Hybrid Deep Learning Networks Based on Self-Organization and their Applications

Hybrid Deep Learning Networks Based on Self-Organization and their Applications

By

Yevgeniy Bodyanskiy, Yuriy Zaychenko and Galib Hamidov

Cambridge Scholars Publishing

Hybrid Deep Learning Networks Based on
Self-Organization and their Applications

By Yevgeniy Bodyanskiy, Yuriy Zaychenko
and Galib Hamidov

This book first published 2024

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-1-0364-1431-3
ISBN (Ebook): 978-1-0364-1432-0

CONTENTS

List of Abbreviations: vi

Preface vii

Chapter One 1
Deep Neural Networks: Structure, Properties, Deep Learning Algorithms, and Regularization

Chapter Two 27
Hybrid GMDH Dl-Fuzzy Networks, Architecture, Training Algorithms and Optimization

Chapter Three 37
Investigations of GMDH-Neo-Fuzzy Neural Networks and Structure Optimization in Forecasting Problems

Chapter Four 82
Experimental Investigations of Hybrid GMDH-Fuzzy Networks in Forecasting Problems

References 115

Addendum 125
GMDH General Characteristics and Main Principles

LIST OF ABBREVIATIONS

Adagrad – Adaptive Gradient Descent
Adam – Adaptive Moment Estimator
ANN – Artificial Neural Network
Bagging – Bootstrap Aggregating
CD – Contrastive Divergence
CNFN – Cascade Neuro-fuzzy Network
CNFNN – Cascade Neo-fuzzy Neural Network
DL – Deep Learning
EBM – Energy-based Models
ENFN – Extended Neo-fuzzy Neuron
ENS_i – Extended Nonlinear Synapse
FNN – Fuzzy Neural Network
GMDH – Group Method of Data Handling
HSCI – Hybrid System of Computational Intelligence
LSM – Least Squares Method
MSE – Mean Square Error
MAE – Mean Absolute Error
MAPE – Mean Absolute Percentage Error
NFN – Neo-fuzzy Network
PCA – Principal Components Algorithm
RBM – Restricted Boltzmann Machine
SAE – Stacked Autoencoder
SRBM – Stacked Restricted Boltzmann Machine

PREFACE

This book is devoted to the problem of the development, analysis and investigation of a new class of deep learning networks – so-called hybrid neuro-fuzzy networks based on the principle of self-organization. This class of networks was suggested and developed by the joint research of the scientific team of the Institute for Applied System Analysis (IASA) Igor Sikorsky Kyiv Polytechnic Institute headed by Professor Yuriy Zaychenko and the team at the Kharkiv National University of Radio Electronics (NURE) headed by Professor Yevgeniy Bodyanskiy. This monograph sums up the results of more than ten years of intensive scientific research that have been presented at numerous international scientific conferences and published in many articles but never in books. This is the first book in which our main results in the field of hybrid DL fuzzy networks based on the GMDH have been presented and discussed.

The first chapter is devoted to a review of the architecture, training algorithms and applications of conventional DL networks. First, the structure, properties and algorithms of encoders and decoders, and restricted Boltzmann machines (RBMs), are considered and analyzed. The algorithm of contrasting divergence for training RBMs is studied and analyzed. The structure of stacked autoencoders and RBMs is presented and the training algorithms are considered. Two main parts of deep learning – transfer learning and fine tuning – are considered and analyzed. The main deep learning algorithms used for loss function optimization – stochastic gradient descent, adaptive gradient descent (Adagrad), the RMSprop algorithm and the Adam (adaptive moment estimator) algorithm – are described in detail and their properties are analyzed. Much attention is paid to the problem of regularization of deep learning. The main approaches and methods of regularization are considered: the L_p-regularization of linear regression, early stopping, dropout and bagging (the ensemble method) and their properties are discussed. At the end of the chapter, an analysis of the advantages and drawbacks of conventional DL networks is performed. It's noted that in DL networks only connection weights are adjusted, not the structure. The efficient method of DL network structure optimization has not been developed up till now.

An adequate method for solving this problem is based on the principle of self-organization and connected with a new class of DL networks –

hybrid fuzzy networks based on GMDH. The theories of these networks, the methods of structure optimization/investigations of their properties and numerous applications are presented in the following chapters.

In the second chapter a hybrid neuro-fuzzy network based on self-organization is considered and investigated. Elementary fuzzy Wang-Mendel neural networks with two inputs are used as the nodes of this network. Their important property is that it's necessary to train not only the weights but also the parameters of membership functions.

An algorithm for synthesizing the structure of a hybrid neural network has been developed based on the online GMDH self-organization method. The advantage of this algorithm is that the synthesis of the network structure occurs simultaneously with its training, layer by layer, until the stop condition is met.

An experimental exploration of the proposed hybrid GMDH neuro-fuzzy network was carried out during the tasks of predicting the share prices of the Microsoft Corporation and the RTS Index, as a result of which the optimal values of the network parameters were found: the number of inputs and the number of fuzzifiers, as well as the optimal network structure for this task.

Comparative forecasting experiments were also carried out with the GMDH method and the full cascaded neo-fuzzy network, which showed that the hybrid neural network has a higher accuracy in predicting stock prices and stock indices in comparison with alternative forecasting methods.

As a result of the experiments, the following conclusions were drawn:

- Variation in the number of inputs in a hybrid neuro-fuzzy network affects the quality of the forecast: with an increase in the number of inputs, the MAPE criterion first falls, reaches a minimum, and then begins to grow;
- The application of the proposed method for the synthesis of the structure and training of the hybrid network allows a reduction in the training time and an increase in the accuracy of prediction in comparison with the cascade neural network. In general, the proposed method for training a hybrid network based on the GMDH has the following advantages;
- There are no problems of deep learning algorithms: the decay or explosion of the gradient due to the proposed learning algorithm.

Due to the use of network nodes with a small number of adjustable parameters, the total training time is significantly reduced, which is extremely important for deep learning networks with many neuron layers.

In the third chapter, another new class of deep learning networks – the

hybrid GMDH neo-fuzzy network – is considered and investigated in forecasting the Google share prices, the NASDAQ Index and the Producer Price Index. This class of hybrid networks differs from neuro-fuzzy networks in that a node of the hybrid network is used for the neo-fuzzy neuron with two inputs. The training of this node refers only to the connection weights. This enables computational operations for training and training time to be cut as compared to neuro-fuzzy networks.

The optimal parameters of the hybrid neo-fuzzy network were found during experimentation: the number of inputs, training/test sample ratio and type of membership function. The optimal structure of the hybrid neo-fuzzy network was generated using the GMDH method.

Comparison experiments of the hybrid neo-fuzzy network with alternative methods – the GMDH and neo-fuzzy cascade network – were performed. These showed that the suggested hybrid network has a higher forecasting accuracy.

The hybrid GMDH network with extended neo-fuzzy neurons as nodes is described and investigated. The system architecture allows the model structure to be constructed in real time due to the proposed neo-fuzzy node-neuron synaptic weights being adjusted. The main property of the proposed architecture and its learning algorithm is the ability to work with small training samples. The hybrid GMDH neo-fuzzy network with extended neo-fuzzy neurons was applied for the recognition of emotions on human faces. The experimental results are presented and discussed.

The architecture and learning algorithms of the hybrid computational intelligence system which is built on the basis of the group method of the data handling (GMDH) method and bagging approach, are proposed for the first time. The process of increasing the number of stacks is based on GMDH principles until the desired accuracy of the final results is achieved. The proposed system does not require large volumes of training samples, it provides online work and automatically determines the number of its layers – stacks in the learning process. Experimental investigations of the suggested system and the problem of short-term and middle-term forecasting of the Dow Jones Index were carried out and these confirmed the efficiency of proposed bagging based on the GMDH.

The fourth chapter is devoted to the numerous experimental investigations of a new generation of deep learning networks – hybrid GMDH – neo-fuzzy networks.

Experimental investigations of the hybrid networks were carried out and compared with conventional DL networks. The problem of forecasting the Dow Jones and NASDAQ Indexes with the application of hybrid neo-fuzzy networks was considered, investigated and compared with the FNN ANFIS

at the different forecasting intervals: one month, one week and one day.

The optimal parameters of hybrid neo-fuzzy networks and sets of informative features for forecasting problems were found. The experimental results have shown that the forecasting accuracy of hybrid neo-fuzzy networks is much better than for the FNN ANFIS.

The training time is the lowest for a hybrid neo-fuzzy network as compared with alternative models – the hybrid neuro-fuzzy network and the ANFIS network.

The problem of forecasting the financial market at different forecasting intervals was also considered (short-term and middle-term forecasting). For its solution it was suggested to apply the GMDH and hybrid deep learning (DL) networks based on the GMDH and ARIMA.

The experimental investigations were performed and there was a problem of forecasting the Apple share prices during the period from January to August 2022 and the Dow Jones Index Average.

The optimization of parameters of the hybrid DL networks and ARIMA was performed during the experiments. The optimal structure of the hybrid DL network was constructed using the GMDH method.

The experimental investigations of the optimized ARIMA, GMDH and hybrid DL networks were carried out at different forecasting intervals. Their results were presented and their accuracy was compared.

In the results, it was established that the application of the GMDH has the best forecasting accuracy for short-term forecasting at the interval of 1-3 days while the hybrid DL neo-fuzzy network is the best for middle-term forecasting – 7-20 days. Overall, the intelligent methods based on the GMDH outperform the statistical method ARIMA in respect of the problems of short-term and middle-term forecasting at stock exchanges.

This book is oriented towards scientists and researchers who are engaged in the development, design and application of neural networks, especially deep learning NN in forecasting, pattern recognition, medical image processing and express diagnostics. It will be useful for bachelor's and master's students specializing in computer science and artificial intelligence, and for the wider community interested in advanced neural networks and their possible prospective applications.

CHAPTER ONE

DEEP NEURAL NETWORKS: STRUCTURE, PROPERTIES, DEEP LEARNING ALGORITHMS, AND REGULARIZATION

Introduction

One of the modern and efficient tools for big data analytics is deep neural networks [52-54]. At the present time, the theory and practice of machine learning are undergoing a profound transformation due to the widespread success of deep learning networks which represent the third generation of neural networks. In contrast to classic neuron networks (the second generation), 80-90 years of 20th-century new training paradigms have allowed us to rid ourselves of some problems which hinder the successful application of traditional neural networks. Neural networks trained with deep learning algorithms not only overcame by accuracy the best alternative approaches but in some cases displayed an understanding of the sense of input information (image recognition, text analysis and other problems).

The most successful industrial systems of computer vision and speech recognition are built on deep networks, and giants of the IT industry such as Apple, Google and Facebook created large research teams dealing with deep learning. The term "deep network" means a big neural network with many hidden layers of neurons [54, 58]. Deep learning represents a set of methods and techniques for training complex neural networks (NN) with many layers. For such networks, traditional machine learning algorithms developed for conventional NN had become inadequate due to some drawbacks, in particular the problem of decay and the explosion of the gradient in the back propagation algorithm [53, 59]. Therefore, the large dimensions of modern neural networks with applications for 3D image recognition and automatic speech recognition demanded the development of new efficient training methods called deep learning.

But the most serious drawback of deep learning networks is the problem of determining their proper structure and how to choose an adequate number of their layers.

Autoassociators. Autoencoders

The implementation of deep learning has led to the development of the special learning structure based on the application of so-called autoassociators [87, 89].

The main task of the autoassociator is to obtain at the output the most accurate mapping of the input vector (pattern).

The first autoassociator (AA) was the neo-cognitron suggested by Fukushima. Its schema is presented in Fig. 1.1 [67].

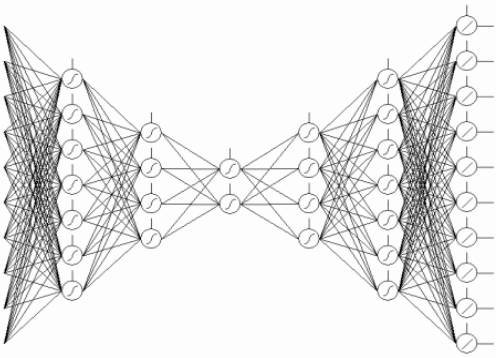


Figure 1.1. The neo-cognitron of Fukushima

There are two types of AA: generating and synthesizing ones.

For the first type, the restricted Boltzmann machine (RBM) is used, and for the second type, autoencoders (AE) are used.

Autoencoder

The autoencoder was one of the first deep learning algorithms. It's an algorithm of non-supervised learning whose output vector equals the input vector [87]. One of the most widespread autoencoder architectures is the feedforward neural network containing input, hidden and output layers.

Unlike the perceptron output, the autoencoder layer has the same number of neurons as the input layer. The data at the input layer are compressed and restored so the hidden features are retrieved.

The goal of an autoencoder is to ensure that the NN output is maximally close to the input vector. For a non-trivial solution to this problem the special constraints are set by the network topology:

1) the number of neurons in the hidden layer should be less than the number of input neurons; and
2) the number of non-active neurons in the hidden layer should significantly exceed the number of active neurons.

The first constraint enables the compression of data while transferring the input signal to the network output. Such compression is possible if there are hidden interconnections in the data, and a correlation among features. The second constraint – the demand for a great number of non-active neurons – allows us to obtain non-trivial results even when the number of neurons in a hidden layer exceeds the dimensionality of the input data. In other words, the goal of an autoencoder is to obtain the most significant features.

Let us consider that a neuron is active if its activation is close to one, and non-active if its activation is close to zero. These constraints force an autoencoder to search for correlations and generalization in the input data and perform its compression.

In this way, the network automatically learns to extract from the input data general features which are encoded in network weights. It's necessary that the mean value of the transfer function of each hidden neuron gets the maximal value close to a given sparsity parameter of about s=0.05. To achieve this, the sparsity parameter *p* was introduced in each neuron of the hidden layer:

$$\widehat{\rho_j} = \frac{1}{m}\sum_{i=1}^{m} \left[a_j^{(2)}(x^{(i)})\right] \tag{1.1}$$

It's necessary that the mean value of the transfer function of each hidden neuron takes the closest value to *p*:

$$\widehat{\rho_j} = \rho \tag{1.2}$$

Let's introduce a penalty function:

$$S = \sum_{j=1}^{S_2} KL(\rho|\widehat{\rho_j}) \tag{1.3}$$

where,

$$KL\left(\widehat{\rho_j}\right) = \rho log\frac{\rho}{\widehat{\rho_j}} + (1-\rho) log\frac{1-\rho}{1-\widehat{\rho_j}} \tag{1.4}$$

A remarkable property of the penalty function is its derivative:

$$\frac{\partial KL(\widehat{\rho_j})}{\partial \rho_j} = -\frac{\rho}{\widehat{\rho_j}} + \frac{1-\rho}{1-\widehat{\rho_j}} \tag{1.5}$$

The example of the encoder is presented in Fig. 1.2. The autoencoder tends to build the function **h(x)=x**. In other words, it tends to find such an approximation of this function that the neural network output is equal to the input vector. This makes the solution of this problem non-trivial; the number of hidden layer neurons should be less than the dimensionality of the input data (see Fig. 1.2).

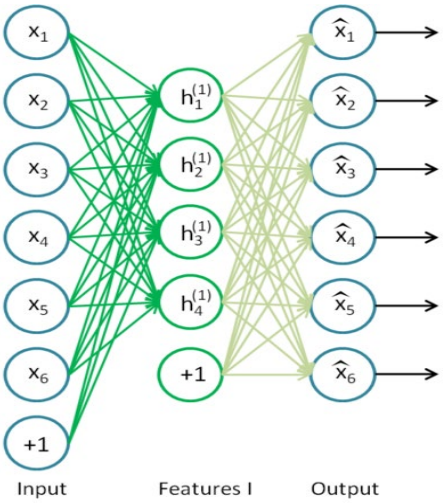


Figure 1.2. Architecture of the autoencoder

This allows us to obtain data compression by the transfer of the input signal to the output.

For example, if the input vector presents a set of brightness levels of an image of 10 x 10 pixels (100 features overall), the number of hidden layer neurons is 50, and the network is forced to learn to compress an image.

Really, the demand h(x)=x means that on the base of the activation levels of 50 neurons, the output layer should restore 100 pixels of the initial image. Such compression is possible if there are hidden interconnections, correlation in features, and in general a certain structure in the data. In this way, the functions of the autoencoder very much resemble the principal components algorithm (PCA) in the sense of cutting the dimensionality of

the input data.

Later, as the sparsity idea was discussed, the so-called sparse autoencoder appeared and gained a wide application [87, 89]. The sparse autoencoder is an autoencoder whose number of hidden neurons is much greater than the dimensionality of the input vector. Sparse activation means that the number of non-active neurons in the hidden layer significantly exceeds the number of active ones. When describing sparsity informally, a neuron is considered active if its transfer function is about 1. If the sigmoid transfer function is used then for a non-active neuron its value should be close to 0 (for tanh – close to -1).

There is a variant of autoencoder called a denoising autoencoder [63, 90]. It's the same autoencoder but its training is specific. During training, randomly distorted data (several input values are changed to 0) are input. Non-distorted values are shown for comparison. In this way autoencoders are compelled to restore distorted input data (Fig. 1.3).

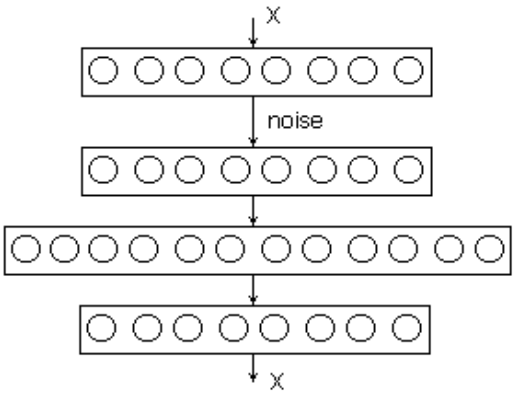


Figure 1.3. Structure of denoising autoencoder

Artificial feed-forward neural networks (ANNs) with large numbers of layers are badly trained by conventional methods; but these are good for ANNs with a small number of hidden layers due to the problem of the vanishing gradient [54]. The farther its layer from the output, the lower are the values of the gradient norm.

This problem may be solved by correctly chosen initial weights. In this case, these do not need to change significantly during the training process.

Boltzmann Machines (BM)

Energetic Models

Boltzmann machines represent a special form of the log-linear Markov field (MRF), i.e., its energy function is linear by parameters. Therefore, let's first consider energy-based models (EBMs). EBMs connect scalar energy with each configuration of the variables. The training corresponds to a modification of the energy function so that its form obtains the desired properties. For example, we would like the desired configurations to have low energy. Probabilistic models that are energy-based thus determine the probability distribution [70, 72]:

$$p(x) = \frac{e^{-E(x)}}{z} \tag{1.6}$$

Normalizing the multiplier Z is called the statistical sum by analogy with the physical systems

$$Z = \sum_x \quad e^{-E(x)} \tag{1.7}$$

Energy-based models may be explored by using the stochastic gradient descent at the empirical negative-logarithmic probability function of the data.

As for the logistic regression, we first determine the logarithmic-likelihood function and then the loss function as a negative logarithmic-likelihood function.

$$L(\theta, D) = \frac{1}{N}\sum_{x^{(i)}} \quad log\, p\left(x^{(i)}\right), \tag{1.8}$$
$$l(\theta, D) = -L(\theta, D)$$

Restricted Boltzmann Machine (RBM)

The historical development of the RBM began with a recurrent neural network (RNN).

One such model was the Hopfield network. Hopfield also introduced the energy concept after comparing neurodynamics with thermodynamics.

The next step was the usual Boltzmann machines which differ from the Hopfield network by their stochastic nature and their neurons divided into two groups which describe hidden and visible states.

The restricted Boltzmann machines differ from the usual ones in that there are no connections among the neurons of the same layer (similar to hidden Markov models). The structure of the RBM is presented in Fig. 1.4.

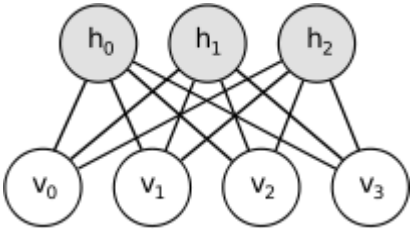


Figure 1.4. The RBM structure

The property of this model is that, at a given state of one group of neurons the states of another group of neurons would be independent of each other. Now consider some theoretical results wherein this property plays a key role.

RBM Interpretation

RBMs are interpreted like hidden Markov models. They have a layer of states which we can observe (visible neurons) and a layer of states which are hidden and we can't see (hidden neurons). But we can make a probabilistic inference concerning the hidden states based on the visible ones. After training such a model we also get the opportunity to make conclusions about visible states knowing hidden ones (using Bayes theorem), and by this generate data from that probabilistic distribution on which the model was trained.

Therefore, we can formulate the RBM training goal: *it's necessary to tune model parameters so that the restored vector is maximally close to the original*.

By restored vector, we imply the vector obtained by probabilistic inference from visible states.

RBM Algorithm

Often, we are not interested in completely observing the instance X or we want to introduce some not-observed variables that increase the model's descriptive force. So let us consider the visible part of the model [70, 71] (denoted by X) and the invisible part (denoted by h). Then we can write:

$$P(x) = \sum_h P(x,h) = \sum_h \frac{e^{-E(x,h)}}{Z} \tag{1.9}$$

The energy function **E(v,h)** of the restricted Boltzmann machine is presented as:

$$E(v,h) = b'v + c'h + h'Wv \tag{1.10}$$

where W is the weight connecting visible and non-visible neurons, and b and c are biases of visible and hidden layers correspondingly.

This is transferred directly to the following formula for free energy:

$$\mathcal{F}(v) = -b'v - \sum_i log \sum_{h_i} e^{h_i(c_i + W_i v)} \tag{1.11}$$

Owing to the specific RBM structure, visible and non-visible neurons are conditionally dependent on each other. Using this property, we can write:

$$p(h|v) = \prod_i p(h_i|v)$$
$$p(v|h) = \prod_j p(v_j|h) \tag{1.12}$$

The network consists of stochastic neurons taking 0 or 1 (where v_j and $h_i \in \{0,1\}$). From the formulas (1.10) and (1.11) we obtain the probabilistic variant of usual neuron activation

$$p(v, W, b_h) = \sigma(W * v + b_h) \tag{1.13}$$

where v is the neuron input, W is the weight vector (matrix), b_h is the bias, and σ(x) is the sigmoidal function. This is the basic variant for binary inputs (Bernoulli-Bernoulli RBM), and there are also modifications for real inputs (Gaussian-Bernoulli RBM, etc.).

The algorithm of the RBM runs as follows:

1. Set the initial values for the input variables v: = x.
2. Compute the probabilities p_h of the change neuron states of the second layer (non-visible) $p_h = \sigma(v * W + b_v)$ where W is the weight matrix, b_v is the bias vector of the first layer, and σ is the activation function (sigmoid).

Store the old values of the input neurons v′:= v.

3. Determine the states of the second layer neurons h, and assign to the neurons state 0 or 1 with the probability p_h.
4. Compute the probabilities p_v of the change states of the first layer neurons
$$p_v := \sigma(h * W^T + b_h),$$
where b_h is the bias vector of the second layer and σ is the activation function (sigmoid), and assign to the neurons state 1 with the probabilities pv (or 0 with the probabilities 1-pv).
5. If v≠v′ then repeat from step 2. Otherwise go to the next step.
6. Release result v.
7. End.

Training Method Contrastive Divergence (CD)

The RBM training algorithm is called contrastive divergence and represents the modified gradient descent [72, 76, 77]. As the estimation function is to be optimized, the likelihood function L is used. Let's search for its maximum. The likelihood function L for the parameters (W,b_v,b_h) and pattern v are determined under the given values of the parameters W, h as $p(v,h) = \frac{1}{z}e^{-E(v,h)}$,

$$L(v) = p(\theta),$$
$$L(v) = p(v|W, b_v, b_h) \ . \quad (1.14)$$

For the simplicity of computations, we'll use the logarithm:

$$lnL(v) = lnp(v) = ln\ \tfrac{1}{z}\textstyle\sum_h \quad exp\left(-E(v,h)\right) =$$
$$ln\textstyle\sum_h \quad exp\left(-E(v,h)\right) - ln\textstyle\sum_{v,h} \quad exp\ (-E(v,h)). \quad (1.15)$$

For a great number of visible and hidden neurons, numerical methods like gradient ascent are usually applied to find the maximum of this logarithmic likelihood function. The maximization of the likelihood function is equivalent to the minimization of weights as the weights are linearly connected with the energy function E (see (1.16)).

$$E(v,h) = (b_v * v + b_h * h + v * h * W) \quad (1.16)$$

As it follows from (1.16), the minimization of weights leads to the minimization of the energy function E. At each iteration of the gradient descent, the algorithm parameters are adjusted in the dependence of the likelihood function whose derivatives are presented below:

$$\frac{\partial lnL(v)}{\partial\theta} = \frac{\partial}{\partial\theta}\left(ln\sum_h \quad e^{-E(v,h)}\right) - \frac{\partial}{\partial\theta}\left(ln\sum_{v,h} \quad e^{-E(v,h)}\right) = -\frac{1}{\sum_h \quad e^{-E(v,h)}}\sum_h \quad e^{-E(v,h)}\frac{\partial E(v,h)}{\partial\theta} + \frac{1}{\sum_{h,v} \quad e^{-E(v,h)}}\sum_{h,v} \quad e^{-E(v,h)}\frac{\partial E(v,h)}{\partial\theta} = -\sum_h \quad p(v)\frac{\partial E(v,h)}{\partial\theta} + \sum_{h,v} \quad p(h,v)\frac{\partial E(v,h)}{\partial\theta}. \tag{1.17}$$

The gradient of this function may be divided into three parts as follows:

$$: \{\frac{\partial lnL(W,b_v,b_h|v)}{\partial W} = \nabla W = (v*h)_{data} - (v*h)_{model}\ ;\ \frac{\partial ln\,L(W,b_v,b_h|v)}{\partial b_v} = \nabla b_v = (v)_{data} - (v)_{model};\ \frac{\partial ln\,L(W,b_v,b_h|v)}{\partial b_h} = \nabla b_h = (h)_{data} - (h)_{model} \tag{1.18}$$

where (·) data is the values of layer states at the initial state of the RBM, and the (·) model is the mathematical expectation of the layer states.

The mathematical expectation of the neuron states is calculated by so-called sampling, i.e., the (·) model is the layer state after some iterations (in practice, for algorithm work one step of sampling (one iteration) is enough). The weights are changed as follows:

$$\{W \coloneqq \varepsilon*(\nabla W + \mu*\Delta W)\ b_v \coloneqq \varepsilon*(\nabla b_v + \mu*\Delta b_v)\ b_h \coloneqq \varepsilon*(\nabla b_h + \mu*\Delta b_h)\ , \tag{1.19}$$

where μ is the so-called moment parameter, ε is the training speed, and $\nabla W, \nabla b_v \nabla b_h$ are the parameters changed at the previous iteration.

As a stop criterion, we'll use the MSE between the input and output of the BRM – $E(v_0, v_k)$, and this value should decrease to the established threshold Emin.

The training algorithm consists of the following steps [77]:

1. Initialize (by zeros) the weight matrix W and the bias vectors $b_v, b;_h$.
2. Choose a random mini-batch from the entire training set (mini-batch) X.
3. For all the examples in the mini-batch assign initial values to the first layer v:= x.
4. Execute k cycles in the network, and determine the initial and final states of layers v_0,h_0, v_k, h_k, (where k is a parameter).
5. Compute the gradient according to (1) and adjust the weights by (2).
6. Calculate the network MSE E.

7. If E <Emin, then go to 8, otherwise go to 2.
8. End.

Training Algorithm Contrastive Divergence (CD-k)

This algorithm was developed by Professor Hinton in 2002, and it differs through its simplicity [72]. The main idea lies in the mathematical expectation being replaced by certain values. The concept of sampling is introduced (*Gibbs sampling*).

The algorithm CD-k runs as follows:

1. The states of the visible neurons are set equal to the input pattern;
2. The probabilities of the hidden layer neurons are calculated;
3. Each neuron of the hidden layer of the state "1" is assigned with a probability equal to its current state;
4. The probabilities of states of the visible layer are determined based on the states of the hidden layer;
5. If the number of current iterations is less than k, return to step 2;
6. The probabilities of the neuron states of the hidden layer are obtained.

The work of the corresponding algorithm is presented in Fig. 1.5.

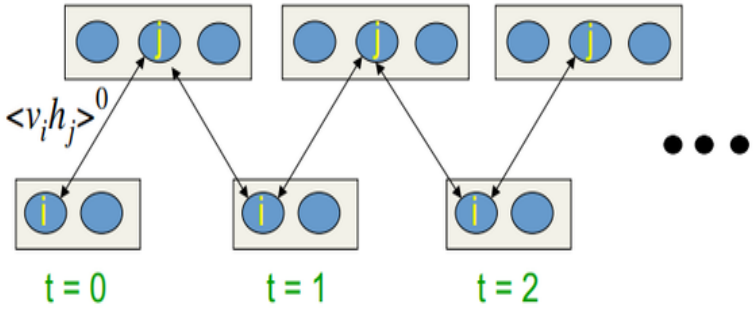
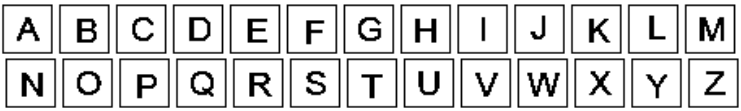


Figure 1.5. Training algorithm CD-k

The longer we make the sampling process, the more accurate is the CD-algorithm.

Example

Consider the implementation of the above presented model. At the start, several images of Latin letters are stored in the memory. Then the system is shown other similar patterns that are distorted; by using these, the original patterns should be restored. The training is set.



The results of the algorithm work:

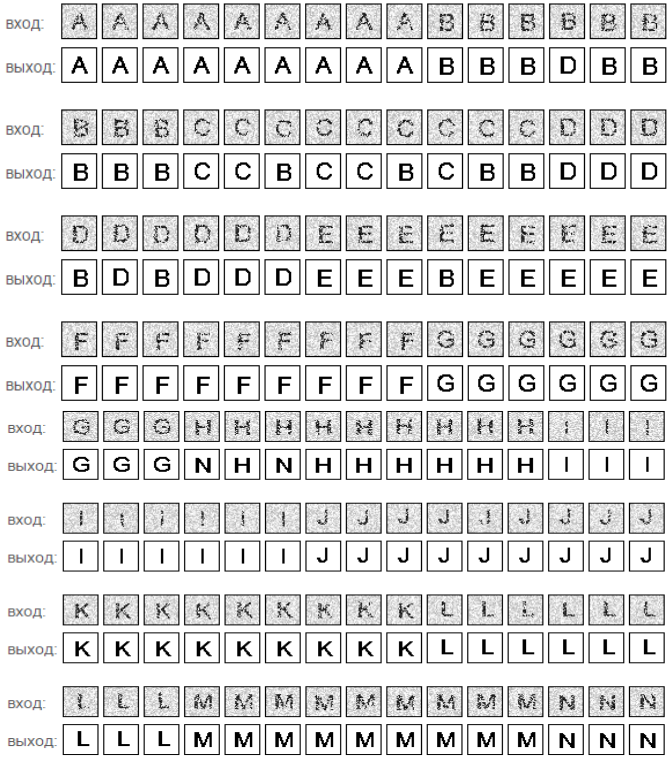


Figure 1.6. Results of algorithm CD-k

Stacked Auto Associator Networks

Stacked Autoencoder (SAE)

Autoassociators are stacked in a network to retrieve high-level abstractions from the input set. In Fig. 1.7 the structure schema of a stacked autoencoder is shown which overall, represents a deep learning network with the weights initialized by a stacked autoencoder [90].

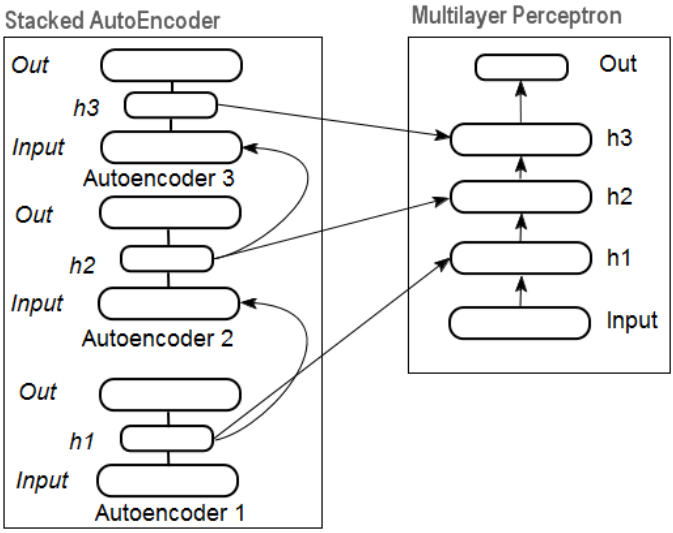


Figure 1.7. Deep network with SAE Structure

Stacked RBM

In Fig. 1.8, a structure schema of a stacked restricted Boltzmann machine (SRBM) and neural networks is presented which represents a deep neural network with the weights initialized by the SRBM.

Structures of deep networks are shown in just such a way, underlining that information is retrieved upwards (from bottom to top).

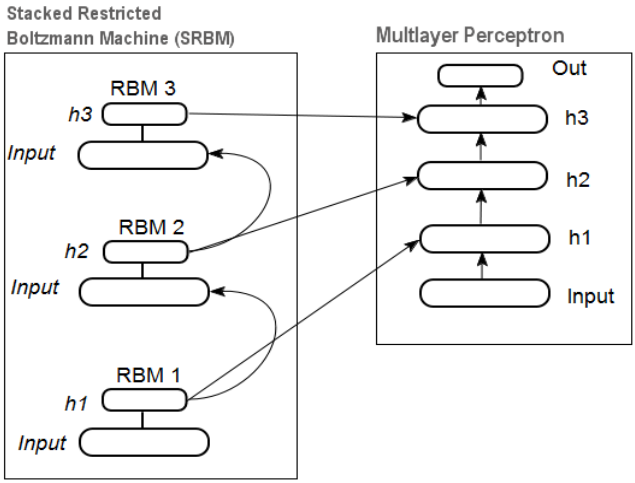


Figure 1.8. Structure of the DN SRBM

Deep Network Learning

The process of learning deep networks is split into two stages [52-54]:

1. Pretraining; and
2. Weight fine-tuning.

Deep Network Pretraining

At the first stage, the autoassociative network (SAE or SRBM) is non-supervised trained on an array of not-marked data after which neurons of the MLP hidden layer are initialized with weights obtained after training. This process of training and transfer is shown in Fig. 2.8. After the training of the first AE/RBM weights of the hidden layer, the neurons become inputs of the second layer and so on. In this way, more and more general information about the structure of data (line, contour, etc.) is retrieved from the data.

Let's consider the pretraining procedure in more detail. Pretraining represents the following procedure: we take pairs of neighboring layers of the deep learning network beginning from the first layer and construct from these pairs the autoencoder, by adding an output layer identical to the input layer. This procedure is repeated sequentially for all network layers. This procedure may be described as follows:

1. Load a training data set X0;
2. Determine a network parameter – the number (N) and size of layers;
3. Set the number of the current layer i=0;
4. Build an autoencoder for the layers i, i+1;
5. Train an autoencoder at the set $X_{i;}$
6. Take away the auxiliary (output) layer of the autoencoder;
7. Preserve the connection weights of the layers i, i+1;
8. If there is still a pair of layers to be processed (i<N-2), then go to the next step; otherwise go to step 10;
9. Generate the data set X_{i+1} for the next autoencoder; for this, propagate through a pair of layers i, i+1 the data set Xi, and go to step 3;
10. End of work.

After this procedure the network is trained by one of the gradient methods.

Besides, G. Hinton also suggested performing fine-tuning in two stages for deep networks with more than three hidden layers. For the first stage, only train the two upper layers and for the second, train the whole network.

It is worth noting that with non-supervised learning the SRBM gives fewer stable results than the SAE.

Fine-Tuning

At the second stage, the fine-tuning of MLP weights (training with a teacher) is performed by known methods. *It was practically proved that this initialization sets the weights of the neurons of the MLP hidden layers in the region of a global minimum and the next fine-tuning is performed for a very short time.*

Fine-tuning is a process of weighing small changes for improving or optimizing results. As a rule, it aims to increase process efficiency. Fine-tuning may be executed by a number of methods which are dependent on optimized processes which include gradient methods of the first order and gradient methods of the second order: Newton and quasi-Newton methods and others.

Algorithms for Optimizing The Cost Function in Machine Learning for NN

Currently, CNN learning packages have a large set of cost function optimizers. The work of most optimizers is based on the calculation of the gradient, but there are special optimizers designed to solve the problem of

local minima [52, 54]. This section considers the best known algorithms for optimizing the cost function.

Gradient Descent

Gradient descent is a basic method of finding the local minimum of a function [100]. The rule of updating the weights θ_i is expressed by the following formula:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \lambda \frac{\partial C^{(t)}}{\partial \theta_i} \tag{1.20}$$

where λ is the learning rate; $\theta_i^{(t)}, \theta_i^{(t+1)}$ are the parameter values i at iterations t and t+1 accordingly; and $C^{(t)}$ is the cost function.

Adagrad

Adaptive gradient descent (Adagrad) normalizes the learning speed for each dimension on which the cost function depends. At each iteration, the global learning rate is divided by the l2-norm of the previous gradients for each dimension [101]. The rule of updating parameters is expressed by the following formula:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\lambda}{\sqrt{\sum_{\tau=1}^{t} \theta_i^{(\tau)2} + \varepsilon}} \frac{\partial C^{(t)}}{\partial \theta_i} \tag{1.21}$$

where λ is the learning rate; $\theta^{(t)}, \theta^{(t+1)}$ are the parameter vectors at the iterations t and $t+1$ correspondingly; $C^{(t)}$ is the cost function; and $\sqrt{\sum_{\tau=1}^{t} \theta_i^{(\tau)2} + \varepsilon} - l^2$ is the norm of the parameters of all preceding iterations for the feature i.

In the Adagrad algorithm, the parameters are updated at different learning speeds. The more frequently the feature is met, the lower is its learning speed and the more seldom the feature is met, the higher is its learning speed. This is due to the fact that, for the scattered features, the value of the l2-norm will be smaller, so the overall speed of learning will be higher. This algorithm works well in the areas of natural language and image processing, which are characterized by scattered data [99].

RMSprop

The RMSprop algorithm [100] implements a mini-package version of the resilient backpropagation optimization technique (Rprop), which is best suited to full-package learning. The Rprop algorithm solves the problem whereby the gradient vectors are not oriented towards the minimum in the elliptical contours of the value function. A characteristic feature of the Rprop algorithm is that gradient signs are solely used for each weight when updating [99, 100].

In the first step, the Rprop algorithm starts with the same update rates for all weights, i.e.,

$$\Delta_{ij}^{(t=0)} = \Delta_{ij}^{(0)} = \Delta \tag{1.22}$$

where $\Delta_{ij}^{(0)}$ is the coefficient of the weight update w_{ij} at the first iteration. The minimum and maximum values of the weighting coefficients $\Delta_{\min}$ and Δ_{max}, respectively, are also set. At each iteration, the algorithm checks the signs of the previous and current gradient components, i.e., the partial derivatives of the cost function at different weights.

If the signs of the current and preceding iterations coincide, that is $sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\frac{\partial C^{(t-1)}}{\partial w_{ij}}\right)$ >0, then the learning rate increases with the coefficient $\lambda_+ = 1{,}2$.

The update factor is calculated by the formula:

$$\Delta_{ij}^{(t+1)} = min\left(\lambda_+ \Delta_{ij}^{(t)}, \Delta_{max}\right) \tag{1.23}$$

where λ_+ is the coefficient of the increased learning rate; $\Delta_{ij}^{(t)}$ is the update coefficient w_{ij} at iteration t; and Δ_{max} is the maximal value of the update coefficient [1]. If the signs of the current and previous gradient components are different, i.e., $sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\frac{\partial C^{(t-1)}}{\partial w_{ij}}\right)$ <0, then the learning rate decreases with the factor $\lambda_- = 0{,}5$.

The update rule then takes the form:

$$\Delta_{ij}^{(t+1)} = max\left(\lambda_- \Delta_{ij}^{(t)}, \Delta_{min}\right) \tag{1.24}$$

where λ_- is the coefficient of the decreased learning rate; $\Delta_{ij}^{(t)}$ is the update coefficient w_{ij} at iteration t; and Δ_{min} is the maximal value of the update coefficient.

If $sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\frac{\partial C^{(t-1)}}{\partial w_{ij}}\right)$ is equal to 0 then the update coefficient does not change:

$$\Delta_{ij}^{(t+1)} = \Delta_{ij}^{(t)}.$$

The rule of updating parameters is expressed by the following formula:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right)\Delta_{ij}^{(t+1)}. \tag{1.25}$$

The RMSprop optimizer combines the capabilities of adaptive learning of each weight of the Rprop algorithm with the efficiency of the stochastic gradient descent. The Rprop algorithm does not use a gradient sign for each weight.

The problem with the stochastic gradient descent is that the cost function and gradients change with each new mini-package. The basic idea of RMSprop is to choose a gradient value that does not fluctuate significantly within several neighboring mini-packages. To do this, it is necessary to use the root mean square values of the gradients for each weight on the last few mini-packages [100].

$$g_{ij}^{(t)} = \alpha g_{ij}^{(t+1)} + (1-\alpha)\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right)^2, \tag{1.26}$$

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\lambda}{\sqrt{g_{ij}^{(t)} + \varepsilon}}\frac{\partial C^{(t)}}{\partial w_{ij}} \tag{1.27}$$

where $g_{ij}^{(t)}$ is the average of the gradient square for the weight w_{ij} at the iteration t; α is the coefficient of the decrease of the gradient square; and $\frac{\partial C^{(t)}}{\partial w_{ij}}$ shows the values of the gradient components.

Adam

The Adam (adaptive moment estimator) algorithm supports not only the moving average of the gradient squares, but also the moving average of past gradients. The Adam algorithm is another algorithm that has an adaptive learning speed for each of the parameters [99, 101].

Let β_1 та β_2 be a rate decrease of an average $m_{ij}^{(t)}$ and the average of gradients square $v_{ij}^{(t)}$ for all weight w_{ij} correspondingly similar to momentum. Then the momentum can be considered as a ball running on a slope, and the Adam algorithm behaves like a heavy ball with friction, thus preferring the flat minima on the surface of the error function [9]. Then the update rules for the Adam algorithm are written as follows:

$$m_{ij}^{(t)} = \beta_1 m_{ij}^{(t-1)} + (1-\beta_1)\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right) \ , \tag{1.28}$$

$$v_{ij}^{(t)} = \beta_2 v_{ij}^{(t-1)} + (1-\beta_2)\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right)^2 \tag{1.29}$$

where $\frac{\partial C^{(t)}}{\partial w_{ij}}$ are values of the gradient components.

Normalized values of the gradients average $\hat{m}_{ij}^{(t)}$ and the average of gradients squares $\hat{v}_{ij}^{(t)}$ are calculated by the following formulas

$$\hat{m}_{ij}^{(t)} = \frac{m_{ij}^{(t)}}{(1-\beta_1^t)}, \tag{1.30}$$

$$\hat{v}_{ij}^{(t)} = \frac{v_{ij}^{(t)}}{(1-\beta_2^t)}$$

The rule of the update for each weight has the following form:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\lambda}{\sqrt{\hat{v}_{ij}^{(t)}+\varepsilon}}\hat{m}_{ij}^{(t)}. \tag{1.31}$$

Deep Learning Regularization

There exist two types of errors in neural network training: 1) the so-called training error ε_{tr} and the generalization error ε_{gen}. The training error is determined in the training sample and the generalization error is in the test sample. These two errors are functions of the number of training iterations n and display different behavior: ε_{tr} is a monotonous decrease with n, while the generalization error ε_{gen} first decreases then attains a minimum, and then begins to rise with an increase of n (this phenomenon is called overfitting). The goal of training lies in the minimization of the

generalization error.

Regularization is any modification of a training algorithm which is aimed to decrease the generalization error at the expense of a certain increase in the training error. Regularization refers to one of the central problems in machine learning which competes, by its significance, with the problem of optimization.

Due to the theorem of costless breakfast the best algorithm for machine training doesn't exist. In particular, there is not a method of regularization which can be classified as the absolute best.

Instead, we need to choose the regularization form that is a good fit for our problem to be solved. In the philosophy of deep learning overall, there lies a wide range of problems (such as all the intelligent problems) which can be efficiently solved with the application of general forms (methods) of regularization. Consider the most popular regularization methods and forms.

L_p-regularization of linear regression

Consider the classic linear regression model

$$t = \sum_{j=1}^{d} w_j x(j) + \varepsilon, \varepsilon \sim N(0, \sigma^2). \quad (1.32)$$

The search of weights w by the maximization of the likelihood function of the sample in this model is equivalent to the LSM method:

$$\sum_{n=1}^{N} \left[t_n - \sum_{j=1}^{d} w_j x_n \right]^2 = \|t - w_1 x_1 - \cdots - w_d x_d\|^2 = \|t - Xw\|^2 \to \min.$$

where $x_i \in R^N$ is a value of the i-h feature for all objects in the sample X=[$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d$].

Note that the denotation x_i introduced here, differs from the standard when by x_i is implied the i-h sample object. Here, and further on, the sample is assumed to be normalized.

This task has a simple geometric interpretation – the search for a projection of vector t onto a hyperplane with direction vectors $x_1, \ldots, x_d$ (see Fig. 1.9). This problem can be solved analytically: $w = (X^T X)^{-1} X^T t, t_{pr} = Xw = X(X^T X)^{-1} X^T t$.