of the RHN as shown in Figure 3-3, complying with the discrete-time model of Section 3-2. Let assume that a set of desired patterns, $\{\mathbf{y}^p | \, p = 1,2, \dots, P\}$, are presented at the input of the network. The error at the output for each particular (desired) pattern, $\mathbf{y}$, is given by $\delta = \boldsymbol{V}^{out} - \mathbf{y}$. The goal is to train the network in a way that the error turns to zeros at least when there is no distortion at the input.

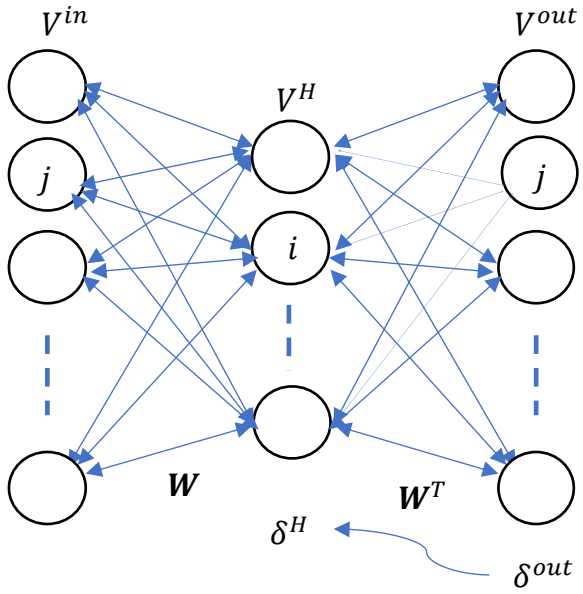


**Figure 3-3: Implementing back-propagation over time for the RHN**

To measure how well the network is trained, we opt for the logarithmic loss function, $J(\boldsymbol{W})$, which is commonly used in the context of Neural networks:

$$J(W) = -\frac{1}{P}[\sum_{p=1}^{P}\sum_{i=1}^{L} y_i^{(p)} \, log\left(V_i^{(p)}\right) \quad + \left(1 - y_i^{(p)}\right) log\left(1 - V_i^{(p)}\right)], \tag{3-3}$$

where $V_i^{(p)} \in (0,1)$ is the output for element $i$ of pattern $p$, and $y_i^{(p)} \in \{\pm 1\}$ is the desired output for the same element. The loss function, $J(\boldsymbol{W})$, measures how every element of each pattern, $V_i^{(p)}$, differs from the desired output. The objective is to find the weight matrix $\boldsymbol{W}$ such that the loss function, which has a lower bound of zero, is minimized.

To find the derivative of the loss function with respect to every element of the weight matrix, we can follow a similar procedure as in the conventional back-propagation scheme. Particularly, by back-propagating the error at the output layer, $\boldsymbol{\delta}^{out}(p)$, for each

pattern $p$, we can find the error at hidden nodes while assuming a sigmoid activation function:

$$\delta^H(p) = (W^T)^T \delta^{out}(p) .* \left(V^H .* (1 - V^H)\right)^{(p)}. \quad (3\text{-}4)$$

In the general, the derivative of $J(\boldsymbol{W})$ with respect to $w_{i,j}$, corresponding to an edge connecting node $j$ to node $i$ in a subsequent layer, is given by $\delta_i$ (i.e., error at the destination node) times the output of activation function at the source node $j$ for a specific input pattern $p$. Since $w_{i,j}$ (for an edge connecting visible node $j$ to hidden node $i$), is chosen to be the same as $w_{j,i}$ (corresponding to the edge connecting the hidden node $i$ to the visible node $j$), the derivative of the loss function for the RHN with symmetric weights is given by

$$\frac{\partial J(W)}{\partial w_{i,j}} = \sum_{p=1}^{P} V_j^{in(p)} \delta_i^H(p) + V_i^{H(p)} \delta_j^{out}(p), \qquad \forall i, j. \quad (3\text{-}5)$$

Given the derivative of the loss function with respect to every element of the weight matrix, we can use the gradient, $\boldsymbol{\nabla} J = [\partial J(W)/\partial W_{i,j}]$ as the moving direction to update $\boldsymbol{W}$. Specifically, we can update each element $w_{i,j}$ by moving in the opposite direction of the gradient, so as to reduce the loss function. That is

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial J(W)}{\partial W_{i,j}}, \qquad \forall i, j. \quad (3\text{-}6)$$

Here, $\alpha$ is a step size that is used for updating the weight matrix. We employ backtracking to set $\alpha$ so that $J(W)$ is decreased in each iteration. Particularly, starting with some initial value for $\alpha$, $\alpha = \alpha_0$, in each iteration $k$, we repeatedly divided $\alpha$ by 2 until $J(\boldsymbol{W}^{k+1}) - J(\boldsymbol{W}^k) < -th$, where $th > 0$ can be a fixed threshold. Alternatively,

$th$ can be chosen proportional to the gradient norm in each iteration, $th \propto \|\nabla J(W)\|$ [46].

The gradient descent algorithm, as presented here, reduces the loss function while the gradient has a positive norm. It means that the algorithm eventually converges to a local optimum point where $\|\nabla J(W)\| \to 0$. A local optimum value for the loss function $J(W)$, however, might be different than the potential global optimum solution. Particularly, the lower bound of zero for the loss function could be achieved only at a global optimum solution (provided that RHN consists of enough hidden nodes for a given number of input patterns). However, a local optimum solution may take on some value *greater than zero*. In the next section, we discuss our proposed method to get out of a local optimum solution.

### 3-4-2- Modified SPSA algorithm

As already discussed, the gradient-decent algorithm, which is implemented based on back-propagation over time, may end up with a local optimum solution. To avoid getting stuck in the vicinity of a local optimum solution, one needs to take a random moving direction when the gradient-descent direction may not further reduce the loss function. To do so, we propose a modified variant of the SPSA scheme, which can effectively find an appropriate moving direction. Particularly, the conventional SPSA scheme (as presented in Section 3-3) is based on generating a simultaneous perturbation matrix, $\mathbf{\Delta}$, for which all elements are selected randomly to be either +1 or -1 with a probability of 0.5. A randomly selected direction may help to get out of a local optimum solution, but it may cancel all the efforts that are made by the gradient-descent sub-routine. So, the main challenge is which random direction is appropriate to choose? To address this issue, we propose a criterion for choosing a randomly selected direction. Moreover, we propose an idea which confines the set of possible random directions so that it becomes easier to find a direction that satisfies the desired condition.

In the proposed method, first, we find the output element(s) which has the maximum contribution to the loss function:

$$i^{max} = \underset{i}{argmax}\, J_i(\boldsymbol{W}), \tag{3-7}$$

$$J_i(W) = -\frac{1}{P}[\sum_{p=1}^{P} y_i^{(p)}\, log\left(V_i^{(p)}\right) \quad + \left(1 - y_i^{(p)}\right) log\left(1 - V_i^{(p)}\right)], \tag{3-8}$$

We choose the perturbation matrix such that only a subset of edges, which are connected to node index $i^{max}$ (see Figure 3-4), take on a non-zero value. That is, we set

$$\Delta_{i,j} = 0\ , \quad i \notin i^{max}, \forall\, j \tag{3-9}$$

$$\Delta_{i,j} \in \{\pm 1\},\ \ i \in i^{max}, \forall j. \tag{3-10}$$

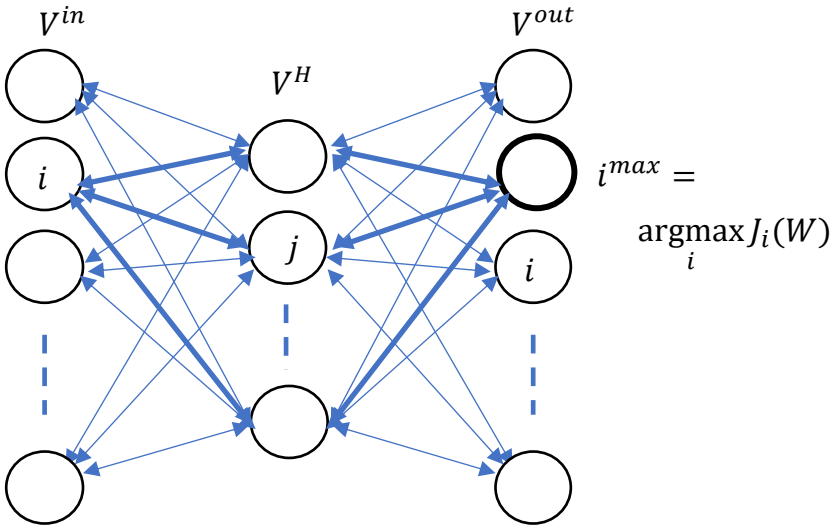


**Figure 3-4: A subset of weights, corresponding to the edges connected to the particular node $i^{max}$, are updated according to the modified SPSA scheme.**

If two or more output elements result in the same loss component, that is $J_{i_1}(W) = J_{i_2}(W) = \max_i J_i(W)$, we let $i^{max}$ denote the set of all of them. Then, we provisionally update the weights for every $w_{i,j}, i \in i^{max}, \forall\, j$, according to (3-1) and (3-2), in the same way as discussed in Section 3-3. The change, however, becomes effective only if the selected random direction either reduces $J(W)$ or reduces $J_{i_{max}}(W)$. Otherwise, we try again choosing another random direction. It's worth noting that by confining the search space only to the edges which affect the output element(s) of interest (i.e., the ones with maximum-loss), it becomes easier to find a random direction which satisfy the desired condition. In the next section, we provide an overview of the whole procedure and discuss how to integrate such a modified variant of SPSA into the gradient-descent scheme.

### 3-4-3- The overall procedure

Starting with an initial randomly selected weight matrix, we implement an iterative algorithm to find an optimal weight matrix which minimizes the loss function. We employ back-propagation over time by default (to calculate the gradient for the RHN) to update the weight matrix as long as the gradient-decent direction sufficiently reduces the loss function. However, the reduction in the loss function turns to zero as we get closer to a local optimum solution where $\|\nabla J(W)\| \to 0$. Hence, we use a threshold (in terms of the gradient norm) to identify whether the current point is in the vicinity of a local optimum point (see the pseudo-code in **Table 2**). In this case, we add a perturbation to certain elements of the weight matrix to help to get out of a local optimum solution, as explained in the previous sub-section. Table 2 summarizes different steps of the algorithm in a pseudo-code format. The reader may refer to the Appendix for the detailed implementation of the training algorithm in Python.

## Table 2: Pseudo-code of the proposed training method for RHN

```
ϵ = 0.001; # The desired maximum error
Data = LoadInput(); # The input is mapped to the range of (ϵ, 1 − ϵ)
W = rand(N, M) − 0.5 # Initialize the weight matrix randomly

while J(w) < ϵ

    ∇J = FindGradient (W, Data);
    α = 0.1;
    # Back tracking line search
    while J(W − α∇J) − J(W) > −γ‖∇J‖ & numBackTrack < 8
        α = α/2;
        numBackTrack++;

    if ‖∇J‖ > th & J(W − α∇J) < J(W)
        W = W − α∇J;

    else # Try Random Directions

        numSearch = 0;

        while true
            numSearch++;
            imx = argmax([J_i]);
            ΔW = zeros(N,M);
            ΔW[imx,:]= random.choice([-1,1],(1,M));
            C = 0.01;
            slope = (J(W+cΔW)-J(W-cΔW))/2c;
            α = 0.1 * slope;

            if J(W − αΔW) − J(W) < ϵ or
               J_imx(W − αΔW) − J_imx(W) < ϵ or
               numSearch ≥ 50
                W = W − αΔW;
                Break
```

## 3-5 Possible extensions

Here we investigated different techniques to enhance the training algorithm performance for a RHN. Some of the presented ideas, however, may have more generic applications. E.g., the idea that we presented to modify the SPSA algorithm can also be useful in the context of hyperparameter optimization. The main idea is to confine the search space and update only a subset of variables which have a direct impact on a parameter/objective which is of interest. E.g., in the case of SPSA, we only update the weights which have a direct impact on the output elements of interest (i.e., the ones with maximum error). In this way, not only the computational complexity is reduced, but also the random search becomes more efficient since we update only the weights which affect the output element(s) of interest.

The same idea is applicable for hyperparameter optimization/tuning. There are usually several model-related parameters which should be properly tuned prior to training the network. Optimizing such parameters usually needs performing several rounds of pilot training which could be computationally demanding. Following the same idea, one may reduce the computational complexity of pilot training by confining the training variables, considering only those which directly impact (or get impacted by) the parameters of interest. Alternatively, one may identify certain input patterns which has a greater contribution to the training error. In this way, pilot training can be performed using only a subset of input patterns, so that reducing the associated complexity.

# Chapter 4

# Experimental results

## 4-1 Overview of the chapter

In this chapter, we perform several experiments to evaluate the efficiency of the proposed training method for RHNs, compare its performance (in terms of the convergence and the capability to recreate desired patterns) against the conventional SPSA scheme, and investigate the impact of different parameters on training and performance of the network. We also compare the performance of the RHN against the Hopfield network (which is trained by the Hebbian rule), as well as the RBM that we reviewed in Chapter 2. Before presenting the results, we first describe the simulation setup and various datasets (including Tensorflow.Keras dataset) that we use for different experiments.

## 4-2 Simulation setup

To evaluate the performance of different training methods, we deploy the RHN, following the discrete-time model of Chapter 3, wherein the network comprises a certain

number of hidden nodes (denoted by “numH”). Operating recurrently, the RHN goes through a certain number of iterations, which resemble multiple layers, denoted by “num. layers” in the following.

We perform several experiments using different datasets as input. First, to evaluate the capability of the RHN compared to the Hopfield network (c.f. Section 4-3), we consider simple input patterns comprising 4 alphabet characters. Each character is represented by a matrix of $7 \times 5$ binary pixels, resulting in 35 elements for each input pattern/character. The Hebbian rule is used for training the Hopfield network to learn the desired input patterns. Hopfield network, however, is observed to show a poor performance in re-creating the inputs when being distorted, even for such a simple dataset, compared to the RHN and RBM.

We further evaluate and compare the performance of the RHN with a more extensive dataset comprising different numeric characters (i.e., 0, 1, …9), each of them represented by a matrix of $7 \times 5$ binary pixels. We use this dataset to perform various experiments, analyzing the impact of different parameters (such as the number of hidden nodes and or the number of layers) on the training and performance of the network, while comparing our training method against the conventional SPSA scheme. We also compare the performance of RHN against the RBM.

We finally use an extensive dataset comprising a plurality of handwritten numeric characters (loaded from the “Tensorflow.Keras” dataset) each represented with $28 \times 28$ pixels. In this case, again we evaluate the capability of the network in memorizing the patterns and recreating them from distorted inputs.

All simulations are performed on a standalone computer with Intel®Core™ i5-3320M CPU, and 8.00 GB RAM. The language which is used for coding is Python 3.7.1, using “NumPy” library in “Pychram IDE”. The Windows 10, 64 bit Professional is the underlying operating system.

## 4-3 RHN against RBM and Hopfield network

Since Hopfield has a limited capability in memorizing patterns (as we discussed in Chapter 2), we consider a simple dataset comprising four characters, A, U, S, T, as shown in Figure 4-1, and use the Hebbian rule to train the Hopfield network and evaluate its performance. We also consider an RHN with 35 visible nodes and 10 hidden nodes. The RHN network is also trained to memorize the four alphabet characters, using each of the SPSA and our proposed training method, referred to as modified Back-Propagation (or modified BP[1]).

To demonstrate the ability of the network in recreating the distorted patterns, we distort the inputs by changing a certain number of pixels of each character. The erroneous number of pixels is defined as the Hamming distance of the input pattern from the desired pattern. Figure 4-2 shows distorted images of A, U, S, T, which have a Hamming distance of 5 compared to the desired patterns shown in Figure 4-1.

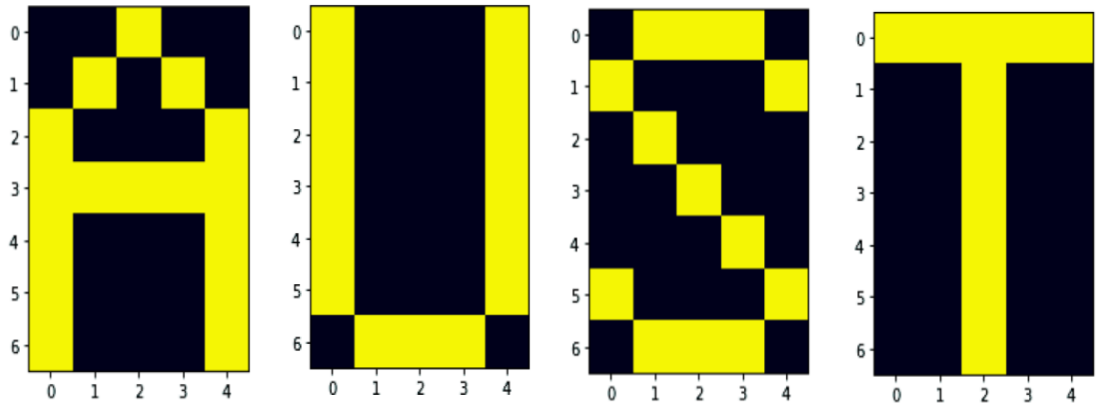

**Figure 4-1: Desired patterns which are used for training the RHN and Hopfield network**

[1] Our proposed training method, as described in Chapter 3, combines the BP method with a modified variant of the SPSA. Hence, we refer to it as *"modified BP"* in this chapter.

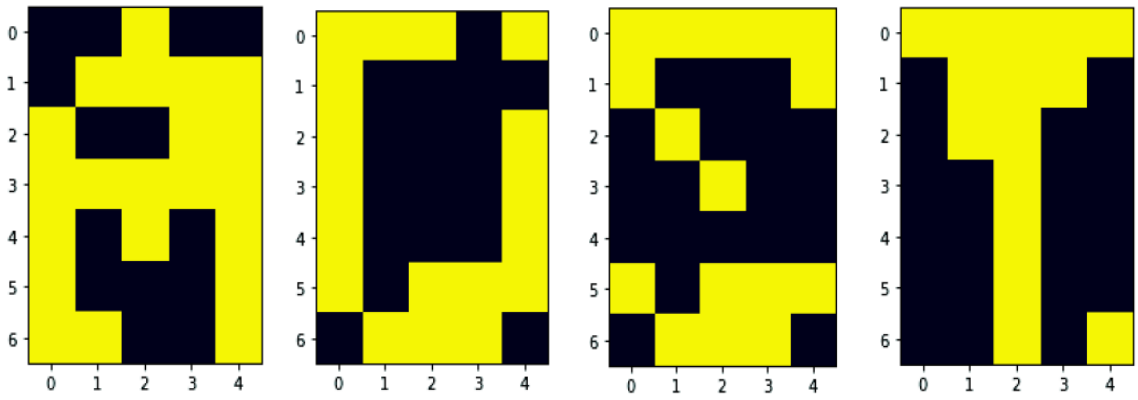

**Figure 4-2: Distorted patterns which are fed to the RHN and Hopfield network to evaluate their capability in recreation of randomly distorted input images.**

To evaluate the performance of the RHN in recreating distorted inputs and compare it against the RBM and Hopfield network, we feed each of them with 10,000 randomly distorted samples of each pattern. This process is repeated for error patterns with a Hamming distance ranging from 1 to 10. The "output pattern error rate" (i.e., the percentage of the patterns which are recreated erroneously for distorted patterns with a specific Hamming distance) is calculated by finding the average failure rate in the recreation of the patterns over different iterations. Particularly, to compute the pattern error rate, we compared the output of the network with the desired output and if they are not the same, the number of errors (numError) is incremented by one. By dividing numError by the number of iterations (10,000) the average error rate is estimated.

We employ two realizations of the RHN both comprising 10 hidden nodes, where one of them is trained with the existing SPSA scheme while the other one is trained with the modified BP technique which we presented in Chapter 3. We also deploy an RBM with 10 hidden nodes following the model and the training method presented in Chapter 2. We report the results for the performance of the Hopfield network and RBM and compare them against the two realizations of RHN (trained with the two training methods) in Figure 4-3.

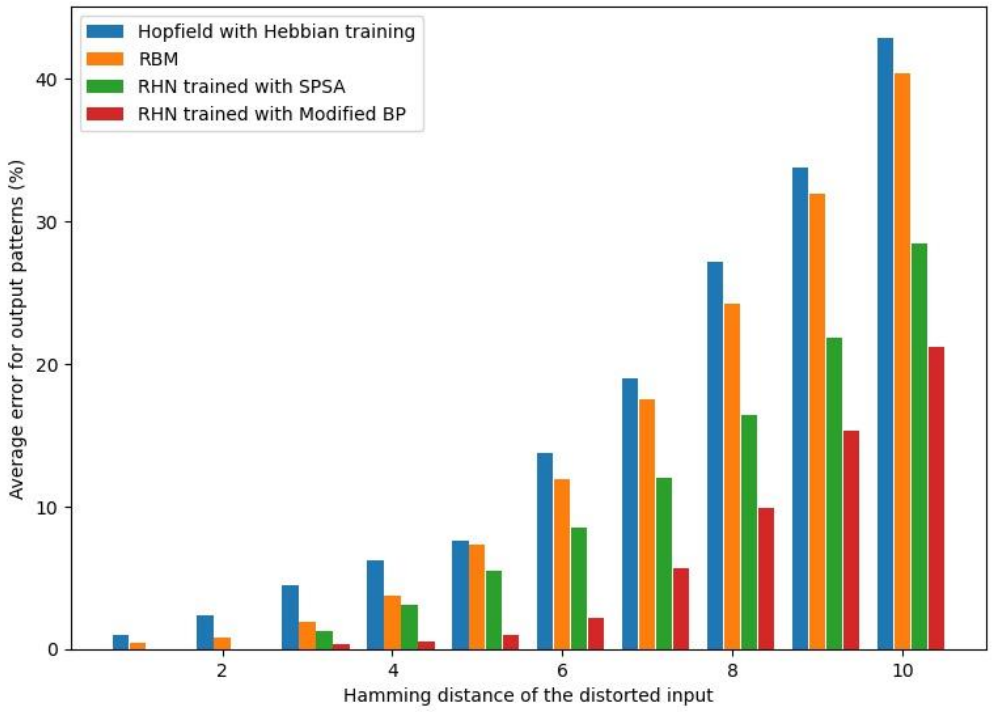


**Figure 4-3. Performance comparison of RHN trained with SPSA and modified BP against Hopfield network trained with Hebbian rule and Restricted Boltzmann Machine. All networks are functioning as an associative memory to store patterns of A, U, S, T characters.**

As shown in Figure 4-3, RHN trained with our modified BP method can make the perfect re-creation of the training images with an average error rate of 1.03% when the Hamming distance is 5 which is a significant improvement compared to the case where RHN is trained with the SPSA scheme (which results in an error pattern of more than 5.5% for the same input). The classical Hopfield network can only achieve an error rate of 8.4% for input with the same Hamming distance. The RBM shows a better performance compared to the Hopfield network, but its performance is way worse than the RHN, especially when the RHN is trained with the modified BP.

An intuitive reason for performance improvement of our proposed training method (compared to the SPSA scheme) is that it strives to uniformly minimize the error across

different output components. Hence, it turns out to be more effective in correcting the errors. Moreover, the loss function that we use to minimize is the logarithmic loss function (which has exponentially increasing components for erroneous output elements), whereas SPSA in [5] is used to minimize the mean-square error of the output. This can be another reason for the improvement of the training.

It is worth noting that we employ a threshold unit at the very last layer of the RBM, so as to avoid errors caused by stochastic mapping[2]. However, the hidden nodes take on binary values in the same way as described in Chapter 2. Even with this improvisation, the RBM is still far behind the RHN as shown in Figure 4-3. An intuitive reason is that the hidden nodes in RHN can take on any real-valued numbers in the range of (0,1), whereas the state of hidden nodes in RBM is mapped to the binary values {0,1}. Hence, the hidden nodes in an RHN can convey more information compared to the RBM.

As shown in [5] for the sake of counterexamples and as we discussed in Chapter 2, the Hopfield network has a limited capability in memorizing the patterns. That is why it may not effectively recreate distorted inputs even for such a simple dataset with four training examples. Hence, we rule it out from further investigation and instead will study the two alternative training methods for RHN in more detail and compare them against the RBM which is a well-understood recurrent neural network that can be used for extensive datasets.

## 4-4 SPSA versus our proposed training method: Convergence

In this section, we evaluate our proposed training method in terms of the convergence performance and compare it against the existing SPSA scheme. For the experiments in this section, we use a dataset comprising 10 numeric characters, as shown in Figure 4-4.

[2] We make this improvisation for all of the experiments in this chapter.

First, we investigate the convergence of the SPSA algorithm. As already discussed, there is no guarantee for the SPSA algorithm to converge. In practice, however, the step size variables $a_k$ and $c_k$ (as described in Chapter 3) are chosen to be gradually decreasing in subsequent iterations, $a_k$ and $c_k \propto 1/(k+1)$, to prevent the algorithm from oscillation when updating the weight matrix. Yet there is no guarantee that it converges to an optimal weight matrix.

In Figure 4-5, we plot the variations of the loss function over 10000 iterations of the SPSA training algorithm for an RHN with 35 visible nodes (corresponding to 35 pixels of input images), 2 layers (recursions), and 30 hidden nodes. As shown in Figure 4-5, with gradually decreasing step sizes, the algorithm approaches a stable weight matrix with the loss function value of around 1. However, it is not known whether it can further reduce the loss function to zero, or how many iterations are needed for the algorithm to converge to an optimal weight matrix.

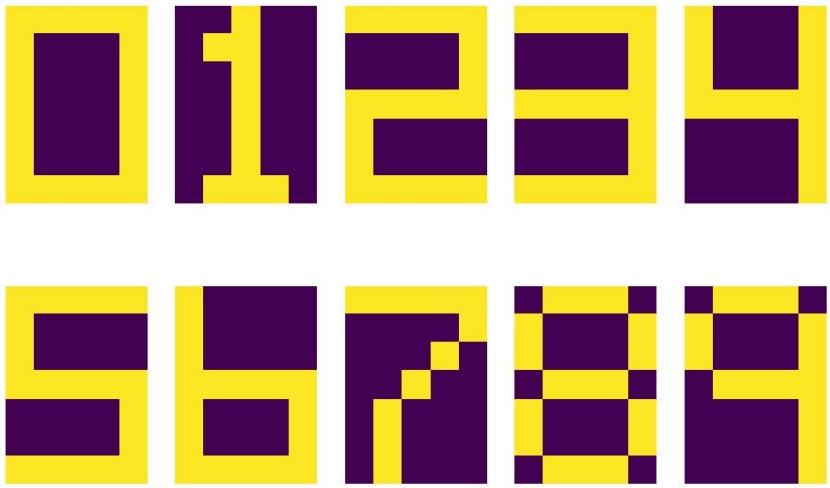

**Figure 4-4: Numeric characters which are used as desired output to train the RHN**

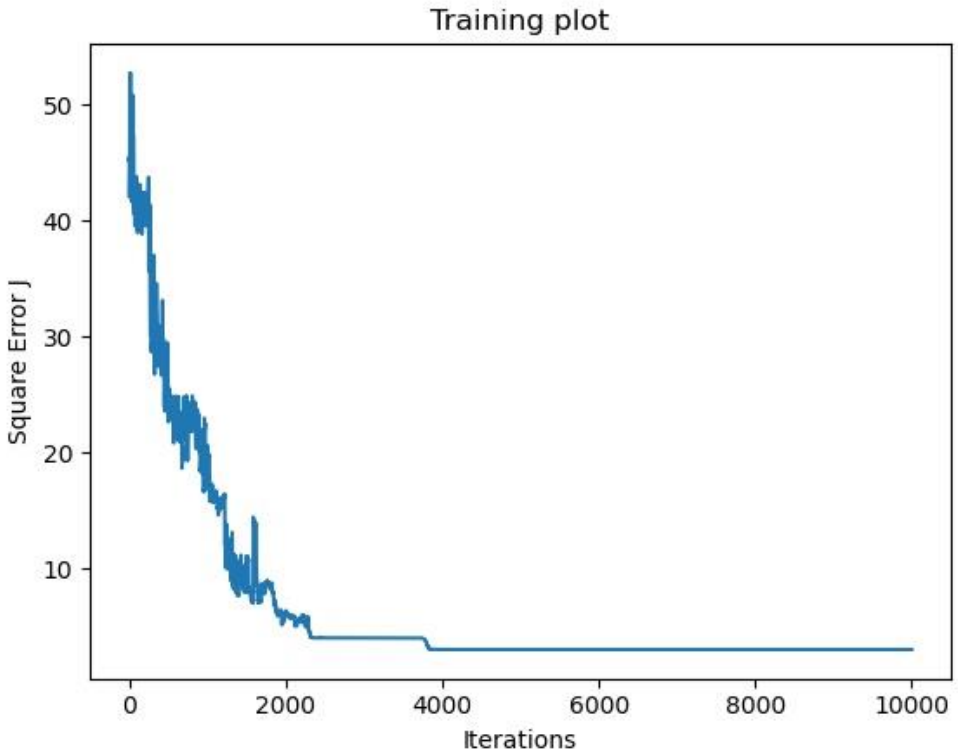


**Figure 4-5: Variations of the mean square error of the output over different iterations using the SPSA scheme to update the weight matrix**

The situation gets worse when considering a larger network with so many hidden nodes or a larger dataset. Figure 4-6 shows the variation of the loss function for an RHN with 70 hidden nodes when using the SPSA algorithm to iteratively update the weight matrix. It is observed that for a larger network, the SPSA algorithm does not tend to converge or even reduce/limit the loss function value, though it has been running for several hours.

As discussed in Chapter 3, the dynamic nature of the SPSA scheme is useful to get out of local optimum solutions. But with randomly selected moving directions it is difficult to approach an optimal solution, especially in a larger size network with so many weight variables. In our proposed training method, we use BP overtime to find the gradient-descent direction, so that the algorithm can reduce the loss function in each iteration, while we use a modified variant of the SPSA to get out of local optimum solutions.

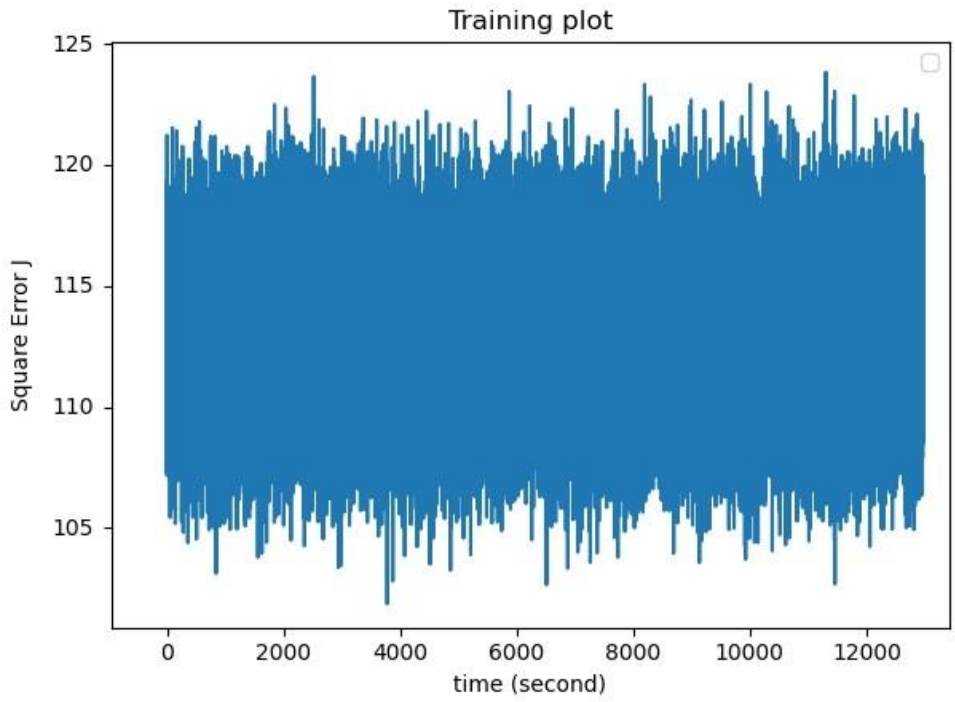


**Figure 4-6: Variation of the mean square error of the output over time for an RHN with 70 nodes when using the SPSA scheme to update the weight matrix. The algorithm does not tend to converge in this experiment.**

Figure 4-7 shows the variation of the loss function value over time when our proposed training method (referred to as "modified BP") is used to train an RHN with 35 visible nodes, 2 layers (i.e., recursions), while comparing the convergence for the various number of hidden nodes. It is observed that in this experiment the modified BP scheme constantly reduces the loss function value in every iteration. An interesting observation here is that the convergence of the training sub-routine is facilitated for an RHN with more number of hidden nodes when using the modified BP scheme.

The rationale for this observation is that it becomes easier for a network with a larger number of hidden nodes to extract different features of the input and then reconstructs the input based on the output of the hidden nodes. In this case, the BP scheme can more efficiently reduce the loss function.

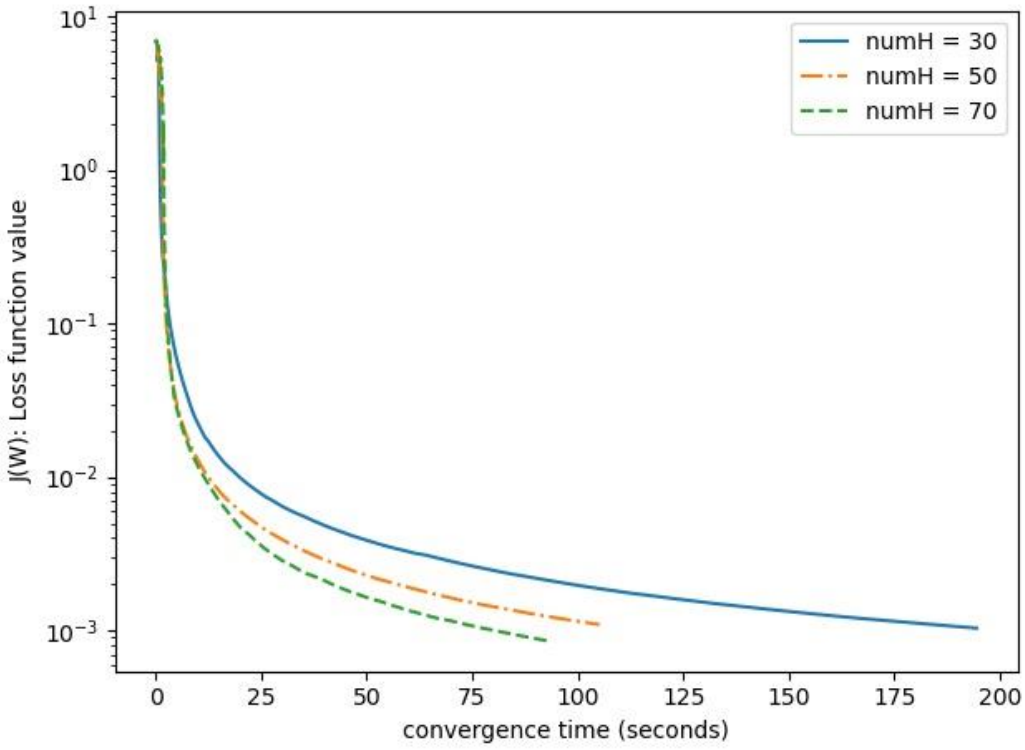


**Figure 4-7: Evolution of the loss function value over time for the modified BP scheme when used to train an RHN with 2 layers (recursions), 35 visible nodes, and various number of hidden nodes**

In another experiment, we study the impact of the number of layers (i.e., number of recursions) on the training complexity. Performing more recursions (or equivalently, including more number of layers) introduces more non-linearity to the network, which is expected to increase the complexity of training. In this case, the loss function also shows more non-linearity effects, such as local optimum solutions. The impact of the number of layers on training complexity is captured in Figure 4-8, where we compare the evolution of loss function value over time for an RHN with the same number of 30 hidden nodes, and the different number of layers.

It is observed that the loss function shows more non-linearity effects when increasing the number of layers from 2 to 4, so that the algorithm faces a number of local optimum

points at the beginning of the simulation (in time interval (0,50) seconds). As discussed in Chapter 3, in the vicinity of a local optimum solution, a modified variant of the SPSA algorithm is used which selectively reduces the components which have the greatest contribution to the loss function. In this way, the overall loss function value might be increased at some points (as depicted in the Figure), but eventually, it helps to get out of local optimum solutions and converge to a global optimum weight matrix.

To further facilitate the training for an RHN, one can train the network with a fewer number of recursions (e.g. numLayer = 2 or 3), and then implement the network using more recursions (e.g., multiples of numLayer used for training) to better recreate the input. In the next section, we further investigate the impact of the number of layers and the number of hidden nodes on the performance of an RHN in re-creating noisy/distorted images.

## 4-5 Network performance

In this section, we evaluate the performance of the RHN in recreating distorted/noisy inputs, while comparing it (using the two alternative training methods) against the RBM. We also study the impact of different parameters (such as the number of hidden nodes and or recursions) on the performance of the network.

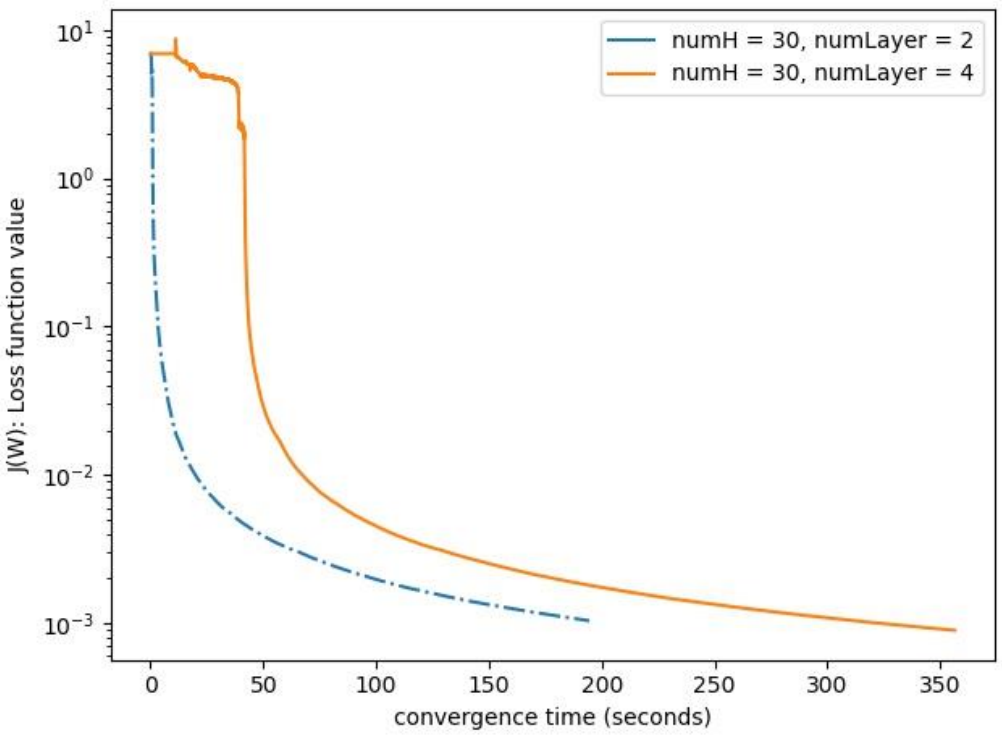


**Figure 4-8: The impact of the number of layers (i.e., recursions) on the complexity of training an RHN using the modified BP method**

### 4-5-1 Performance comparison

In the first experiment, we train an RHN with 50 hidden nodes, and 4 layers, to memorize the numeric characters (shown in Figure 4-4) using each of the SPSA and the modified BP scheme. Then, we compare the performance of the RHN in recreating distorted patterns against an RBM (comprising the same number of Hidden nodes and layers) while the Hamming distance of the input (with respect to the desired patterns) is ranging from 0 to 10. For each Hamming distance number, we populate 10,000 distorted images which are feed to the two realizations of RHN (one using the weight matrix generated by SPSA, and the other one trained with the modified BP scheme). As shown in Figure 4-9, our proposed training method results in a significantly lower error rate in recreating

the distorted patterns (compared to the SPSA scheme), especially for smaller Hamming distances.

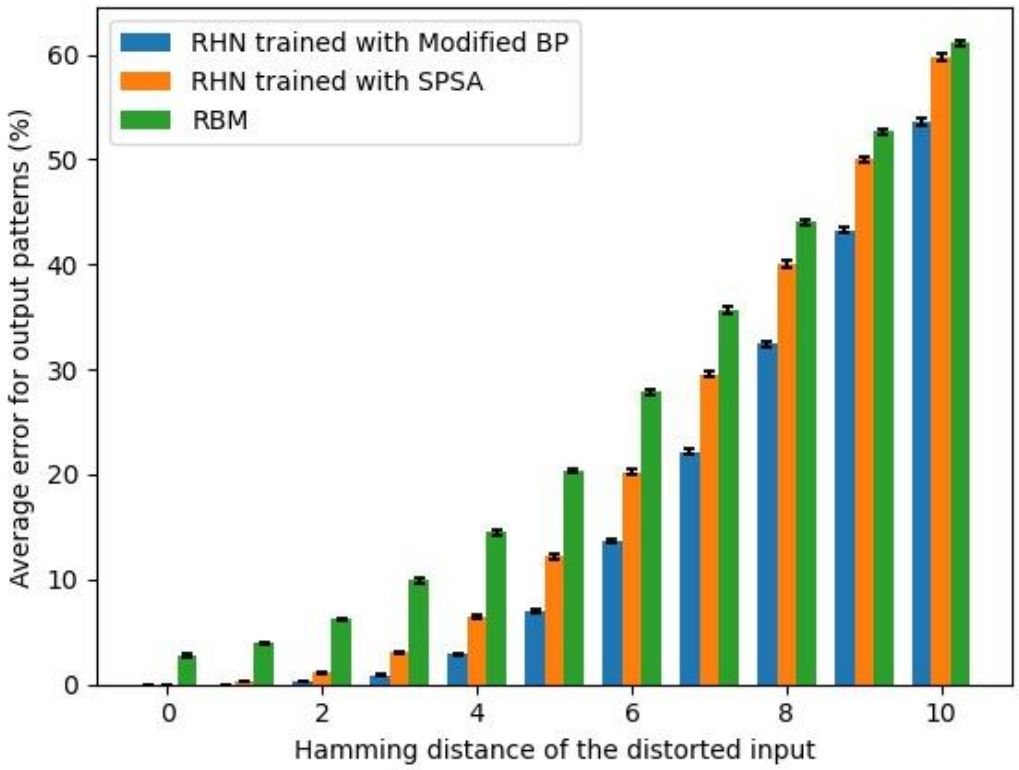


**Figure 4-9: The average error rate in recreating distorted patterns (with a Hamming distance ranging from 0 to 10) when the RHN is trained with modified BP compared to the SPSA scheme and Restricted Boltzmann Machine. The 95% confidence interval is shown on the top of each graph.**

To better observe the efficiency of the RHN and the efficiency of the proposed scheme for training the network, we report the average error rates for inputs with smaller Hamming distances (up to 5) in Table 3. Particularly, the RHN trained with modified BP is shown to recreate all the distorted patterns with a Hamming distance of 1, whereas SPSA results in an error rate of 0.294%. Also, for a distorted input with Hamming distance of 3, the modified BP results in an error rate of 0.8% which means that the error

is reduced by a factor of four compared to the SPSA training method. The performance of RHN in recreating the patterns is also compared with RBM in Table 3.

**Table 3: The average error rate in recreating distorted patterns (with a Hamming distance ranging from 1 to 5) for an RHN trained with SPSA compared to the modified BP**

| Hamming Distance | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Modified BP | 0 | 0.243 % | 0.806 % | 2.67 % | 6.385 % |
| SPSA | 0.294 % | 1.13 % | 3.07 % | 6.44 % | 12.16 % |
| RBM | 3.9 % | 6.19 % | 9.89 % | 14.47 % | 20.34 % |

As in the experiment of Section 4-3, RBM shows a considerably inferior performance compared to the RHN. The reason is mainly due to the flexibility of RHN to take on a continuum of values (in the range of (0,1)) at each hidden node. The performance gap here, however, is less significant (compared to the experiment of Figure 4-3), although we consider a more extensive dataset. The intuitive reason for this observation is the "plurality of hidden nodes" in this experiment which helps to better capture the features of the input patterns despite the limitation that each node can take on only a binary value.

As indicated by Figure 4-9, a considerable portion of distorted patterns may not be recreated when the Hamming distance of error is greater than 5. This is mainly because of the similarities in the input patterns, rather than being related to the capability of the network. Particularly, many of the characters of Figure 4-4 have a Hamming distance of 4 pixels from each other. Hence, distorting only two pixels suffices to become undecided between two patterns. Because of this reason, it is reasonable to investigate the performance of the network using other sorts of noise/distortion at the input.

To address this point, we perform another experiment wherein each pixel of the input is distorted with an analog Gaussian noise with a certain power. Particularly, considering a specific noise-power $\sigma^2$, we add a randomly generated analog distortion $n_i$ to each pixel $i$ where $n_i$ is chosen according to a normal (Gaussian) distribution with the mean value of zero and the variance of $\sigma^2$. Figure 4-11 shows the average error in recreating the noisy patterns for each of the SPSA and modified BP schemes where the noise power is ranging from zero to 0.25. The noise power of 0.25 (which is equivalent to a standard deviation of 0.5) is sufficient to swap 5 input bits/pixels (on average), while other pixels are also distorted from their original binary values (see Figure 4-10). Figure 4-11 shows the capability of the RHN in filtering the input noise and recreating the desired output compared to the RBM. It also shows the improvement that is achieved by the modified BP training method compared to the SPSA scheme.

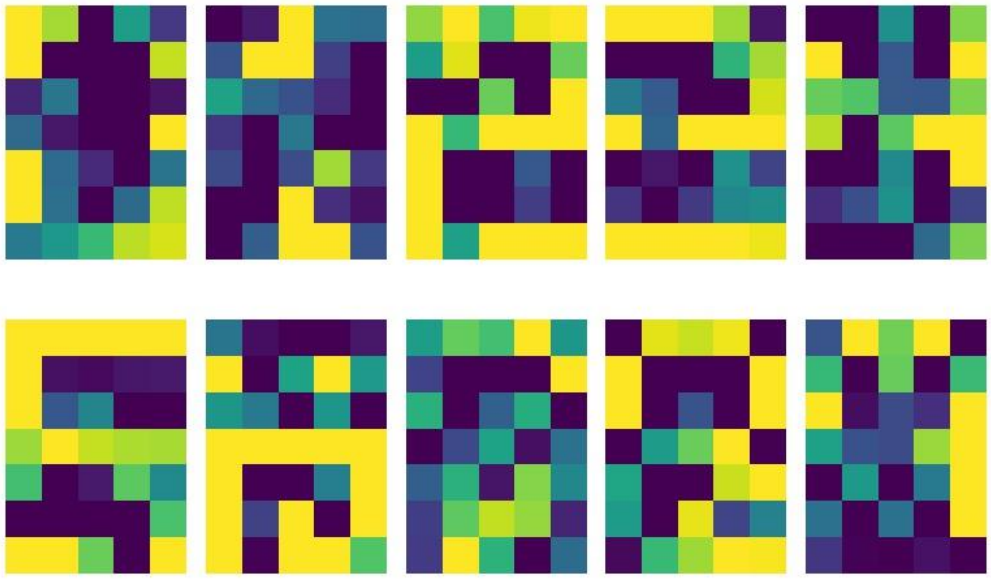

**Figure 4-10: A sample input pattern distorted by an analogue Gaussian noise with a zero mean and the standard deviation of 0.5**

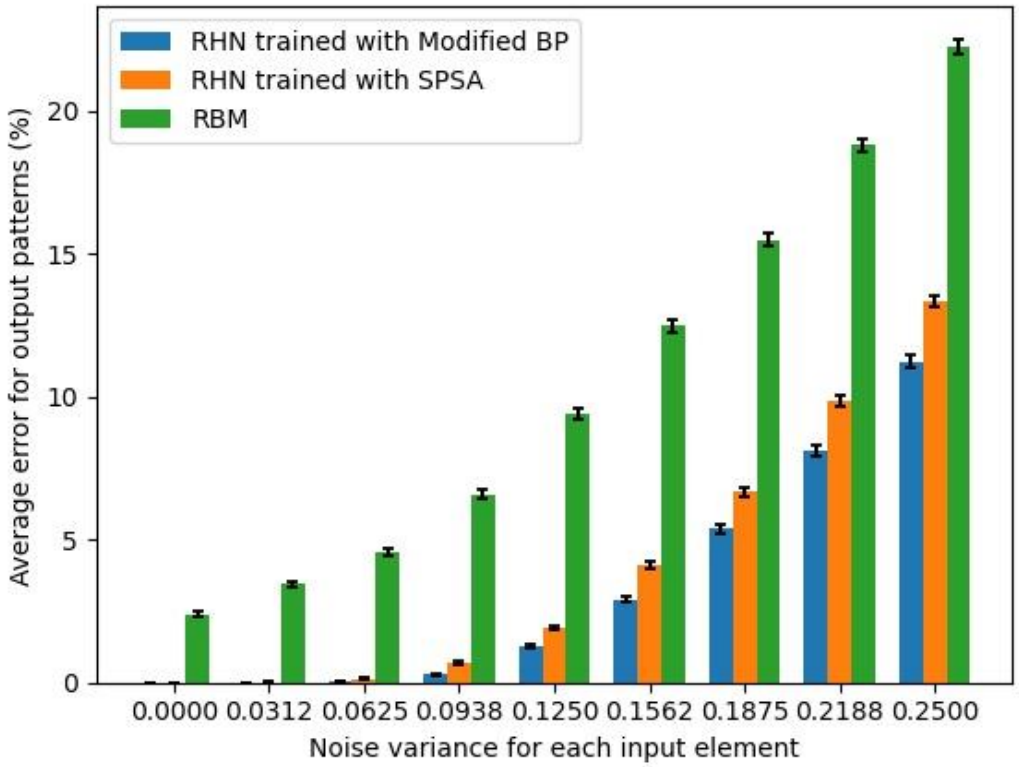


**Figure 4-11: The average error rate in recreating noisy inputs which are distorted by analogue noise with the noise power (i.e., variance) ranging from zero to 0.25.**

To better observe the improvement that is achieved by the RHN trained with modified BP compared to the SPSA scheme, and compared to the RBM, we report the performance improvement ratio of the average output error for various noise power values (ranging from 1/32 to ¼) in Table 4. It is observed that an improvement of around 20% is achieved by the modified BP compared to the SPSA in recreating the inputs which are heavily noisy (as shown in Figure 4-10 for instance for the noise power of ¼), while the achieved improvement is more significant for smaller values of noise power (see Table 4).

In this experiment, again the RHN is significantly outperforming the RBM, while the performance gap is significantly higher than the experiment of Figure 4-9. This could be

due to the flexibility of RHN to take on real-valued states at the hidden nodes which seem to have been more effective to filter out the input noise with analog quantities.

**Table 4: The improvement ratio of the output error for the modified BP training method compared to the SPSA scheme and Restricted Boltzmann Machine.**

| Noise power | $\frac{1}{32}$ | $\frac{1}{16}$ | $\frac{3}{32}$ | $\frac{1}{8}$ | $\frac{5}{32}$ | $\frac{6}{32}$ | $\frac{7}{32}$ | $\frac{1}{4}$ |
|---|---|---|---|---|---|---|---|---|
| Improvement Ratio compared to SPSA | Inf | 5.54 | 2.34 | 1.53 | 1.42 | 1.24 | 1.21 | 1.19 |
| Improvement Ratio compared to RBM | Inf | 207.72 | 22.48 | 7.55 | 4.28 | 2.87 | 2.31 | 1.98 |

### 4-5-2 Analyzing the impact of different parameters

In this subsection, we study the impact of different parameters, such as the number of hidden nodes and the number of recursions/layers on the performance of an RHN in recreating distorted inputs.

To evaluate the impact of the number of hidden nodes, we consider three realizations of the RHN, each comprising 4 layers with 30, 50, and 70 hidden nodes, respectively. The networks are trained to memorize the numeric characters of Figure 4-4 using the modified BP training method. Figure 4-12 compares the error rate of recreating distorted patterns for the realizations of the RHN with a different number of hidden nodes. As expected, the performance (in recreating distorted patterns) improves as the number of hidden nodes increases. The performance improvement, however, get saturated as we add more hidden nodes to the network. To better observe this effect, we report the improvement ratio of the output error when the number of hidden nodes is increased

from 30 to 50 nodes compared to the case that the number of hidden nodes is increased from 50 to 70 nodes. Indeed, the performance improvement diminishes if we keep increasing the number of hidden nodes. Hence, a moderate number of hidden nodes suffices in practice.

To observe the impact of the number of recursions on the performance of the network, we carry out another experiment wherein an RHN with 50 hidden nodes is recurrently running on the same input performing 4, 8, and 12 recursions, respectively. Figure 4-13 reports the average error rate of the output patterns for the RHN with different number of recursions. As expected, the network performance in recreating the distorted inputs is improved (i.e., pattern errors are reduced), though the performance improvement is negligible (around 5% for distorted inputs with larger Hamming distances).

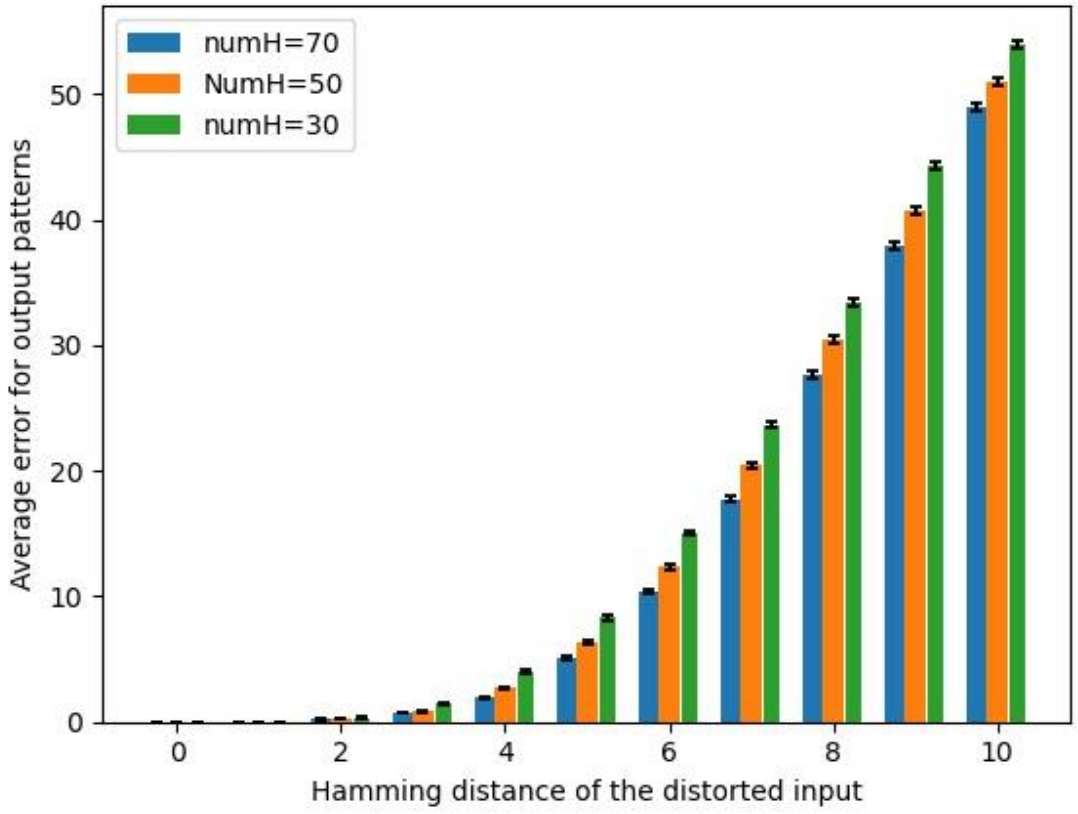


**Figure 4-12: The average error rate in recreating distorted patterns for an RHN with 4 layers, and various numbers (30, 50, and 70, respectively) of hidden nodes when the Hamming distance of the input is ranging from 0 to 10.**

**Table 5: The improvement ratio of the output error (for RHN in recreating distorted patterns with Hamming distances ranging from 2 to 10) when the number of hidden nodes is increased from 30 to 50 nodes, and from 50 to 70 nodes.**

| Improvement ratio | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| For numH=70 over numH=50 | 1.17 | 1.15 | 1.38 | 1.26 | 1.19 | 1.15 | 1.12 | 1.07 | 1.04 |
| For numH = 50 over numH = 30 | 1.31 | 1.78 | 1.5 | 1.3 | 1.22 | 1.16 | 1.1 | 1.08 | 1.06 |

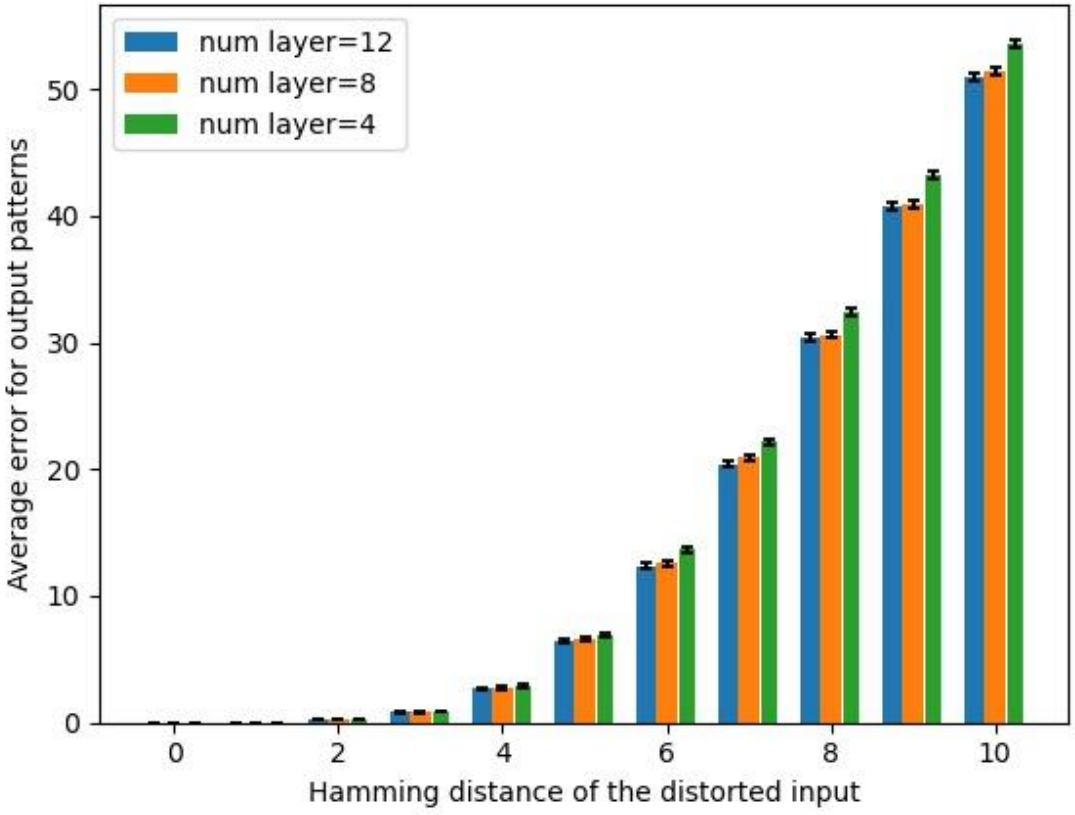


**Figure 4-13: The average error rate in recreating distorted patterns (with a Hamming distance ranging from 0 to 10) for an RHN with 50 hidden nodes compared for various numbers of recursions.**

### 4-5-3 Performance evaluation using a practical dataset

In this subsection, we evaluate the performance of RHN in memorizing the patterns and recreating them from distorted/noisy inputs for a real-world dataset comprising hand-written digits and compare it against the performance of an RBM network. Particularly, we use the "Tensorflow.Keras" dataset which contains a plurality of handwritten numeric characters (as shown in Figure 4-14) each represented with $28 \times 28$ pixels. We use 100 patterns of this dataset to train both the RHN and the RBM network, and then compare their capability to memorize the patterns and also their robustness against the possible distortions at the input.

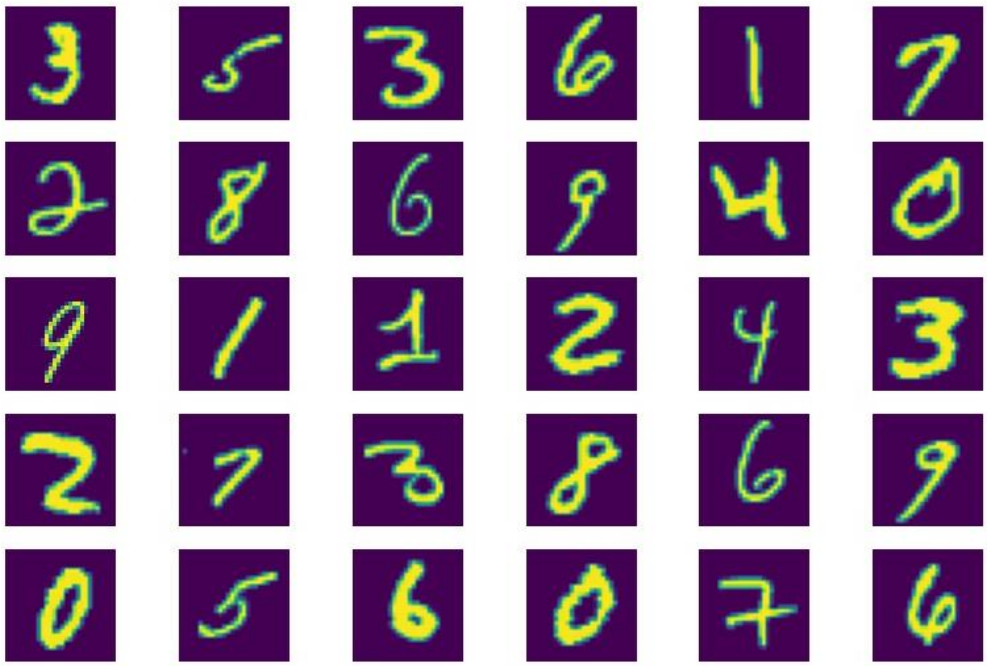

**Figure 4-14: A sample of "Tensorflow.keras" dataset comprising a plurality of handwritten numeric characters**

Our experiments indicate that we need a network with at least 100 hidden nodes to memorize such a large dataset. Particularly, we could train both RHN and RBM with 100 nodes which could successfully memorize all the patterns. However, the resulting

networks are not much robust against the distortions at the input. Hence, we performed another experiment training both RHN and RBM networks with 150 hidden nodes. We couldn't make the SPSA algorithm converge to a stable weight matrix, even for an RHN with 100 nodes. Hence, we report the RHN performance using our proposed training method. Since RBM is a stochastic neural network, we find 100 replicas of the weight matrix for given input patterns (using the training method presented in Chapter [2]), and then evaluate the network performance by taking a statistical average over different realizations of the network. Both of the RHN and RBM networks are shown to be extremely robust against the distortions at the input (as reported in Figure 4-15 and Figure 4-16, and discussed in the following).

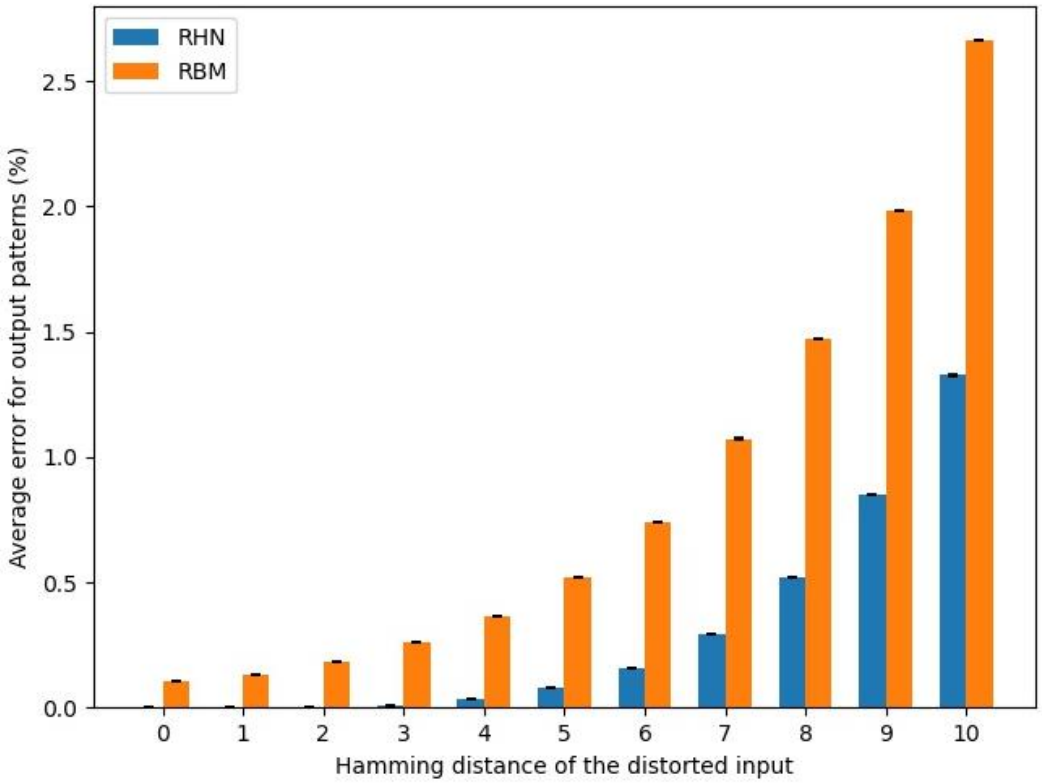


**Figure 4-15: The average error rate in recreating distorted keras patterns for an RHN with 2 layers, and 150 hidden nodes compared to an RBM network with the same parameters when the Hamming distance of the input is ranging from 0 to 10.**

Figure 4-15 shows the average percentage of the patterns which are erroneously re-created at the output when the Hamming weight of the input error is ranging from 0 to 10 pixels. To find this percentage, we generate 10,000 randomly distorted replicas of each pattern (for a given Hamming distance) and then count all the patterns for which even one pixel is different than the desired output. According to Figure 4-15, the RHN can fix around 99% of the distorted patterns which have a Hamming distance of up 10 (with respect to the corresponding desired pattern), whereas this number is around 97.5% for the RBM. The percentage of patterns which are erroneously recreated at the output is less than 0.2% for the RHN (and around 0.8 % for the RBM, respectively) when the input has a Hamming distance of 6 pixels.

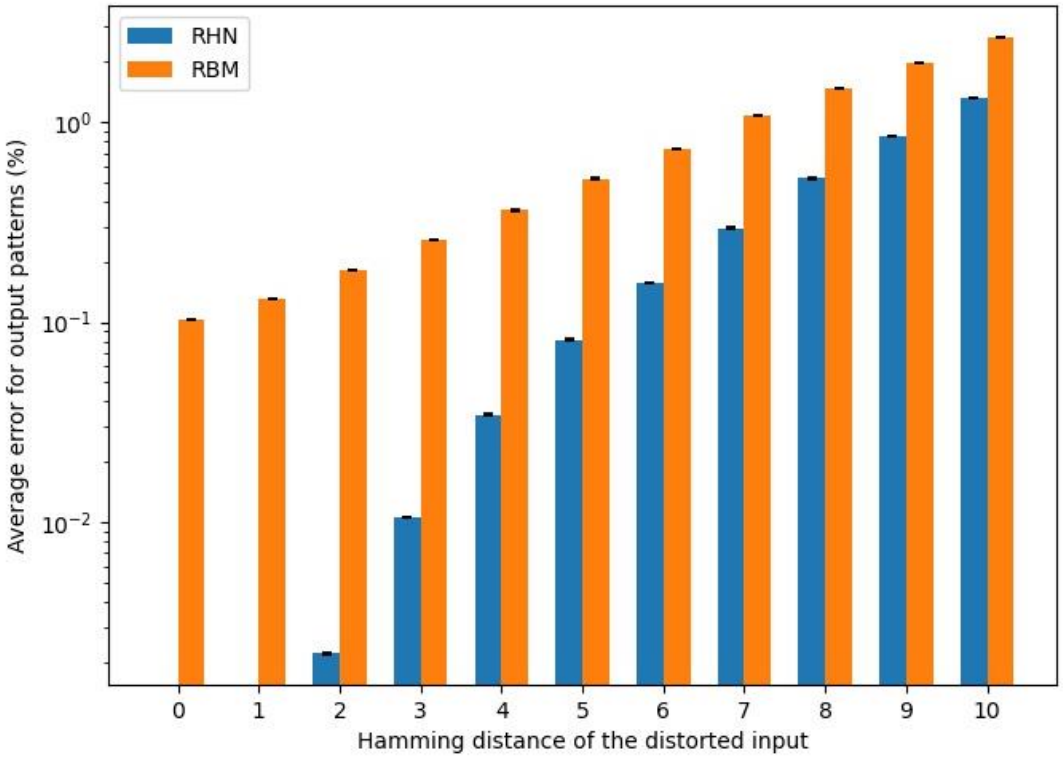


**Figure 4-16: The average error rate (in logarithmic scale) for an RHN compared to an RBM network (both comprising 2 layers and 150 hidden nodes) when the Hamming distance of the input is ranging from 0 to 10.**

To better observe the improvement of RHN compared to RBM for inputs with smaller Hamming distances, we plot in Figure 4-16 the average error rate of patterns in a logarithmic scale. We also report in Table 6 the average improvement ratio for different Hamming distances ranging from 1 to 6 in. It can be observed that the improvement ratio is more significant for inputs with smaller Hamming distances. In this experiment, we implemented both networks with two layers of recursions. The performance of each of them can be further enhanced by applying more recursions as we discussed in the previous subsections.

**Table 6: The average error rate in recreating distorted patterns (with a Hamming distance ranging from 1 to 6) for an RHN compared to the RBM network**

| Hamming Distance | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Improvement Ratio | Inf | 90.6 | 25.9 | 10.6 | 6.3 | 4.7 |

It is worth noting both networks may fix part of the distorted pixels for most of the distorted inputs. That is, the number of erroneous pixels of the pattern recreated at the output could be reduced compared to the input (though we count patterns even with one altered pixel as erroneous). To capture the capability of the networks in correcting erroneous pixels, we report in Figure 4-17 the average number of erroneous bits that remain in error at the output of the network for the inputs with different Hamming distances. It is observed that for RBM network the number of erroneous bits in the output is less than 0.1 pixel (on average), for a considerably distorted input with a Hamming distance of 10, whereas for RHN the erroneous bits are further reduced by a factor of around 2 for the same input. It means that RHN can fix 99.995% of erroneous pixels of the distorted inputs with the Hamming distance of 10.

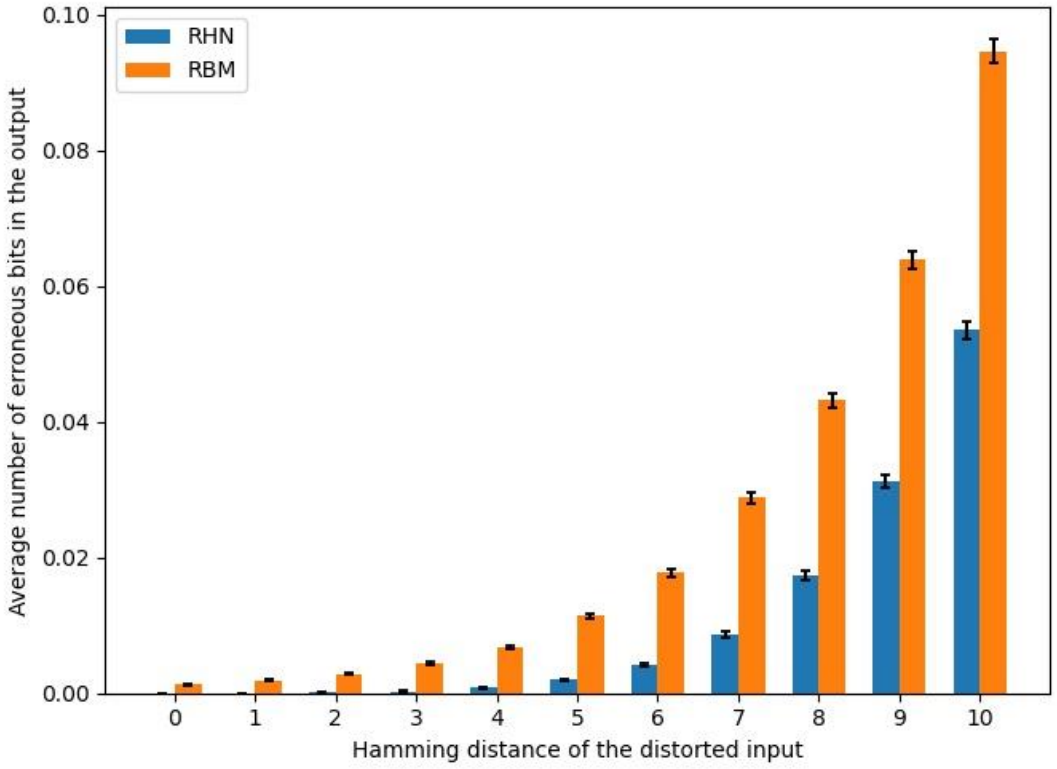


**Figure 4-17: The average number of erroneous bits at the output of an RHN compared to an RBM network (both comprising 150 hidden nodes and 2 recursions) for inputs with different Hamming distances ranging from 0 to 10.**

To evaluate the robustness of the RHN against other sorts of distortions, we perform another experiment wherein the inputs are distorted with an analog Gaussian noise. In this experiment, we generate 10,000 randomly distorted patterns where a Gaussian noise with a specific noise-power $\sigma^2$ is added independently to 15% of the pixels of each pattern. We perform this experiment for various noise powers ranging from 1/32 to ¼. Figure 4-18 shows a number of patterns from the "Tensorflow.Keras" dataset which are distorted with an analog noise as described above. The noise power of 0.25 (which is equivalent to a standard deviation of 0.5) is sufficient to alter tens of bits/pixels in each pattern; while some other pixels are also distorted from their original binary values (see Figure 4-18).

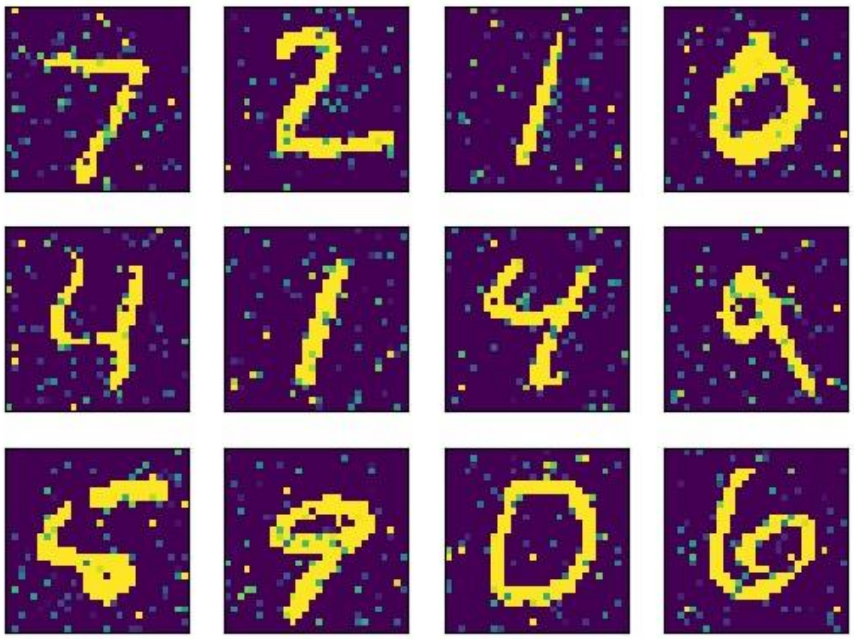

**Figure 4-18: Sample inputs from "Tensorflow.Keras" dataset, where 15% of the pixels of each pattern are distorted with an analog noise with a zero-mean and standard deviation of $\sigma = 0.5$**

Figure 4-18 compares the capability of RHN against RBM to filter the input noise and recreate the desired outputs from noisy inputs. It is observed that the average error rate in recreating the patterns is significantly improved by RHN compared to the RBM network, especially with noisy inputs with a smaller noise power. The performance gap decreases for larger noise powers, but still, RHN outperforms RBM.

The same observation is made in Figure 4-20 where the number of erroneous bits/pixels at the output is compared for RHN against RBM for inputs with different levels of noise powers. It is observed that RHN is extremely robust against the noisy inputs when the standard deviation of the noise is rather small (e.g., less than 0.1).

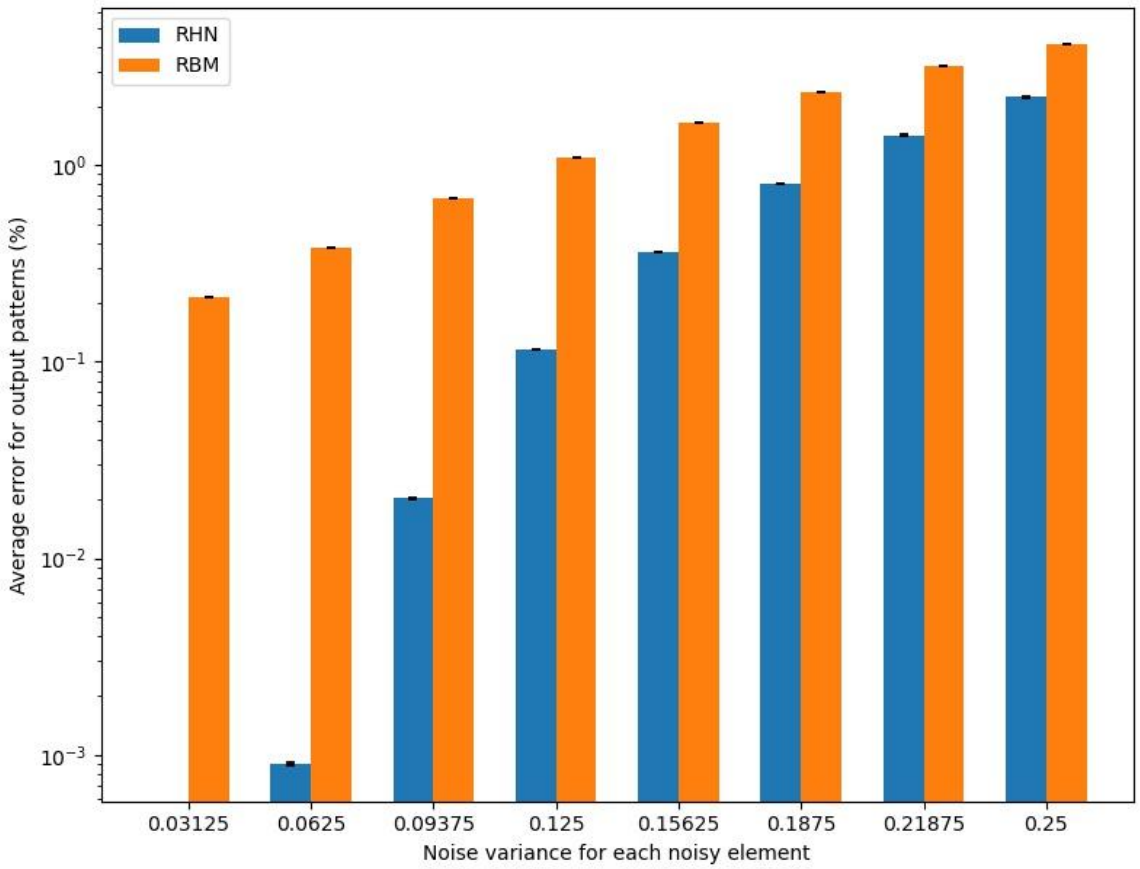


**Figure 4-19: The average error rate percentage (in logarithmic scale) for an RHN compared to an RBM network (both comprising 2 layers and 150 hidden nodes) when the input is distorted with an analog noise with the noise power of distorted pixels ranging from 1/32 to 1/4.**

Intuitively, an input with a small noise power may randomly alter the state of some hidden nodes in a stochastic RBM network. Hence, RBM could be more susceptible to small variations of the input (e.g. in the range of 0.1) as opposed to RHN which can flexibly take analog quantities as the state of hidden nodes. This helps RHN better filter out analog distortions of the input compared to the RBM network. That is why a more superior performance is achieved by RHN (compared to RBM) for inputs with smaller noise powers.

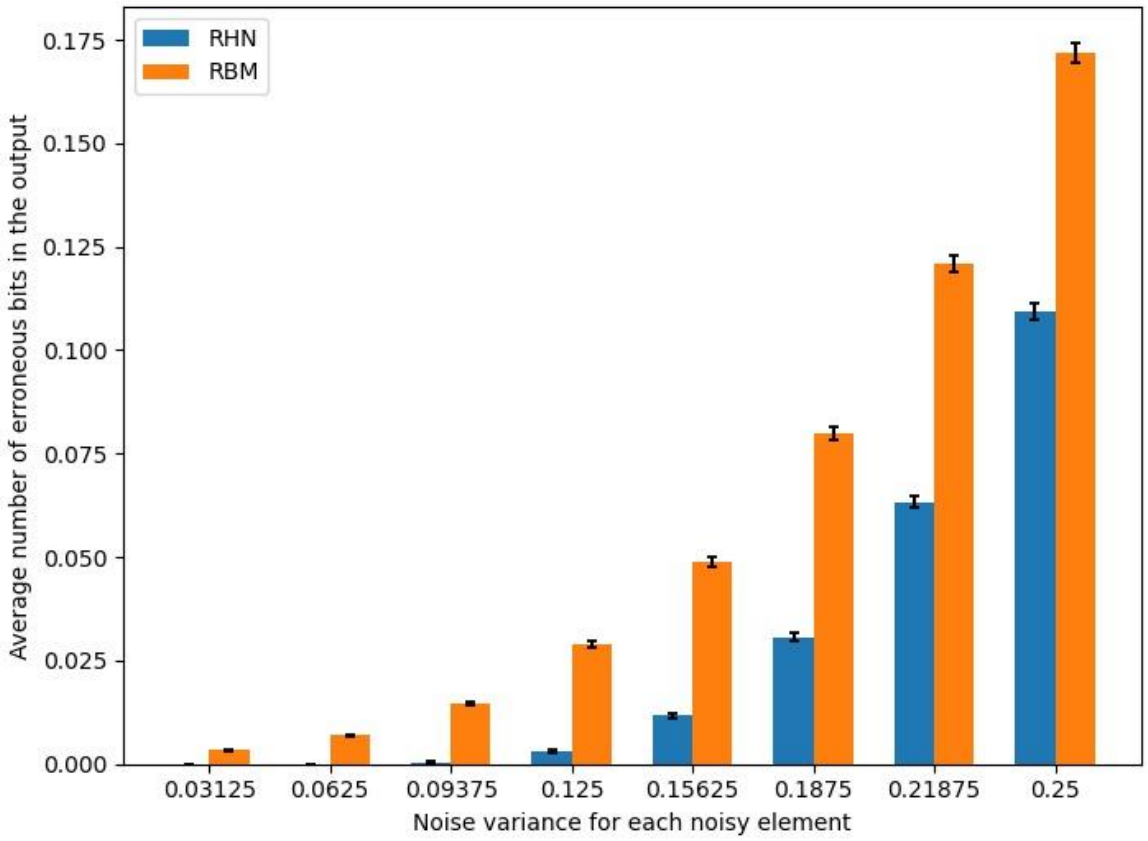


**Figure 4-20: The average number of erroneous bits at the output of an RHN compared to an RBM network (both comprising 150 hidden nodes and 2 recursions) when the input is distorted with an analog noise with the noise power of distorted pixels ranging from 1/32 to 1/4.**

These experiments show the great capability of RHN in memorizing the patterns as well as its robustness against the distortions at the input. They also demonstrate the superior performance of RHN compared to the conventional RBM network.

## 4-6 Summary of the chapter and some concluding remarks

In this chapter, we performed several experiments to evaluate the efficiency of the training method that we proposed for RHNs. The experiments in this chapter particularly demonstrated the enhanced performance of the proposed method (in terms of convergence and the capability of the network to recreate distorted patterns) compared

to the conventional SPSA scheme. We also compared the performance of RHN against the Hopfield network (which is trained by the Hebbian rule), as well as the RBM network (which is a stochastic recurrent neural network with a similar architecture). The original Hopfield network was shown to offer a limited capability in memorizing the patterns and re-creating them from distorted inputs. Moreover, as argued in Chapter 2, there is no way to improve the performance of a Hopfield network, as opposed to RBM and RHN for which the performance can be enhanced by increasing the number of hidden nodes.

Hence, we performed various experiments (using both synthetic and real data-sets) to compare RHN performance against the conventional RBM network. Particularly, RBM showed a considerably inferior performance compared to RHN for small data-sets where a network with a limited number of hidden nodes is implemented. The performance gap, however, reduces as we add more hidden nodes to the network for more extensive data-sets. We also observed that RHN shows more robustness against different sort of distortions (and especially analog noise) of the input. An intuitive reason is that the hidden nodes in RHN can take on any analog quantities in the range of (0,1), whereas the states of hidden nodes in an RBM are stochastically mapped to the binary values $\{0,1\}$. Hence, the hidden nodes in an RHN can convey more information compared to the RBM network. The "plurality of hidden nodes" in larger networks, however, helps to better capture the features of the input patterns despite the limitation that each node can take on only a binary value.

# Chapter 5

# Conclusion and Future Works

## 5-1 Summary and conclusion

In this thesis, we proposed a new training method for RHN which showed a significant improvement in terms of convergence compared to the existing SPSA method. The proposed method not only enhanced the convergence of the training sub-routine but also was shown to enhance the learning capability of the network. To achieve this enhancement, we first strived to analytically calculate the gradient of the loss function for RHN using the back-propagation technique through time. Then, we employed the gradient descent algorithm to iteratively update the weight matrix of the network. As the gradient-decent algorithm might end up with a local optimum solution, we proposed a new variant of SPSA, which is used in conjunction with back-propagation, updating the weight matrix in a random direction in the vicinity of a local optimum solution. The complexity of random search in the proposed scheme, however, is reduced by confining the search space only to components which have the greatest contribution to the loss

function. In this way, we reduced not only the computational complexity but also improved the agility of the solution by searching in more effective directions.

We implemented the proposed scheme by programming in Python, to verify and evaluate the convergence of the training sub-routine, analyze the impact of different parameters, and evaluate the performance of the whole network compared to existing solutions. The numerical experiments demonstrated the enhanced performance of the training method not only in terms of convergence but also in terms of performance of the network (in re-creating distorted patterns) compared to the conventional SPSA scheme. We also compared the performance of RHN (with different training methods) against the Hopfield network (programmed with the Hebbian rule) and the RBM network. The experiments demonstrated the superior and robust performance of RHN compared to the RBM for both synthetic and practical datasets (such as Tensorflow Keras package) in the presence of different sorts of distortions at the input.

## 5-2 Future Work

There are a number of directions which can be further investigated in a future study. We briefly describe a few of them in the following.

- Given our proposed enhanced training method for the RHN, one can study the implementation of RHN for other applications wherein a recurrent neural network is desired. One of the applications, for instance, which can benefit from our proposed training method, is moving object detection. In this context, the goal is to detect the physical movement of an object in a certain area [54], [55]. Particularly, assume multiple consecutive frames of a video which should be used to determine if an object is moving. This potentially results in a huge temporal data which needs to be processed by a recurrent neural network with a high memory capacity such as an RHN. Our proposed training method can be useful to achieve optimum and accurate training (without getting stuck in a local

optimum) while reducing the convergence time which can be crucially important for such applications.

- Some of the ideas/techniques that we presented here may have more generic applications. E.g., the idea that we presented to modify the SPSA algorithm can also be useful in the context of *hyper-parameter optimization.* Particularly, there are usually several model-related parameters which should be properly tuned prior to training the network. Optimizing such parameters usually involves several rounds of pilot training which could be computationally demanding. By confining the search space and updating only a subset of variables that have direct impact on a desired objective/output, not only the computational complexity is reduced, but also the random search can get more efficient. We leave this promising direction for further investigation in the future.
- Another promising direction is investigating the application of the presented ideas for the training of feed-forward MLP neural networks. As we discussed in Chapter 2, the back-propagation method is conventionally used to train a MLP network. BP scheme, however, is effective to train MLP networks with a limited number of hidden layers (e.g., networks with 2 or 3 hidden layers). For deep neural networks with several hidden layers (e.g., 5 layers of hidden nodes), however, the BP method becomes less effective.

  Intuitively, the degree of non-linearity of the network increases as we incorporate more hidden layers to a FF MLP network. As a result, there could be various local optimum solutions which could disturb the performance of the BP training method which is based on the "gradient descent" scheme. The back-propagated error (which gives an estimate of the gradient) vanishes at the initial layers, especially when approaching a local minimum solution.

  A possible solution to alleviate the global convergence issue of the BP method for a FF MLP network is to incorporate the SPSA method to induce a random search direction when approaching a local optimum solution. To make the

random search more effective, however, one can employ the idea that we presented in Chapter 3 to confine the search space. Particularly, one can find the set of nodes with the maximum error at the output of each layer, and then update (in a random direction) only the weights of edges that are connected to the nodes with the maximum error. In this way, it is expected to expedite the training subroutine by making the random searches more effective. We leave this potential extension for further investigation in the future.

# References

[1] K. O'Shea, R. Nash, "An Introduction to Convolutional Neural Networks", arXiv:1511.0845v2[cs.NE], Dec 2015.

[2] C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

[3] R. J. Mceliece, E. C. Posner, E. R. Rodemich, S. S. Venkatesh, "The Capacity of the Hopfield Associative Memory", IEEE Transactions on Information Theory, VOL. 33, NO. 4, pp. 461-482, July 1987

[4] J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," In Proc. National Academy of Sciences of the United States of America, April 1982.

[5] T. Yeap, "Implementation of an Associative Memory Using a Restricted Hopfield Network", In Proc. Int. Research Conference, pp. 112-116, May 2020.

[6] J. J. Hopfield, D. W. Tank, "Neural Computation of Decisions in Optimization Problems", Biolagical Cybernetics, Vol. 52, pp. 141-152, 1985.

[7] A.J. Storkey, R. Valabregue, "The Basins of Attraction of a New Hopfield Learning Rule", Neural Networks, Vol. 12, pp. 869-876, 1999.

[8] E. Gardner, "Maximum Storage Capacity in Neural Networks", Europhys. Letters, Vol. 4, pp. 481-485, 1987.

[9] E. Gardner, “The Space of Interactions in Neural Network Models”, Journal of Physics A: Mathematical and General, Vol. 21, pp. 257-270, 1988.

[10] E. Gardner, B. Derrida, “Optimal Storage Properties of Neural Network Models”, Journal of Physics A: Mathematical and General, Vol 21, pp. 271-284, 1988.

[11] R. Salakhutdinov, G. Hinton, “Restricted Boltzmann Machines”, In Proc. of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS), Florida, USA, 2009.

[12] I. Sutskever, G. Hinton, G. Taylor, “The Recurrent Temporal Restricted Boltzmann Machine”, In Proc. of the Neural Information Processing Systems, 2008.

[13] R. R. Salakhutdinov, A. Mnih, G. E. Hinton, “Restricted Boltzmann machines for Collaborative Filtering”, In Proc. 24th ACM International Conference on Machine Learning, Vol. 24, pp. 791-798, 2007.

[14] J.C. Spall, “Multivariate Stochastic Approximation Using Simultaneous Perturbation Gradient Approximation”, IEEE Transaction on Automatic Control, Vol. 37, No. 3, pp. 333-341, March 1992.

[15] J.C. Spall, “An Overview of the Simultaneous Perturbation Method for Efficient Optimization”, Johns Hopkins Apl Technical Digest, Vol. 19, No. 4, pp. 482-492, 1998.

[16] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview", Neural Networks, Vol. 61, pp. 85-117, 2015

[17] I. Goodfellow, Y. Bengio, A. Courville, “Deep Learning”, Cambridge: MIT Press, 2016.

[18] A. Zell, “Simulation Neuronaler Netze”, Bone: Addison-Wesley, 1994.

[19] M. H. Sazly, “A Brief Review of Feed-Forward Neural Networks”, Ankara University, Technical Report, Department of Electronics Engineering, Ankara, Turkey, 2006.

[20] N. Japkowicz, “Supervised Versus Unsupervised Binary-Learning by Feedforward Neural Networks”, Machine Learning, Kluwer Academic Publishers， Vol. 42, pp. 97–122, 2001.

[21] A. S. Fernandez, C. Delgado-Mata, R. Velazquez, “Training a Single-Layer Perceptron for an Approximate Edge Detection on a Digital Image”, In Proc. IEEE Technologies and Applications of Artificial Intelligence, 2011.

[22] A. K. Jain, J. Mao, and K. Mohiuddin, “Artificial Neural Networks: A tutorial” Computer, vol. 29, pp. 31–44, 1996.

[23] Z. Yanling, D. Bimin, and W. Zhanrong, “Analysis and Study of Perceptron to Solve XOR Problem”, In Proc. IEEE 2nd International Workshop on Autonomous Decentralized System, 2002.

[24] S. Sharma, S. Sharma, A. Athaiya, “Activation Functions in Neural Networks”, International Journal of Engineering Applied Sciences and Technology, Vol. 4, No. 12, pp. 310-316, 2020.

[25] P. Sibi, S.A. Jones, P.Siddarth, “Analysis of Different Activation Functions Using Back Propagation Neural Networks”, Journal of Theoretical and Applied Information Technology, Vol. 47 No.3, pp. 1264-1268, 2013.

[26] F. Rosenblatt, “The Perceptron: A Theory of Statistical Separability in Cognitive Systems”, Cornell Aeronautical Laboratory, Report No. VG1196-G-1, Jan 1958.

[27] B. M. Wilamowski, “Neural Networks Architectures and Learning Algorithms”, IEEE Industrial Electronics Magazine, Vol. 3, No. 4, pp. 56-63, 2009

[28] V.K. Singh, “One Solution to XOR Problem Using Multilayer Perceptron Having Minimum Configuration”, International Journal of Science and Engineering, Vol. 3, No. 2, pp. 32-41, 2015.

[29] V.K., Singh, “Two Solutions to the XOR Problem Using Minimum Configuration MLP”, International Journal of Advanced Engineering Science and Technological Research, Vol. 3, No. 3, pp. 16-20, 2015.

[30] H. Ramchoun, M. A. Janati Idrissi, Y. Ghanou, M. Ettaouil, “Multilayer Perceptron: Architecture Optimization and Training”, International Journal of Interactive Multimedia and Artificial Intelligence, Vol. 4, No. 1, 2016.

[31] D.Salamon, “Data Compression”, Springer, 2004

[32] H. Ramchoun, M A. Janati Idrissi, Y. Ghanou, and M. Ettaouil, “New Modeling of Multilayer Perceptron Architecture Optimization with Regularization: An Application to Pattern Classification”, IAENG International Journal of Computer Science, Vol. 44, No. 3, pp.261-269, Aug 2017.

[33] J.J. Hopfield, D.W. Tank, “Neural computation of decisions in optimization problems”, Biological Cybernetics, Vol. 52, No. 3, pp. 141-152, 1985.

[34] M. Schuster, K. K. Paliwal, “Bidirectional Recurrent Neural Networks”, IEEE Transactions on Signal Processing, Vol. 45, No. 11, pp.2673-2681, 1997

[35] J.J. Hopfield, D.W. Tank, “Computing with Neural Circuits: A Model”, Science, Vol. 233, No. 4764, pp. 625-633, Aug 1986.

[36] G. A. Carpenter, “Neural Network Models for Pattern Recognition and Associative Memory”, Neural Networks, Vol 2, No. 4, pp 243-257, 1989.

[37] Ch. Li, “Deep Learning for Object Detection and Tracking and for Field of View Prediction in 360-Degree Videos”, PhD. Dissertation, Tandon School of Engineering, New York University, 2019

[38] S. Amari, “Backpropagation and Stochastic Gradient Descent Method”, Neurocomputing, Vol. 5, No. 4-5, pp.185-196, 1993.

[39] D. E. Rumelhart, Y. Chauvin, “Backpropagation Theory, Architectures, and Applications”, Psychology Press, 1995.

[40] S. Amari, “Theory of Adaptive Pattern Classifiers”, IEEE Transactions on Electronic Computers, Vol. 3, pp. 299-307, 1967.

[41] R. Salakhutdinov, G. Hinton, “Deep Boltzmann Machines”, In Proc. International Conference on Artificial Intelligence and Statistics, 2009.

[42] M. Bod´en, “A Guide to Recurrent Neural Networks and Backpropagation”, School of Information Science, Computer and Electrical Engineering, Halmstad University, 2001.

[43] O. Hee-Kuck, “The Relaxation Method for Learning in Artificial Neural Networks”, PhD dissertation, Dept. Computer Science, Iowa State University, 1992

[44] Y. Maeda, M. Wakamura, “Bidirectional associative memory with learning capability using simultaneous perturbation”, Neurocomputing, Vol. 69, No. 1-3, pp. 182–197, 2005.

[45] D. Hebb, “The Organization of Behavior: A Neuropsychological Theory”, John Wiley and Sons, Vol. 62, P. 78, 1949

[46] S. Boyd, L. Vandenberghe, “Convex Optimization*”*. Cambridge University Press, 2004.

[47] Y. S. Abu-Mostafa, St. J. M. Jacques, "Information Capacity of the Hopfield Model," IEEE Transactions on Information Theory, Vol. 31, No. 4, pp. 461-464,

July 1985

[48] K. Lee, S. C. Kothari, D. Shin, "Probabilistic Information Capacity of Hopfield Networks," Complex Systems, Vol. 6, No. 1, pp. 31-46, 1992.

[49] B. Kosko, "Bidirectional Associative Memories" IEEE Transactions on System, Man, and Cybernetics, Vol. 18, No. 1, pp. 49-60, 1988.

[50] P. Simpson, "Bidirectional Associative Memory Systems," General Dynamics Electronics Division, Technical Report GDE-ISG-PKS-02, 1988.

[51] S. Agmon, "The Relaxation Method for Linear Inequalities" Canadian Jornal of Mathematics, Vol. 6, pp. 382-392, 1954.

[52] M. H. Hassoun, "Dynamic Heteroassociative Neural Memories" Neural Networks, Vol. 2, No. 4, pp. 275-287, 1989.

[53] D. H. Ackley, G.E. Hinton, T. J. Sejnowski, “A Learning Algorithm for Boltzmann Machines”, Cognitive Science, Vol. 9, No. 1, pp. 147-169, 1985.

[54] G.E. Hinton, R.R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks." Science, Vol. 313, No. 5786, pp. 504-507, 2006.

[55] J. S. Kulchandani, K. J. Dangarwala, "Moving Object Detection: Review of Recent Research Trends", In Proc. IEEE International Conference on Pervasive Computing, 2015.

[56] R. R. Salakhutdinov, A. Mnih, G. E. Hinton, “Restricted Boltzmann Machines for Collaborative Filtering”, In Proc. of the International Conference on Machine Learning, Vol. 24, pp. 791-798, 2007.

[57] R. Salakhutdinov, G. Hinton, “Deep Boltzmann machines”, In Proc. 12th International Conference on Artificial Intelligence and Statistics (AISTATS), 2009.

[58] G.E. Hinton, "Deep belief networks", Scholarpedia, Vol. 4, No. 5, 2009.

[59] P. Smolensky, “Information Processing in Dynamical Systems: Foundations of Harmony Theory”, Parallel Distributed Processing, Vol. 1, pp. 194-281. MIT Press, Cambridge, 1986.

[60] Y. Freund, D. Haussler, “Unsupervised Learning of Distributions on Binary Vectors Using Two Layer Networks”, In proc. Advances in Neural Information Processing Systems, Vol. 4, pp. 912-919, 1992.

[61] G.E Hinton, “Training products of experts by minimizing contrastive divergence”, Neural Computation, Vol. 14, No. 8, pp. 1711-1800, 2002.

[62] M.A. Carreira-Perpinan, G.E. Hinton, “On contrastive divergence learning”, Artificial Intelligence and Statistics”, Vol. 10, pp 33-40, 2005.

# Appendix

Enclosed, is the code that we developed in Python to implement our proposed training method. Particularly, the main sub-routine in Table 7 implements the training algorithm which we presented in Chapter 4. The implementation of RHN class itself is presented in Table 8, while the functions which are called by the main sub-routine to calculate the derivative of the loss function, and to find a random update direction, respectively, are brought in Table 9 and Table 10.

**Table 7: The main sub-routine**

```
import numpy as np
import matplotlib.pyplot as plt
from rhn_class import RHN
from calculateError_and_Derivative import calculateErrorDer
from find_modifiedSPSA_direction import findRandDirection
import time
import matplotlib.image as img
from PIL import Image
import math

######################################################################
# Initialize the input (parameters and desired patterns) to the network
from digit_INP import desiredOTP_mpvd

desiredOTPOrigin = np.array(desiredOTP_mpvd)
desiredOTP = np.maximum(np.minimum(desiredOTPOrigin, 1 - 0.01), 0.01)
numP = np.size(desiredOTPOrigin, 0)  # number of patterns
dimP = np.size(desiredOTPOrigin, 1)  # Dimension of each pattern

# Initialize the network parameters
numH = 70  # number of hidden layers
numV = dimP  # number of input nodes
```

```
numLayers = 2  # number of layers of the network
rHnet = RHN(numV, numH, numLayers)
Wloaded = rHnet.GetWeitghs()
biasV = Wloaded[1:numV + 1, 0]
biasH = Wloaded[0, 1:numH + 1]
W = Wloaded[1:numV + 1, 1:numH + 1]

# Input parameters to the training module
numItr = 200000  # Maximum number of iterations
itrArr = np.arange(1, numItr, 1).tolist()  # array for main loop
iterations
maximumError = 0.0015  # stopping criterion
ak_max = 1 / 20  # Initial value for back tracking parameter
numBackTracking = 10  # Number of back tracking retries
dviStep = 2  # Division factor for back tracking
logError = np.zeros(np.size(itrArr) + 1)  # Keep track of the loss
function values in each iteration
logTime = np.zeros(np.size(itrArr) + 1)
thshld = 1e-5

# Find the initial value of the loss function
[logError[0], tmpErr, deltaW, deltaBV, deltaBH] =
calculateErrorDer(desiredOTP, rHnet)
minW = W  # the best W matrix (along with the biases) ever found by the
training algorithm
minBV = biasV
minBH = biasH
minLog = np.sum(np.abs(tmpErr))  # minimum value for the loss function
ever found

############## The main loop for iterative training #################
t0 = time.clock()
print("Start")

for kk in itrArr:
    # In each iteration, we strive to find a decent direction
      so as to reduce the error function
    ak = ak_max
    WL = W - ak * deltaW
    biasVL = biasV - ak * deltaBV
    biasHL = biasH - ak * deltaBH
    numTries = 1  # num back tracking
    rHnet.setWB(WL, biasVL, biasHL)

    # The function "calculateErrorDer(...)" is called to calculate
      total error (qL), component-wise error (tmpErr),
    # and the gradient components as a potential moving direction
    [qL, tmpErr, tmpDW, tmpDBV, tmpDBH] = calculateErrorDer(desiredOTP,
                                          rHnet)

    # Back tracking line search (is meant to find a proper step size)
    while (qL > logError[kk - 1]) and numTries <= numBackTracking:
        ak = ak / dviStep
        WL = W - ak * deltaW
        biasVL = biasV - ak * deltaBV
        biasHL = biasH - ak * deltaBH
```

```
        rHnet.setWB(WL, biasVL, biasHL)
        [qL, tmpErr, tmpDW, tmpDBV, tmpDBH] =
              calculateErrorDer(desiredOTP,rHnet)
        numTries = numTries + 1

    normGrd = ak * np.sqrt(np.mean(np.square(tmpDW)))
    # To check if the gradient direction in conjunction
      with back-tracking would reduce the error function.
    # A small norm for the gradient (i.e., normGrd < thshld) is
      indicative of vicinity to a local optimum point.
    # Hence, we use the gradient direction to update only if
      normGrd > thshld
      or if approaching the global optimum point (qL < 1)
    if qL < logError[kk - 1] and (normGrd > thshld or qL < 1):
        W = WL
        biasV = biasVL
        biasH = biasHL
        logError[kk] = qL

        logTime[kk] = time.clock() - t0
        deltaW = tmpDW
        deltaBV = tmpDBV
        deltaBH = tmpDBH
        # print(numTries)
    else:
        # Take a random moving direction if we are in the vicinity of
         a local optimum point
        [logError[kk], tmpErr, W, biasV, biasH, deltaW, deltaBV,
        deltaBH] = findRandDirection(desiredOTP, logError[kk - 1],
        tmpErr, W, biasV, biasH, deltaW, deltaBV, deltaBH, rHnet)
        logTime[kk] = time.clock() - t0
    if  minLog > np.sum(np.abs(tmpErr)):
    # Log if a minimum value is achieved for the loss function
        minLog = np.sum(np.abs(tmpErr))
        minW = W
        minBV = biasV
        minBH = biasH
    if np.sum(np.abs(tmpErr)) < maximumError or kk >= numItr:
        break
    print(logError[kk], np.sum(np.abs(tmpErr)))

# Save the output result
W = minW
biasV = minBV
biasH = minBH
Woutput = np.zeros((numV + 1, numH + 1))
Woutput[1:numV + 1, 0] = biasV
Woutput[0, 1:numH + 1] = biasH
Woutput[1:numV + 1, 1:numH + 1] = W
np.save("WEIGHT_MPVD_NH{}NL{}MaxE{}".format(numH, numLayers, minLog),
Woutput)
```

**Table 8: Sub-routines to implement RHN class.**

```
import numpy as np

class RHN:
    def __init__(self, num_visible, num_hidden, numLayer):
        self.num_visible = num_visible
        self.num_hidden = num_hidden
        self.numLayer = numLayer

        # Initialize a weight matrix, of dimensions (num_visible x
          num_hidden), using a Gaussian distribution with mean 0
          and standard deviation 0.05.
        self.weights = 0.05 * np.random.randn(self.num_visible,
                                    self.num_hidden)
        # Insert weights for the bias units into the first row
          and first column.
        self.weights = np.insert(self.weights, 0, 0, axis=0)
        self.weights = np.insert(self.weights, 0, 0, axis=1)

    def run_visible(self, data):

    """
    Assuming the RHN has been trained (so that weights for the network
    have been learned), run the network on a set of visible units, to
    get a sample of the hidden units.

    Parameters
    ----------
    data: A matrix; each row consists of the states of visible units.

    Returns
    -------
    hidden_states: A matrix; each row consists of the hidden units
    activated from the visible units when the data matrix passed in.
    """
        num_examples = data.shape[0]
        # Create a matrix, where each row is to be the hidden units
          (plus a bias unit)
        hidden_states = np.zeros((num_examples, self.num_hidden + 1))

        # Insert bias units of 1 into the first column of data.
        data = np.insert(data, 0, 1, axis=1)
        # Calculate the activations of the hidden units.
        hidden_activations = np.dot(data, self.weights)
        # Calculate the output of the hidden units
        hidden_states = self._logistic(hidden_activations)
        # Ignore the bias units.
        hidden_states = hidden_states[:, 1:]

        return hidden_states

    def run_hidden(self, data):
    """
    Assuming the RHN has been trained (so that weights for the network
    have been learned), run the network on a set of hidden units, to
```

```
    get a sample of the visible units.

    Parameters
    ----------
    data: A matrix; each row consists of the states of hidden units.

    Returns
    -------
    visible_states: A matrix where each row consists of the visible
    units activated from hidden units when the data matrix passed in.
    """

        num_examples = data.shape[0]
        # Create a matrix, where each row is to be the visible units
          (plus a bias unit)
        visible_states = np.zeros((num_examples, self.num_visible + 1))

        # Insert bias units of 1 into the first column of data.
        data = np.insert(data, 0, 1, axis=1)
        # Calculate the activations of the visible units.
        visible_activations = np.dot(data, self.weights.T)
        # Calculate the output of the visible units
        visible_states = self._logistic(visible_activations)
        # Ignore the bias units.
        visible_states = visible_states[:, 1:]

        return visible_states

    def read_machine(self, filename):
        weights = np.load(filename)
        self.weights = np.reshape(weights, (self.num_visible + 1,
             self.num_hidden + 1))

    def setWeights(self, weightsIN):
        self.weights = np.reshape(weightsIN, (self.num_visible + 1,
             self.num_hidden + 1))

    def setWB(self, W, bV, bH):
        Win = np.zeros((self.num_visible + 1, self.num_hidden + 1))
        Win[1:self.num_visible + 1, 0] = bV
        Win[0, 1:self.num_hidden + 1] = bH
        Win[1:self.num_visible + 1, 1:self.num_hidden + 1] = W
        self.weights = Win

    def GetNumV(self):
        return self.num_visible

    def GetNumH(self):
        return self.num_hidden

    def GetWeitghs(self):
        return self.weights

    def _logistic(self, x):
        x = np.maximum(np.minimum(x, 300), -300)
        return 1.0 / (1 + np.exp(-x))
```

```
    def calculateOTP(self, vIn):
        vr = np.asarray(vIn)
        for i in range(self.numLayer):
            hr = self.run_visible(vr)
            vr1 = self.run_hidden(hr)
            vr = vr1
        return vr, hr

    def calculateOTP_detailed(self, vIn):
        numP = np.size(vIn, axis=0)
        vVLayers = np.zeros((self.numLayer, numP, self.num_visible))
        vHLayers = np.zeros((self.numLayer, numP, self.num_hidden))
        vr = np.asarray(vIn)
        for i in range(self.numLayer):
            vVLayers[i] = vr
            hr = self.run_visible(vr)
            vr1 = self.run_hidden(hr)
            vHLayers[i] = hr
            vr = vr1
        return vr, vVLayers, vHLayers
```

**Table 9: Sub-routine implemented to calculate the derivative of the error function.**

```
import numpy as np
from rhn_class import RHN

def calculateErrorDer(desiredOTPP, rHNET):

    """
    Calculate the logarithmic loss/error function for a network rHNET
    (with a fixed weight matrix) operating on desired patterns

    param desiredOTPP: The matrix of desired output patterns
    param rHNET: An object of the RHN class with preset weights
    return: errorLogarithm: The value of the logarithmic loss function
            arrTmp: The components of loss function for different
                    output elements and for each pattern
            dW: The derivative of loss function w.r.t. weight matrix W
            dbV: The derivative of loss function w.r.t. bias vector, bV
            dbH: The derivative of loss function w.r.t. bias vector, bH
    """

    m = np.size(desiredOTPP, 0)
    numV = rHNET.GetNumV()
    numH = rHNET.GetNumH()

    # Call "rHNET.calculateOTP_detailed" to calculate the output at
      the very last layer/iteration (calculatedOut)
    # The function "rHNET.calculateOTP_detailed" also returns the
      state of visible and hidden nodes at intermediate levels
    calculatedOut, vVLay, vHLay =
                          rHNET.calculateOTP_detailed(desiredOTPP)
    cOut = np.minimum(np.maximum(calculatedOut, 1e-16), 1 - 1e-16)
```

```
    Wloaded = rHNET.GetWeitghs()
    WW = Wloaded[1:numV + 1, 1:numH + 1]
    biasV = Wloaded[1:numV + 1, 0]
    biasH = Wloaded[0, 1:numH + 1]

    # Back-propagate the output error in order to find the derivative
    deltaOut = cOut - desiredOTPP.round() # error at the output
    derivt1 = np.zeros((numV, numH))
    derivt2 = np.zeros((numV, numH))
    dbV = np.zeros((1, numV))
    dbH = np.zeros((1, numH))
    nItr = np.size(vVLay, axis=0)

    for n in np.arange(nItr):
        # Back-propagate the error from the very last layer back to
         the input
        ind = nItr - 1 - n
        deltaH = np.dot(deltaOut, WW) * vHLay[ind] * (1 - vHLay[ind])
        for i in np.arange(m):
            # The gradient vector is given by summation of the
             derivative of the error function with respect to the
             weight matrix across different layers
            derivt1 = derivt1 + deltaOut[i].reshape(numV, 1) *
                      vHLay[ind][i].reshape(1, numH)
            derivt2 = derivt2 + vVLay[ind][i].reshape(numV, 1) *
                      deltaH[i].reshape(1, numH)
        dbV = dbV + np.sum(deltaOut, axis=0)
        dbH = dbH + np.sum(deltaH, axis=0)
        deltaOut = np.dot(deltaH, np.transpose(WW)) * vVLay[ind] *
                                                (1 - vVLay[ind])

    # Find return parameters
    dW = derivt1 + derivt2
    dV = dbV/m
    dH = dbH/m
    arrTmp = (desiredOTPP.round() * np.log(cOut) +
     (1 - desiredOTPP.round()) * (np.log(1 - cOut)))/m
    lmbda = 0.00001
    errorLogarithm = - np.sum(arrTmp) +
                     lmbda*(np.sum(np.power(WW, 2)))/2/m
    dW = dW / 2 + lmbda * WW / m

    return [errorLogarithm, -arrTmp, dW, dV, dH]
```

**Table 10: Sub-routine implemented to find a random update direction.**

```
import numpy as np
import matplotlib.pyplot as plt
from rhn_class import RHN
from calculateError_and_Derivative import calculateErrorDer


""" Find a random direction, which introduces perturbation to the
output elements with the maximum contribution to the loss function.
Inputs - desiredOTP: desired output patterns
       - prevQ: previous (total) value of loss function
       - tempError: previous element-wise values of loss function
       - W, biasV, biasH: Current weight matrix and biases
       - deltaK_old, dbv_old, dbh_old: Current moving direction
(gradient)
       - rHNET: The network object
Outputs: q2: New (total) value of the loss function
         tempError: New element-wise values of loss function
         W2, biasV2, biasH2: New weight matrix and biases
         deltaWout, deltaBVout, deltaBHout: New moving directions
"""
#########################################################################

def findRandDirection(desiredOTP, prevQ, tempError, W, biasV, biasH,
deltaK_old, dbv_old, dbh_old, rHNET):
    keepSearching = 1
    numSearch = 20
    ak_max = 1 / 20
    numBackTracking = 6
    ck_min = 0.01
    ak = ak_max
    ck = ck_min * max(1, ak_max/ak)
    alpha = 0.9
    rangeMaxW = 5
    epsilon = 0.01  #

    numV = rHNET.GetNumV()
    numH = rHNET.GetNumH()
    prevError = tempError
    idxListV = np.arange(0, numV, 1)

    # Keep searching for random directions until a decent direction
      is found or a maximum numSearch is achieved
    while keepSearching < numSearch:
        ak = ak_max
        # Call findIndexMax to find the indices which have a
          maximum error at the output
        indMax = findIndexMax(prevError)
        maxNotIndices = idxListV[~indMax]
        maxIndices = idxListV[indMax]
        # Choose a random update direction
         (being +1/-1 with 0.5 probability)
        deltaKnew = np.random.choice([-1, 1], size=(numV, numH))
        dbvNew = np.zeros((1, numV))
        dbhNew = np.zeros((1, numH))
```

```
        # Only the edges connected to  maxIndices remain non-zero
        for ind in maxNotIndices:
            deltaKnew[ind] = 0

        # The search direction, deltaK is set as a combination of the
          previous decent direction (deltaK_old which is
        # provided as input and is possibly the gradient) and the
          randomly chosen update direction deltaKnew.
        # We take deltaK as the update direction and break the loop
          if it sufficiently reduces the loss/error function
        # The weight of the previous direction is reduced as we
          keep searching.
        deltaK = alpha * deltaK_old + (1 - alpha) * deltaKnew
        beta = np.sqrt(alpha)
        dbv = beta * dbv_old + (1 - beta) * dbvNew
        dbh = beta * dbh_old + (1 - beta) * dbhNew

        # As in SPSA, we initially use the random direction to
         approximate the slope of the function
        ckDeltaK = ck * deltaK
        W1 = np.minimum(np.maximum(W + ckDeltaK, -rangeMaxW),
                        rangeMaxW)
        biasV1 = biasV + dbv
        biasH1 = biasH + dbh
        rHNET.setWB(W1, biasV + dbv, biasH + dbh)
        [q1, tempError, deltaWout, deltaBVout, deltaBHout] =
                       calculateErrorDer(desiredOTP, rHNET)

        # Find the reduction of the error for the components with
          the maximum error at the output
        # We may let the total error increases if at least the
          maximum error is reduced (i.e., maxIndReduce < 0)
        # This is to help getting out of a local optimum point
        maxIndReduce = np.maximum(np.sum((np.sum(tempError, axis=0)
                       - np.sum(prevError, axis=0))[maxIndices]),
                                  -10 * epsilon)
        # We quit if the initial random direction sufficiently reduces
          the total error at the output,
        # or at most increases the total error by half of the decrease
          in the maximum error
        if q1 < prevQ - epsilon or (q1 < prevQ - maxIndReduce):
            W2 = W1
            biasV2 = biasV1
            biasH2 = biasH1
            q2 = q1
            break

        # Otherwise,the search direction is adjusted based on the
          slope of function,
        # and then back-tracking line search is used to find an
          appropriate step-size.
        else:
            deltaW = np.multiply(np.minimum(np.abs(np.divide(q1 -
            prevQ, ckDeltaK)), 1), np.sign(np.divide(q1 - prevQ,
                  ckDeltaK)))
            numTries = 0
```

```
            WL = np.minimum(np.maximum(W - ak*deltaW, -rangeMaxW),
                  rangeMaxW)
            biasVL = biasV - ak * dbv
            biasHL = biasH - ak * dbh
            rHNET.setWB(WL, biasV - ak * dbv, biasH - ak * dbh)
            [qL, tempError, deltaWout, deltaBVout, deltaBHout] =
                   calculateErrorDer(desiredOTP, rHNET)
            maxIndReduce = np.maximum(np.sum((np.sum(tempError, axis=0)
             - np.sum(prevError, axis=0))[maxIndices]), -10 * epsilon)

            # The loop for back-tracking line search
            while qL > prevQ - epsilon and (qL > prevQ - maxIndReduce):
                ak = ak / 2
                WL = np.minimum(np.maximum(W - ak * deltaW,
                                -rangeMaxW), rangeMaxW)
                biasVL = biasV - ak * dbv
                biasHL = biasH - ak * dbh
                rHNET.setWB(WL, biasV - ak * dbv, biasH - ak * dbh)
                [qL, tempError, deltaWout, deltaBVout, deltaBHout]
                      = calculateErrorDer(desiredOTP, rHNET)
                maxIndReduce =
                np.maximum(np.sum((np.sum(tempError,axis=0) - np.sum
                  (prevError, axis=0))[maxIndices]), -10 * epsilon)
                numTries = numTries + 1
                if numTries >= numBackTracking:
                    break
            W2 = WL
            biasV2 = biasVL
            biasH2 = biasHL
            q2 = qL
            if qL <prevQ -epsilon
            or (qL < prevQ -epsilon -maxIndReduce)
            or (qL < prevQ + epsilon*(keepSearching - 0.95*numSearch)):
                break
            else:
                keepSearching = keepSearching + 1
                ck = min(max(ck_min * max(1, np.power(ak_max / ak, 2)),
                                          ck * 2), 10 / ak_max)
                alpha = np.maximum(alpha/1.2, 0.1)
    return [q2, tempError, W2, biasV2, biasH2, deltaWout,
                                deltaBVout, deltaBHout]

##################################################################
def findIndexMax(atmp):
    eps = 0.01
    errorMax = np.sum(np.abs(atmp - np.max(atmp)) < eps, axis=0)
    errorMax = (errorMax == np.max(errorMax))
    errorSum = np.abs(np.sum(atmp, axis=0) - np.max(np.sum(atmp,
axis=0))) < eps
    if np.sum(errorSum & errorMax):
        maxIndices = errorSum & errorMax
    else:
        maxIndices = errorMax
    return maxIndices
```