AALTO UNIVERSITY
SCHOOL OF SCIENCE

Tuomas Nikoskinen

FROM NEURAL NETWORKS TO DEEP NEURAL NETWORKS

Mat-2.4108 Independent Research Project in Applied Mathematics

Espoo 2015

Supervisor:

Prof. Ahti Salo

Instructor:

Prof. Ahti Salo

Aalto University
School of Science

AALTO UNIVERSITY SCHOOL OF SCIENCE P.O. Box 1100, FI-00076 AALTO http://www.aalto.fi	ABSTRACT

Author:	Tuomas Nikoskinen		
Title:	From Neural Networks to Deep Neural Networks		
Degree programme:	Engineering Physics and Mathematics		
Major subject:	Systems and Operations Research	Minor subject:	Computational Science and Engineering
Chair (code):	Mat-2		
Supervisor:	Prof. Ahti Salo		
Instructor:	M.Sc. Ahti Salo		

Neural networks are based on the idea of mathematically representing information processing in human brains. They imitate the structure of brain by consisting of fundamental computing units, perceptrons, assembled in a connected network. By introducing nonlinear transformations to the flow of information between the layers of perceptrons, neural networks can model nonlinear phenomena.

The overarching limitation of classical multilayer perceptron (MLP) neural networks is that some complex, nonlinear functions cannot be efficiently represented by architectures that are too shallow, that is, have too few hidden layers and thus too few levels of nonlinear transformations. Consequently, there has been a desire to build neural networks with a deep architecture, but the results have been poor until the shift of paradigm in training deep neural networks.

Deep neural networks (DNN) have the exact same structure as classical MLP neural networks, except that the number of hidden layers is greater so that they can be considered deep. DNNs differ from MLP neural networks in the training phase. Instead of directly learning a DNN, a generative deep belief network (DBN) is trained first and then transformed into a DNN. Properties of DBNs are easily confused with DNN properties due to the critical role of DBNs in the DNN training phase.

This study presents and evaluates the principles behind the classical MLP neural networks and the modern deep neural networks. It also reviews the differences and similarities between the two.

Date: March 19, 2015	Language: English	Number of pages: 23
Keywords:	Deep neural network (DNN), Deep belief network (DBN), Restricted Bolzmann machine (RBM), MLP neural network, Deep learning	

Contents

Symbols and Abbreviations

Matrices are capitalized and vectors are in bold type.

Operators and miscellaneous notation

$1 : n$	$1, \ldots, n$
$x_{1:d}$	Scalars $x_1, x_2, \ldots, x_d$
$\mathbf{x}$	Vector $\mathbf{x}$
$\mathbf{A}^\top$	Transpose of matrix $\mathbf{A}$
$\mathbf{I}$	Identity matrix
$\mathrm{E}(\mathbf{b})$	Expectation of $\mathbf{b}$
$\mathcal{N}(\boldsymbol{m}, \boldsymbol{\Sigma})$	Gaussian distribution with mean $\mathbf{m}$ and covariance $\boldsymbol{\Sigma}$
$p(\mathbf{y} \mid \mathbf{x})$	Conditional probability density of $\mathbf{y}$ given $\mathbf{x}$
$\mathbb{R}$	The real numbers

Abbreviations

DBN	Deep belief network
DNN	Deep neural network
MLP	Multilayer perceptron
MAP	Maximum a posteriori
MSE	Mean-square error
RBM	Restricted boltzmann machine

1 Introduction

The origins of neural networks date back to attempts to find a mathematical representation for information processing in human brains. (McCulloch and Pitts, 1943). The brain far exceeds the capabilities of modern engineering devices in many information processing domains including learning, vision and speech recognition. Thus, a natural approach to developing computational methodology for information processing is to use the characteristics of brain as a starting point. The brain consists a huge number (around 10^{11} according to Azevedo *et al.*, 2009) of processing units, neurons, that operate in parallel and are highly inter-connected through synaptic connections. Neural networks are designed after these same characteristics: the computational power of neural networks is based on a large number connected of processing units operating in parallel.

Artificial neural networks have been of scientific interest ever since Rosenblatt (1958) first introduced the perceptron. Perceptrons are used as the fundamental processing units in standard neural networks. Research on neural networks did stagnate after Minsky and Papert (1969) criticized perceptrons of their limited capabilities, but, was rekindled in 1980's through the inventions of multilayer perceptron (MLP) neural networks (Rumelhart, David E and McClelland, James L and PDP Research Group, 1986) and the famous backpropagation algorithm for training them (Rumelhart *et al.*, 1986). Since then, various kinds of neural networks or variants of have been proposed for machine learning purposes (see, e.g. Alpaydin, 2004; Bishop, 2006). In the following, we refer to these models as having a *shallow* architecture to make the distinction between *deep* models having a deep architecture. The depth of a model architecture refers to the number of levels of nonlinear operations in the function learned.

The overarching limitation of shallow models is that some complex, highly nonlinear functions cannot be efficiently represented by architectures that are too shallow (Bengio, 2009). In other words, if a function can be efficiently represented by a deep architecture, it might require an exponential number of computational units to be represented ,by a more shallow architecture. This suggests that a deep model might be required to successfully model some complex functions. In practice for example, a deep neural network (DNN)

clearly outperforms shallow models in a handwritten digit classification task with the large well-known MNIST dataset (Hinton *et al.*, 2006).

For a long time, there has been a desire to train neural networks with a deep architecture (Utgoff and Stracuzzi, 2002; Bengio *et al.*, 2007) but the results have been poor (Glorot and Bengio, 2010). In general, the challenge in training deep neural networks in the classic way—by conducting the training in a supervised manner using a gradient-based optimization starting from a random initialization—is that the parameter estimation iteration often appears to get stuck in poor solutions and thus the model produces poor results. The breakthrough was achieved when Hinton *et al.* (2006) proposed a new approach to training deep neural networks, which led to a shift of paradigm in approaching the training of deep models. In short, the idea in the new approach is to first pre-train a generative deep belief network (DBN) and then to transform the pre-trained DBN to a discriminative deep neural network. Although this may appear straightforward, the new training approach consists of many new computational elements, such as restricted Boltzmann machines (RBM) that are not familiar from the context of classical neural network theory.

Since 2006, deep neural networks and deep models in general have become increasingly popular and they have successfully been applied to a broad range of machine learning tasks (see, e.g., Bengio, 2009). Although the popularity is much due to the success of these methods, assumedly the human-like aim of deep learning methods to learn feature hierarchies, where higher lever features are composed of lower level features, plays a part in the popularity as well.

In this study, we review the main ideas and principles behind the classical MLP neural networks and the modern deep neural networks. We conclude by discussing the differences and similarities between the two.

2 Neural Networks

Multilayer perceptron (MLP) neural networks are built by assembling the fundamental computing units, perceptrons (Rosenblatt, 1958), to a network in parallel and layer-wise fashion. Perceptrons as such are linear computing units, but by introducing nonlinear transformation to the intermediate perceptron layers, the hidden layers, neural networks can also be applied to nonlinear problems. The objective in supervised training of neural networks is to estimate the model free parameters, the network connection weights, to mimic patterns in the training input/output data set. Backpropagation algorithm is the original and most basic method for neural network training.

2.1 Perceptrons

Perceptrons are the fundamental processing units in a neural network. A perceptron computes an output $y \in \mathbb{R}$ from inputs $x_i \in \mathbb{R}, k = 0, 1, \ldots, d$ where each input unit x_i is associated with a corresponding connection weight $w_i \in \mathbb{R}, i = 0, 1, \ldots, d$. The architecture of a single perceptron is illustrated in Figure 1a.

A perceptron is a linear processing unit that transforms the input units x_i to an output y with a linear transformation

$$y(x_{1:d}, w_{1:d}) = \sum_{i=1}^{d} w_i x_i + w_0, \tag{1}$$

where $x_{1:d} = x_1, x_2, \ldots, x_d$ and x_0 is a bias unit that generalizes the model and is set to constant $x_0 = 1$. We can write the perceptron output equation (1) in compact vector form of $y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\mathsf{T}\mathbf{x}$, where the input units and corresponding weights are represented by vectors $\mathbf{x} = [1, x_1, \ldots, x_d]^\mathsf{T}$ and $\mathbf{w} = [w_0, w_1, \ldots, w_d]^\mathsf{T}$ respectively. The output equation (1) defines a line when the input is one dimensional and a hyperplane in a multidimensional case.

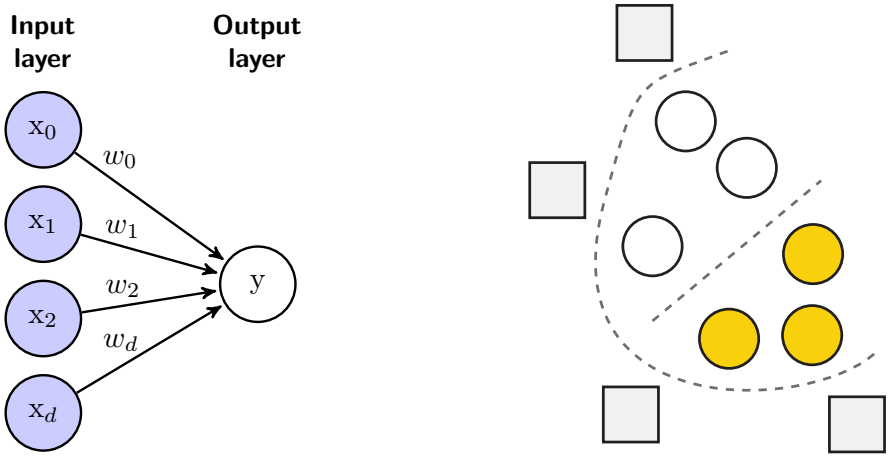
As such, perceptrons can be applied to linear regression and binary classification. The former application corresponds to the case where a set of N input/output pairs is given $D = \{\mathbf{x}^{(n)} \in \mathbb{R}^d, d^{(n)} \in \mathbb{R} \mid n = 1, 2, \ldots, N\}$ and

the objective is to choose $\mathbf{w}$ to best fit the model outputs $y(\mathbf{x}, \mathbf{w})$ against the known outputs $d^{(n)}$ given the inputs $\mathbf{x}^{(n)}$. In binary classification the given outputs $d^{(n)}$ belong to two distinct sets (classes) and thus can be treated as binary variables $d^{(n)} = \{0, 1\}\ \forall n$. In this case the objective is to choose $\mathbf{w}$ so that the model $y(\mathbf{x}, \mathbf{w})$ defines a line or hyperplane that separates the different sets of binary outputs from each other. This is illustrated in Figure 1b, where a perceptron could separate the filled and empty circles—two distinct classes of outputs—from each other, because the circles are linearly separable, but could not separate the squares from the circles, because the solution (separating line) in this case is nonlinear.

A perceptron can be used as a binary classifier by applying the Heaviside step function ϕ on the perceptron output equation (1)

$$\phi(y) = \begin{cases} 1, & \text{if } y(\mathbf{x}, \mathbf{w}) > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

and by classifying the output y depending on whether $\phi(y)$ is 1 or 0.



(a) Perceptron architecture

(b) Linear and nonlinear separators (dashed lines)

Figure 1: Panel (a) shows the architecture of a single perceptron. The inputs $x_i, i = 0, 1, 2, \ldots, d$ are connected to the output y through directed connections with a weight $w_i, i = 0, 1, \ldots, d$ corresponding each input respectively. Panel (b) shows three arbitrary sets where the filled and empty circles are linearly separable where as the squares are nonlinearly separable from the circles. A perceptron could separate the two sets of circles from each other but not the squares from the circles.

2.2 Neural Networks—Networks of Perceptrons

Neural networks are networks of perceptrons. Perceptrons are typically assembled in parallel and in layers in a neural network, which is illustrated in Figure 2. Parallel perceptrons share the same input units and extend the computational capabilities of a single perceptron by allowing computation of a multidimensional output, that is $\mathbf{y} \in \mathbb{R}^m, m > 1$. The idea behind organizing perceptrons in layers is that the intermediate layers, often called hidden layers, introduce a nonlinear transformation to the information flowing through the network. A network of perceptrons with the addition of nonlinearity through the hidden layers makes it possible to apply neural networks to nonlinear regression and classification problems.

The architecture of a *multilayer perceptron* (MLP) network depicted in Figure 2 shares the same essential features with a single perceptron. The input units x_i are connected to the hidden units h_j through directed connections with weights w_{ij} and the hidden units are connected to the output units y_k in similar manner but with different weights v_{jk}. The weight subscripts indicate which units are connected, for example w_{ij} is the weight for the connection of input unit x_i and hidden unit h_j. The significant difference between a perceptron architecture and a multilayer perceptron network architecture is the nonlinearity of the hidden units' output. Each hidden unit is a perceptron

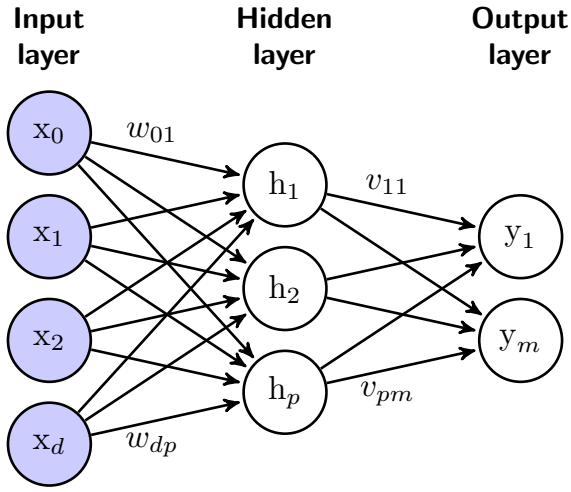


Figure 2: Architecture of a network of perceptrons (i.e. a neural network). Perceptrons are assembled in the network both in parallel sharing the same inputs and in layers taking the previous layer outputs as new inputs.

and applies the perceptron output equation (1) to its inputs. In addition to this, however, a hidden unit also applies a nonlinear transformation $\varphi(x)$ to the linear output it computes via perceptron output equation. Thus, the output from a hidden unit h_j is

$$h_j = \varphi(\mathbf{w}_j^\mathsf{T}\mathbf{x}), \tag{3}$$

where $\mathbf{x}$ is the augmented input vector and $\mathbf{w}_j = [w_{0j}, w_{1j}, \ldots, w_{dj}]^\mathsf{T}$ contains the weights connecting inputs to the hidden unit h_j.

The nonlinear transformation $\varphi(x)$ is often taken to be the sigmoid function

$$\varphi(x) = \frac{1}{1 + \exp(-x)}, \tag{4}$$

but for example a hyperbolic tangent function or a Gaussian are often used as well. Sigmoid, hyperbolic tangent or Gaussian are continuous and differentiable versions of the Heaviside step function (2) restricting the output to range $[-1, 1]$ or $[0, 1]$. The nonlinear transformation needs to be differentiable because gradient-based methods are used to train the network. Here, training refers to the procedure of choosing the weights $\mathbf{w}$ and $\mathbf{v}$ given the modeling objective and input/output dataset.

The output unit y_k of a neural network is a perceptron that applies the linear perceptron output equation to its inputs, that are the hidden units' outputs, that is

$$y_k(\mathbf{h}, \mathbf{v}_k) = \mathbf{v}_k^\mathsf{T}\mathbf{h}, \tag{5}$$

where $\mathbf{h} = [h_1, h_2, \ldots, h_p]^\mathsf{T}$ (p hidden units) and $\mathbf{v}_k = [v_{1k}, v_{2k}, \ldots, v_{pk}]^\mathsf{T}$ (p hidden units and thus p output connection weights). For classification purposes, the output equation (5) can be used by choosing the class label k^* that maximizes y_k over the output units $k = 1, 2, \ldots, m$. Posterior probabilities $p(y_k)$ for class labels are readily available through the use of softmax function

$$p(y_k) = \frac{\mathbf{v}_k^\mathsf{T}\mathbf{h}}{\sum_{k=1}^{m} \mathbf{v}_k^\mathsf{T}\mathbf{h}}. \tag{6}$$

Note that the neural network architecture is not limited to a single hidden layer. However, additional hidden layers introduce additional complexity to the neural network model, which makes it harder to analyze the network and

its behavior. Also, all input or hidden units do not need to be connected to all units on the next layer. If prior information is available about the modeled phenomena that implies local structures in the data, it may be sensible to introduce similar local structures to the network to account for this information.

2.3 Training Neural Networks

Training a neural network refers to the action of inferring or learning values for the model free parameters, weights of the hidden units $\mathbf{w}$ and weights of the outputs $\mathbf{v}$. In supervised learning there is available a set of known outputs $\mathbf{d}^{(n)}$ corresponding to inputs $\mathbf{x}^{(n)}$, a *training set*, and the objective of training is to choose appropriate weights given the training set.

A straightforward approach to supervised learning is to formulate training as an optimization problem

$$\mathbf{w}^*, \mathbf{v}^* = \arg\min_{\mathbf{w},\mathbf{v}} \xi(\mathbf{d}, \mathbf{y}), \tag{7}$$

where the objective is to find the weights $\mathbf{w}$ and $\mathbf{v}$ so that the error function ξ, that measures the discrepancy between known outputs and the model produced outputs, is minimized. A typical choice for error function in regression problems is the mean-square error (MSE)

$$\begin{aligned} \xi_{MSE}(\mathbf{d}, \mathbf{y}) &= \frac{1}{2}\mathrm{E}\left(\|\mathbf{d} - \mathbf{y}\|^2\right) \\ &= \frac{1}{2N}\sum_{n=1}^{N}\sum_{k=1}^{m}(d_k^{(n)} - y_k^{(n)})^2 \end{aligned} \tag{8}$$

and cross entropy in classification problems with more than two classes

$$\xi_{ent}(\mathbf{d}, \mathbf{y}) = -\sum_{n=1}^{N}\sum_{k=1}^{m} d_k^{(n)} \log y_k^{(n)}, \quad m > 2. \tag{9}$$

A Bayesian approach to training neural networks is to treat the weights $\mathbf{w}$, $\mathbf{v}$ as random variables which have a posterior probability distribution

$$p(\mathbf{w}, \mathbf{v} \mid \mathbf{D}) = \frac{p(\mathbf{D} \mid \mathbf{w}, \mathbf{v})\, p(\mathbf{w}, \mathbf{v})}{p(\mathbf{D})}, \tag{10}$$

where $\mathbf{D}$ is the dataset containing the known inputs and outputs. The weights are then solved from the problem of maximizing the log posteriori and the solution is called the maximum a posteriori (MAP) solution

$$\mathbf{w}^*, \mathbf{v}^* = \arg\max_{\mathbf{w},\mathbf{v}} \log p(\mathbf{w}, \mathbf{v} \mid \mathbf{D}). \tag{11}$$

If we specify a Gaussian prior for the weights $p(\mathbf{w}, \mathbf{v}) \sim \mathcal{N}(\mathbf{0}, \lambda^{-1}\mathbf{I})$, it turns out that the MAP solution (11) minimizes the augmented error function

$$\xi_\lambda(\mathbf{d}, \mathbf{y}) = \xi_{MSE}(\mathbf{d}, \mathbf{y}) + \frac{\lambda}{2}\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\theta}, \tag{12}$$

where $\boldsymbol{\theta}$ contains all the model parameters, that is $\boldsymbol{\theta} = [\mathbf{w}^\mathsf{T}, \mathbf{v}^\mathsf{T}]^\mathsf{T}$. This augmented error function ξ_λ corresponds to the case when a popular regularizer, *weight decay*, is used to regulate the values of weights and thus mitigate the problem of overfitting neural networks in the training phase.

2.3.1 Backpropagation

The *backpropagation* algorithm is the original method to train neural networks by solving the problem (8) or an equivalent optimization problem. The basic backpropagation algorithm is a two-step instantaneous stochastic gradient algorithm used to update the weights of a neural network. In the forward step, the network outputs are computed from the given inputs and fixed weights. In the backward step, the weights of the network are updated. To update the network weights the computed model approximation errors are propagated backwards in the network, hence the algorithm name backpropagation.

The forward step equations are presented in the Section 2.2 but we restate them here for convenience

$$\begin{aligned} h_j(\mathbf{x}, \mathbf{w}_j) &= \varphi(\mathbf{w}_j^\mathsf{T}\mathbf{x}), \\ y_k(\mathbf{z}, \mathbf{v}_k) &= \mathbf{v}_k^\mathsf{T}\mathbf{h}. \end{aligned} \tag{13}$$

In these equations $\mathbf{x}$ is the input vector, h_j is the hidden unit j output, y_k is the output unit k output, $\mathbf{w}_j$ and $\mathbf{v}_k$ are the weight vectors for the hidden unit j and output unit k and $\varphi(x)$ is the sigmoid function (4).

For the backward step equations, we use here the mean-square error function

(8) and derive the basic backpropagation update equations for the weights $\mathbf{w}$ and $\mathbf{v}$ that can be used to solve the problem (7) in a regression problem. By differentiating (8) with respect to the weights over the whole dataset $n = 1, 2, \ldots, N$ we get

$$\begin{aligned} -\frac{\partial \xi_{MSE}}{\partial v_{jk}} &= \frac{1}{N}\sum_{n=1}^{N}(d_k^{(n)} - y_k^{(n)})h_j \\ -\frac{\partial \xi_{MSE}}{\partial w_{ij}} &= -\frac{1}{2N}\sum_{n=1}^{N}\frac{\partial \xi_{MSE}}{\partial y_k^{(n)}}\frac{\partial y_k^{(n)}}{\partial h_j^{(n)}}\frac{\partial h_j^{(n)}}{\partial w_{ij}} \\ &= \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{m}\left((d_k^{(n)} - y_k^{(n)})v_{jk}\right)h_j^{(n)}(1 - h_j^{(n)})x_i^{(n)}. \end{aligned} \tag{14}$$

The equations (14) are derived from a *batch learning* perspective in which the weights are updated after a complete pass of the dataset instead of doing the update after each forward and backward pass of a single input/outpair data point. The backpropagation backward step equations are derived from the equations (14) by defining a learning parameter η and writing

$$\begin{aligned} v_{jk}(l+1) &= v_{jk}(l) - \eta\frac{\partial \xi_{MSE}}{\partial v_{jk}} \\ w_{ij}(l+1) &= w_{ij}(l) - \eta\frac{\partial \xi_{MSE}}{\partial w_{ij}}. \end{aligned}$$

Here l indicates the iteration round. The learning parameter η defines the step size in the iterative updates. More advanced approach is to adjust value of η dynamically during the iteration instead of fixing it to a constant value. In addition to the treatment of the learning parameter, there are numerous other variations and enhancements to the standard backpropagation algorithm presented here.

3 Deep Neural Networks

A deep neural network (DNN; Bengio, 2009) is built from a deep belief network (DBN; Hinton *et al.*, 2006) by transforming a generative belief network into a discriminative neural network. The transformation is done by adding a discriminative layer of computing units on top the DBN architecture that consists of layers of stacked restricted Boltzmann machines (RBM; Smolensky, 1986). DBN is a generative model because its processing units, RBMs, are generative by nature. The training of DNNs is carried out in two phases. First the underlying generative DBN is pretrained in an unsupervised training phase. After training the underlying DBN, the weights of a discriminative DNN are fine-tuned in a supervised training phase.

3.1 Restricted Boltzmann Machines

A restricted Boltzmann machine is a generative stochastic neural network which contains stochastic binary units on two layers: the visible layer and the hidden layer. The units are fully connected between the layers but no intra-layer connections between the units exist—this is unlike in a Boltzmann machine, hence the name restricted Boltzmann machine. The architecture of an RBM is illustrated in Figure 3. Note that although RBMs and parallel perceptrons have a very similar network structure (see the first two layers in Figure 2 for a parallel perceptron structure), the stochastic and generative nature of RBMs set them apart from parallel perceptrons.

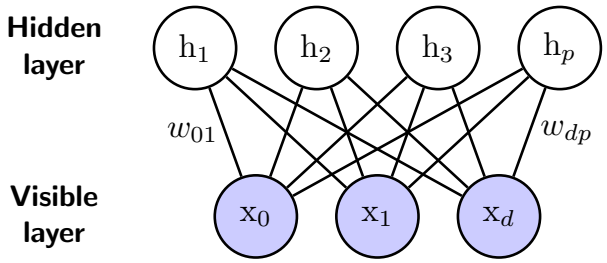


Figure 3: Architecture of a restricted Boltzmann machine (RBM). The visible units x_i and hidden units h_j are connected through undirected and symmetric connections. There are no intra-layer connections exist unlike in Boltzmann machines, hence the name restricted Boltzmann machine.

The generative nature of RBMs stems from their design. RBMs do not have output units as such. Rather, the connections between the units on the visible layer (hereafter visible units) and on the hidden layer (hereafter hidden units) are symmetric and undirected. Undirected and symmetric connections between the units mean that the state x_i of a visible unit affects the state h_j of a hidden unit, and vice versa, and that the connection weight w_{ij} between the units i and j is symmetric $w_{ij} = w_{ji}$. In addition to weights, RBM parameters include a constant bias term for each visible and hidden unit, $b_i \in \mathbb{R}$ and $c_i \in \mathbb{R}$, respectively. In learning RBMs, and generative models in general, the objective is to infer the model parameters from known input data in such way that the model can generate data with similar distribution.

The stochastic nature of RBMs follows from the fact that the visible and hidden states are stochastic. The states are binary, i.e. $x_i, h_j \in \{0, 1\} \, \forall \, i, j$, and the joint probability characterizing the RBM configuration is the Boltzmann distribution

$$p(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta}) = \frac{1}{Z} e^{-E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})}, \tag{15}$$

where $\mathbf{x} = [x_1, x_2, \ldots, x_d]^\mathsf{T}$, $\mathbf{h} = [h_1, h_2, \ldots, h_p]^\mathsf{T}$ and $\boldsymbol{\theta}$ contains all the model parameters: the weights $w_{ij} \in \mathbf{W} \in \mathbb{R}^{d \times h}$ and the bias terms $b_i \in \mathbf{b} \in \mathbb{R}^d$ and $c_j \in \mathbf{c} \in \mathbb{R}^h$ for visible and hidden units respectively. The normalization constant is $Z = \sum_{\mathbf{x}, \mathbf{h}} e^{-E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})}$ and the energy $E(\cdot)$ is defined by

$$E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta}) = -(\mathbf{x}^\mathsf{T} \mathbf{W} \mathbf{h} + \mathbf{b}^\mathsf{T} \mathbf{x} + \mathbf{c}^\mathsf{T} \mathbf{h}). \tag{16}$$

In RBMs, the hidden units are conditionally independent due to the absence of connections between them; the hidden units are independent given the visible units, hence the conditional independency. Therefore, and given the exponential form of the joint probability (15), the conditional distributions $p(\mathbf{x} \mid \mathbf{h}, \boldsymbol{\theta})$ and $p(\mathbf{h} \mid \mathbf{x}, \boldsymbol{\theta})$ factorize

$$p(\mathbf{x} \mid \mathbf{h}, \boldsymbol{\theta}) = \textstyle\prod_{i=1}^{d} p(x_i \mid \mathbf{h}, \boldsymbol{\theta}) \tag{17}$$

$$p(\mathbf{h} \mid \mathbf{x}, \boldsymbol{\theta}) = \textstyle\prod_{j=1}^{h} p(h_j \mid \mathbf{x}, \boldsymbol{\theta}), \tag{18}$$

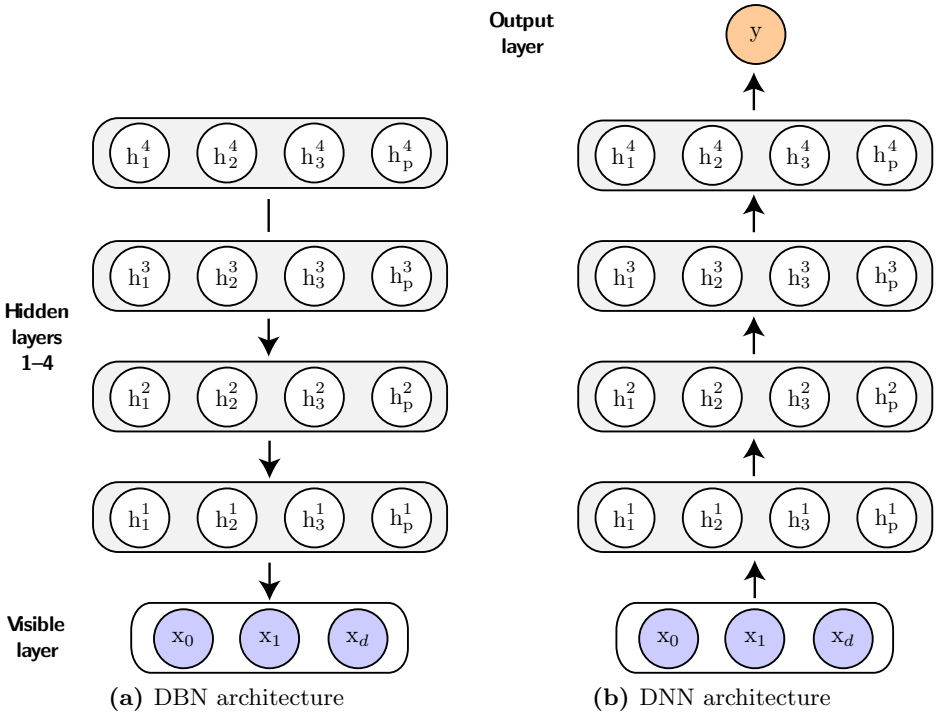
where, given $\varphi(x) = \frac{1}{1 + \exp(-x)}$, we have (see, e.g., Fischer and Igel, 2014)

$$p(x_i = 1 \mid \mathbf{h}, \boldsymbol{\theta}) = \varphi\left(\textstyle\sum_{j=1}^{h} w_{ij} h_j + b_i\right) \tag{19}$$

$$p(h_j = 1 \mid \mathbf{x}, \boldsymbol{\theta}) = \varphi\left(\textstyle\sum_{i=1}^{h} w_{ij} x_i + c_j\right). \tag{20}$$

3.2 Deep Belief Networks—Networks of RBMs

Restricted Boltzmann machines (RBM) can be combined into a stack to form a deep belief network (DBN), that is a multilayer stochastic generative model. A DBN contains multiple hidden layers on top of the lowest level RBM. The idea in having multiple hidden layers is that the preceding hidden layer acts as the visible layer for the next hidden layer and thus the model can incrementally learn more complex features of data. Although the connections in an RBM are undirected, the connections between the DBN layers are top-down directed expect between the two hidden layers on top of the stack that have undirected, symmetric connections. Thus, a DBN does not have a one-to-one correspondence with a stack of RBMs. Although in the generative pretraining phase the connections of a DBN are treated as undirected and symmetric. As in RBMs, the units between the DBN layers are fully connected and there are no intra-layer connections. The architecture of a DBN with four hidden layers is shown in Figure 4a.



(a) DBN architecture **(b)** DNN architecture

Figure 4: Panel (a) shows the architecture of a DBN. The connections between the top two level hidden layers are undirected and symmetric where as the other connections are top-down directed. Panel (b) shows the architecture of a DNN where there are only feedforward connections. The weights of a trained DBN can be used to initialize the weights of a DNN for a discriminative fine tuning.

The top two levels of a DBN, $\mathbf{h}^{L-1}$ and $\mathbf{h}^{L}$, form an RBM and the lower levels form a directed sigmoid belief net, $\mathbf{h}^{L-2} \Rightarrow \mathbf{h}^{L-3} \Rightarrow \cdots \Rightarrow \mathbf{x}$, (see, e.g., Neal, 1992). The top-level RBM provides a prior for the hidden layer $\mathbf{h}^{L-2}$. The joint distribution of a DBN is thus

$$p(\mathbf{x}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^L) = p(\mathbf{h}^{L-1}, \mathbf{h}^L) \prod_{l=0}^{L-2} p(\mathbf{h}^l \mid \mathbf{h}^{l+1}), \tag{21}$$

where $\mathbf{h}^0 = \mathbf{x}$ and $p(\mathbf{h}^{L-1}, \mathbf{h}^L)$ is the top-level RBM joint distribution (Hinton *et al.*, 2006; Bengio, 2009). The lower level layers $\mathbf{h}^l, l \in [0, L-2]$ are conditional only on the layer above, $p(\mathbf{h}^l \mid \mathbf{h}^{l+1})$, because the DBN is top-down directed expect for the top-level RBM. Note that in a deep Boltzmann machine (Salakhutdinov and Hinton, 2009) the connections between the lower level layers are also undirected and the conditional distribution of $\mathbf{h}^l$ is conditional on both the layer above and below, that is $p(\mathbf{h}^l \mid \mathbf{h}^{l+1}, \mathbf{h}^{l-1})$.

Samples from the generative DBN model can be obtained as follows (Bengio, 2009):

1. Draw samples of each hidden unit on the penultimate hidden layer $\mathbf{h}^{L-1}$.

2. For each lower level layer $\mathbf{h}^l, l \in [0, L-2]$, draw samples of the hidden units given the samples of the layer above.

3. The visible layer samples $\mathbf{x} = \mathbf{h}^0$ are the DBN samples.

The samples in step 2 above are drawn from the conditional distributions $p(\mathbf{h}^l \mid \mathbf{h}^{l+1})$. The initial samples from the layer $\mathbf{h}^{L-1}$ are produced in step 1 by conducting an alternating Gibbs sampling between the top-level RBM distributions $p(\mathbf{h}^{L-1} \mid \mathbf{h}^L)$ and $p(\mathbf{h}^L \mid \mathbf{h}^{L-1})$ until the Gibbs chain has converged and the samples of $\mathbf{h}^{L-1}$ represent $p(\mathbf{h}^{L-1} \mid \mathbf{h}^L)$.

A trained generative deep belief network is transformed to a deep neural network by adding a discriminative layer $\mathbf{y}$ on top of the top-level RBM and converting all the connections in the DBN to feedforward ones, see Figure 4b. The estimated DBN parameters can directly be used as the DNN parameters but typically, these are fine-tuned through a classical supervised parameter estimation, for example by using the backpropagation algorithm.

3.3 Training Deep Neural Networks

Training deep neural networks in the standard discriminative and supervised way, that is approach in training the classical feedforward neural networks, is very hard and does not perform well (Glorot and Bengio, 2010). In essence, the difficulty in training DNNs arises from the very large number of free model parameters that are due to the deep model architecture.

Instead of training a DNN in a discriminative manner from the beginning, the successful approach to training DNNs has been to first pretrain a generative model transform this to a discriminative mode and finally fine tune the model parameters with a standard discriminative training (Hinton *et al.*, 2006; Hinton and Salakhutdinov, 2006). The generative pretraining initializes the model parameters to a parameter-space that forms a much better starting point for the discriminative fine tuning phase and thus allows it make rapid progress (Hinton *et al.*, 2012).

In this section, we review the principles for training a DNN using the generative pretraining and discriminative fine tuning approach. In short, a DBN is generatively pretrained first and then a discriminative fine tuning training is performed for the DNN that is formed from the pretrained DBN. To train a DBN one must know how to train an RBM. Thus, we start with the training of RBMs. Then we show how to conduct the layer-wise generative pretraining of a DBN.

Transforming a DBN to a DNN can done simply by including an additional discriminative layer on top of the DBN architecture and turning all the connections in the network in to directed feedforward connections, see the Figure 4. Performing the discriminative fine tuning for the DNN corresponds to a standard discriminative training of MLP neural network, which we have covered in the Section 2.3 and do not touch upon in this section. The DNN parameters are initialized to the values found during the generative pretraining and the discriminative fine tuning for these can be done, for example, with the classical backpropagation algorithm, see Section 2.3.1.

3.3.1 Training Restricted Boltzmann Machines

Let us denote the likelihood of model parameters $\boldsymbol{\theta}$ with $L(\boldsymbol{\theta}) = p(\mathbf{x} \mid \boldsymbol{\theta})$. The general approach to model parameter estimation is to find the parameter estimates $\boldsymbol{\theta}^*$ by maximizing the log-likelihood

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \log L(\boldsymbol{\theta}).$$

We derive the log-likelihood for RBMs by first marginalizing the joint distribution (15) by summing over the hidden states

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_{\mathbf{h}} p(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})$$

and then taking a log-transformation of the resulting marginal distribution

$$\begin{aligned} \log L(\boldsymbol{\theta}) &= \log \sum_{\mathbf{h}} p(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta}) \\ &= \log \sum_{\mathbf{h}} \frac{1}{Z} e^{-E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})} \\ &= \log \sum_{\mathbf{h}} e^{-E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})} - \log Z \\ &= \log \sum_{\mathbf{h}} e^{-E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})} - \log \sum_{\mathbf{x}, \mathbf{h}} e^{-E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})}. \end{aligned} \tag{22}$$

The partial derivatives of the likelihood with respect to the RBM model parameters θ_i are defined by the equation (for a straightforward derivation, see e.g., Fischer and Igel, 2014)

$$\frac{\partial \log L(\boldsymbol{\theta})}{\partial \theta_i} = \sum_{\mathbf{x}, \mathbf{h}} p(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta}) \frac{\partial E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})}{\partial \theta_i} - \sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{x}, \boldsymbol{\theta}) \frac{\partial E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})}{\partial \theta_i}, \tag{23}$$

which is often in literature defined as follows, because the two terms in the formula are expectations of the same argument but over different distributions, that is,

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_i} = \mathrm{E}_{d_1} \left[\frac{\partial E(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})}{\partial \theta_i} \right] - \mathrm{E}_{d_2} \left[\frac{\partial E(\mathbf{h} \mid \mathbf{x}, \boldsymbol{\theta})}{\partial \theta_i} \right].$$

Here $E(\cdot)$ is energy function defined in (16) and E_d denotes expectation taken over distribution d. Here we have $d_1 = p(\mathbf{x}, \mathbf{h} \mid \boldsymbol{\theta})$ and $d_2 = p(\mathbf{h} \mid \mathbf{x}, \boldsymbol{\theta})$.

The second term in (23) can be computed analytically, see for example Fischer and Igel (2014). This follows from the fact that the posterior $p(\mathbf{h} \mid \mathbf{x}, \boldsymbol{\theta})$ of the hidden states can be factorized, see the equation (18), and also from the derivatives $\frac{\partial E(\mathbf{x},\mathbf{h}|\boldsymbol{\theta})}{\partial \theta_i}$ being analytically computable for the RBM parameters. However, the first term in (23) is generally too burdensome for direct computation. For this reason, MCMC sampling methods (see, e.g., Gelman *et al.*, 2004; Bishop, 2006) are employed to approximate the second term by generating samples from the corresponding distribution and approximating the expectation with an sample average. Particularly, Gibbs sampling (Geman and Geman, 1984) is typically used in training RBMs.

Contrastive divergence (CD, Hinton, 2002; Carreira-Perpinan and Hinton, 2005) learning is computationally a much more efficient approach to training RBMs than the approach of approximating the second term in the exact log-likelihood gradient (23) using MCMC methods. In k-step CD learning (typically $k = 1$) the log-likelihood gradient is approximated with a $\mathrm{CD}_k(\mathbf{x}^{(0)}, \boldsymbol{\theta})$ function

$$\begin{aligned} \mathrm{CD}_k(\mathbf{x}^{(0)}, \boldsymbol{\theta}) = &\sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{x}^{(\tilde{k})}, \boldsymbol{\theta}) \frac{\partial E(\mathbf{x}^{(\tilde{k})}, \mathbf{h} \mid \boldsymbol{\theta})}{\partial \theta_i} \\ &- \sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{x}^{(0)}, \boldsymbol{\theta}) \frac{\partial E(\mathbf{x}^{(0)}, \mathbf{h} \mid \boldsymbol{\theta})}{\partial \theta_i}. \end{aligned} \tag{24}$$

Here $\mathbf{x}^{(0)}$ is a known training input and $\mathbf{x}^{(\tilde{k})}$ is a sample produced by running a Gibbs sampler for k steps from the initial state $\mathbf{x}^{(0)}$. In principle, the sample $\mathbf{x}^{(\tilde{k})}$ is produced by sampling $\mathbf{h}^{(t)}$ on step t from $p(\mathbf{h} \mid \mathbf{x}^{(t)}, \boldsymbol{\theta})$ and subsequently $\mathbf{x}^{(t+1)}$ from $p(\mathbf{x} \mid \mathbf{h}^{(t+1)}, \boldsymbol{\theta})$.

The contrastive divergence approximation (24) is biased for two reasons. First, the sample $\mathbf{x}^{(\tilde{k})}$ is not from the converged, stationary model distribution (unless $k \to \infty$. Second, the approximation does not maximize the likelihood of the data given the model (but rather a difference of two Kullback–Leibler divergences). For detailed analysis and justification for using contrastive divergence, see Bengio and Delalleau (2009).

3.3.2 Generative Pretraining of Deep Belief Networks

An efficient two-stage algorithm to training DBNs has been proposed by Hinton *et al.* (2006). We present here the first stage of the algorithm which

corresponds to a layer-wise generative pretraining of a DBN—the second stage presented by Hinton *et al.* (2006) corresponds to fine tuning the model for generative purposes, which is not in our interests here.

In the generative pretraining stage a DBN is trained layer-wise starting from the bottom hidden layer and subsequently moving upwards layer per layer until the top hidden layer has been trained. The layer below the layer being trained is treated in turn as if it was a visible layer—what it indeed is in the case of the first hidden layer. This way, each pair of layers in turn is treated as if it was an RBM. The connections between the pair of layers are taken to be undirected and symmetric during the pretraining stage.

The layer-wise training is conducted as follows (see, e.g. Bengio, 2009; Cho, 2014). First, the lowest level RBM is trained. That is, the units on the first hidden layer $\mathbf{h}^1$ and on the visible layer $\mathbf{x}$ are trained to model a given a set of training inputs $D_0 = \{\mathbf{x}^{(n)} \mid n = 1, 2, \ldots, N\}$. Training here refers to estimating the RBM parameters, the weights and biases. The RBM training can be done for example using the contrastive divergence algorithm. Having estimated the parameters $\boldsymbol{\theta}^1$ of the first (the lowest level) RBM, a set of samples D_1 of the first hidden layer states $\mathbf{h}^1$ is drawn from the posteriors $Q(\mathbf{h}^1 \mid \mathbf{x}^{(n)}, \boldsymbol{\theta}^1)$, $n = 1, 2, \ldots, N$, for the training of the next level RBM. The posteriors are denoted with $Q(\cdot)$ because they only approximate the true posterior that is also dependent on the above hidden layers. The aggregate posterior $Q(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)$, from which the samples are effectively collected, is an average

$$Q(\mathbf{h}^1 \mid \boldsymbol{\theta}^1) = \frac{1}{N} \sum_{n=1}^{N} Q(\mathbf{h}^1 \mid \mathbf{x}^{(n)}, \boldsymbol{\theta}^1). \tag{25}$$

The next step in the layer-wise pretraining is to train the second level RBM by using the set D_1, which contains samples of the hidden layer $\mathbf{h}^1$ states, as an input for training the second level RBM. The second level RBM is thus trained to model the aggregate posterior $Q(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)$ through the samples. The hidden layers $\mathbf{h}^2$ and $\mathbf{h}^1$ of the DBN are the hidden and visible layer of the second level RBM, respectively. That is, the hidden layer of the first level RBM, $\mathbf{h}^1$, is the visible layer of the next level RBM. After training the second level RBM, and thus having estimated the corresponding RBM parameters $\boldsymbol{\theta}^2$, samples from the aggregate posterior $Q(\mathbf{h}^2 \mid \boldsymbol{\theta}^2)$ are collected for training the next level RBM. The generative pretraining proceeds subsequently by

treating each pair of consecutive hidden layers, $\mathbf{h}^l$ and $\mathbf{h}^{l-1}$, as an RBM and training them to model the lower level aggregate posterior $Q(\mathbf{h}^{l-1} \mid \boldsymbol{\theta}^{l-1})$ until the top layer L is reached.

We can summarize the generative pretraining algorithm for DBNs as follows. Let use denote here the visible layer $\mathbf{x}$ with $\mathbf{h}^0$. For each hidden layer $\mathbf{h}^l$ in a DBN, repeat the two steps below until a stopping criterion is reached:

1. Draw k samples of each hidden unit $h_j^l, j = 1, 2, \ldots, p$ from the distribution $Q(h_j^{l-1} \mid \mathbf{h}^{l-2}, \boldsymbol{\theta}^{l-1})$ using the formula (20).

2. Train the l-level RBM, that consists of the hidden layer $\mathbf{h}^l$ and visible layer $\mathbf{h}^{l-1}$, using the samples drawn in the step 1 and, as a result, obtain the parameters $\boldsymbol{\theta}^l$ including the weights and bias units.

Naturally, the first step does not need to be performed for the first hidden layer $\mathbf{h}^1$ as its training set is given and consists of the inputs $\mathbf{x}^{(n)}$.

The key motivation in training a DBN with multiple hidden layers is the following. Given the inputs $\mathbf{x}$, the first level RBM learns the distribution

$$Q(\mathbf{x} \mid \boldsymbol{\theta}^1) = \sum_{\mathbf{h}^1} p(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)\, p(\mathbf{x} \mid \mathbf{h}^1, \boldsymbol{\theta}^1), \tag{26}$$

where $p(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)$ is the prior distribution of the hidden units $\mathbf{h}^1$. The first level RBM can be improved if we can replace the prior $p(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)$ with a better prior while keeping the input posterior $p(\mathbf{x} \mid \mathbf{h}^1, \boldsymbol{\theta}^1)$ fixed. A better prior is a prior closer to the aggregate posterior $Q(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)$, which is exactly the distribution that the second level RBM is trained to model. Consequently, in DBNs the prior $p(\mathbf{h}^1 \mid \boldsymbol{\theta}^1)$ is effectively replaced with a prior so that it is produced by the second level RBM

$$p(\mathbf{h}^1 \mid \boldsymbol{\theta}^2) = \sum_{\mathbf{h}^2} p(\mathbf{h}^1, \mathbf{h}^2 \mid \boldsymbol{\theta}^2), \tag{27}$$

and likewise for the higher level hidden layers (Cho, 2014). Hinton *et al.* (2006) have shown that it is indeed possible to train DBNs one layer at a time and that training DBNs layer-wise never decreases the log likelihood of the data under the full model—although, the conditions are in practice often violated, for example through using CD in training RBMs.

4 Conclusions

In this study we have presented the computational principles underlying neural networks. Traditional multilayer perceptron neural networks and modern deep neural networks have the exact same network structure and computations within the networks are performed similarly, once the network training has been completed. Both MLP and deep neural networks consist of perceptrons, which are linear computing units. Nonlinear transformations between layers of perceptrons enable neural networks to be used for modeling nonlinear phenomena.

Deep neural networks differ from MLP neural networks in the network depth, that is determined by the number of hidden layers in the network. A neural network with three or more hidden layers is typically considered as a deep neural network. This difference is conceptual because from a purely computational perspective there is no difference between a MLP network with three or more hidden layers and a deep neural network with the same number of hidden layers. However, the size of a neural network, and especially the number of hidden layers, is an important factor because networks with many hidden layers are hard to train in practice. This is because the parameter estimation in the network training phase tends to converge to local optima resulting in a neural network with poor modeling performance.

The difficulties in training MLP neural networks with several hidden layers have given rise to the paradigm shift in training deep neural networks. Instead of directly training the network in a supervised manner through a standard discriminative learning setup, a generative deep belief network is trained first and its parameters are used to initialize the parameter estimation of the corresponding deep neural network. This training approach consisting of two, significantly different phases has enabled successful training of deep neural networks. In addition to the standard classification and regression tasks, deep neural networks have been successfully applied to machine learning tasks in various fields such as natural language processing, computer vision and information retrieval.

The journey from the first neural networks consisting of parallel perceptrons to the large-scale multi-layer deep neural networks has taken half a century. The pinnacle of deep neural networks is twofold. First, more complex ma-

chine learning problems can be solved more successfully with deep neural networks. Second, deep neural networks are pretrained through deep belief networks without labeled data. In practice, this is a significant advantage over traditional MLP neural networks which require labeled training data, because much of the data in the world is unlabeled.

References

Alpaydin, E. (2004). *Introduction to Machine Learning*. The MIT Press, Cambridge, second edition.

Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., Lent, R., and Herculano-Houzel, S. (2009). Equal Numbers of Neuronal and Nonneuronal Cells Make the Human Brain an Isometrically Scaled-up Primate Brain. *Journal of Comparative Neurology*, 513(5):532–541.

Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127.

Bengio, Y. and Delalleau, O. (2009). Justifying and Generalizing Contrastive Divergence. *Neural Computation*, 21(6):1601–1621.

Bengio, Y., LeCun, Y., *et al.*(2007). Scaling Learning Algorithms Towards AI. *Large-Scale Kernel Machines*, 34(5).

Bishop, C. (2006). *Pattern Recognition and Machine Learning*, volume 4. Springer, New York.

Carreira-Perpinan, M. A. and Hinton, G. E. (2005). On Contrastive Divergence Learning. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*.

Cho, K. (2011). *Improved Learning Algorithms for Restricted Boltzmann Machines*. Master's thesis, Aalto University, School of Science.

Cho, K. (2014). *Foundations and Advances in Deep Learning*. Doctoral dissertation, Aalto University.

Fischer, A. and Igel, C. (2014). Training Restricted Boltzmann Machines: an Introduction. *Pattern Recognition*, 47(1):25–39.

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (2004). *Bayesian Data Analysis*. Chapman & Hall/CRC Press, Boca Raton, Florida, second edition.

Geman, S. and Geman, D. (1984). Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(1):721–741.

Glorot, X. and Bengio, Y. (2010). Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics*.

Hinton, G. (2002). Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14(8):1771–1800.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., and Sainath, T. N. (2012). Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97.

Hinton, G., Osindero, S., and Teh, Y.-W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507.

McCulloch, W. S. and Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, Cambridge.

Neal, R. M. (1992). Connectionist Learning of Belief Networks. *Artificial Intelligence*, 56(1):71–113.

Rosenblatt, F. (1958). The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386–408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-Propagating Errors. *Nature*, 323(6088):533–536.

Rumelhart, David E and McClelland, James L and PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*. MIT Press, Cambridge.

Salakhutdinov, R. and Hinton, G. E. (2009). Deep Boltzmann Machines. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS'09)*, volume 5, pages 448–455.

Smolensky, P. (1986). Information Processing in Dynamical Systems: Foundations of Harmony Theory. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: foundations. The MIT Press, Cambridge.

Utgoff, P. E. and Stracuzzi, D. J. (2002). Many-Layered Learning. *Neural Computation*, 14(10):2497–2529.