Scientific Research Publishing

American Journal of Computational Mathematics, 2018, 8, 296-313
http://www.scirp.org/journal/ajcm
ISSN Online: 2161-1211
ISSN Print: 2161-1203

Square Neurons, Power Neurons, and Their Learning Algorithms

Ying Liu

Department of Engineering Technology, Savannah State University, Savannah, Georgia
Email: liuy@savannahstate.edu

How to cite this paper: Liu, Y. (2018) Square Neurons, Power Neurons, and Their Learning Algorithms. *American Journal of Computational Mathematics*, **8**, 296-313.
https://doi.org/10.4236/ajcm.2018.84024

Received: September 5, 2018
Accepted: December 4, 2018
Published: December 7, 2018

Abstract

In this paper, we introduce the concepts of square neurons, power neurons, and new learning algorithms based on square neurons, and power neurons. First, we briefly review the basic idea of the Boltzmann Machine, specifically that the invariant distributions of the Boltzmann Machine generate Markov chains. We further review ABM (Attrasoft Boltzmann Machine). Next, we review the θ-transformation and its completeness, *i.e.* any function can be expanded by θ-transformation. The invariant distribution of the ABM is a θ-transformation; therefore, an ABM can simulate any distribution. We review the linear neurons and the associated learning algorithm. We then discuss the problems of the exponential neurons used in ABM, which are unstable, and the problems of the linear neurons, which do not discriminate the wrong answers from the right answers as sharply as the exponential neurons. Finally, we introduce the concept of square neurons and power neurons. We also discuss the advantages of the learning algorithms based on square neurons and power neurons, which have the stability of the linear neurons and the sharp discrimination of the exponential neurons.

Keywords

AI, Boltzmann Machine, Markov Chain, Invariant Distribution, Completeness, Deep Neural Network

1. Introduction

Neural networks and deep learning currently provide the best solutions to many supervised learning problems. In 2006, a publication by Hinton, Osindero, and Teh [1] introduced the idea of a "deep" neural network, which first trains a simple supervised model, and then adds on a new layer on top and trains the parameters for the new layer alone. You keep adding layers and training layers

in this fashion until you have a deep network. Later, this condition of training one layer at a time is removed.

After Hinton's initial attempt of training one layer at a time, Deep Neural Networks train all layers together. Examples include TensorFlow [2], Torch [3], and Theano [4]. Google's TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and also used for machine learning applications, such as neural networks [5]. It is used for both research and production at Google. Torch is an open source machine learning library and a scientific computing framework. Theano is a numerical computation library for Python. The approach using the single training of multiple layers gives advantages to the neural network over other learning algorithms.

In addition to neural network algorithms, there are numerous learning algorithms. We select a few such algorithms below.

Principal Component Analysis [6] [7] is a statistical procedure that uses an orthogonal transformation to convert a set of vectors into a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables.

Sparse coding [8] [9] minimizes the objective:

$$L_{sc} = \|WH - X\|_2^2 + \lambda \|H\|_1$$

where, W is a matrix of transformation, H is a matrix of inputs, and X is a matrix of the outputs. λ implements a trade of between sparsity and reconstruction.

Auto encoders [10]-[15] minimize the objective:

$$L_{ae} = \left\|W\sigma\left(W^{\mathrm{T}}X\right) - X\right\|_2^2$$

where σ is some neural network functions. Note that L_{sc} looks almost like L_{ae} once we set $H = \sigma\left(W^{\mathrm{T}}X\right)$. The difference is that: 1) auto encoders do not encourage sparsity in their general form; 2) an auto encoder uses a model for finding the codes, while sparse coding does so by means of optimization.

K-means clustering [16] [17] [18] [19] is a method of vector quantization which is popular for cluster analysis in data mining. K-means clustering aims to partition n observations into k clusters. Each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into k clusters.

If we limit the learning architecture to one layer, all of these algorithms have some advantages for some applications. The deep learning architectures currently provide the best solutions to many supervised learning problems, because two layers, when "properly" constructed, are better than one layer. One question is the existence of a solution for a given problem. This will often be followed by an effective solution development, *i.e.* an algorithm for a solution. This will often be followed by the stability of the algorithm. This will often be followed by an efficiency study of solutions. Although these theoretical approaches are not necessary for the empirical development of practical algorithms, the theoretical stu-

dies do advance the understanding of the problems. The theoretical studies will prompt new and better algorithm development of practical problems. Along the direction of solution existence, Hornik, Stinchcombe, & White [20] have shown that the multilayer feedforward networks with enough hidden layers are universal approximators. Roux & Bengio [21] have shown the same. Restricted Boltzmann machines are universal approximators of discrete distributions.

Hornik, Stinchcombe, & White [20] establish that the standard multilayer feedforward networks with hidden layers using arbitrary squashing functions are capable of approximating any measurable function from one finite dimensional space to another to any desired degree of accuracy, provided many hidden units are sufficiently available. In this sense, multilayer feedforward networks are a class of universal approximators.

Deep Belief Networks (DBN) are generative neural network models with many layers of hidden explanatory factors, recently introduced by Hinton, Osindero, and Teh, along with a greedy layer-wise unsupervised learning algorithm. The building block of a DBN is a probabilistic model called a Restricted Boltzmann machine (RBM), used to represent one layer of the model. Restricted Boltzmann machines are interesting because inference is easy in them and because they have been successfully used as building blocks for training deeper models. Roux & Bengio [21] proved that adding hidden units yield a strictly improved modeling power, and RBMs are universal approximators of discrete distributions.

An alternative to the direction of "deep layers", higher order is another direction. In our earlier paper [22], we provided yet another proof: Deep Neural Networks are universal approximators. The advantage of this proof is that it will lead to multiple new learning algorithms. In our approach, Deep Neural Networks implement an expansion and this expansion is complete. These two directions are equivalent [22] [23]. There are several learning algorithms characterized by θ-transformation, which are in the direction of higher order, which form a new family of learning algorithms [22] [23]. The conversion between these two directions of deep layers and higher orders is beyond the scope of this paper. The first learning algorithm characterized by higher orders and θ-transformation [24] [25] [26] [27] is ABM [28], which has a problem of stability.

Once we accept that the deep learning architectures currently provide the best solutions, the next question is what is in each layer; in this paper, we intend to fill these layers with the square and power neurons.

In [23], by identifying that the ABM algorithm uses exponential neurons, a second learning algorithm was developed to replace the exponential neurons with linear neurons [23], which solved the stability problem. However, the linear neurons do not discriminate the wrong answers from the right answers as sharply as the exponential neurons. In this paper, we will present a third algorithm after [28] and [23]. We will take the middle ground between the exponential neurons [28] and the linear neurons [23], which has the advantages

of both algorithms [23] [28] and avoids the disadvantages of the both algorithms.

In Section 2, we briefly review how to use probability distributions in a Supervised Learning Problem. In this approach, given an input A, an output B, and a mapping from A to B, one can convert this problem to a probability distribution [29] [30] [31] [32] [33] of (A, B): $p(a,b), a \in A, b \in B$. If an input is $a \in A$ and an output is $b \in B$, then the probability $p(a,b)$ will be higher than 0. One can find a Markov chain [34] such that the equilibrium distribution of this Markov chain, $p(a,b)$, realizes, as faithfully as possible, the given supervised training set.

In Section 3, the Boltzmann machines [29] [30] [31] [32] [33] are briefly reviewed. Our discussion concentrates on the distribution space of the Boltzmann machine rather than the neural aspects. All possible distributions together form a distribution space. All of the distributions, implemented by Boltzmann machines, define a Boltzmann Distribution Space, which is a subset of the distribution space [24] [25] [26] [27]. Given an unknown function, one can find a Boltzmann machine such that the equilibrium distribution of this Boltzmann machine realizes, as faithfully as possible, the unknown function. A natural question is whether such an approximation is possible. The answer is that this approximation is not yet a good approximation.

In Section 4, we review the ABM (Attrasoft Boltzmann Machine) [28] which has an invariant distribution. An ABM is defined by two features: 1) an ABM with n neurons has neural connections up to the n^{th} order; and 2) all of the connections up to n^{th} order are determined by the ABM algorithm [28]. By adding more terms in the invariant distribution compared to the second order Boltzmann Machine, ABM is significantly more powerful to simulate an unknown function. Unlike the Boltzmann Machine, ABM's emphasize higher order connections rather than lower order connections. The Boltzmann Machine (order 0, 1, 2) and the ABM (order n, $n-1$, $n-2$) are at the opposite end of the neuron orders.

In Section 5, we review θ-transformation [24] [25] [26] [27].

In Section 6, we review the completeness of the θ-transformation [24] [25] [26] [27]. The θ-transformation is complete, *i.e.* given a function, one can find a θ-transformation by converting it from the x-coordinate system to the θ-coordinate system.

In Section 7, we discuss how the invariant distribution of an ABM implements a θ-transformation [11] [12] [13] [14], *i.e.* given an unknown function, one can find an ABM such that the equilibrium distribution of this ABM realizes precisely the unknown function. We introduce the exponential neurons; if we keep only lower orders, this will be the standard Boltzmann machine.

In Section 8, we discuss the stability problem of the exponential neurons.

In Section 9, we review linear neurons [23], which solves the stability problem. However, the linear neurons do not discriminate the wrong answers from the right answers as sharply as the exponential neurons.

In Section 10, we review the linear neuron learning algorithms.

In Section 11, we will take the middle ground between the exponential neurons

and the linear neurons, which has the advantages of both algorithms and avoids the disadvantages of the both algorithms. The new contribution of this paper is that we introduce the concept of square neurons and power neurons.

In Section 12, we also discuss the advantages of the two new learning algorithms based on square neurons and power neurons, which has the stability of the linear neurons and the sharp discrimination of the exponential neurons.

In Section 13, we introduce a simple example to demonstrate the improvement of the square neurons and power neurons over linear neurons.

2. Basic Approach

The basic supervised learning [29] problem is: given a training set {*A*, *B*}, where $A=\{a_1,a_2,\cdots\}$ and $B=\{b_1,b_2,\cdots\}$, find a mapping from *A* to *B*. It turns out that if we can reduce this from a discrete problem to a continuous problem, it will be very helpful. The first step is to convert this problem to a probability [29] [30] [32] [33]:

$$p=p(a,b), a\in A, b\in B.$$

If a_1 does not match with b_1, the probability is 0 or close to 0. If a_1 matches with b_1, the probability is higher than 0. This can reduce the problem of inferencing of a mapping from *A* to *B* to inferencing a distribution function.

An irreducible finite Markov chain possesses a stationary distribution [34]. This invariant distribution can be used to simulate an unknown function. It is the invariant distribution of a Markov chain which eventually allows us to prove that the DNN is complete.

3. Boltzmann Machine

A Boltzmann machine [29] [30] [31] [32] [33] is a stochastic neural network in which each neuron has a certain probability to be 1. The probability of a neuron to be 1 is determined by the so called Boltzmann distribution. The collection of the neuron states:

$$x=(x_1,x_2,\cdots,x_n)$$

of a Boltzmann machine is called a configuration. The configuration transition is mathematically described by a Markov chain with 2^n configurations $x\in X$, where *X* is the set of all points, $(x_1,x_2,\cdots,x_n)$. When all of the configurations are connected, it forms a Markov chain. A Markov chain has an invariant distribution [34]. Whatever initial configuration of a Boltzmann starts from, the probability distribution converges over time to the invariant distribution, $p(x)$. The configuration $x\in X$ appears with a relative frequency $p(x)$ over a long period of time.

The Boltzmann machine [29] [30] [31] [32] [33] defines a Markov chain. Each configuration of the Boltzmann machine is a state of the Markov chain. The Boltzmann machine has a stable distribution. Let *T* be the parameter space of a family of Boltzmann machines. An unknown function can be considered as a stable distribution of a Boltzmann machine. Given an unknown distribution, a Boltzmann

machine can be inferred such that its invariant distribution realizes, as faithfully as possible, the given function. Therefore, an unknown function is transformed into a specification of a Boltzmann machine.

More formally, let F be the set of all functions. Let T be the parameter space of a family of Boltzmann machines. Given an unknown $f \in F$, one can find a Boltzmann machine such that the equilibrium distribution of this Boltzmann machine realizes, as faithfully as possible, the unknown function [29] [30] [31] [32] [33]. Therefore, the unknown, f, is encoded into a specification of a Boltzmann machine, $t \in T$. We call the mapping from F to T as a Boltzmann Machine Transformation: $F \to T$ [24] [25] [26] [27].

Let T be the parameter space of a family of Boltzmann machines, and let F_T be the set of all functions that can be inferred by the Boltzmann Machines over T; obviously, F_T is a subset of F. It turns out that F_T is significantly smaller than F and it is not a good approximation for F. The main contribution of the Boltzmann machine is to establish a framework for inferencing a mapping from A to B.

4. Attrasoft Boltzmann Machines (ABM)

The invariant distribution of a Boltzmann machine [29] [30] [31] [32] [33] is:

$$p(x) = b\mathrm{e}^{\sum_{i<j} M_{ij} x_i x_j} \tag{1}$$

If the threshold vector does not vanish, the distributions are:

$$p(x) = b\mathrm{e}^{\sum_{i<j} M_{ij} x_i x_j - \sum T_i x_i} \tag{2}$$

By rearranging the above distribution, we have:

$$p(x) = \mathrm{e}^{c - \sum T_i x_i + \sum_{i<j} M_{ij} x_i x_j}$$

It turns out that the third order Boltzmann machines have the following type of distributions:

$$p(x) = \mathrm{e}^{c - \sum T_i x_i + \sum_{i<j} M_{ij} x_i x_j + \sum_{i<j<k} M_{ijk} x_i x_j x_k} \tag{3}$$

An ABM [24] [25] [26] [27] is an extension of the higher order Boltzmann Machine to the maximum order. An ABM with n neurons has neural connections up to the n^{th} order. All of the connections up to the n^{th} order are determined by the ABM algorithm [28]. By adding additional higher order terms to the invariant distribution, ABM is significantly more powerful to simulate an unknown function.

By adding additional terms, the invariant distribution for an ABM is:

$$p(x) = \mathrm{e}^{H},$$

$$H = \theta_0 + \sum \theta_1^{i_1} x_{i_1} + \sum \theta_2^{i_1 i_2} x_{i_1} x_{i_2} + \sum \theta_3^{i_1 i_2 i_3} x_{i_1} x_{i_2} x_{i_3} + \cdots + \theta_n^{12\cdots n} x_1 x_2 \cdots x_n$$

ABM is significantly more powerful to simulate an unknown function. As more and more terms are added, from the second order terms to the n^{th} order terms, the invariant distribution space will become larger and larger. Like the Boltzmann Machines in the last section, ABM implements a transformation, $F_B \to T$. We hope ultimately that this ABM transformation is complete so that given any

function $f \in F$, we can find an ABM, $t \in T$, such that the equilibrium distribution of this ABM realizes precisely the unknown function. We show that this is exactly the case.

5. θ-Transformation

5.1. Basic Notations

We first introduce some notations used in this paper [24] [25] [26] [27]. There are two different types of coordinate systems: the x-coordinate system and the θ-coordinate system [24] [25] [26] [27]. Each of these two coordinate systems has two representations, x-representation and θ-representation. An N-dimensional vector, p, is:

$$p = (p_0, p_1, \cdots, p_{N-1}),$$

which is the x-representation of p in the x-coordinate systems.

In the x-coordinate system, there are two representations of a vector:

- $\{p_i\}$ in the x-representation, and
- $\left\{p_m^{i_1 i_2 \cdots i_m}\right\}$ in the θ-representation.

In the θ-coordinate system, there are two representations of a vector:

- $\{\theta_i\}$ in the x-representation, and
- $\left\{\theta_m^{i_1 i_2 \cdots i_m}\right\}$ in the θ-representation.

The reason for the two different representations is that the x-representation is natural for the x-coordinate system, and the θ-representation is natural for the θ-coordinate system.

The transformations between $\{p_i\}$ and $\left\{p_m^{i_1 i_2 \cdots i_m}\right\}$, and those between $\{\theta_i\}$ and $\left\{\theta_m^{i_1 i_2 \cdots i_m}\right\}$, are similar. Because of the similarity, in the following, only the transformation between $\{p_i\}$ and $\left\{p_m^{i_1 i_2 \cdots i_m}\right\}$ will be introduced. Let $N = 2^n$ be the number of neurons. An N-dimensional vector, p, is:

$$p = (p_0, p_1, \cdots, p_{N-1}) \tag{4}$$

Consider p_x, because $x \in \{0,1,\cdots,N-1=2^n-1\}$ is the position inside a distribution, then x can be rewritten in the binary form:

$$x = x_n 2^{n-1} + \cdots + x_2 2^1 + x_1 2^0 \tag{5}$$

Some of the coefficients x_i might be zero. In dropping those coefficients which are zero, we write:

$$x = x_{i_1} x_{i_2} \cdots x_{i_m} = 2^{i_m - 1} + \cdots + 2^{i_2 - 1} + 2^{i_1 - 1}. \tag{6}$$

This generates the following transformation:

$$p_m^{i_1 i_2 \cdots i_m} = p_x = p_{2^{i_m-1}+\cdots+2^{i_2-1}+2^{i_1-1}} \tag{7}$$

where

$$1 \le i_1 < i_2 < \cdots < i_m \le n \tag{8}$$

In this θ-representation, a vector p looks like:

$$\left\{p_0, p_1^1, p_1^2, p_1^3, \cdots, p_2^{12}, p_2^{13}, p_2^{23}, \cdots, p_3^{123}, \cdots\right\}$$

The 0-th order term is p_0, the first order terms are: $p_1^1, p_1^2, p_1^3, \cdots, \ldots$ The first few terms in the transformation between $\{p_i\}$ and $\left\{p_m^{i_1 i_2 \cdots i_m}\right\}$ are:

$$\begin{aligned} &p_0 = p_0, \quad p_1^1 = p_1, \quad p_1^2 = p_2, \\ &p_2^{12} = p_3, \quad p_1^3 = p_4, \quad p_2^{13} = p_5, \\ &p_2^{23} = p_6, \quad p_3^{123} = p_7, \quad p_1^4 = p_8, \cdots \end{aligned} \tag{9}$$

The x-representation is the normal representation, and the θ-representation is a form of binary representation.

5.2. θ-Transformation

Denote a distribution by p, which has a x-representation in the x-coordinate system, $p(x)$, and a θ-representation in the θ-coordinate system, $p(\theta)$. When a distribution function, $p(x)$ is transformed from one coordinate system to another, the vectors in both coordinates represent the same abstract vector. When a vector q is transformed from the x-representation $q(x)$ to the θ-representation $q(\theta)$, then $q(\theta)$ is transformed back to $q'(x)$, $q'(x) = q(x)$.

The θ-transformation uses a function F, called a generating function. The function F is required to have the inverse:

$$FG = GF = I, \quad G = F^{-1}. \tag{10}$$

Let p be a vector in the x-coordinate system. As already discussed above, it can be written either as:

$$p(x) = \left(p_0, p_1, \cdots, p_{N-1}\right) \tag{11}$$

or

$$p(x) = \left(p_0; p_1^1, \cdots, p_1^n; p_2^{12}, \cdots, p_2^{n-1,n}; p_3^{123}, \cdots, p_n^{12\cdots n}\right). \tag{12}$$

The θ-transformation transforms a vector from the x-coordinate to the θ-coordinate via a generating function. The components of the vector p in the x-coordinate, $p(x)$, can be converted into components of a vector $p(\theta)$ in the θ-coordinate:

$$p(\theta) = \left(\theta_0; \theta_1^1, \cdots, \theta_1^n; \theta_2^{12}, \cdots, \theta_2^{n-1,n}; \theta_3^{123}, \cdots, \theta_n^{12\cdots n}\right), \tag{13}$$

or

$$p(\theta) = \left(\theta_0, \theta_1, \cdots, \theta_{N-1}\right). \tag{14}$$

Let F be a generating function, which transforms the x-representation of p in the x-coordinate to a θ-representation of p in the θ-coordinate system. The θ-components are determined by the vector $F[p(x)]$ as follows:

$$F\left[p(x)\right] = \theta_0 + \sum \theta_1^{i_1} x_{i_1} + \sum \theta_2^{i_1 i_2} x_{i_1} x_{i_2} + \sum \theta_3^{i_1 i_2 i_3} x_{i_1} x_{i_2} x_{i_3} + \cdots + \theta_n^{12\cdots n} x_1 x_2 \cdots x_n \tag{15}$$

where

$$1 \le i_1 < i_2 < \cdots < i_m \le n \tag{16}$$

Prior to the transformation, $p(x)$ is the x-representation of p in the x-coordinate; after transformation, $F[p(x)]$ is a θ-representation of p in the θ-coordinate system.

There are N components in the x-coordinate and N components in the θ-coordinate. By introducing a new notation X:

$$\begin{aligned} &X_0 = X_0 = 1, \quad X_1^1 = X_1 = x_1, \quad X_1^2 = X_2 = x_2, \quad X_2^{12} = X_3 = x_1 x_2, \\ &X_1^3 = X_4 = x_3, \quad X_2^{13} = X_5 = x_1 x_3, \quad X_2^{23} = X_6 = x_2 x_3, \\ &X_3^{123} = X_7 = x_1 x_2 x_3, \quad X_1^4 = X_8 = x_1 x_2 x_3 x_4, \cdots \end{aligned} \tag{17}$$

then the vector can be written as:

$$F\left[p(x)\right] = \sum \theta_J X_J \tag{18}$$

By using the assumption $GF = I$, we have:

$$p(x) = G\left\{\sum \theta_J X_J\right\} \tag{19}$$

where J denotes the index in either of the two representations in the θ-coordinate system.

The transformation of a vector p from the x-representation, $p(x)$, in the x-coordinate system to a θ-representation, $p(\theta)$, in the θ-coordinate system is called θ-transformation [24] [25] [26] [27].

The θ-transformation is determined by [24] [25] [26] [27]:

$$\begin{aligned} \theta_m^{i_1 i_2 \cdots i_m} = {} & F\left[p_m^{i_1 i_2 \cdots i_m}\right] + F\left[p_{m-2}^{i_1 \cdots i_{m-2}}\right] + \cdots + F\left[p_{m-2}^{i_3 \cdots i_m}\right] + F\left[p_{m-4}^{\cdots}\right] + \cdots \\ & - F\left[p_{m-1}^{i_1 \cdots i_{m-1}}\right] - \cdots - F\left[p_{m-1}^{i_2 \cdots i_m}\right] - F\left[p_{m-3}^{i_1 \cdots i_{m-3}}\right] - \cdots \end{aligned} \tag{20}$$

The inverse of the θ-transformation [24] [25] [26] [27] is:

$$p_m^{i_1 i_2 \cdots i_m} = G\left(\theta_0 + \theta_1^{i_1} + \theta_1^{i_2} + \cdots + \theta_1^{i_m} + \theta_2^{i_1 i_2} + \theta_2^{i_1 i_3} + \cdots + \theta_2^{i_{m-1} i_m} + \cdots + \theta_m^{i_1 i_2 \cdots i_m}\right) \tag{21}$$

6. θ-Transformation Is Complete

Because the θ-transformation is implemented by normal function, $FG = GF = I$, as long as there is no singular points in the transformation, any distribution function can be expanded. If we require $p_i \geq \varepsilon$, which is a predefined small number, then there will be no singular points in the transformation.

7. Exponential Neurons

An ABM with n neurons has neural connections up to the n[th] order. The invariant distribution is:

$$p(x) = \mathrm{e}^H,$$

$$H = \theta_0 + \sum \theta_1^{i_1} x_{i_1} + \sum \theta_2^{i_1 i_2} x_{i_1} x_{i_2} + \sum \theta_3^{i_1 i_2 i_3} x_{i_1} x_{i_2} x_{i_3} + \cdots + \theta_n^{12 \cdots n} x_1 x_2 \cdots x_n.$$

An ABM implements a θ-transformation [24] [25] [26] [27] with:

$$F(y) = \log(y), \quad G(y) = \exp(y).$$

We call the neurons in the ABM algorithm the exponential neurons, because of its exponential generating function. Furthermore, the "connection matrix" element can be calculated as follows [24] [25] [26] [27]:

$$\theta_m^{i_1 i_2 \cdots i_m} = \log \frac{p_m^{i_1 i_2 \cdots i_m} p_{m-2}^{i_1 \cdots i_{m-2}} \cdots p_{m-2}^{i_3 \cdots i_m} p_{m-4}^{\cdots}}{p_{m-1}^{i_1 \cdots i_{m-1}} \cdots p_{m-1}^{i_2 \cdots i_m} p_{m-3}^{i_1 \cdots i_{m-3}} \cdots} \quad (22)$$

The reverse problem is as follows: given an ABM, the invariant distribution can be calculated as follows [24] [25] [26] [27]:

$$p_m^{i_1 i_2 \cdots i_m} = \exp\left(\theta_0 + \theta_1^{i_1} + \theta_1^{i_2} + \cdots + \theta_1^{i_m} + \theta_2^{i_1 i_2} + \theta_2^{i_1 i_3} + \cdots + \theta_2^{i_{m-1} i_m} + \cdots + \theta_m^{i_1 i_2 \cdots i_m}\right) \quad (23)$$

Therefore, an ABM can realize a θ-expansion, which in turn can approximate any distribution. The starting point of the algorithm is a complete expansion; thus, it has the advantage of accuracy [24] [25] [26] [27]. Write the above equation:

$$\begin{aligned} p_m^{i_1 i_2 \cdots i_m} &= \exp\left(\theta_0\right)\exp\left(\theta_1^{i_1}\right)\exp\left(\theta_1^{i_2}\right)\cdots\exp\left(\theta_1^{i_m}\right) \\ &\cdot \exp\left(\theta_2^{i_1 i_2}\right)\exp\left(\theta_2^{i_1 i_3}\right)\cdots\exp\left(\theta_2^{i_{m-1} i_m}\right)\cdots\exp\left(\theta_m^{i_1 i_2 \cdots i_m}\right) \end{aligned}$$

We call the neurons in the ABM algorithm the exponential neurons, because of its exponential generating function. The ABM algorithm uses multiplication expansion, which raises the question of stability. Therefore, we expect to improve this algorithm.

8. Stability of Exponential Neurons

If we take derivative of the expression:

$$\theta_m^{i_1 i_2 \cdots i_m} = \log \frac{p_m^{i_1 i_2 \cdots i_m} p_{m-2}^{i_1 \cdots i_{m-2}} \cdots p_{m-2}^{i_3 \cdots i_m} p_{m-4}^{\cdots}}{p_{m-1}^{i_1 \cdots i_{m-1}} \cdots p_{m-1}^{i_2 \cdots i_m} p_{m-3}^{i_1 \cdots i_{m-3}} \cdots},$$

let

$$p_a \in \left\{p_m^{i_1 i_2 \cdots i_m}\right\},$$
$$\theta_b \in \left\{\theta_m^{i_1 i_2 \cdots i_m}\right\},$$

then depending whether p_a appears in numerator, absent, or denominator, a partial derivative can be

$$\frac{\partial \theta_b}{\partial p_a} = \frac{1}{p_a}, 0, -\frac{1}{p_a}.$$

Because of $1/p_a$, a small change in $p = \left(p_0, p_1, \cdots, p_{N-1}\right)$ can cause a large change in

$$p\left(\theta\right) = \left(\theta_0; \theta_1^1, \cdots, \theta_1^n; \theta_2^{12}, \cdots, \theta_2^{n-1,n}; \theta_3^{123}, \cdots, \theta_n^{12\cdots n}\right).$$

Expand:

$$\mathrm{e}^y = 1 + y + \frac{y^2}{2!} + \cdots$$

As we will argue in the next few sections,

- If we replace $G(y) = \mathrm{e}^y$ with $G(y) = y$, the θ-transformation will be stable, *i.e.* a small ∂p_a will cause a small $\partial \theta_b$;
- If a generating function, $G(y) = y$, can classify a problem correctly, the generating function, $G(y) = y^2$, can discriminate the wrong answers from

the right answers more sharply than $G(y)=y$;

- The generating function, $G(y)=y^{n+1}$, can discriminate the wrong answers from the right answers more sharply than $G(y)=y^n$.

In the next section, we will replace $G(y)=e^y$ with $G(y)=y$. On one hand, this replacement will stabilize the θ-transformation. On the other hand, the linear term does not discriminate the wrong answers from the right answer as sharply as the exponential neurons, because we can view the exponential neurons consisting of the contributions from linear term, square term, cubic term, …

9. Linear Neuron

If we can convert the multiplication expansion to addition expansion, then the performance will be more stable.

Let :

$$G(y)=y,\quad F(y)=y,$$

From section 5, we have:

$$\theta_m^{i_1i_2\cdots i_m}=p_m^{i_1i_2\cdots i_m}+p_{m-2}^{i_1\cdots i_{m-2}}+\cdots+p_{m-2}^{i_3\cdots i_m}+p_{m-4}^{\cdots}-p_{m-1}^{i_1\cdots i_{m-1}}-\cdots-p_{m-1}^{i_2\cdots i_m}-p_{m-3}^{i_1\cdots i_{m-3}}-\cdots$$

$$p_m^{i_1i_2\cdots i_m}=\theta_0+\theta_1^{i_1}+\theta_1^{i_2}+\cdots+\theta_1^{i_m}+\theta_2^{i_1i_2}+\theta_2^{i_1i_3}+\cdots+\theta_2^{i_{m-1}i_m}+\cdots+\theta_m^{i_1i_2\cdots i_m}$$

We call these neurons linear neurons. The new algorithm uses summation in expansion, thus it is more stable compared to exponential neurons. The partial derivatives do not have singular points.

Example: let an ANN have 3 neurons,

$$(x_1,x_2,x_3)$$

and let a distribution be:

$$\{p_0,p_1,p_2,p_3,p_4,p_5,p_6,p_7\},$$

Then,

$$p_0=(\theta_0),\ p_1=(\theta_0+\theta_1),p_2=(\theta_0+\theta_2),\ p_3=(\theta_0+\theta_1+\theta_2+\theta_3),$$
$$p_4=(\theta_0+\theta_4),\ p_5=(\theta_0+\theta_1+\theta_4+\theta_5),\ p_6=(\theta_0+\theta_2+\theta_4+\theta_6),$$
$$p_7=(\theta_0+\theta_1+\theta_2+\theta_3+\theta_4+\theta_5+\theta_6+\theta_7).$$

When the expansion uses addition, it has the advantage of stability. As we will show below, it also has a third advantage of fast training (low time complexity).

10. A Linear Neuron Learning Algorithm

In [23], we introduced the linear neuron learning algorithm. The L_1-distance between two configurations is:

$$d(x',x)=|x_1'-x_1|+|x_2'-x_2|+\cdots$$

For example, $d(111,111)=0$; $d(111,110)=1$.

The linear neuron learning algorithm can be summarized into a single formula:

$$\theta_m^{i_1i_2\cdots i_m}=2\,\hat{}\,\left(D-d\left(x_m^{i_1i_2\cdots i_m},x\right)\right),\ \text{if}\ \ 0\le d\left(x_m^{i_1i_2\cdots i_m},x\right)\le D$$

$$\theta_m^{i_1 i_2 \cdots i_m} = 0, \text{ if } \; d\left(x_m^{i_1 i_2 \cdots i_m}, x\right) > D$$

where $d\left(x_m^{i_1 i_2 \cdots i_m}, x\right)$ is the distance between a neuron configuration, x, and a training neuron configuration, $x_m^{i_1 i_2 \cdots i_m}$, and D is called connection radius. Beyond this radius, all connections are 0.

The linear neuron learning algorithm is [23] [28]:

Step 1. The First Assignment ($d = 0$)

The first step is to assign the first connection matrix element for training vector, $x = x_m^{i_1 i_2 \cdots i_m}$. We will assign:

$$\theta_x = \theta_m^{i_1 i_2 \cdots i_m} = 2^D,$$

while D is the radius of connection space.

Step 2. The Rest of the Assignment

The next step is to assign the rest of the weight:

$$\theta_m^{i_1 i_2 \cdots i_m} = 2 \wedge \left(D - d\left(x_m^{i_1 i_2 \cdots i_m}, x\right)\right), \text{ if } \; 0 \le d\left(x_m^{i_1 i_2 \cdots i_m}, x\right) \le D$$

$$\theta_m^{i_1 i_2 \cdots i_m} = 0, \text{ if } \; d\left(x_m^{i_1 i_2 \cdots i_m}, x\right) > D$$

Step 3. Modification

The algorithm uses bit "1" to represent an input pattern or an output class; so, the input vectors or the output vectors cannot be all 0's; otherwise, these coefficients are 0.

Step 4. Retraining

Repeat the last three steps for all training patterns; if there is an overlap, take the maximum values:

$$\theta_m^{i_1 i_2 \cdots i_m}(t+1) = \max\left\{\theta_m^{i_1 i_2 \cdots i_m}(t), \theta_m^{i_1 i_2 \cdots i_m}\right\}.$$

11. Square Neurons and Power Neurons

The linear neurons do not discriminate the wrong answers from the right answers as sharply as the exponential neurons. We will use a numerical example to demonstrate this in the later section.

To improve the accuracy of the linear neurons, we define the square neurons using the following generating function:

$$F(y) = (y)^{1/2}, \quad G(y) = (y)^2.$$

We define the power neurons using the following generating function:

$$F(y) = (y)^{1/L}, \quad G(y) = (y)^L.$$

For square neurons, we have:

$$\theta_m^{i_1 i_2 \cdots i_m} = \left(p_m^{i_1 i_2 \cdots i_m} + p_{m-2}^{i_1 \cdots i_{m-2}} + \cdots + p_{m-2}^{i_3 \cdots i_m} + p_{m-4}^{\cdots} - p_{m-1}^{i_1 \cdots i_{m-1}} - \cdots - p_{m-1}^{i_2 \cdots i_m} - p_{m-3}^{i_1 \cdots i_{m-3}} - \cdots\right) \wedge (1/2)$$

$$p_m^{i_1 i_2 \cdots i_m} = \left(\theta_0 + \theta_1^{i_1} + \theta_1^{i_2} + \cdots + \theta_1^{i_m} + \theta_2^{i_1 i_2} + \theta_2^{i_1 i_3} + \cdots + \theta_2^{i_{m-1} i_m} + \cdots + \theta_m^{i_1 i_2 \cdots i_m}\right) \wedge 2$$

For power neurons, we have:

$$\theta_m^{i_1 i_2 \cdots i_m} = \left(p_m^{i_1 i_2 \cdots i_m} + p_{m-2}^{i_1 \cdots i_{m-2}} + \cdots + p_{m-2}^{i_3 \cdots i_m} + p_{m-4}^{\cdots} - p_{m-1}^{i_1 \cdots i_{m-1}} - \cdots - p_{m-1}^{i_2 \cdots i_m} - p_{m-3}^{i_1 \cdots i_{m-3}} - \cdots\right) \wedge (1/L)$$

$$p_m^{i_1 i_2 \cdots i_m} = \left(\theta_0 + \theta_1^{i_1} + \theta_1^{i_2} + \cdots + \theta_1^{i_m} + \theta_2^{i_1 i_2} + \theta_2^{i_1 i_3} + \cdots + \theta_2^{i_{m-1} i_m} + \cdots + \theta_m^{i_1 i_2 \cdots i_m}\right) \wedge L$$

12. Square Neurons and Power Neurons Algorithms Are Better

The square neuron learning algorithm is similar to the linear neuron learning algorithm except for the linear neurons:

$$p_m^{i_1 i_2 \cdots i_m} = \left(\theta_0 + \theta_1^{i_1} + \theta_1^{i_2} + \cdots + \theta_1^{i_m} + \theta_2^{i_1 i_2} + \theta_2^{i_1 i_3} + \cdots + \theta_2^{i_{m-1} i_m} + \cdots + \theta_m^{i_1 i_2 \cdots i_m}\right) \wedge 1$$

And for the square neurons:

$$p_m^{i_1 i_2 \cdots i_m} = \left(\theta_0 + \theta_1^{i_1} + \theta_1^{i_2} + \cdots + \theta_1^{i_m} + \theta_2^{i_1 i_2} + \theta_2^{i_1 i_3} + \cdots + \theta_2^{i_{m-1} i_m} + \cdots + \theta_m^{i_1 i_2 \cdots i_m}\right) \wedge 2$$

If the linear neuron can classify a problem correctly, then the square neurons will do better. We will not formally prove this, but we will use a simple example to show the point.

Example. Assume a linear neuron distribution is (1, 2, 3, 4)/10, then based the above expressions, the square neuron distribution is (1^2, 2^2, 3^2, 4^2)/30. The largest probability is increased from 4/10 to 16/30.

13. An Example

In this section, we will first use the linear neuron algorithm [23]; then we will use the square and power neuron algorithms. The example is to identify simple digits in **Figure 1** [1] [24] [25] [26] [27] [29]. Each digit is converted into 7 bits: 0, 1, …, 6. **Figure 2** shows the bit location.

The 10 input vectors for digits in **Figure 2** have 7 bits:

$$I_0 = (1,1,1,0,1,1,1),$$

$$I_1 = (0,0,1,0,0,1,0),$$

$$I_2 = (1,0,1,0,1,0,1),$$

…

where I_0 is an image of "0". The 10 output vectors for digits in **Figure 2** have 10 bits:

$$O_0 = (1,0,0,0,0,0,0,0,0,0),$$

$$O_1 = (0,1,0,0,0,0,0,0,0,0),$$

$$O_2 = (0,0,1,0,0,0,0,0,0,0),$$

…

where O_0 is a classification. The 10 training vectors have 17 bits:

$$T_0 = (I_0, O_0) = ((1,1,1,0,1,1,1),(1,0,0,0,0,0,0,0,0,0)),$$

$$T_1 = (I_1, O_1) = ((0,0,1,0,0,1,0),(0,1,0,0,0,0,0,0,0,0)),$$

…

We will set the radius: $D = 2$; the possible elements are: $2^{D-d} = 4$, 2, 1, and 0.

We will work out a few examples. As the first example, we rewrite T_1 as:

$$T_1 = (I_1, O_1) = x_3^{259} = ((0,0,1,0,0,1,0),(0,1,0,0,0,0,0,0,0,0)).$$

The first connection element (0-distance) is: $\theta_3^{259} = 4$. There are two coefficients for $d = 1$: $\theta_2^{59} = \theta_2^{29} = 2$. T_1 generates 3 coefficients.

Figure 1. An example.

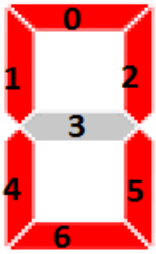

Figure 2. Input bit assignments.

As the second example, we rewrite T_7 as:

$$x_4^{1,2,5,14} = \left((1,0,1,0,0,1,0),(0,0,0,0,0,0,0,1,0,0)\right).$$

The first connection element (0-distance) is: $\theta_4^{1,2,5,14} = 4$. There are 3 coefficients for $d = 1$: $\theta_3^{1,2,14} = \theta_3^{1,5,14} = \theta_3^{2,5,14} = 2$. There are 3 coefficients for $d = 2$: $\theta_2^{1,14} = \theta_2^{2,14} = \theta_2^{5,14} = 1$. T_7 generates 7 coefficients.

As the last example, we rewrite T_4 as:

$$x_5^{1,2,3,5,11} = \left((0,1,1,1,0,1,0),(0,0,0,0,1,0,0,0,0,0)\right),$$

the first connection element (0-distance) is: $\theta_5^{1,2,3,5,11} = 4$. There are 4 coefficients for $d = 1$: $\theta_4^{1,2,3,11} = \theta_4^{1,2,5,11} = \theta_4^{1,3,5,11} = \theta_4^{2,3,5,11} = 2$. There are 6 coefficients for $d = 2$: $\theta_3^{1,2,11} = \theta_2^{1,3,11} = 1$. T_4 generates 11 coefficients.

After training the linear neuron algorithm with $\{T_0, T_1, \cdots, T_9\}$, all of the connection coefficients, $\theta_m^{i_1 i_2 \cdots i_m}$, are calculated. Section 9 provides the formula to calculate the probability of each (input, output) pair. For example, the probability is p_3^{259}, if the input is "1" and the output is in class 1; the probability is p_3^{258}, if the input is "1" and the output is in class 0; the probability is $p_3^{2,5,10}$, if the input is "1" and the output is in class 2; …

The character recognition results [23] are given in Table 1, where the first column is input, then the next 10 columns are output. The output probability is not normalized in Table 1. The relative probability for (input = 0, output = 0) is 31; those for (input = 0, output = 1) are 8; those for (input = 0, output = 2) are 6; …. So if the input is digit 0, the output is identified as 0. In this problem, the output is a single identification, so the largest weight determines the digit classification. In each case, all input digits are classified correctly.

The worst case is input = 8, see Table 2. Using the largest probability as a classification, if input = 8, then output = 8, which is a correct classification. But the 16% probability for (input = 8, output = 8) is too low compared to the next one, 12.7% for (input = 8, output = 9), or (input = 8, output = 6), or (input = 8, output = 0). Some improvements must be made.

This is the main reason for the square neurons and power neurons, which will improve all digits in Table 1. The character recognition results for the square neurons simply square every number in Table 1, see Table 3. In the following, we will only study the worst case of Table 2.

For the square neuron algorithm, the results are in Table 4. Now the probability of the correct output in the worst case is increased from 16% to 22.7%.

For the power neuron algorithm with $L = 4$, the results are in Table 5. Now the probability of the correct output in the worst case is increased from 16% to 37%. This can be further improved by increasing L.

To summarize, the square and power neurons add leverage to the linear neurons performance. The square neuron and power neurons have sharply increased the discrimination of the linear neurons between the correct answer and wrong answer.

Table 1. The classifications from the linear neuron algorithm without normalization.

Input	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
0	31	8	6	6	5	6	7	13	8	7
1	0	8	0	0	1	0	0	4	0	0
2	1	2	24	6	1	1	1	4	1	1
3	1	8	6	24	5	6	1	13	1	7
4	0	8	0	1	18	1	0	4	0	1
5	1	2	1	6	5	24	7	4	1	7
6	7	2	6	6	5	24	31	4	8	7
7	0	8	0	1	1	0	0	13	0	0
8	**31**	**8**	**24**	**24**	**18**	**24**	**31**	**13**	**39**	**31**
9	7	8	6	24	18	24	7	13	8	31

Table 2. The classifications of "8" from the linear neuron algorithm.

Input	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
8	**31**	**8**	**24**	**24**	**18**	**24**	**31**	**13**	**39**	**31**
8	0.1276	0.03292	0.09877	0.09877	0.07407	0.09877	0.12757	0.0535	0.16049	0.1276

Table 3. The classifications from the square neuron algorithm without normalization.

Input	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
0	961	64	36	36	25	36	49	169	64	49
1	0	64	0	0	1	0	0	16	0	0
2	1	4	576	36	1	1	1	16	1	1
3	1	64	36	576	25	36	1	169	1	49
4	0	64	0	1	324	1	0	16	0	1
5	1	4	1	36	25	576	49	16	1	49
6	49	4	36	36	25	576	961	16	64	49
7	0	64	0	1	1	0	0	169	0	0
8	961	64	576	576	324	576	961	169	1521	961
9	49	64	36	576	324	576	49	169	64	961

Table 4. The square neuron algorithm.

Input	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
8	**961**	**64**	**576**	**576**	**324**	**576**	**961**	**169**	**1521**	**961**
	0.1437	0.00957	0.08611	0.08611	0.04844	0.08611	0.14367	0.02527	0.22739	0.1437

Table 5. The power (L = 4) neuron algorithm.

Input	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
8	**0.14**	**0.00**	**0.05**	**0.05**	**0.01**	**0.05**	**0.14**	**0.00**	**0.37**	**0.14**

14. Conclusion

In conclusion, we have introduced two new learning algorithms: the square neuron learning algorithm and the power neuron learning algorithm, which are superior to the earlier ABM algorithm [10] and the linear neuron algorithm [15]. The reason for this improvement is that the ABM has a problem of stability and the linear neuron algorithm has a problem of discriminating wrong answers. In this paper, we have introduced the concepts of square neurons and power neurons. We have shown that these two new learning algorithms, based on square neurons and power neurons, have advantages over both the ABM learning algorithm and the linear neuron algorithm.

Acknowledgements

I would like to thank Gina Porter for proof reading of this paper.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Hinton, G.E., Osindero, S. and Teh, Y. (2006) A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, **18**, 1527-1554. https://doi.org/10.1162/neco.2006.18.7.1527

[2] TensorFlow. https://www.tensorflow.org/

[3] Torch. http://torch.ch/

[4] Theano. http://deeplearning.net/software/theano/introduction.html

[5] Byrne, W. (1992) Alternating Minimization and Boltzmann Machine Learning. *IEEE Transactions on Neural Networks*, **3**, 612-620. https://doi.org/10.1109/72.143375

[6] Jolliffe, I.T. (2002) Principal Component Analysis, Series: Springer Series in Statistics. 2nd Edition, Springer, New York, 487 p.

[7] Abdi, H. and Williams, L.J. (2010) Principal Component Analysis. *Wiley Interdisciplinary Reviews*: *Computational Statistics*, **2**, 433-459.

[8] Olshausen, B.A. (1996) Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images. *Nature*, **381**, 607-609.

https://doi.org/10.1038/381607a0

[9] Gupta, N. and Stopfer, M. (2014) A Temporal Channel for Information in Sparse Sensory Coding. *Current Biology*, **24**, 2247-2256. https://doi.org/10.1016/j.cub.2014.08.021

[10] Bengio, Y. (2009) Learning Deep Architectures for AI (PDF). *Foundations and Trends in Machine Learning*, **2**, No. 1. https://doi.org/10.1561/2200000006

[11] Liou, C.-Y., Huang, J.-C. and Yang, W.-C. (2008) Modeling Word Perception Using the Elman Network. *Neurocomputing*, **71**, 3150-3157. https://doi.org/10.1016/j.neucom.2008.04.030

[12] Liou, C.-Y., Cheng, C.-W., Liou, J.-W. and Liou, D.-R. (2014) Autoencoder for Words. *Neurocomputing*, **139**, 84-96.

[13] Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986) Learning Internal Representations by Error Propagation.

[14] Bourlard, H. and Kamp, Y. (1988) Auto-Association by Multilayer Perceptrons and Singular Value Decomposition. *Biological Cybernetics*, **59**, 291-294. https://doi.org/10.1007/BF00332918

[15] Hinton and Salakhutdinov. (2006) Reducing the Dimensionality of Data with Neural Networks.

[16] MacQueen, J.B. (1967) Some Methods for Classification and Analysis of Multivariate Observations. *Proceedings of* 5*th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, Vol. 1, 281-297.

[17] Steinhaus, H. (1957) Sur la division des corps matériels en parties. *Bulletin L'Académie Polonaise des Science*, **4**, 801-804. (In French)

[18] Lloyd, S.P. (1982) Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, **28**, 129-137. https://doi.org/10.1109/TIT.1982.1056489

[19] Forgy, E.W. (1965) Cluster Analysis of Multivariate Data: Efficiency versus Interpretability of Classifications. *Biometrics*, **21**, 768-769.

[20] Hornik, K., Stinchcombe, M. and White, H. (1989) Multilayer Feedforward Networks Are Universal Approximators. *Neural Networks*, **2**, 359-366. https://doi.org/10.1016/0893-6080(89)90020-8

[21] Le Roux, N. and Bengio, Y. (2008) Representational Power of Restricted Boltzmann Machines and Deep Belief Networks. *Neural Computation*, **20**, 1631-1649. https://doi.org/10.1162/neco.2008.04-07-510

[22] Liu, Y. and Wang, S. (2018) Completeness Problem of the Deep Neural Networks. *American Journal of Computational Mathematics*, **8**, 184-196. https://doi.org/10.4236/ajcm.2018.82014

[23] Liu, Y. (2018) Identity Neurons and Their Learning Algorithms. *Journal of Computer Science and Information Technology*.

[24] Liu, Y. (1993) Image Compression Using Boltzmann Machines. *Proceedings of SPIE*, **2032**, 103-117. https://doi.org/10.1117/12.162027

[25] Liu, Y. (1995) Boltzmann Machine for Image Block Coding. *Proceedings of SPIE*, **2424**, 434-447. https://doi.org/10.1117/12.205245

[26] Liu, Y. (1997) Character and Image Recognition and Image Retrieval Using the Boltzmann Machine. *Proceedings of SPIE*, **3077**, 706-715. https://doi.org/10.1117/12.271533

[27] Liu, Y. (1993) Two New Classes of Boltzmann Machines. *Proceedings of SPIE*, **1966**, 162-175. https://doi.org/10.1117/12.152648

[28] Liu, Y. (2002) US Patent No. 7,773,800. http://www.google.com/patents/US7773800

[29] Sanches-Sinencio, E. and Lau, C. (1992) Artificial Neural Networks. IEEE Press.

[30] Amari, S., Kurata, K. and Nagaoka, H. (1992) Information Geometry of Boltzmann Machine. *IEEE Transactions on Neural Networks*, **3**, 260-271. https://doi.org/10.1109/72.125867

[31] Dagli, C.H. (1995) Intelligent Engineering Systems through Artificial Neural Networks. ASME Press, Vol. 5, 757-850.

[32] Anderson, N.H. and Titterington, D.M. (1995) Beyond the Binary Boltzmann Machine. *IEEE Transactions on Neural Networks*, **6**, 1229-1236. https://doi.org/10.1109/72.410364

[33] Lin, C.T. and Lee, C.S.G. (1995) A Multi-Valued Boltzmann Machine. *IEEE Transactions on Systems*, *Man and Cybernetics*, **25**, 660-668.

[34] Feller, W. (1968) An Introduction to Probability Theory and Its Application. John Wiley and Sons, Hoboken.