ISSN 1060-992X, Optical Memory and Neural Networks, 2017, Vol. 26, No. 4, pp. 221–248. © Allerton Press, Inc., 2017.

Shallow and Deep Learning for Image Classification

G. Ososkov[a], * and P. Goncharov[b], **

[a] *Joint Institute for Nuclear Research, Dubna, Moscow oblast, Russia*

[b] *Sukhoi State Technical University of Gomel, Gomel, Belarus*

**e-mail: ososkov@jinr.ru*

***e-mail: kaliostrogoblin3@gmail.com*

Received September 1, 2017; in final form, September 11, 2017

Abstract—The paper is focused on the idea to demonstrate the advantages of deep learning approaches over ordinary shallow neural network on their comparative applications to image classifying from such popular benchmark databases as FERET and MNIST. An autoassociative neural network is used as a standalone program realized the nonlinear principal component analysis for prior extracting the most informative features of input data for neural networks to be compared further as classifiers. A special study of the optimal choice of activation function and the normalization transformation of input data allows to improve efficiency of the autoassociative program. One more study devoted to denoising properties of this program demonstrates its high efficiency even on noisy data. Three types of neural networks are compared: feed-forward neural net with one hidden layer, deep network with several hid-
1 den layers and deep belief network with several pretraining layers realized restricted Boltzmann machine. The number of hidden layer and the number of hidden neurons in them were chosen by cross-validation procedure to keep balance between number of layers and hidden neurons and classification efficiency. Results of our comparative study demonstrate the undoubted advantage of deep networks, as well as denoising power of autoencoders. In our work we use both multiprocessor graphic card and cloud services to speed up our calculations. The paper is oriented to specialists in concrete fields of scientific or experimental applications, who have already some knowledge about artificial neural networks, probability theory and numerical methods.

Keywords: artificial neural networks, deep learning, autoencoder, restricted Bolzmann machine, image recognition

DOI: 10.3103/S1060992X1704004X

1. INTRODUCTION

The new challenging era of scientific data management in the coming decade of "Big Data" requires a new paradigm named data intensive science to deal with Exabyte data scale in many modern social, economic and scientific areas. Experimental High Energy and Nuclear Physics, in particular, demands now not only the giant complexes for distributed computing and corresponding grid-cloud internet services, but what is the most important in context of this paper, – Machine Learning approaches to search for unevident regularities in data for getting the most probable forecast of phenomena under study. Machine learning algorithms use input data to detect patterns in data and adjust program actions accordingly.

Artificial neural networks with their ability for learning and self-learning are effective machine learning tools, so physicists accumulated a quite solid experience in various neural net applications in many high energy and nuclear physics experiments for the recognition of charged particle tracks, Cherenkov rings, physical hypotheses testing, and image processing [1–4]. Two types of neural nets were mainly applied, – multi-level perceptrons (MLP) and recurrent neural nets of the Hopfied type.

There is still the handy MLP realization with the error back-propagation algorithm for its training in the Toolkit for Multivariate Data Analysis with ROOT [5]. Besides, namely physicists have a privilege to generate training samples of any arbitrary needed length by Monte Carlo simulations on the basis of some well-known physical theory and use to apply MLP intensively [6].

At the same time the first neural net applications to the particle track reconstruction problems in 1988 [7, 8] brought physicists to the world of dynamical binary neuron systems with a symmetric weight matrix and the network energy function which minima correspond to equilibrium points of the system. Using a statistical treatment of neuron states known as the "mean field theory" [9], it became possible to induce regulated amounts of thermal noise with the noise temperature T into the evolving neural net rule so that it escapes the local minima. Larger temperatures correspond to larger amount of noise, and the evolving procedures benefit from various rules of the simulated annealing scheme [10], where the temperature decreases as the network approaches its global minimum of the energy function. Due to the flexibility of the energy function definition it was possible to solve the track recognition problem by constructing the energy function in a form which causes the Hopfield network evolution to converge to the state where neurons corresponded to smooth tracks are the most probable [9, 11].

Nevertheless, at the new "Big Data" reality both neural net types quite rare could show their needed efficiency due to many shortcomings. The shallow architecture with 1-2 hidden layers is typical, at present, for feed-forward neural nets with the back-propagation training algorithm [12]. Since problems to be solved are now getting more and more complicated, the neural net architecture needs to become deeper by adding more hidden layers for better parametrization and more adequate problem modelling. However in most cases deepening, i.e. the fast grow of inter-neuron links, faces the problem known as the "Curse
2 of dimensionality", which leads to overfitting or to sticking the minimized error function of the back propagation in poor local optima. As it was noted in [13, 14], Hopfield networks could not also be effective enough in cases of noisy and dense events.

At these circumstances it is just the time to remember about series of researches devoted to the deep learning concepts published during the last decade [15–18], which authors dealing with neural networks were inspired by the knowledge about the deep hierarchical structure of mammals visual cortex [19–21].

There were two ways of making deeper a neural network hierarchy: either just adds more hidden layers to multilayer perceptron with its supervised learning paradigm, or to change this paradigm by including to neural network a hierarchy of additional layers, but with different, unsupervised learning.

The first way was well described in [17, 22] and mostly used the state of art approach to overcome difficulties of training deep feedforward neural networks by various tricks to optimize network parameters.

The second way is more similar to the bio-analog of the animal visual cortex and was proposed by G. Hinton [23] on the basis of combining several stacked layers of recurrent neural net of the Bolzmann machine type realizing unsupervised representations of input data with the last supervised layer completing a classification task. At the end the whole network is fine-tuned by providing supervised data to it. This type of deep networks was named deep belief networks where the word "belief" means not "faith", but "confidence" referring one to stochastic neural networks and probability distribution of their neurons. These layers are restricted Boltzmann machine (RBM) [15, 24, 25] that can learn themselves a probability distribution over its set of inputs. The evolution of Restricted Boltzmann machine layers is realized by Markov chain Monte Carlo method [26], although its convergence needed to obtain mean values of wanted parameters is always too long to be practical. However, Hinton's idea of so-called "contrastive divergence" function helps to speed up the convergence [27].

One quite significant approach is to build out a deep neural nets of layers of autoencoders. Autoencoder algorithms consist of training a neural network to produce an output that is identical to the input, but having fewer nodes in the hidden layer than in the input forming a "bottle neck". In this case one has built a tool for compressing data to a smaller number of the most informative features [28, 29].

Alternatively, you could allow for a large number of hidden units, but require that, for a given input, most of the hidden neurons only produce a very small activation. Nevertheless, if we impose a sparsity constraint on the hidden neurons neglecting weak ones with very small activation, then an autoencoder will discover important structures in the data, even if the number of hidden units is large. Such a process of unsupervised pre-training is named sparsifying. As it is shown in [30], a sparsified representation before the classification task should greatly reduce the learning problem because sparsity enforces a hidden layer to learn a code dictionary that minimizes reconstruction error while restricting the number of code-words required for reconstruction.

There are other meaningful properties of autoencoders. Stacking autoencoders, when each layer of the network learns an encoding of the layer below creates a network with unsupervised learning hierarchical features [31, 32]. Training stacked layers allows a deep network to learn a good representation in a greedy layerwise fashion instead of trying to train the whole network from a random initialization of weights.

Autoencoder stacking purposes two objects: one concerns to denoising autoencoders proposed in [33] for unsupervised learning based on the idea of making the initial learned representations robust to partial corruption of the input pattern. The robustness to partially destroyed inputs should yield almost the same representation. This approach can be used to train denoising autoencoders to be stacked to initialize deep architectures.

The second object of autoencoder stacking relates to convolutional autoencoders intended to deal with 2D image structure unlike fully connected and denoising autoencoders [34]. It allows to discover localized features that repeat themselves all over the input. The convolutional network architecture consists of three basic building blocks to be stacked: the convolutional layer, the max-pooling layer and the classification layer [35]. At present convolutional networks are among the most successful models for supervised image classification [36].

There are many other types of deep learning neural nets with quite different structures such as Long Short Term Memory [37, 38], Gated Recurrent Unit networks [39], Reinforcement learning networks [40, 41]. The most intriguing results were shown recently by Microsoft deep residual networks with hundreds of layers and millions of parameters [42, 43]. During the last decade these novel ultra-deep networks have shown remarkable results not only in image classification, but also in regression, dimensionality reduction, modeling motions, information retrieval, object segmentation, natural language processing and many others.

However in this paper we are not going to survey deep learning networks. Our goal is focused on a more practical idea to demonstrate the advantages of deep learning approaches over ordinary shallow neural network on their comparative applications to image classifying from the such popular benchmark databases as FERET [44] and MNIST [45].

At the first step of preparing images to input to neural network we use autoassociative or autoencoders to extract the most informative features of input data. According to M. Kramer [28], three hidden layers of an autoencoder with a "bottle neck" in the middle allows to accomplish the nonlinear principal component analysis which realize more complex mapping functions than principal component analysis removing not only linear, but also non-linear correlations.

Three types of neural networks are compared: feed-forward neural net with one hidden layer, deep
1 neural net with several hidden layers and deep belief neural net with several pretraining RBM layers. The
number of hidden layers and the number of hidden neurons in them were chosen by cross-validation pro-
cedure [46] to keep balance between number of layers and hidden neurons and classification efficiency.

It is worth mentioning here that one of the main obstacles of applying deep learning methods is their slowness. To solve real big data problems they demand quite essential computing resources. It leads to the challenge of evolving multicore computational systems and multiprocessor graphic cards or GPUs to speed up the convergence of deep networks. Now we can see a real deep learning breakthrough because, for instance, GPUs allow the fast matrix and vector multiplications required as for convincing virtual realities, as for neural net training, where they can speed up learning by a factor of 50 and more [38]. Therefore in our work we use both GPU and cloud services to speed up our calculations.

This paper is oriented to specialists in concrete fields of scientific or experimental applications, who have already some knowledge about artificial neural networks, probability theory and numerical methods.

2. METHODS AND DATABASES

The intrinsic dimension of the data set is the least dimension that describes a data set without significant loss of feature. Biometric datasets such as face images are typical examples of high-dimensional datasets with low intrinsic dimensionality. The dimensionality of a multicomponent image vector to be input to a neural net is redundant due to correlations and non-linear dependencies between pixels forming this image and, therefore, is much greater than its intrinsic dimensionality. The image dimensionality reduction to its intrinsic dimensions simplifies considerably any image processing. It is especially important for deep neural net applications where the number of inter-neuronal links grows drastically with the dimension of input vector what makes the training process too long and leads to the bad convergence of the neural net cost function. However traditional linear estimators of intrinsic dimensions, such as principal component analysis can identify only linear correlations while reduction of non-linear dependencies with minimal loss of information could be also important.

2.1. Nonlinear Principal Component Analysis by Autoencoders

We propose to apply an autoencoder with unsupervised learning for data compression to a smaller number of the most informative features by realizing the nonlinear principal component analysis (NLPCA) [28]. Besides we refuse the popular idea of including AE into the hierarchy of a learning neural net in a stacked manner [34] and prefer to use an autoencoder as a standalone program due to the following reasons:

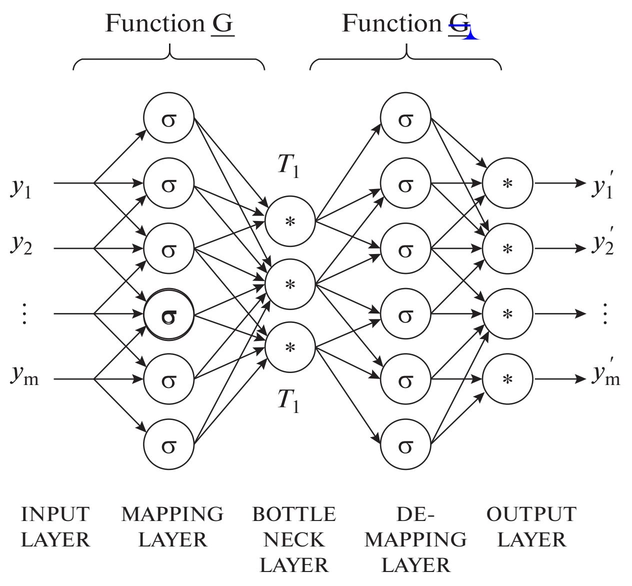


Fig. 1 [28]. Autoaccosiative neural network for simultaneous determination of *f* nonlinear factors, σ indicates sigmoidal nodes, * indicates sigmoidal or linear nodes.

• it gives us the same set of compressed data as an input vector for three neural networks to be compared further as classifiers;

• autoencoder realized as an autoassociative neural network needs to optimize its structure and tune parameters of its activation function and a procedure of input data normalization.

We use M. Kramer's scheme of NLPCA network where the non-linearity is obtained by the special architecture with three hidden layers including the bottle neck layer in the middle. This scheme is depicted in Fig. 1 from [28].

Bottle-neck layer T extracts f-principal components. Training is performed with sigmoidal activations by self-supervised backpropagation minimizing the sum of squared errors

$$E = \sum_{p=1}^{n}\sum_{i=1}^{m}(Y_i - Y_i')_p^2. \quad (1)$$

However we accomplish a special study of applicability sigmoidal activations and optimal choice of the normalization transformation of input data (see expounding and results in section 3.1 below). Then we modified Kramer's scheme considerably by replacing sigmoidal activations on to Leaky Rectified Linear activations [47] with a specific input normalization.

One more study devoted to denoising properties of our autoencoder algorithm demonstrates its high efficiency even on noisy data (see details in section 6 below).

2.2. Deep Belief Network

Deep belief networks (DBN) were proposed as probabilistic generative models that are composed of multiple layers of stochastic, latent variables [25]. The scheme of the DBN from [48] is presented on the Fig. 2. DBN runs a pretraining procedure, using latent restricted Boltzmann machine (RBM) layers (see below). Discriminative fine-tuning can be performed by adding a final layer of variables that represent the desired outputs and backpropagating error derivatives.

2.2.1. Restricted Boltzmann machines and contrastive divergence. Boltzmann machine is a type of stochastic recurrent neural network and Markov random field, for which the bilinear energy function is determined as follows [15]:

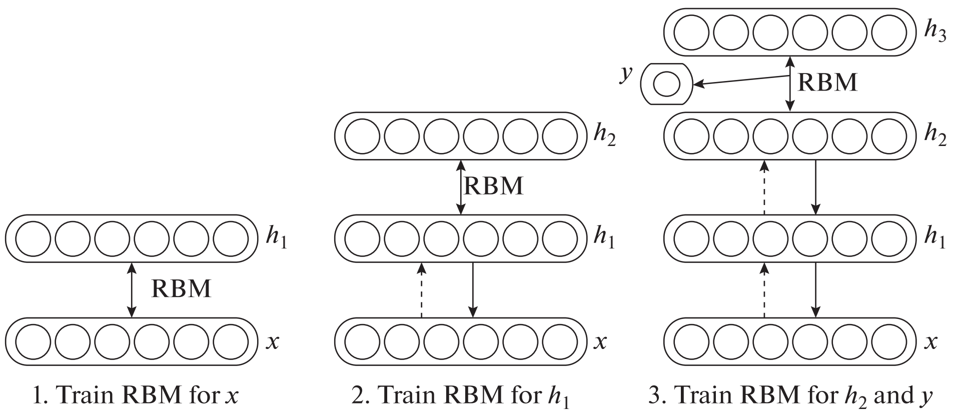
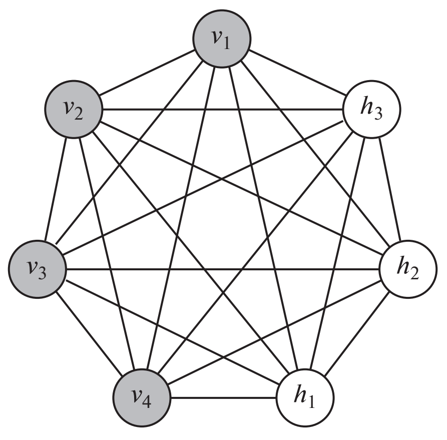
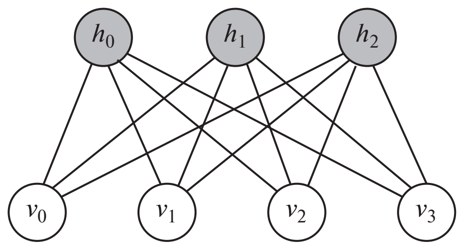


Fig. 2 [48]. Deep Belief Network model.



(a) Boltzmann machine



(b) RBM

Fig. 3. Comparison of the BM and RBM models architectures.

$$E(v,h) = -\sum_{i\in \text{visible}} a_i v_i - \sum_{j\in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_j W_{i,j},$$

where v_i, h_j are states of visible unit i and hidden unit j, a_i, b_j are they biases and $W_{i,j}$ is their synoptic weight. RBMs further restrict BMs to those states, but without visible-visible and hidden-hidden connections. The comparative schemes of BM and RBM architectures are presented in Fig. 3.

The RBM network assigns a probability to every possible pair of a visible and a hidden vector via the energy function:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)}, \tag{2}$$

where the partition function, Z, is given by summing over all possible pairs of visible and hidden vectors:

$$Z = \sum_{v,h} e^{E(v,h)}.$$

The probability of network assigning a visible vector, v, is given by summing over all possible hidden vectors:

$$p(v) = \frac{1}{Z} \sum_{h} e^{E(v,h)}.$$

The probability that the network assigns to a training image can be derived by adjusting the weights and biases to lower the energy of a given image and to raise the energy of other images, especially those that have low energies and therefore make a big contribution to the partition function. The derivative of the log probability of a training vector (2):

$$\frac{\partial \log p(v)}{\partial w_{i,j}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}},$$

where the angle brackets are used to denote expectations under the distribution specified by the subscript that follows. This leads to a very simple learning rule for performing stochastic steepest descent in the log probability of the training data, where η is a learning rate:

$$\Delta w_{i,j} = \eta\left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}}\right).$$

Because there are no direct connections between RBM hidden units, it is very easy to get an unbiased sample of $\langle v_i h_j \rangle_{\text{data}}$. Given a randomly selected training image, v, the binary state, h_j, of each hidden unit is set to 1 with probability (3). It is also quite easy to get an unbiased sample of the state of a visible unit, given a hidden vector (4).

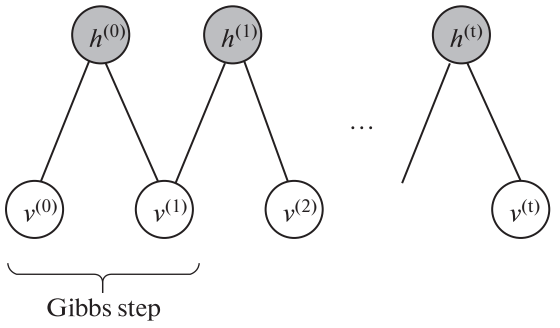


Fig. 4. Sampling in an RBM.

$$p(h_j = 1|v) = \sigma\left(b_j + \sum_i v_i w_{i,j}\right), \tag{3}$$

$$p(v_i = 1|h) = \sigma\left(a_j + \sum_i h_i w_{i,j}\right), \tag{4}$$

where $\sigma = 1/(1 + \exp(-x))$.

Getting an unbiased sample of $\langle v_i h_j \rangle_{\text{model}}$, however, is much more difficult. It can be done by starting at any random state of the visible units and performing alternating Gibbs sampling for a very long time [25].The scheme of this procedure called Markov Chain Monte Carlo (MCMC) [26] is shown in Fig. 4.

Unfortunately, the MCMC convergence needed to obtain mean values of wanted parameters is always too long to be practical; however, G. Hinton found a "trick" to speed up sampling process, which works surprisingly well.

He showed, if the learning approximately minimizes a different function called "contrastive divergence", then Markov chain can run only for just one step [27].

Contrastive divergence learning algorithm is as follows:

—since, we eventually want $p(v) \approx p_{\text{train}}(v)$, we initialize the Markov chain with some of training examples;

—Contrastive divergence learning does not wait for the whole chain to converge. Samples are obtained after only k-steps of Gibbs sampling. In practice, $k = 1$ is enough.

After training one RBM layer, the activities of its hidden units can be treated as data for training a higher-level RBM. This method of stacking RBMs makes it possible to train many layers of hidden units efficiently and it is one of the most common deep learning strategies. As each new layer is added, the overall generative model gets better.

During the last decade deep belief family networks have been applied with success not only in classification, but also in regression, dimensionality reduction, modeling motions, information retrieval, object segmentation, natural language processing and many others [49].

2.3. Datasets

As databases for training and testing networks, two well-known datasets are used:

—MNIST handwritten digit database;

—FERET face database.

MNIST handwritten digits database [45] includes 70000 training images, 10000 validation images (for hyper-parameter selection), and 10000 test images, each showing a 28 × 28 grey-scale pixel image of one of the 10 digits. In Fig. 5 a small sample of MNIST handwritten digit database is shown.

The Facial Recognition Technology (FERET) database includes of 400 people's faces photos (40 people to 10 photos for each) taken from different angles, which format is 55 × 45 pixels [44]. A fragment of FERET images is presented in Fig. 6.

3. OPTIMAL CHOICE OF ACTIVATION FUNCTION AND INPUT NORMALIZATION

Learning of many hidden layers in the artificial neural network causes the famous hampering problem named "vanishing and exploding gradients" because there is an intrinsic instability associated to learning by gradient descent in deep, many-layer neural networks. [22]. There are several recommendations how to avoid such problems [35, 22, 50]. Keeping them in mind, we made a study of the optimal choice of activation functions and normalization transformations of input data.

The important note should be made about optimality criteria used to choose network parameters. It is reasonable to use for classifying networks such criteria as classification efficiency and calculation speed, but for the standalone autoencoder which is used as the data compressor for all three classifiers we have

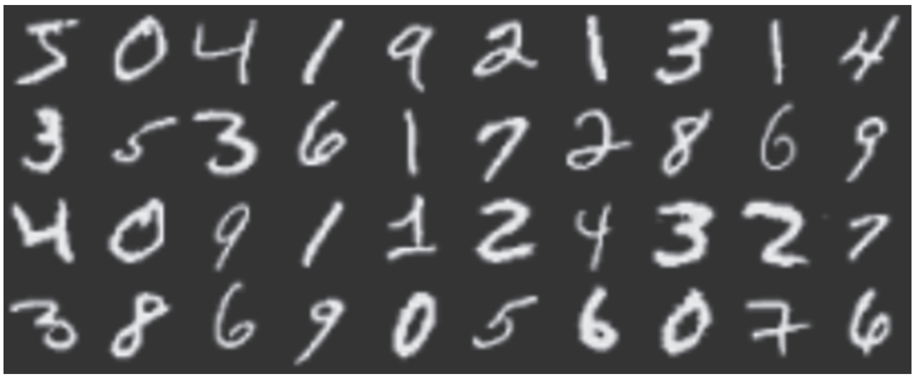


Fig. 5. MNIST handwritten database sample.

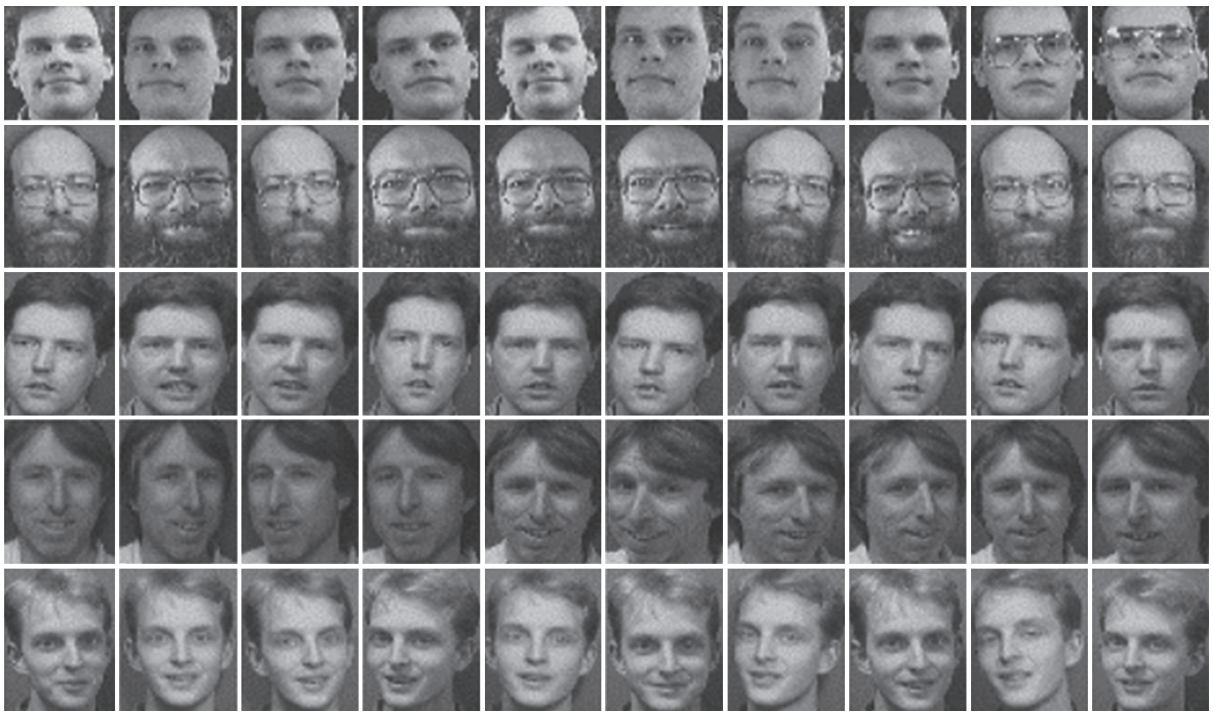


Fig. 6. FERET face database sample.

to measure the quality of data compression by backpropagation error, i.e. the sum of squares of differences between input and output images.

3.1. Activation Functions and Normalizing Features for the Autoencoder

In this section, we introduce seven choice strategies using 5 types of activation functions, including sigmoid, hyperbolic tangent (Tanh), softplus, rectified linear (ReLU), leaky rectified linear (LReLU). (see their illustrations in Fig. 7) and 5 variants of dataset normalizing with MinMax scale and "mean face" subtraction.

3.1.1. Activation functions. Sigmoid. Commonly used sigmoid activation formula is as follows:

$$f(x) = \frac{1}{1+e^{-x}}.$$

Hyperbolic Tangent. The tanh activation is also very popular

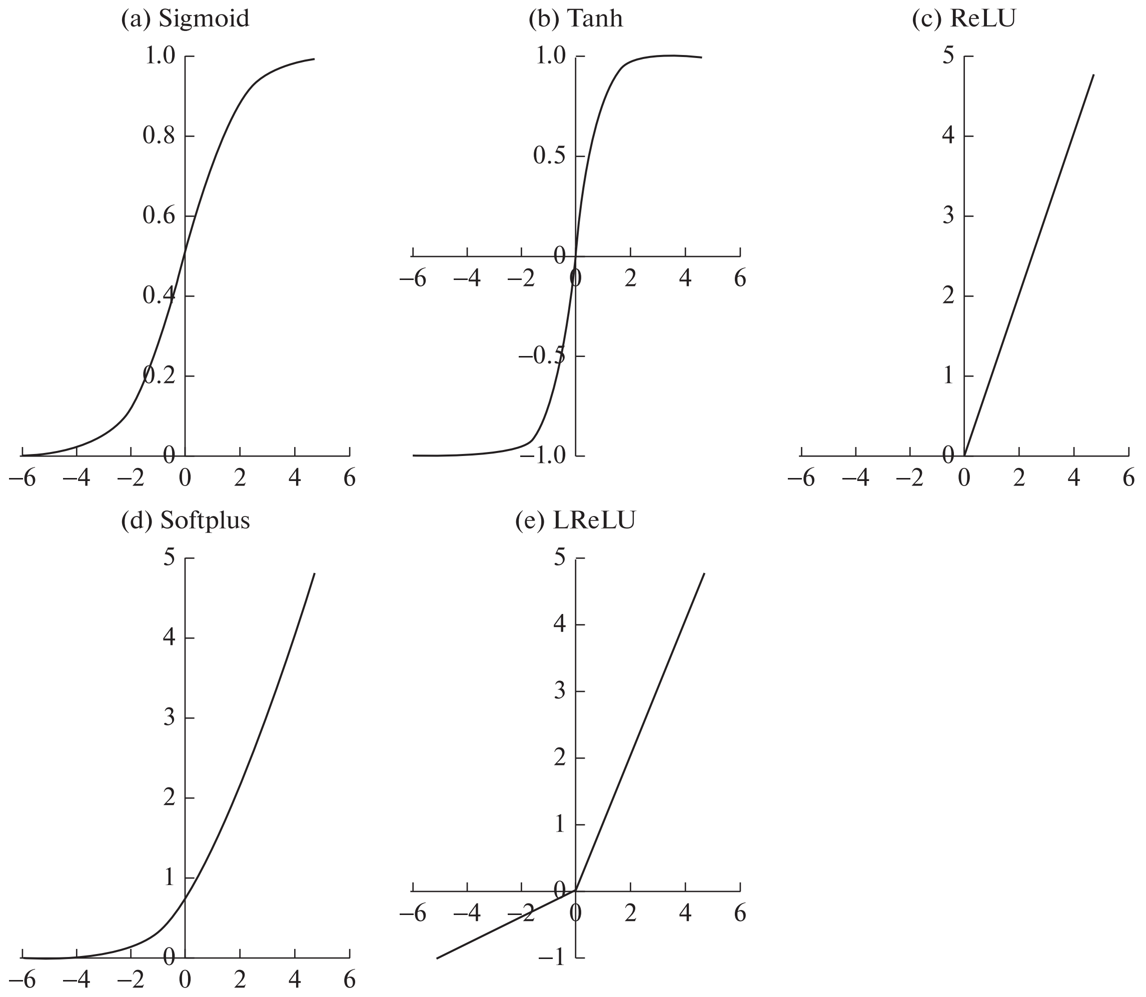


Fig. 7. Sigmoid, Tanh, ReLU, Softplus and LReLU.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{5}$$

Rectified Linear Unit (ReLU). Rectified Linear is first used in Restricted Boltzmann machines [51].

$$f(x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}. \tag{6}$$

Softplus. Softplus is the smooth (ReLU) approximation proposed by [52]

$$f(x) = \ln\left(1 + e^{x}\right). \tag{7}$$

Leaky Rectified Linear Unit (LReLU). Leaky ReLU activation is first introduced in acoustic model [Maas et al., 2013]. In this paper we performed LReLU following to [47]:

$$f(x) = \begin{cases} x \text{ if } x \geq 0 \\ \dfrac{x}{a} \text{ if } x < 0 \end{cases}, \tag{8}$$

where a is a fixed parameter in range $(1, +\infty)$. In this work we set this parameter equal to 5.5 for the MNIST dataset. The optimal choice of parameter a for the FERET face database was obtained by the search of loss function minimum as it is shown in Fig. 8. We provide $a = 3$ in case of faces.

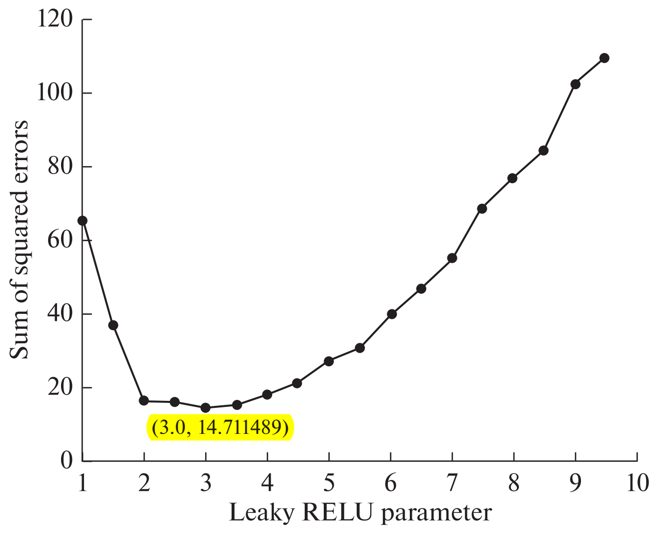


Fig. 8. Choosing the optimal Leaky ReLU parameter a. The marked point illustrates the better $a = 3$.

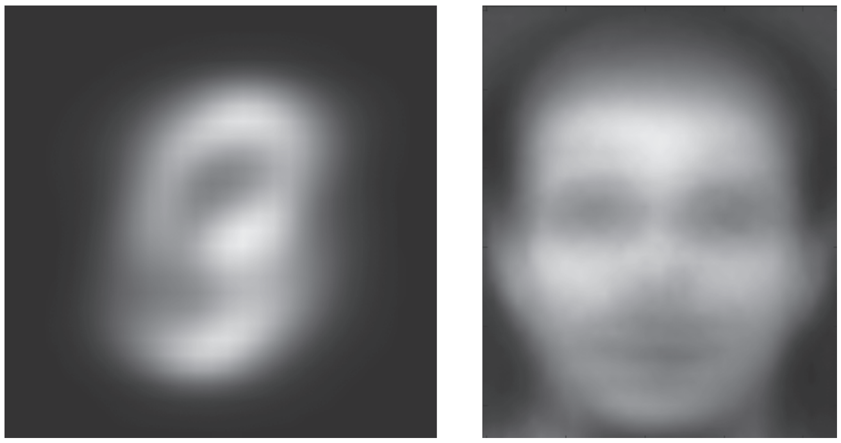


Fig. 9. "Mean faces". On the left side there is a mean value of MNIST dataset, and the right image is FERET mean value.

Nonlinearities of such activation functions, as tanh and sigmoid, are used very often in neural networks, but in practice ReLU activation performs with the same efficiency or even better, but its hard saturation below the threshold is not suited for the reconstruction units.

3.1.2. Importance of normalizing the input. It worth to note, that all input data should be normalized to obtain faster convergence [35] and to lie within the activation range. We use the MinMax normalization technique because it, according to our trials, produced the best result with the highest accuracy, least complexity and shortest learning speed, better than known Z-score standardization [53].

The formula of MinMax normalization is as follows:

$$X_n = \frac{X_o - X_{\min}}{X_{\max} - X_{\min}}.$$

The other thing that we tried is subtracting mean data value ("mean face", further) from each sample. In Fig. 9 one can see the "mean faces" of two images from our databases.

Besides of studying activations, we experiment with several strategies of various normalizations to prevent zero reconstruction in place of non-zero target:

1. Use a softplus activation function (7) for the last reconstruction layer, along with a quadratic cost (1) [54].

2. Apply hyperbolic tangent (5) or sigmoid nonlinearities in contrast to softplus (7).

3. Use a rectifier activation function for the reconstruction layer, along with quadratic cost.

4. Use leaky ReLU activation (8) instead of original rectifier linear units (6).

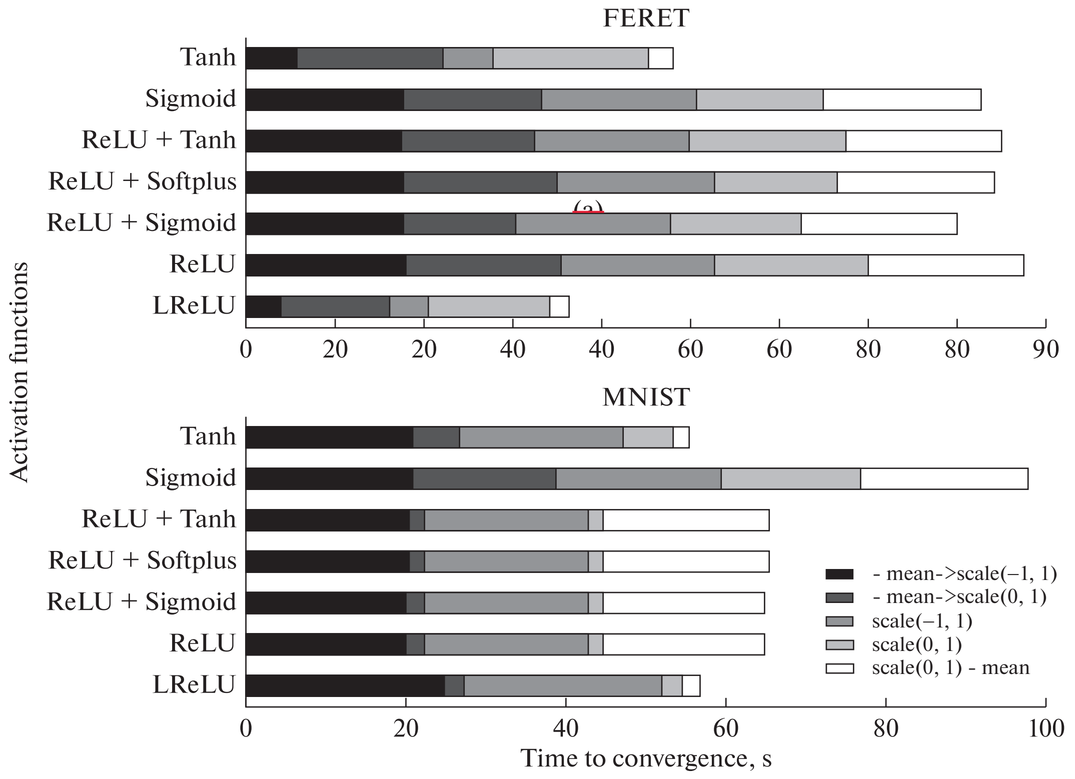


Fig. 10. Convergence speed for 35 options of autoencoder parameters.

Consider this, we have 7 options of activations functions: Sigmoid, Tanh, ReLU+Tanh, ReLU+Sigmoid, ReLU+Softplus, ReLU and LReLU, where "+" means that we apply different nonlinearity to the reconstruction layer, along with general squared-error loss. Additionally we have to consider 5 normalization options:

1. scale (data – "mean face") from –1 to 1;
2. scale (data – "mean face") from 0 to 1;
3. scale (data) from 0 to 1;
4. scale (data) from –1 to 1;
5. (scale (data) from 0 to 1) – "mean face".

It should be noted, that the second item and the last item are not the same, because, for example, if we rescaled data in range $(0,1)$ with mean equals 0.5 and then subtract mean value from each sample, the new range will be $(-0.5,1)$. Also items one and forth are not similar too, because the fourth point ignores the "mean face" subtraction.

To find the best option between so many combinations of feature normalization and activation we define the loss boundary for each dataset, i.e. such a value of the sum of squared errors on which we obtain the highest efficiency. We set these values to 60 and 5000 for the FERET and MNIST databases respectively. Then we check the speed of the model convergence and plot the horizontal bar, as it is depicted in Fig. 10.

Thus, from Fig. 10 is follows the best choice for FERET data is obtained as LReLU activation and normalization by scale (0, 1) – mean face, for MNIST data we obtain the same normalization, while two activations LReLU and Tanh give similar results.

3.1.3. Tied weights and stochastic gradient to solve the convergence problem. Autoassociative neural network that were used for determination of nonlinear factors has three hidden layers. As a result, network training with the backpropagation algorithm suffers also from a bad convergence of the common gradient descent used in backpropagation. An example of the autoencoder output is shown in Fig. 11 where the loss function has likely stuck in a local minimum.

Two methods were used to overcome this issue: tied weights and stochastic gradient descent with so-called Adam optimization [55].

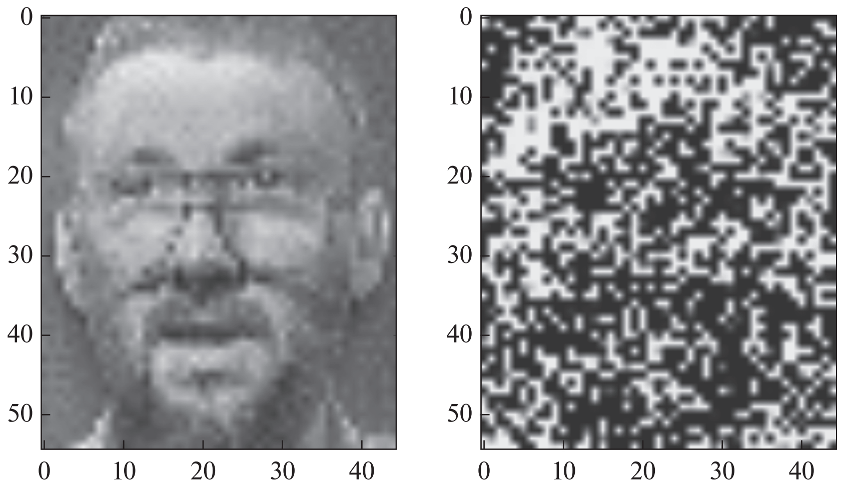


Fig. 11. Example of the autoencoder output (in the right) trained by backpropagation with regular weights.

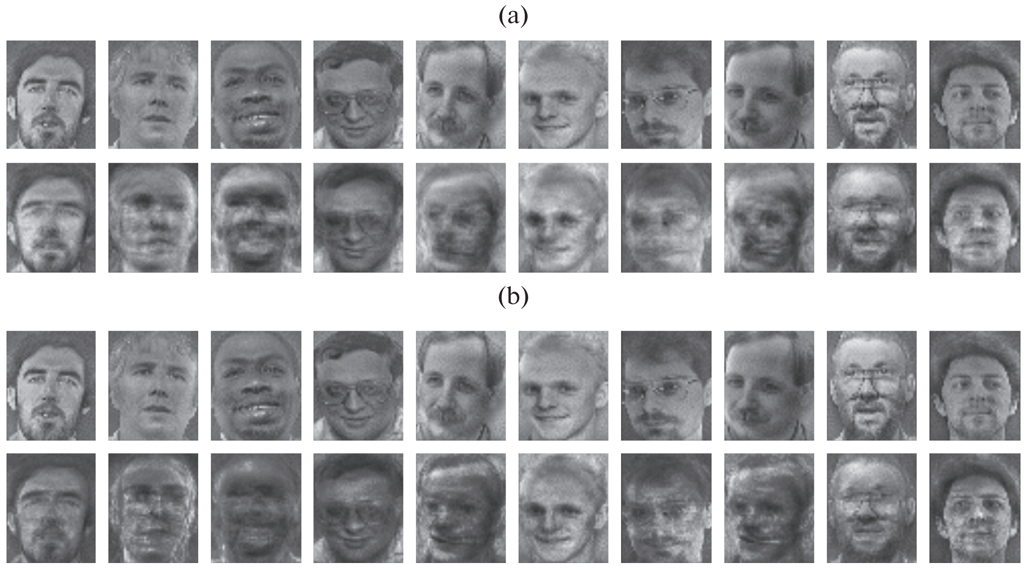


Fig. 12. Images from the FERET database (upper row) recovered with the autoencoder from 64 nonlinear principal components and tied weights (lower row), a) with Tanh activation; b) with LReLU activation.

3.1.4. Tied weights. The tied weights approach could be explained on an easy example of feed forward neural network with one hidden layer and regular weights. Its output looks as

$$f(x) = \sigma_2(b_2 + W_2\sigma_1(b_1 + W_1x)), \tag{9}$$

where σ_2, b_i and W_i $(i = 1,2)$ denotes, correspondingly, activations, biases and weights related to the hidden layer. Since the weight matrices W_1 and W_2 are independent, they got different after autoencoder training by the backpropagation algorithm. A way around this is to use the tied weights defined by formula:

$$f(x) = \sigma_2\left(b_2 + W_1^T\sigma_1(b_1 + W_1x)\right). \tag{10}$$

Here we set $W_2 = W_1^T$ eliminating a lot of freedom degrees. Tied weights work faster and give more accurate results in finding the global minimum of the cost function (1), as it shown in Fig. 12 for the autoencoder which compressed input to 64 nonlinear principal components.

In addition to using tied weights, we propose their normalized initialization such, as described below in section 3.2, with small difference. A numerator is equals to one and a denominator is the square root of the neurons number of the previous layer. It prevents from exploding or vanishing gradients problem.

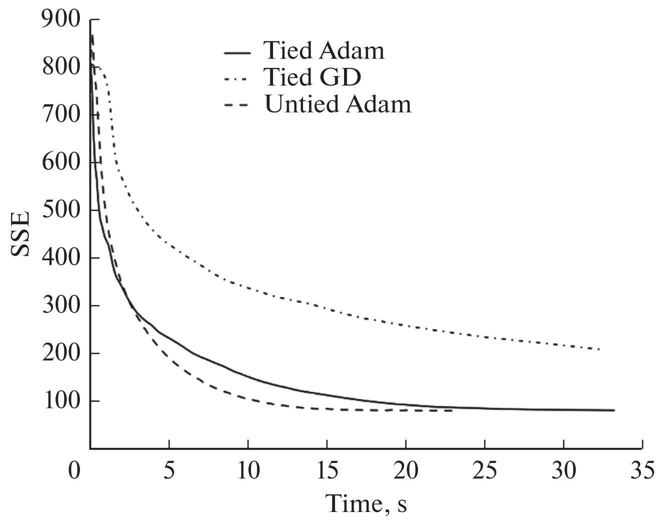


Fig. 13. Decreasing of the loss function in time depending on optimization method and type of weights.

3.1.5. Stochastic gradient descent with Adam optimization. The common batch gradient descent method used in backpropagation performs computations over all training sample, what is too long for large datasets and redundant, since it recomputes gradients of similar examples before each parameter update. In contrast, stochastic gradient descent (SGD) performs one update at a time for each randomly chosen training example x_i and its label. It is much faster and, besides, SGD's fluctuation enables it to jump to new and potentially better local minima. From another hand SGD keeps overshooting the exact minimum of the backpropagation loss function what complicates convergence to it. One of ways to optimize SGD and to dampen its oscillations is adding to the current update vector "a momentum of inertia", as a fraction of the update vector of the past time step.

We use Adaptive Moment Estimation (Adam) method for efficient stochastic optimization that only requires first-order gradients [55]. Adam computes adaptive learning rates for each parameter and also keeps an exponentially decaying average of past gradients, similar to momentum. An important property of Adam's update rule is its careful choice of step sizes.

Results of the applications of the ~~ties~~ weights and SGD Adam methods depicted in Fig. 13. Prefix "Tied/Untied" means the type of network weights: tied or separate weights, respectively. "GD" is a gradient descent abbreviation. As we can see in Fig. 13, Adam optimization method significantly reduces the convergence time. Tied weights do not make a great benefit for speeding up the learning procedure and regular separate weights behave even better, but, worth to note, that in spite of everything, the curve is still stabilized as in the case of tying weights. The other motivation of using transposed representation of encoder weights for the decoder is reducing the number of network parameters twice which is the crucial property of weight tying.

Our experiments show us that untied version of autoencoder even doesn't converge when using gradient descent optimization. At the start of training it stuck in local minima, and then the loss increases very fast.

In our further work with shallow, deep and deep belief neural networks we also use SGD Adam optimization.

3.2. Deep Neural Network

One of the main advantages of deep belief nets used in our work is that these networks do not suffer from "vanishing and exploding gradients" problem. But some learning algorithms (in particular, unsupervised learning algorithms such, as algorithms for training RBMs by approximate maximum likelihood) are problematic in this respect because we cannot directly measure the quantity to be optimized (e.g. the likelihood) because it is intractable. On the other hand, the expected denoising reconstruction error is easy to estimate (by just averaging the denoising error over a validation set, as in [56]). Furthermore, a training time of deep belief nets is always too long. At the same time it is possible to apply another type of classifiers which convergence is much faster, namely Deep Neural Networks (DNN) with the initialization performed by normalized weights [22]:

$$W \sim U\left[-\frac{\sqrt{\{6\}}}{\sqrt{\{n_j + n_{\{j+1\}}\}}}, \frac{\sqrt{\{6\}}}{\sqrt{\{n_j + n_{\{j+1\}}\}}}\right], \tag{11}$$

where $U[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and the n is the size of the previous layer (the number of W columns).

Table 1. Initial parameters of all classifiers

Parameter	Classifier		
	perceptron (1 hidden layer)	deep belief network	deep neural network
Learning rate	0.1	0.1	0.1
Batch size	64	64	32
Hidden units	80	80	80
Optimizer	Adam	Contrastive divergence	Adagrad [57]
Loss function	Cross-entropy	Cross-entropy	Cross-entropy
Dropout	–	0	0
Pretrain epochs	–	1	–
Learning rate pretrain	–	0.1	–
Learning rate decays	–	1.0	–
Gradient clipping	–	–	5.0

Table 2. Comparison of the different classifiers

	Perceptron (1 hidden layer)	Deep neural network	Deep belief network
MNIST	0.7977	0.8713	0.8482
FERET	0.8750	0.9583	0.9416

1 The DNN works much faster than DBN, because it avoids the pretraining procedure for RBM hidden layers. For instance, it is about 28 seconds per training DBN epoch on MNIST dataset on Intel Core i3-4005U versus 2 seconds for deep nets with two hidden layers.

3.3. Shallow Neural Network with One Hidden Layer

We use feed-forward perceptron with one hidden layer and sigmoid activations as shallow neural network estimator. The input of the perceptron is an array of eigenvectors with dimension of 64, which is received from the latent representation of input images of the autoencoder's bottle-neck layer. All perceptron hyperparameters were optimized to obtain the maximum classification efficiency and remain unchanged during all experiments.

The optimal number of hidden neurons was set to 80. The loss function used for training is the mean value of cross-entropy errors. As the training minimizer, the SGD Adam method described in section 3.1.5 is used. Size of output layer corresponds to the number of learning classes (10 for MNIST and 40 for FERET, respectively).

3.4. The First Test of Updated Neural Networks

After updating all three neural networks, – perceptron with one hidden layer, Deep Belief Network and Deep Neural Network with parameters shown in Table 1 we accomplish the first preliminary classification on both databases, – handwritten digits and faces. The obtained classification efficiencies are shown in Table 2.

As one can see the classification results look well and the efficiency of the deep learning models is already higher than of the perceptron with one hidden layer. However, as it is shown in the next section, there are still more ways of optimizing hyperparameters to improve advantages of deep network models.

4. SELECTION OF OPTIMAL HYPERPARAMETERS FOR NETWORK MODELS

Firstly, we need to choose an optimal number of principal components for the best input data recovering by our autoencoder. We suppose the upper boundary of the number of principal components as 250 to prevent the model from an extra growing complexity.

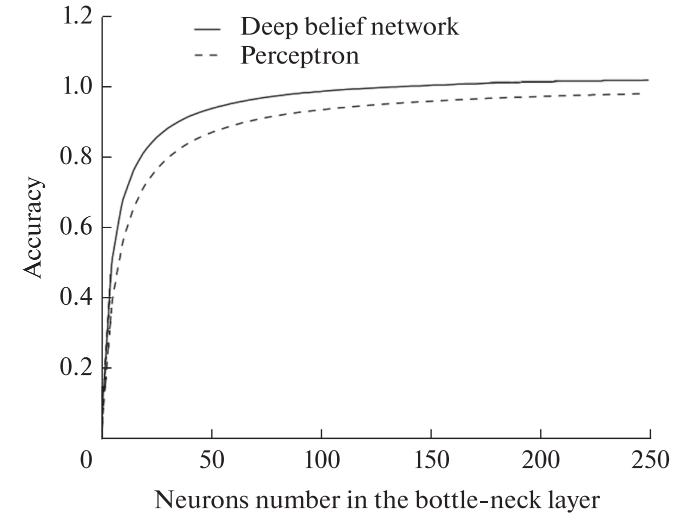


Fig. 14. Dependence of the accuracy on the number of neurons in the bottle-neck layer.

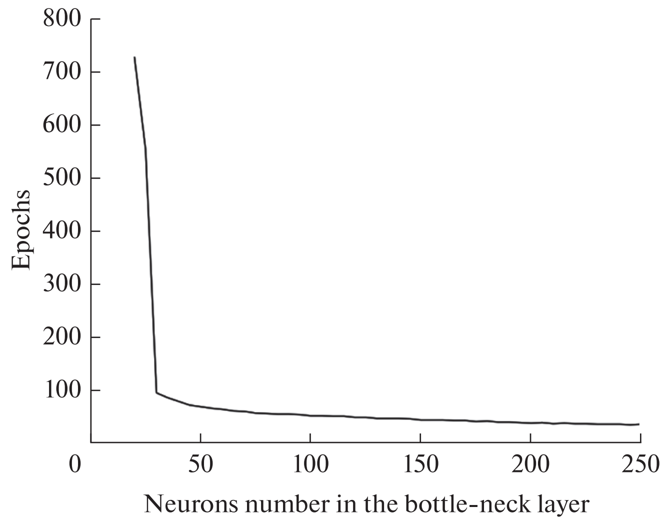


Fig. 15. Dependence of the epoch's number on the number of neurons in the bottle-neck layer.

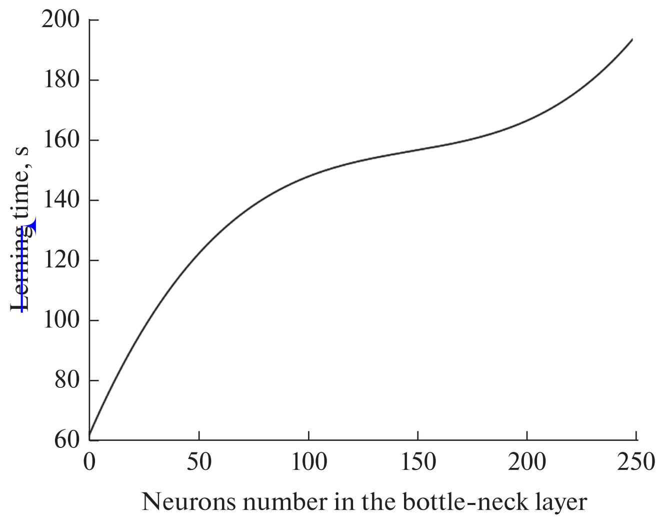


Fig. 16. Dependence of the training time on the number of neurons in the bottle-neck layer.

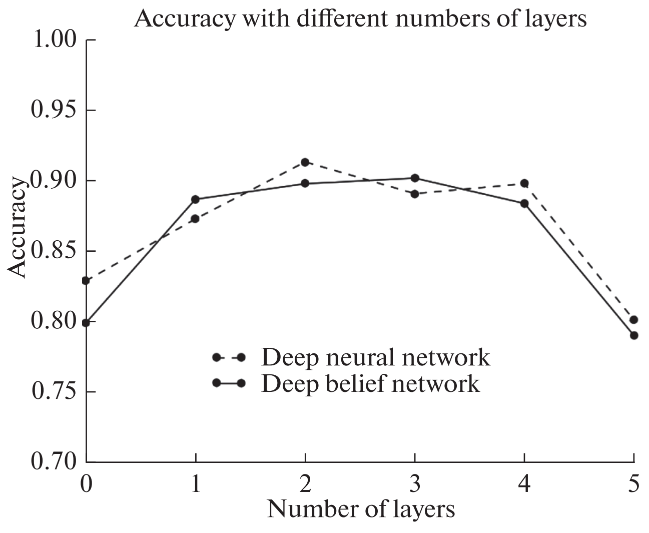


Fig. 17. Efficiency dependence on the number of hidden layers.

4.1. Number of Neurons in Hidden Layers of Autoencoder

Then we calculate the dependence of the AE loss function for different numbers of neurons in the bottle-neck layer. It is shown in Fig. 14. As one can see there, the acceptable number of the principal components begins from 60 components.

The next we calculate how many epoch numbers needed to train the network with the different numbers of principal components. In Fig. 15 it is shown that the epoch number inversely proportional to the number of bottle-neck neurons due to insufficient number of neurons for the input data reconstruction. The curve stabilizes around 70 neurons.

Two previous tests show that the optimal number of principal components lies between 50 and 200. In addition to the epoch number, we consider the time needed to train neural network, which also depends of the number of neurons in the bottle-neck layer, as it shown in Fig. 16. Proceeding from both these dependencies we find that the optimal value is around 70–80 neurons in the bottle-neck layer.

Our experiments with different numbers of principal components bring us, eventually, to the conclusion that the good number is 64, because it ensures the classification efficiency on the level 95% and, as it noted before, a greater number of neurons would increase the model complexity. From similar considerations we found that the number of neurons in both mapping-demapping layers should not exceed 256.

Thus, the final architecture of the autoencoder consists of 5 layers: input and output layers with the same dimensions (length of the input vector), mapping and demapping layers with 256 neurons per layer and the bottle-neck layer with 64 neurons.

4.2. Cross Validation for Tuning Architecture of Both Deep Neural Networks

A good quality of deep architectures means they can model high-level abstractions in data that allows obtain high prediction quality, although too deep neural net models can be easily overfitted, if wrong parameters were chosen. Analogously, if the model is not complex enough, it may not be sufficiently powerful to capture all of the useful information necessary to solve a problem. However, if the model is need-
2 lessly complex (especially if there is a limited amount of training data), it would suffer from overfitting.
2 There are known countermeasures to prevent overfitting in deep learning such as regularization by dropout technics [58] when on each weight update cycle randomly selected nodes are dropped-out with a given probability (e.g. 20%). However we found that the cross validation approach is more preferable in our case for tuning both deep belief and deep network architectures.

Grid search ~~**cross validation**~~ ([59]) provides a rather simple and effective method for both model selection and performance evaluation, which is widely employed by the machine learning community. Under k-fold cross-validation the data are randomly partitioned to form k disjoint subsets of approximately equal size. In the ith fold of the cross-validation procedure, the ith subset is used to estimate the generalization performance of a model trained on the remaining $k-1$ subsets. The average of the generalization perfor-

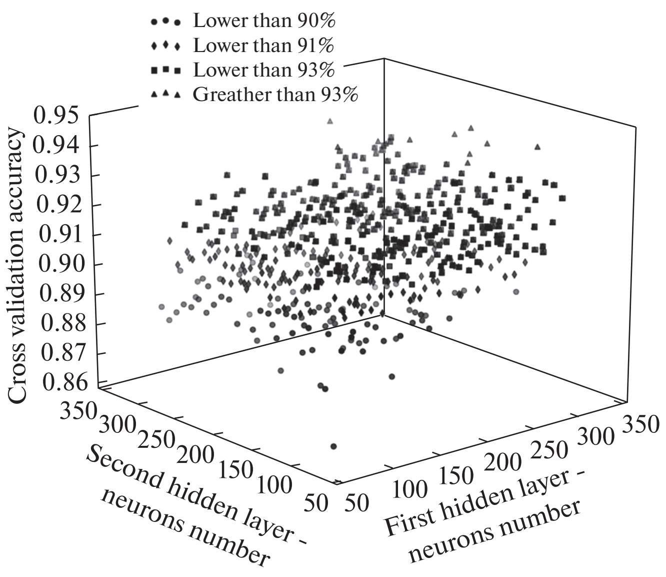


Fig. 18. Efficiency distribution depends on the number of neurons in the hidden layers.

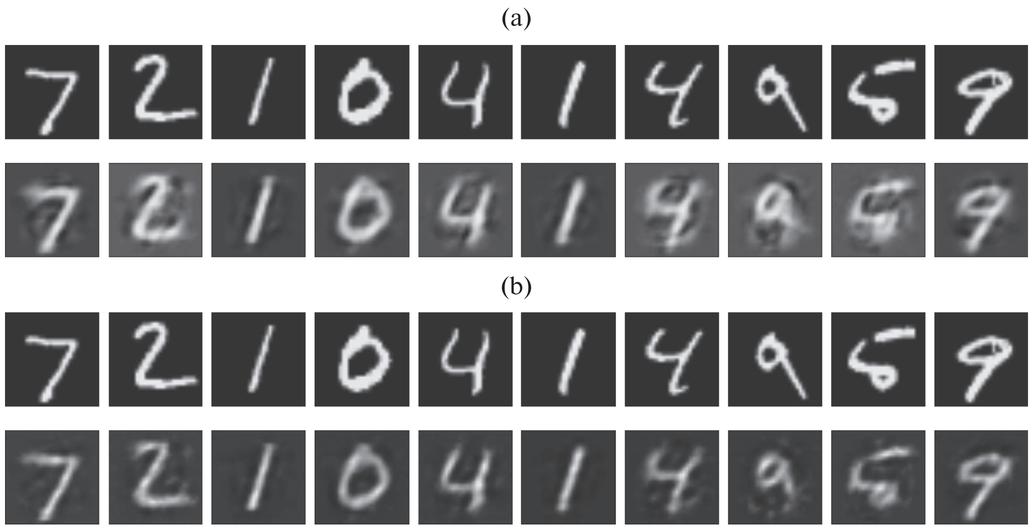


Fig. 19. Images from the MNIST database recovered by the autoencoder with 64 nonlinear principal components; a) with Tanh activation; b) with LReLU activation.

mance observed over all k-folds provides an estimate (with a slightly pessimistic bias) of the generalization performance of a model trained on the entire sample [60].

In our paper we apply a simple grid search cross-validation method for tuning model parameters. According to [61] we split the training database in two parts: 80% as train set and another 20% as cross-validation set. Then we divide cross-validation data into 3-fold batches, calculated the score during the fit of an estimator on each fold and take an average score value. After fitting the estimator on a parameter grid, we choose parameters to maximize the cross-validation score, for which the network classification efficiency is selected.

For the beginning we need to choose the number of the hidden layers in our deep classifiers. Two curves in Fig. 17 represent dependencies the network classification efficiency and the number of the hidden layers. The best choice from Fig. 17 is two hidden layers. Then we have to determine an optimal combination of hidden neurons per layer.

Table 3. 7 best combinations of the hidden neurons numbers

Neurons 1st layer	Neurons 2nd layer	Cross-validation average score
200	230	0.939286
250	300	0.939286
260	210	0.939286
300	160	0.939286
300	200	0.939286
300	100	0.942857
300	180	0.942857

The distribution of the efficiency dependencies on the number of neurons in the hidden layers is shown in Fig. 18. On horizontal axes we can see the number of neurons in the first and second hidden layer, on the vertical axis – the classification efficiency.

Since it is hard to judge on this plot, which combination of the hidden neurons is better, we brought the best combinations and their scores to Table 3. As one can see in Table 3, two last combinations give us the same scores, but 100 neurons in the second layer is apparently better than 180. Thus, the optimal numbers of the hidden neurons per 1st and 2d layers are 300 and 100 respectively.

The other hyperparameters were tuned using the grid search cross-validation (see [59]). The next chapter presents the results of testing the final tuned models.

5. FINAL TEST OF UPDATED NEURAL NETWORKS

5.1. Feature Extraction by Autoencoder

The autoassociative network is used for testing, as the ~~auotoencoder~~ with 5 layers: input and output layers with the same dimensions (length of the input vector), mapping and demapping layers with 256 neurons per layer and the bottle-neck layer with 64 neurons. Examples of MNIST data recovering are shown in Fig. 19.

As it shown above, the proposed autoassociative neural network works pretty well for the feature extraction. It rescales data into the principal components vector with 64 dimensionalities and recovers original images from these encoded representations with the following backpropagation loss function measured as SSE (sum of squared errors):

• for FERET with Tanh activation – 79.4 3, with LReLU activation – 28.18;

• for MNIST with Tanh activation – 15777.3, with LReLU activation – 8573.5.

5.2. Classifiers

The images being rescaled to the 64-dimensional eigenvectors were classified then by three different artificial neural networks:

—perceptron with one hidden layer;

—deep neural network with randomly initialized weights;

—deep belief network.

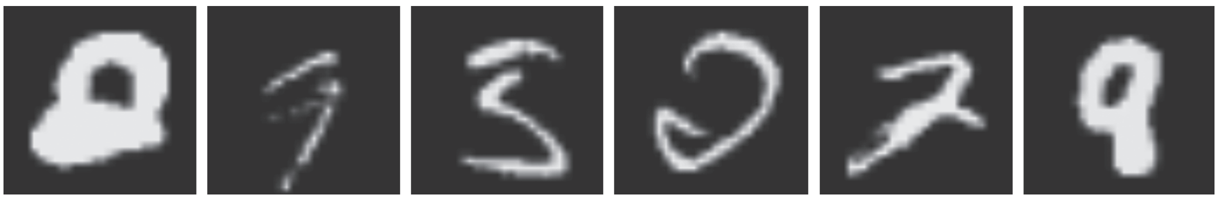


Fig. 20. Examples of incorrect predictions on the MNIST test set.

Table 4. Comparison of the different estimators

	Perceptron	Deep neural network	Deep belief network
MNIST	0.7977	0.9285	0.9839
FERET	0.8750	1.0000	1.0000

The architectures of these networks were specified above in section 4.

Table 4 contains the results of obtained efficiency for testing different classifiers with two datasets:

—MNIST handwritten digits dataset;

—FERET faces database.

The results look quite satisfactory. Deep learning techniques give the 100% classification efficiency on the faces database and the 93–98 percent efficiency on the handwritten digits dataset. Although it is not a problem for the human to distinguish various faces, the DBN can make better predictions in the case of digits, even than the human can.

Examples of the incorrect predictions by the DBN classifier are shown in Fig. 20 to emphasize the special difficulties of classifying handwritten digits even for the human.

6. AUTOENCODER APPLICATION FOR IMAGE DENOISING

Generally, real image examples are not so clean – they can have a lot of noise. We are going to test how our autoencoder model will recover the input images with the different percent of noise imposed on them.

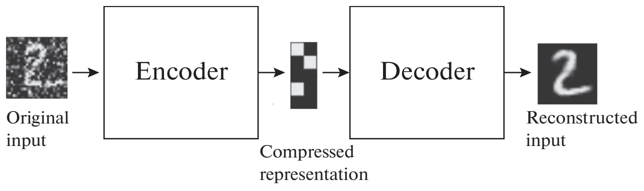


Fig. 21. Denoising autoencoder scheme.

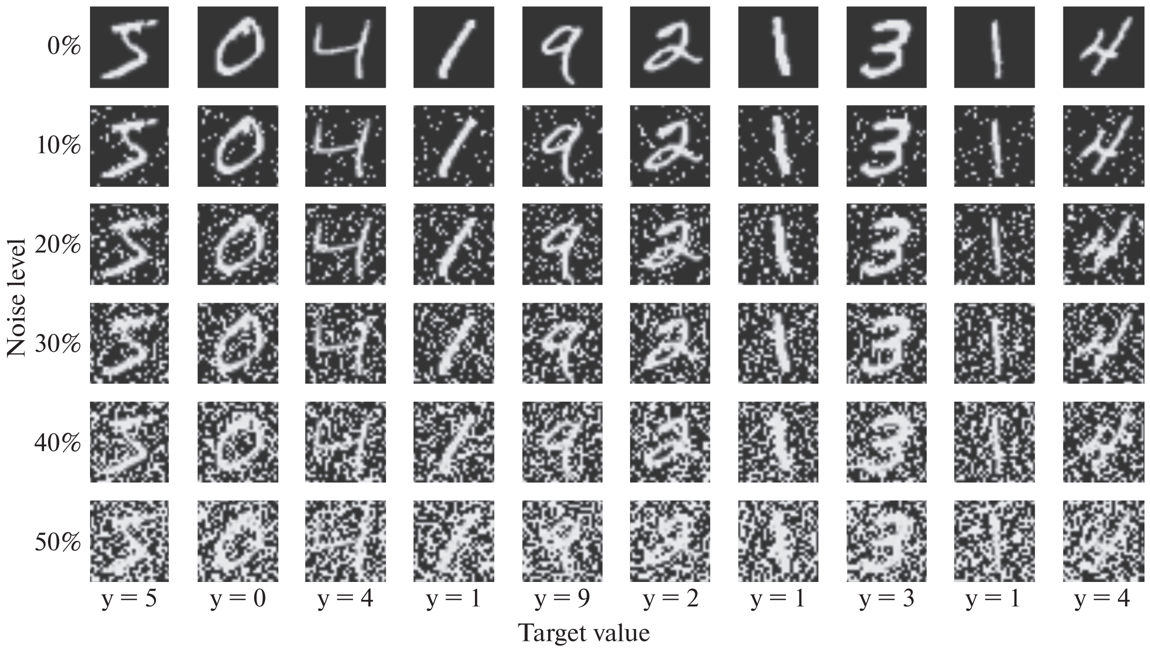


Fig. 22. Examples of the noisy images with different percentage of noise.

Table 5. Comparison of classifying efficiency for noisy images in the case of applying LReLU activations and MinMax normalization of input

	FERET			MNIST		
noise (%)	perceptron	deep NN	deep belief	perceptron	deep NN	deep belief
0	0.9166	0.9867	**1.0000**	0.7449	0.9182	**0.9861**
10	0.9083	**0.9867**	0.9833	0.6901	0.8779	**0.9644**
20	0.9000	**0.9750**	0.9666	0.6115	0.7832	**0.8794**
30	0.8750	0.9583	**0.9583**	0.5001	0.6443	0.6027
40	0.8500	**0.9667**	0.9499	0.4100	0.5202	0.3863
50	0.8000	**0.9583**	0.9333	0.3036	0.3973	0.2283

Table 6 [62]. Comparison of main properties of deep learning toolkits

Frameworks	Base language	Multi-GPU support
TensorFlow	Python, C++, Java, Go	Yes
Torch	LuaJIT	Yes
Theano	Python	By default, no
Caffe	C++	Yes

We did not use any stacked autoencoders for denoising, as in [32] but just the autoencoder described in section 3 above. A denoising procedure takes noisy data, as input, and outputs this data, where the noise is removed. The scheme of denoising neural network is presented in Fig. 21.

A neural network has to be trained to clear images from the noise by minimizing the sum of square errors between the output image, produced by feed forward noisy image through the network, and the clear image that noisy image refers to. Eventually, such a denoising autoencoder network learns how to clear images from noise.

Noise generation is a function that takes a non-corrupted image and corrupts randomly selected pixels bounded by noise factor argument – the corruption dose. The corruptions are different for our data sets: for FERET a corrupted pixel is set to zero, while for MNIST it is set to one. Examples of MNIST noisy images sorted by percent of noise are shown in Fig. 22 (original images correspond to the noise level 0%).

To estimate how effective is denoising power of such a denoising autoencoder we have to apply some classifier to see its efficiency on noisy data compressed by this autoencoder. However at this point we met the question caused by the fact that the noise level of images to be classified is not known in advance. So we have, first, to determine some optimal noise level (a noise factor value) to train a denoising autoencoder on images corrupted by such noise in order to use then this trained denoising autoencoder for compressing images with unknown noise level. After the special study, when we looked over a denoising autoencoders with various noise levels (including zero, i.e. training on non-corrupted images) and evaluating then the classification efficiency after applying the denoising autoencoder to noisy images, we choose 0.2 as the noise factor value. It should be noted that autoencoders trained by clear, non-corrupted data could also recover noisy input, since they find during training nonlinear dependences between input and output data bypassing the image corruption, but they yield 9–10% of classification efficiency to the denoising autoencoder trained with 0.2 noise factor value. Comparative results of noisy images classification by three different estimators are presented in Table 5. The highest efficiencies are marked out by bold-face type in the table.

Our classifiers were trained on non-corrupted images. The denoising autoencoder was trained with 20% noise added to input images. Then the classifiers were tested on six datasets with noise levels in range from 0 to 50% compressed previously by the denoising autoencoder.

As we can see from Table 5, the classification efficiency keeps well even on the data with 30% of noise. Deep Belief Network performs better classification results again, but with noise increasing it yields to the deep neural network. Both deep classifier efficiencies are notably higher than for the perceptron with one hidden layer.

7. SPEED UP DEEP LEARNING BY GPU AND CLOUD SERVICES

7.1. Evaluation of Deep Learning Libraries

Over the past decade deep learning evolution has experienced the great changes. Through growing interest from science, engineering and business this part of machine intelligence endured a lot of extensions and explorations. Now the vital demand of building special frameworks becomes a real challenge for software developers and great IT-giants like Google, Microsoft and IBM. In particular, Google released the beta version of its own framework named TensorFlow.

The current major deep learning frameworks: Theano, TensorFlow, Caffe and Torch are examined here and compared across various features, such as native language of framework, multi-GPU support, training time and aspects of their usability. In the Table 6 from [62] we can see a brief comparison of some properties of these deep learning toolkits.

TensorFlow is an open source software library for numerical computation using data flow graphs [63]. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows one to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

Torch is a scientific computing framework with the wide support for machine learning algorithms that puts GPUs first [64]. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT (it is a Just-In-Time Compiler (JIT) for the Lua programming language), and an underlying C/CUDA implementation. At the heart of Torch are the popular neural networks and optimization libraries which are simple to use, while having maximum flexibility in implementing complex neural network topologies. You can build arbitrary graphs of neural networks, and parallelize them over CPUs and GPUs in an efficient manner.

Theano is a Python library that lets you to define, optimize, and evaluate mathematical expressions, especially ones with multi-dimensional arrays (numpy.ndarray). Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs. Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations [65].

Caffe is a deep learning framework made with expression, speed, and modularity in mind [66]. It is developed by the Berkeley Vision and Learning Center and by community contributors. Expressive architecture encourages application and innovation. Models and optimization are defined by configuration without hard-coding. One can switch between CPU and GPU by setting a single flag to train on a GPU machine then deploy to commodity clusters or mobile devices [66].

As we can see, TensorFlow supports a great variety of programming languages, much more than other frameworks. It should be mentioned, that Caffe has an ability to run in Python by its pretrained models. The syntax of C++ is rather complicated in compared with Python. This also applies to Lua language.

The comparison of these frameworks time consuming is shown in Table 8 from [67] where presented implementation results for different CNNs. The best time marked in bold.

Machine: 6-core Intel Core i7-5930K CPU @ 3. 50GHz + NVIDIA Titan X + Ubuntu 14.04 x86_64.

Table 7 above shows that TensorFlow performs better in benchmarks. Google's library twice faster than Torch or even C++ based Caffe.

There is one more test [72], where the running time of various recurrent neural networks on different deep-learning libraries are compared. Table 9 illustrates the speed of feeding training samples in second trough the network: forward and backward. The best pairs of values marked in bold.

Model: A single LSTM layer

—nn.SeqLSTM for Torch (updated version as of Nov 18, 2016);

—tf.nn.rnn_cell.LSTMCell for TensorFlow 0.11;

—scan from Theano 0.8.2.

Machine: Pascal Titan X + Cuda 8 with cudNN 5.1 drivers

As it seen from Table 8, TensorFlow shows the best results in benchmarking.

Thus, we can say, that TensorFlow is the fastest deep learning tool.

Table 7. Convolution networks benchmarks

library	class	time (ms)	forward (ms)	backward (ms)
AlexNet [68] – Input 128 × 3 × 224 × 224				
TensorFlow	conv2d	**81**	**26**	**55**
Torch-7	SpatialConvolutionMM	342	132	210
Caffe	ConvolutionLayer	324	121	203
Overfeat [69] – Input 128 × 3 × 231 × 231				
TensorFlow	conv2d	**279**	**90**	**189**
Torch-7	SpatialConvolutionMM	878	379	499
Caffe	ConvolutionLayer	823	355	468
OxfordNet [70] – Input 64 × 3 × 224 × 224				
TensorFlow	conv2d	**540**	**158**	**382**
Torch-7	SpatialConvolutionMM	1105	350	755
Caffe	ConvolutionLayer	1068	323	745
GoogleNet V1 [71] – Input 128 × 3 × 224 × 224				
TensorFlow	conv2d	**445**	**135**	**310**
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	462	112	349
Caffe	ConvolutionLayer	1935	786	1148

Table 8. Recurrent networks benchmarks

Library	Sequence length	Batch size	Hidden layer size	Forward samples/sec	Forward + backward samples/sec
Torch	30	32	128	22110	4849
TensorFlow	30	32	128	**2778**	**1410**
Theano	30	32	128	15462	5440
Torch	30	32	512	6722	1582
TensorFlow	30	32	512	**2155**	**1285**
Theano	30	32	512	7127	1874
Torch	30	128	128	74897	15131
TensorFlow	30	128	128	**8656**	**5411**
Theano	30	128	128	53953	14491
Torch	60	32	128	11126	2364
TensorFlow	60	32	128	**1353**	**879**
Theano	60	32	128	5538	3092
Torch	60	32	512	3344	785
TensorFlow	60	32	512	**1272**	**811**
Theano	60	32	512	3951	1060
Torch	60	128	1024	5248	1543
TensorFlow	60	128	1024	**2695**	**1423**
Theano	60	128	1024	4366	1409

We select Python as a programming language for our research because it is easy to understand, supports multiple systems and platforms, object oriented programming-driven, has a plethora of frameworks and modules and has a large number of resources available for it.

With such a consideration we compare how easy to program in two deep learning toolkits, TensorFlow and Theano. They both are flexible and support major types of layers, activation functions and add-on libraries. So we can compare how many lines of code you should write to obtain the same things in each of those libraries [73].

Variables initialization:

```
1.  # TensorFlow
2.  b = tf.Variable(tf.zeros([n_out]))
3.  W = tf.Variable(tf.random_uniform([n_in, n_out], -1.0, 1.0))
4.  y = tf.matmul(W, x_data) + b
5.
6.  # Theano
7.  X = T.matrix()
8.  Y = T.vector()
9.  b = theano.shared(numpy.random.uniform(-1, 1), name="b")
10. W = theano.shared(numpy.random.uniform(1.0, 1.0, (n_in, n_out)),
name="W")
11. y = W.dot(X) + b
```

TensorFlow doesn't require any special treatment of the x and y variables: they just exist in their native forms.

Define loss function and optimizer:

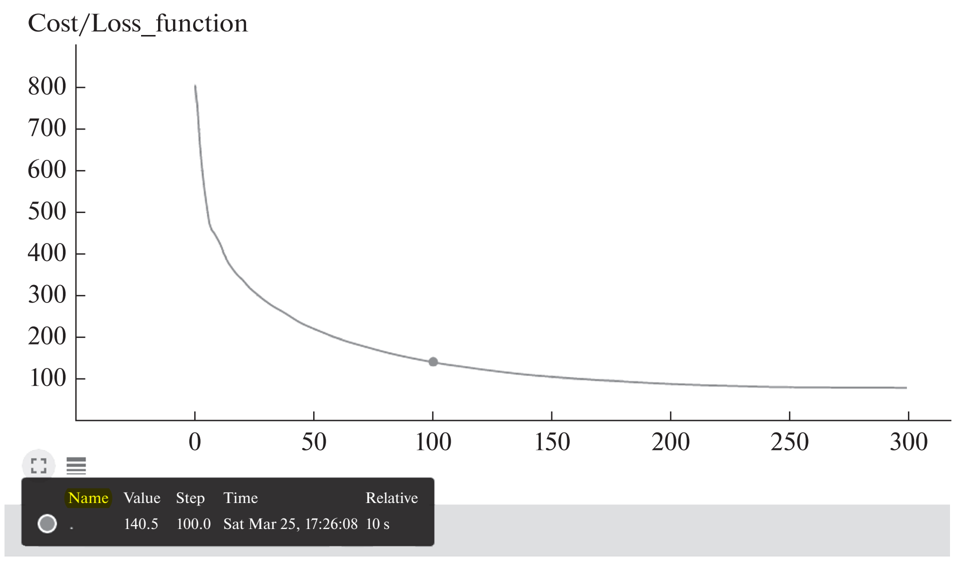
```
1.  # Tensorflow
2.  loss = tf.reduce_mean(tf.square(y - y_data)) # (1)
3.  optimizer = tf.train.GradientDescentOptimizer(0.5) # (2)
4.  train = optimizer.minimize(loss) # (3)
5.
6.  # Theano
7.  cost = T.mean(T.sqr(y - Y)) # (1)
8.  gradientW = T.grad(cost=cost, wrt=W) # (2)
9.  gradientB = T.grad(cost=cost, wrt=b) # (2)
10. updates = [[W, W - gradientW * 0.5], [b, b - gradientB * 0.5]] # (2)
11. train = theano.function(inputs=[X, Y], outputs=cost,
updates=updates, allow_input_downcast=True) #(3)
```

Lines marked as (2, 7) specifying the loss function, in this case mean squared error. The (3, 8-10) marker denotes the gradients definition. TensorFlow gives access to lots of good optimizers out of the box (including gradient descent and Adadelta). Theano makes one to do all the hard work. This is both a good and bad thing: it offers ultimate control, but also affects code comprehension and increases verification effort. Lines with mention #(3) are the training function initialization.

And to train these models we should build a graph and start epochs. The code below does this.

Building graph and training:

```
1.  # TensorFlow
2.  init = tf.initialize_all_variables()
3.
4.  sess = tf.Session()
5.  sess.run(init)
6.  for step in xrange(0, 201):
7.      sess.run(train)
8.
9.  # Theano
10. for i in xrange(0, 201):
```



Name	Value	Step	Time	Relative
.	140.5	100.0	Sat Mar 25, 17:26:08	10 s

Fig. 23. TensorBoard visualization of descending the autoencoder loss function.

```
        train(x_data, y_data)
        print W.get_value(), b.get_value()
```

Theano's training is intuitive, but TensorFlow's philosophy of encapsulating the graph execution in a Session object does feel conceptually clearer than Theano's. TensorFlow protects the user from writing plenty of code defining input tensors, updating gradients and its code is more compact then Theano's one. In addition, Google deep learning library has many debugging tools and can visualize computation graph with a suite of visualization tools called TensorBoard [75]. You can use TensorBoard to visualize your TensorFlow graph, plot quantitative metrics about the execution of your graph, and show additional data like images that pass through it.

Firstly, before running tensorboard you should save needful variables, update logs and the graph session. For example, to save how the loss function is decreased during training, we need add summary scalar and give it a name:

```
# %% cost function measures pixel-wise difference
with tf.name_scope('cost'):
    cost = tf.reduce_sum(tf.square(y - corr))
    tf.summary.scalar('Loss_function', cost)
```

Then we should create session and merge all summaries in one variable. The code below does it:

```
# We create a session to use the graph
sess = tf.Session()
```

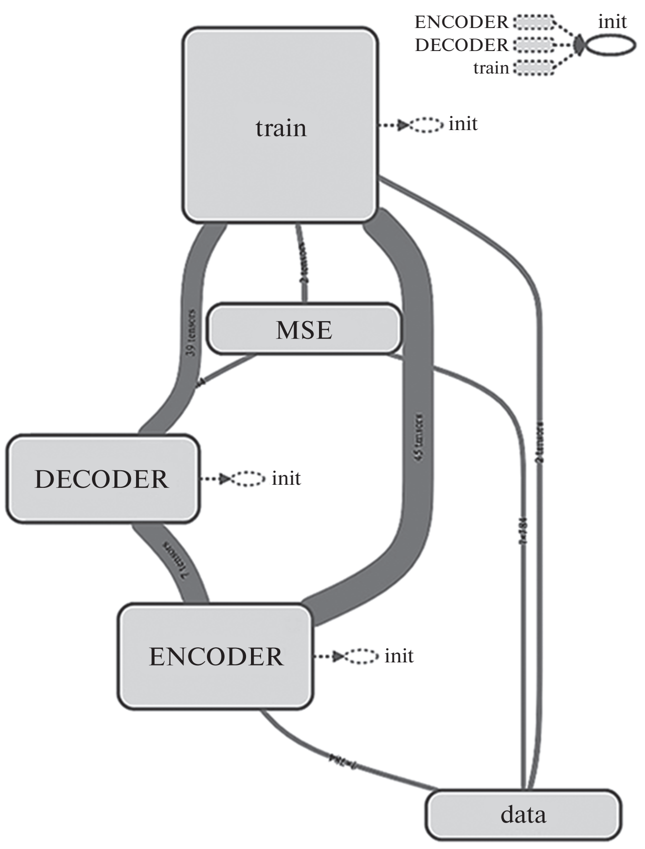


Fig. 24. Computation graph of the denoising autoencoder.

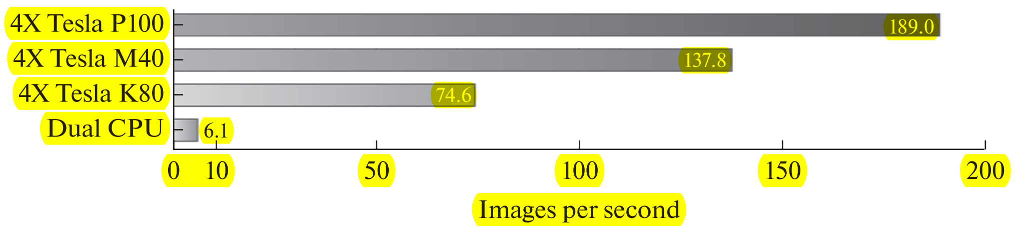


Fig. 25. Images produced per second with different devices. *Dual CPU System: Dual Intel E5-2699 v4 @ 3.6 GHz, 64 batch size, cuDNN v5.1.*

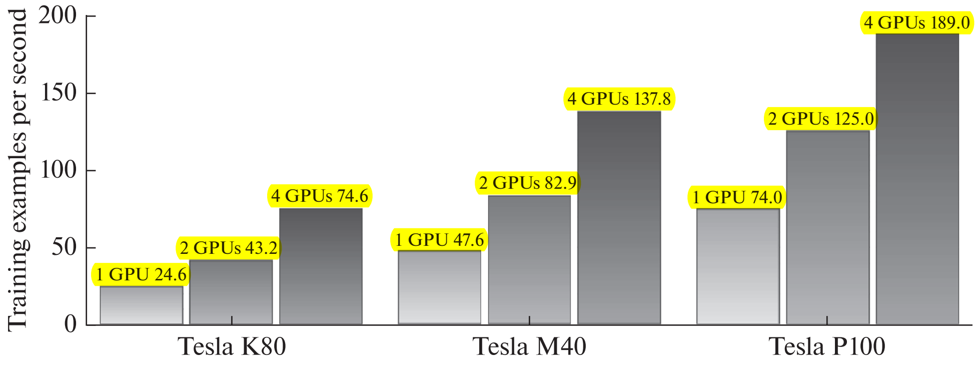


Fig. 26. TensorFlow Inception v3 Training Scalable Performance on multi-GPU node.

```
# Merge all summaries
merged = tf.summary.merge_all()

# Create a log writer
writer = tf.summary.FileWriter('./tmp/logs', sess.graph)

# Initialize all variables and run session
sess.run(tf.global_variables_initializer())
```

Eventually we should update our summary variable for each training step by adding new value in training loop. Then after training close the writer.

```
for i in range(epochs_n):
    for batch_i in range(faces['train_data'].shape[0] // batch_size):
        batch_xs = faces['train_data'][batch_i*batch_size:batch_i*batch_size+batch_size]
        _, summary = sess.run([optimizer, merged], feed_dict={ae['x']: batch_xs, ae['corr']:batch_xs})
    writer.add_summary(summary, i)

writer.close()
```

Finally, to visualize the loss function decreasing in the training process, type in terminal (linux) or command line (windows):

```
tensorboard –logdir=./tmp/logs
```

Then the terminal will print the message with the address, on which it is necessary to navigate to open TensorBoard starting page. Click the menu item "SCALARS" to see all scalar plot. As an example, of the loss function descending for our autoencoder is shown in Fig. 23. If we put the cursor on a certain point in the graph, we will see on the bottom left details about this point: point value, current training step, timestamp and how much time passed before this point occurring.

If we want to view the graph built with all nodes (in our example it is the autoassociative neural network graph) we must click on the menu button "GRAPHS" and then we will see the main graph and its auxiliary nodes (Fig. 24). User may open interesting node and also download a png image of graph.

Thus, TensorFlow has many advantages: it supports two main operation systems – Linux and Windows; based on great variety of programming languages such as Python, C++, Java, Go; it is very fast, twice faster than the other deep learning toolkits; it is more flexible because of its high-level API (version 1.0); it can create a visualization of session graph and training process; it has many debugging tools and you do not need to write long lines of code.

It is the reason to use TensorFlow in our work. All neural network models, except Deep Belief was coded with Google library. Deep Belief Network was created with the help of Nolearn package which is a high-level superstructure of Theano, because building Deep Belief Network from scratch is unnecessary slow, while Nolearn has a prepared function for it.

7.2. Speed up Training Process with GPU Graphics Cards and Cloud Service

Deep learning means deep neural networks with many layers and thousands or even millions of neurons. With growing interest to deep techniques, the model's capacity is increasing extremely fast and go deeper and deeper. For instance, in networks for natural language processing with their gate system even one neuron includes 5 activations and 5 gradients respectively. To train such a giant networks, hundreds of hours could be needed on ordinary computer. When in our work on the notebook Dell Vostro 5480, we
1 train deep belief network, its evolution is rather complicated and one minibatch training epoch with pre-training by contrastive divergence takes 28 seconds on the MNIST dataset. Therefore, there is a great need in speeding up network evolution by switching from CPU to GPU computations.

TensorFlow and Theano frameworks operate with large matrix multiplications and can do calculations in parallel from box. You only need to install required drivers and define special flags. In Theano there is a THEANO_FLAGS variable in which you can specify device as "gpu0" and floatX as "float32" it means that system will run program with GPU support and perform float point computations. TensorFlow has two repositories with regular CPU version of library and GPU. To install GPU version, user must add "gpu" postfix after the name of package, like in this example:

```
pip install --upgrade tensorflow-gpu
```

To switch back from GPU to CPU mode, type the code below and insert in it your computation part.

```
with tf.device('/cpu:0'):
  # your computations.
```

Figure 25 illustrates how many images can be processed per second with different hardware. As benchmark Nvidia's test [74] was used on Inception v3 model. The graphs below shows the expected system performances with a CPU and 1, 2, and 4 Tesla GPUs per node.

It is clear that GPU performs faster for one-two orders of magnitude. Worth to note, that TensorFlow supports multi-GPU calculations. The plot in Fig. 26 shows the TensorFlow training scalable performance on multi-GPU node.

The application of Tensorflow and other libraries was well facilitated by a multicore computational system and multiprocessor graphics cards (GPUs) via the cloud service facilities provided by heterogeneous computing cluster HybriLIT [76] of the Joint Institute for Nuclear Research in Dubna, Russia.

HybriLIT cluster has seven computational nodes, named Supermicro SuperBlade Chassis, with GPU units: Nvidia Tesla K40, K80 and K20X. Also, JINRs cluster includes Dell PowerEdge block at the disposal of which there are four another K80s [76]. Besides, today the system obtains a new hardware of Nvidia Tesla M60 model.

For our experiments we used a virtual machine on HybriLIT cloud with Ubuntu 16.04 and single M60 graphic card as hardware for multiprocessing computations. Experiments showed a significant training speed boosting. Our virtual machine with Tesla M60 outperforms the dual-CPU (Intel Core i3-4500U) in speed more than 6 times. For comparison, one epoch of fitting DAEN with 3 hidden layers and LeakyReLU as activation on MNIST data with batch size equals to 64 resolves only 3 seconds using M60 GPU-card versus 20 seconds with dual-CPU Core i3-4500U.

7. CONCLUSION

In our work, we propose a deep learning model for the image classification, which consists of two neural networks: an autoencoder for preliminary input compression and an estimator for classifying images of two famous benchmark data sets MNIST and FERET.

A comparative study of three neural classifiers: a perceptron with one hidden layer, DBN and DNN was accomplished on images of both data sets being previously compressed by the autoencoder. Comparison of investigated estimators shows that DBN provides the highest classification efficiency, but DNN accomplishes faster training time having competitive efficiency. Both, DBN and DNN perform much better than perceptron with one hidden layer.

It is worth to note, that the main efforts to deal with deep learning networks were spent to quite a painstaking work for optimizing structures of those networks and, their components, as activation functions, weights, as well as the procedures of minimizing their loss function to improve their performances. So, we have to make the special study to find optimal numbers of layers and hidden neurons, to apply tied weights and SGD Adam method, to choose the most suitable activation functions and weight normalization in order to improve performance of all used neural nets and speed up their learning time. Additionally we observe such the remarkable feature of deep autoencoders as their ability for denoising images after being trained on an image set corrupted in a special way. Denoising results of such autoencoder classification efficiency on FERET images are really striking: images corrupted with 50 percent of noise were classified with efficiency greater than 90 percent.

In our study we speed up significantly our deep network calculations by using Tensorflow library which realization was well facilitated via a multicore computational system and multiprocessor graphics cards. Our access to such a system was provided by the cloud service facilities at the JINR heterogeneous computing cluster HybriLIT.

REFERENCES

1. Ososkov, G., Robust tracking by cellular automata and neural network with non-local weights, Rogers, S.K. and Ruck, D.W., Ed., *Appl. Sci. Artificial Neural Networks, Proc. SPIE 2492*, 1995, p. 1180A1192.
2. Lebedev, S., Hoehne, C., Lebedev, A., and Ososkov, G., Electron reconstruction and identication capabilities of the CBM experiment at FAIR, *J. Phys.: Conf. Ser.*, 2012, vol. 396, p. 022029.
3. Baginyan, S. et al., Tracking by modified rotor model of neural network, *Comput. Phys. Commun.*, 1994, vol. 79, p. 95.
4. Galkin, I. et al., Feedback neural networks for ARTIST ionogram processing, *Radio Sci.*, 1996, vol. 31, no. 5, pp. 1119–1128.
5. TMVA Users Guide. http://tmva.sf.net.
6. Kisel, I., Neskoromnyi, V., and Ososkov, G., Applications of neural networks in experimental physics, *Phys. Part. Nucl.*, 1993, vol. 24, no. 6, pp. 657–676.
7. Peterson, C., Track finding with neural networks, *Nucl. Instr. Meth. A*, 1986, vol. 279, p. 537.
8. Denby, B., Neural networks and cellular automata in experimental high energy physics, *Comput. Phys. Commun.*, 1988, vol. 49, p. 429.
9. Peterson, C. and Hartman, E., Explorations of the mean field theory learning algorithm, *Neural Networks*, 1989, vol. 2, pp. 475–494.
10. Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P., Optimization by simulated annealing, *Science*, 1983, vol. 22, p. 671.
11. Ososkov, G.A., Polanski, A., and Puzynin, I.V., Current methods of processing experimental data in high energy physics, *Phys. Part. Nucl.*, 2002, vol. 33, pp. 347–382.
12. Rumelhart, D., Hinton, G., and Williams, R., Learning representations by back-propagating errors, *Nature*, 1986, vol. 323, pp. 533–536.
13. Gyulassy, M. and Harlander, M., Elastic tracking and neural network algorithms for complex pattern recognition, *Comput. Phys. Commun.*, 1991, vol. 66, pp. 31–46.
14. Ohlsson, M., Petereson, C., and Yuille, A.L., Track finding with deformable template – The elastic arm approach, *Comput. Phys. Commun.*, 1992, vol. 71, p. 77.
15. Hinton, G.E. and Salakhutdinov, R., Reducing the dimensionality of data with neural networks, *Science*, 2006, vol. 313, pp. 504–507.
16. Bengio, Y., *Learning Deep Architectures for AI*, Now Publishers Inc., 2009, p. 144.
17. LeCun, Y., Bottou, L., Orr, G., and Müller, K.-R., Efficient backprop, Orr, G. and Müller, K.-R., Eds., *Neural Networks: Tricks Trade*, 1998a, pp. 9–50.

18. Krizhevsky, A., Learning Multiple Layers of Features from Tiny Images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.
19. Felleman, D.J. and van Essen, D.C., Distributed hierarchical processing in the primate cerebral cortex, *Cereb. Cortex*, 1991, vol. 1, pp. 1–47. doi 10.1093/cercor/1.1.1-a
20. Simon Thorpe, Denis Fize, Catherine Marlot, et al., Speed of processing in the human visual system, *Nature*, 1996, vol. 381, no. 6582, pp. 520–522.
21. Lennie, P., The cost of cortical computation, *Curr. Biol.*, 2003, vol. 13, no. 6, pp. 493–497.
22. Glorot, X. and Bengio, Y., Understanding the difficulty of training deep feedforward neural networks, *Aistats*, 2010, vol. 9, pp. 249–256.
23. Hinton, G., Osindero, S., and Yee-Whye Teh, A fast learning algorithm for deep belief nets, *Neural Comput.*, 2006, vol. 18, pp. 1527–1554.
24. Smolensky, P., Information processing in dynamical systems: Foundations of harmony theory, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: *Foundations*, Rumelhart, D.E., McClelland, J.L., Eds., MIT Press, 1986, pp. 194–281.
25. Hinton, G.E., A Practical Guide to training restricted Boltzmann machines, *Tech. Rep. 2010-000*, Toronto: Machine Learning Group, University of Toronto, 2010.
26. Diaconis, P., The Marcov chain Monte Carlo revolution, *Bull. Am. Math. Soc.*, 2009, vol. 46, no. 2, pp. 179–205.
27. Carreira-Perpinan, M.A. and Hinton, G., On contrastive divergence learning, *Aistats*, *Citeseer*, 2005, vol. 10, pp. 33–40.
28. Kramer, M., Nonlinear principal component analysis using autoassociative neural networks, *AIChE J.*, 1991, vol. 37, no. 2, pp. 161–310.
29. Baldi, P. and Hornik, K., Neural networks and principal components analysis: Learning from examples without local minima, *Neural Networks*, 1989, vol. 2, pp. 53–58.
30. Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y., Efficient Learning of Sparse Representations with an Energy-Based Model. http://yann.lecun.com/exdb/publis/pdf/ranzato-06.pdf.
31. Bengio, Y. and LeCun, Y., Scaling learning algorithms towards AI, in *Large Scale Kernel Machines*, Bottou, L., Chapelle, O., DeCoste, D., and Weston, J., Eds., MIT Press, 2007.
32. Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P., Extracting and Composing Robust Features with Denoising Autoencoders, 2008. http://machinelearning.org/archive/icml2008/papers/592.pdf.
33. Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P., Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion, *J. Mach. Learning Res.*, 2010, vol. 11, pp. 3371–3408.
34. Masci, J., Meier, U., Ciresan, D., and Schmidhuber, J., Stacked convolutional auto-encoders for hierarchical feature extraction, *ICANN 2011*, Honkela, T. et al., Ed., Springer-Verlag: Berlin Heidelberg, 2011, Part I, LNCS 6791, pp. 52–59. https://pdfs.semanticscholar.org/1c6d/990c80e60aa0b0059415444cdf94b3574f0f.pdf.
35. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P., Gradient-based learning applied to document recognition, *Proc. IEEE*, 1998, vol. 86, no. 11, pp. 2278–2324.
36. Simonyan, K. and Zisserman, A., Very Deep Convolutional Networks for Large-Scale Image Recognition. https://arxiv.org/pdf/1409.1556.pdf.
37. Sepp, H. and Schmidhuber, J., Long short-term memory, *Neural Comput.*, 1997, vol. 9.8, pp. 1735–1780.
38. Schmidhuber, J., Deep learning in neural networks: An overview, *Neural Networks*, 2015, vol. 61, pp. 85–117.
39. Chung, J., Gulcehre, C., Cho, K., and Bengio, Y., Gated feedback recurrent neural networks, arXiv preprint, 2015. https://arxiv.org/pdf/1502.02367.pdf.
40. Sutton, R. and Barto, A., *Reinforcement Learning: An Introduction*, MIT Press, 1998.
41. Mnih, V. et al., Human-level control through deep reinforcement learning, *Nature*, 2015, vol. 518, pp. 529–533.
42. He, K., Zhang, X., Ren, Sh., and Sun, J., Identity mappings in deep residual networks. https://arxiv.org/pdf/1603.05027.pdf.
43. Mao, X.-J., Shen, C., and Yang, Y.-B., Image restoration using convolutional auto-encoders with symmetric skip connections. https://arxiv.org/pdf/1606.08921.pdf.
44. Phillips, A., Moon, H., Rauss, P., and Rizvi, S., The FERET evaluation methodology for face recognition algorithms, *IEEE Trans. Pattern Analysis Mach. Intelligence*, 2000, vol. 22, no. 10.
45. MNIST. https://www.nist.gov/sites/default/files/documents/srd/nistsd19.pdf.
46. Arlot, S., A survey of cross-validation procedures for model selection, *Statistics Surv.*, 2010, vol. 4, pp. 40–79.
47. Xu, B., Wang, N., Chen, T., and Li, M., Empirical evaluation of rectified activations in convolutional network. https://arxiv.org/abs/1505.00853.
48. Hua, K.L., Hsu, C.H., Hidayati, S.C., Cheng, W.H., and Chen, Y.J., Computer-aided classification of lung nodules on computed tomography images via deep learning technique, *OncoTargets Therapy*, 2014, vol. 8, pp. 2015–2022.

49. Bengio, Y., Courville, A., and Vincent, P., Representation learning: A review and new perspectives. https://arxiv.org/pdf/1206.5538.pdf.

50. Sutskever, I., Training Recurrent Neural Networks, *PhD Thesis*. http://www.cs.utoronto.ca/~ilya/pubs/ilya_-sutskever_phd_thesis.pdf.

51. Nair, V. and Hinton, G., Rectified linear units improve restricted Boltzmann machines, *Proc. 27th Int. Conference on Machine Learning*, Furnkranz, J. and Joachims, Th., Eds., Haifa, Israel, 2010, pp. 807–814C.

52. Dugas, Y., Bengio, F., Bélisle, C., and Nadeau, R., Garcia, Incorporating second-order functional knowledge for better option pricing, *NIPS'2000*, 2001. http://papers.nips.cc/paper/1920-incorporating-second-order-functional-knowledge-for-better-option-pricing.pdf.

53. Zuriani, M. and Yuhanis, Y., A comparision of normalization techiques in predicting dengue outbreak, *2010 International Conference on Business and Economics Research*, 2011, vol. 1, Kuala Lumpur, Malaysia: IACSIT Press, pp. 345–349.

54. Glorot, X., Bordes, A., and Bengio, Y., Deep sparse rectifier neural networks, *Aistats*, 2011, vol. 15, no. 106, pp. 315–323.

55. Kingma, D. and Ba, J.L., ADAM: A method for stochastic optimization, arXiv preprint arXiv:1412.6980, 2014. https://arxiv.org/abs/1412.6980.

56. Bengio, Y., *Practical recommendations for gradient-based training of deep architectures, Neural Networks: Tricks of the Trade*, Springer, 2012, pp. 437–478.

57. Duchi, J., Hazan, E., and Singer, Y., Adaptive subgradient methods for online learning and stochastic optimization, *J. Mach. Learning Res.*, 2011, vol. 12, no. Jul, pp. 2121–2159.

2 58. Srivastava, N. et al., Dropout: A simple way to prevent neural networks from overfitting, *J. Mach. Learning Res.*, 2014, vol. 15, pp. 1929–1958.

59. Stone, M., Cross-validatory choice and assessment of statistical predictions, *J. Royal Statistical Soc., Ser. B: Methodological*, 1974, pp. 111–147.

60. Cawley, G.C. and Talbot, N.L.C., On over-fitting in model selection and subsequent selection bias in performance evaluation, *J. Mach. Learning Res.*, 2010, vol. 11, pp. 2079–2107.

61. Tuning the hyper-parameters of an estimator. http://scikit-learn.org/stable/modules/grid_search.html#grid-search.

62. Deep Learning Frameworks. https://www.microway.com/hpc-tech-tips/deep-learning-frameworks-survey-tensorflow-torch-theano-caffe-neon-ibm-machine-learning-stack/.

63. Tensorflow. https://www.tensorflow.org/.

64. Torch. http://torch.ch/.

65. Theano at a Glance. http://deeplearning.net/software/theano/introduction.html. Cited March 5. 2017.

66. Jia, Y. et al., Caffe: Convolutional architecture for fast feature embedding, *Proceedings of the 22nd ACM International Conference on Multimedia*, ACM, 2014, pp. 675–678.

67. Easy benchmarking of all publicly accessible implementations of convents. https://github.com/soumith/convnet-benchmarks.

68. Cuda-convnet2 project. https://github.com/akrizhevsky/cuda-convnet2.

69. Sermanet, P. et al., Overfeat: Integrated recognition, localization and detection using convolutional networks, arXiv preprint arXiv:1312.6229, 2013.

70. Simonyan, K. and Zisserman, A., Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556, 2014.

71. Szegedy, C. et al., Going deeper with convolutions, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

72. Torch vs. TensorFlow vs. Theano. http://www.ccri.com/2016/12/09/torch-vs-tensorflow-vs-theano/.

73. Porting a model to TensorFlow. https://medium.com/@sentimentron/faceoff-theano-vs-tensorflow-e25648c31800.

74. Speed up training with GPU-accelerated TensorFlow. http://www.nvidia.com/object/gpu-accelerated-applications-tensorflow-benchmarks.html.

75. TensorBoard: Visualizing Learning. https://www.tensorflow.org/get_started/summaries_and_tensorboard.

76. Web-portal of HybriLIT JINR computation facility. http://hybrilit.jinr.ru.

SPELL: 1. pretraining, 2. overfitting