Nizhny Novgorod State University

Institute of Information Technologies, Mathematics and Mechanics

Department of Computer software and supercomputer technologies

Educational course
«Introduction to deep learning
using the Intel® neon™ Framework»

Unsupervised learning: autoencoders, restricted Boltzmann machines, deconvolutional networks

Supported by Intel

Valentina Kustikova,
Phd, lecturer, department of Computer software
and supercomputer technologies

Content

- ❑ The problem statement
- ❑ Autoencoders
- ❑ Restricted Boltzmann machine
- ❑ Deep Boltzmann machine
- ❑ Deep belief network
- ❑ Deconvolutional neural network
- ❑ Example of unsupervised learning application for pre-training the parameters of deep models to predict a person's sex from a photo

THE PROBLEM STATEMENT

The problem

- ❑ The dataset ImageNET: 14 197 122
- ❑ The dataset PASCAL Visual Object Challenge 2012 (semantic segmentation): ~10 000
- ❑ The dataset IMDB-WIKI: 460 723 + 62 328
- ❑ …

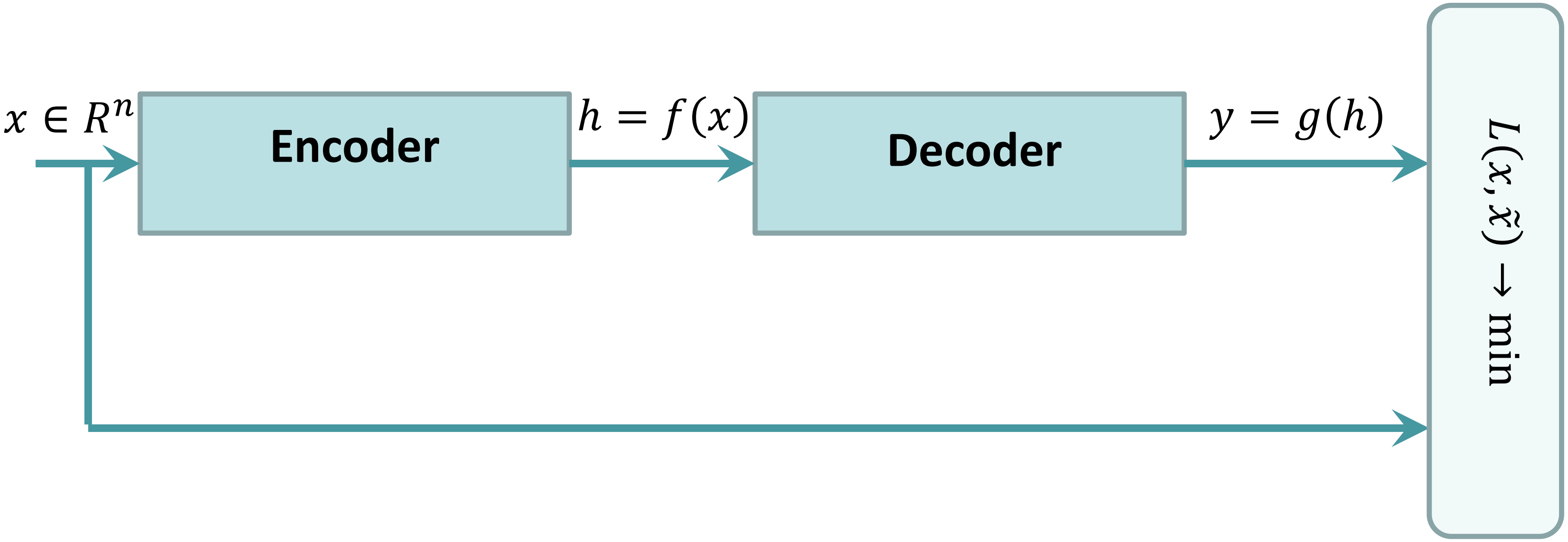
- ❑ ***How to reduce the amount of labeled data for constructing efficient deep neural network models?***

The solution

- Pre-train the network weights
  - Construct a “good” initial approximation of the parameters for further training the network using the labeled training dataset
  - Do not use labeled data during pre-training

AUTOENCODERS

Autoencoder (1)



Autoencoder (2)

- ***Autoencoder*** is a neural network that attempts to approximate the output signal to the input one, i.e. to find the best approximation of the identity transform
- The network is divided into two principal parts:
  - ***Encoder*** $h = f(x)$ to compress input data representation
  - ***Decoder*** $y = g(h)$ to restore input data based on the compressed representation
- Autoencoder can be considered as a feed-forward network, so for training it is possible to use the backpropagation algorithm, based on the gradient optimization methods

Simple autoencoder

- The simplest autoencoder consists of two fully-connected layers
- The layers are described by the transforms as follows:

  $$h = f(x) = s_f(W_h \cdot x + b_h), \qquad y = g(h) = s_g\left(W_y \cdot h + b_y\right),$$

  where $s_f(\cdot)$ is an activation function (sigmoid, hyperbolic tangent, ReLU)
- Training the autoencoder is a problem of minimizing the functional $J(\theta)$ with respect to the set of parameters $\theta = \left\{W_h, W_y, b_h, b_y\right\}$:

  $$J(\theta) = \sum_{x \in D_n} L\left(x, g\left(f(x)\right)\right) \to \min_{\theta}$$

- Cost function $L\left(x, g\left(f(x)\right)\right)$ is a square function or multinomial cross-entropy

Autoencoders and principal component analysis (1)

- If $L\left(x, g\left(f(x)\right)\right)$ is square function then the minimized functional is as follows:

$$J(\theta) = \sum_{x \in D_n} \left\| x - g\left(f(x)\right) \right\|^2$$

- If f and g are linear functions then the functional is as follows:

$$J(\theta) = \sum_{x \in D_n} \| x - VWx \|^2 ,$$

  where W is a matrix of direct transform (corresponds to the encoder), V is a matrix of inverse transform (corresponds to the decoder)

- The optimal solution of the minimization problem corresponds to the solution of linear PCA problem (the projection of the input data onto a lower-dimensional subspace)

Autoencoders and principal component analysis (2)

- If f and g are non-linear functions then the optimal solution of the minimization problem corresponds to the solution of non-linear PCA problem

Regularized autoencoders

- The regularization parameters are introduced – the penalty function
- Square penalty for the hidden layer parameters:

$$J(\theta) = \sum_{x \in D_n} L\left(x, g\left(f(x)\right)\right) + \lambda \sum_{i,j} w_{ij}^2 \to \min_{\theta}, \;\; W = \left(w_{ij}\right)$$

Sparse autoencoders (1)

- ***Sparse autoencoders*** is a kind of regularized autoencoders
- In the simplest case, the sparsity penalty is a L_1-norm of the signal on the hidden layer, and the minimized functional is as follows:

$$J(\theta) = \sum_{x \in D_n} \left(L\left(x, g\left(f(x)\right)\right) + \lambda \sum_i |h_i| \right) \to \min_{\theta}$$

- In general case, the functional is calculated as follows:

$$J(\theta) = \sum_{x \in D_n} \left(L\left(x, g\left(f(x)\right)\right) + \Omega\left(f(x)\right) \right) \to \min_{\theta}$$

Sparse autoencoders (2)

- Let us consider a sparse autoencoder as an approximation of a generative model with latent variables, in which the maximum likelihood method can be used to train parameters
- Suppose that there is a model in which x is a set of the visible variables (the input signal), h is a set of the hidden variables
- Autoencoder cost function:

$$L\left(x, g\left(f(x)\right)\right) = -\log p(x|h)$$

- Then the likelihood function for the generative model is represented as follows:

$$\log p(x) = \log \sum_{h} p(h, x)$$

Sparse autoencoders (3)

- The autoencoder approximates this sum by a point estimate $p(h,x)$ for only the most probable h
- Therefore, for each chosen h, the logarithm of the joint distribution is maximized:

$$\log p(h,x) = \log p(h) + \log p(x|h)$$

- If each variable of the hidden layer belongs to the Laplace distribution $p(h_i) = \frac{\lambda}{2} e^{-\lambda|h_i|}$, then the penalty function proportional to the absolute value:

$$-\log p(h) = \sum_i \left(\lambda|h_i| - \log\frac{\lambda}{2}\right) = \Omega(h) + const,$$

- The penalty represents a L_1-norm
- Choosing another distribution leads to a different kind of penalty function

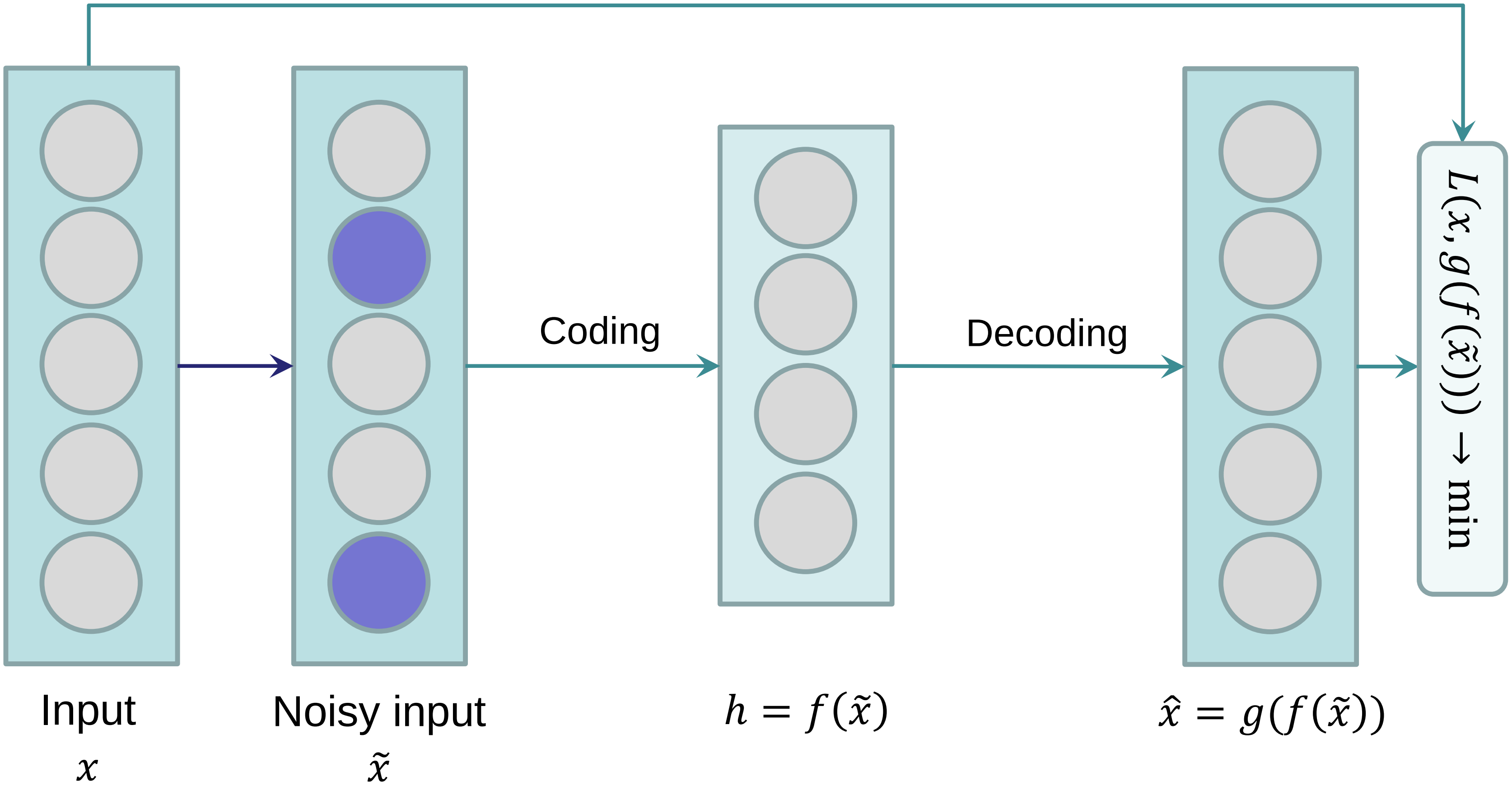
Sparse autoencoders (4)

- ❑ Sparse autoencoders are usually used to learn features for solving a problem (for example, classification tasks)
- ❑ Such autoencoders reflect the statistical properties of the training dataset
- ❑ As a result, we can construct a model that has learned the useful properties of a dataset

Denoising autoencoders (1)

- The goal of ***denoising autoencoders*** is to provide noise reduction of the distorted data, i.e. to restore original input based on the distorted version
- Denoising autoencoders minimize the cost function $L\left(x, g\left(f(\tilde{x})\right)\right)$, where $\tilde{x}$ is a distorted version of the input signal x

Denoising autoencoders (2)



Denoising autoencoders (3)

- The minimized functional is as follows:

$$J(\theta) = \sum_{x \in D_n} \mathbb{E}_{\tilde{x} \sim C(\tilde{x}|x)} \left(L\left(x, g\left(f(\tilde{x}) \right) \right) \right) \to \min_{\theta},$$

  where $\mathbb{E}_{\tilde{x} \sim C(\tilde{x}|x)}(.)$ is a mean value of all noisy signals $\tilde{x}$, obtained from the input signal x in accordance with some random process $C(\tilde{x}|x)$

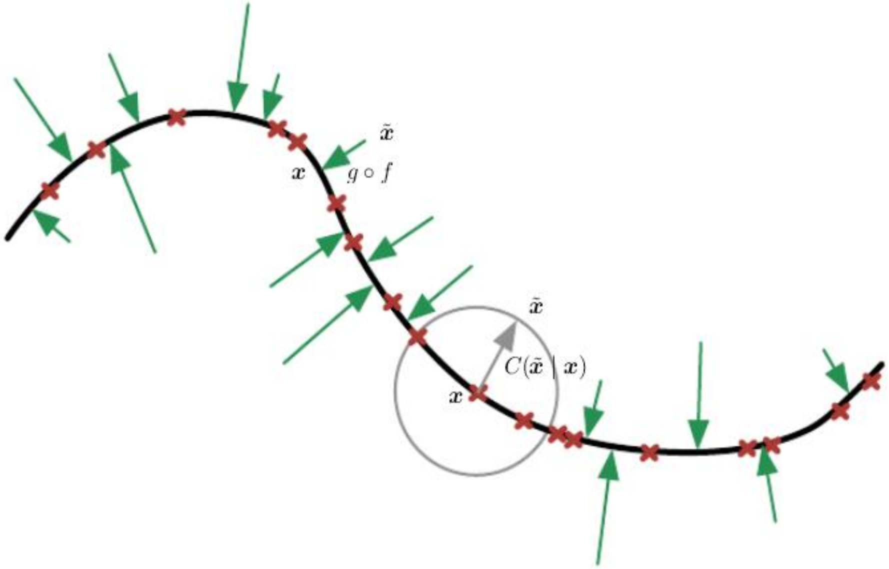
- Possible distortions:
  – Additive isotropic Gaussian noise
  – Noise of the form “salt and pepper” for gray images (randomly appearing black and white pixels)
  – Masking noise, i.e. setting of randomly selected inputs to zero (independently for each training sample)

Denoising autoencoders (4)

- The red crosses are training samples
- The gray circle corresponds to a set of the distorted input x
- The black line is a manifold that approximates the training set
- If the autoencoder is trained to minimize the square error, then $g\big(f(\tilde{x})\big)$ allows us to estimate

$$\mathbb{E}_{\tilde{x}\sim C(\tilde{x}|x)}\bigg(L\Big(x, g\big(f(\tilde{x})\big)\Big)\bigg)$$

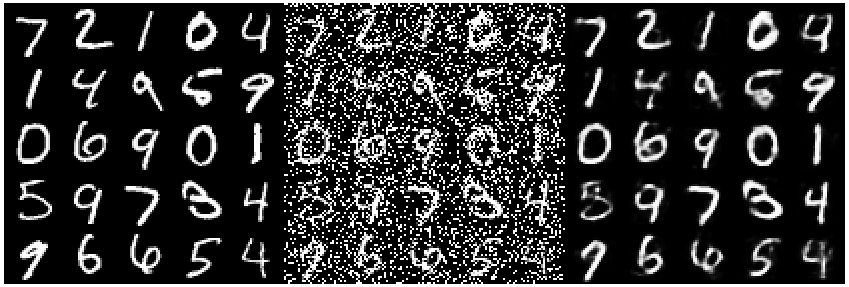
- In this case, each vector $g\big(f(\tilde{x})\big) - \tilde{x}$ points in the direction of the nearest point on the manifold



* Goodfellow I., Bengio Y., Courville A. Deep Learning. – MIT Press. – 2016. – [http://www.deeplearningbook.org].

Application of denoising autoencoders. Example

- ❑ The goal is to construct a network that, on a noisy image, restores the original image without noise (the noise generation law is known)
- ❑ A training data set is a set of images without noise. x is an image of the training set, and $\tilde{x}$ is a noisy image
- ❑ Training is identifying the network parameters, which provide the best restoration of the image based on its noisy copy
- ❑ Testing assumes restoring the original image based on the input noisy image



Original images Noisy images Restored images

* OpenDeep. Tutorial: Your First Model (DAE) [http://www.opendeep.org/v0.0.5/docs/tutorial-your-first-model].

Contractive autoencoder

- ***Contractive autoencoder*** is a kind of regularized autoencoders for which the minimized functional is as follows:

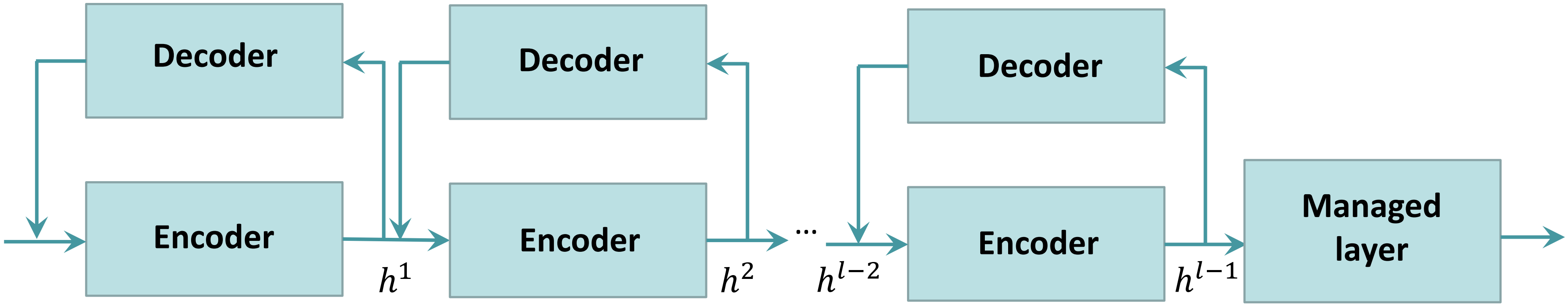
$$J(\theta) = \sum_{x \in D_n} \left(L\left(x, g\left(f(x)\right)\right) + \lambda \left\| J_f(x) \right\|_F^2 \right) \to \min_{\theta},$$

where $J_f(x)$ is a Jacobian of the vector function $f(x)$, $\left\|J_f(x)\right\|_F^2$ is a Frobenius norm of the Jacobian $J_f(x)$:

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \dots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(x) & \dots & \frac{\partial f_n}{\partial x_n}(x) \end{bmatrix}, \qquad \left\|J_f(x)\right\|_F^2 = \sum_{ij} \left(\frac{\partial f_j}{\partial x_i}(x) \right)^2$$

Stack of autoencoders

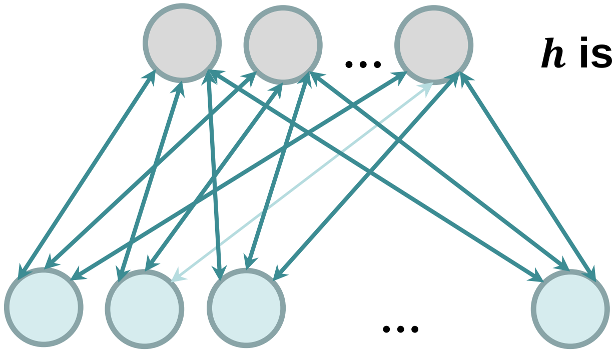
- ❑ For multi-layered networks it is possible to construct a stack of autoencoders
- ❑ Each autoencoder is trained sequentially, which allows you to gradually reduce the dimension of the feature space and adjust the parameters of the encoding layers



RESTRICTED BOLTZMANN MACHINE

Restricted Boltzmann machine

- ***Restricted Boltzmann machine*** is a probabilistic model of an autoencoder
- The Boltzmann machine is a stochastic model, the components of which are stochastic neurons
- ***A stochastic neuron*** can be in two probability states, which can be formally assigned the values $+1$ (on) and -1 (off), or $+1$ and 0 respectively
- The Boltzmann machine is characterized by the presence of ***symmetric synaptic connections***



h is a set of hidden neurons

x is a set of visible neurons

25

Notation

- $x = (x_i)_{0 \le i \le N}$ are visible neurons, $h = \left(h_j\right)_{0 \le j \le K}$ are hidden neurons
- w_{ij} is a synaptic connection between neurons i and j
  - $w_{ij} = w_{ji}$ is a matrix symmetry condition
  - $w_{ii} = 0$ means the absence of neuron connection with itself
- The shift use is achieved by adding fictitious elements between a node with a constant signal $+1$ and a neuron

The goal of training (1)

- By analogy with thermodynamics, the energy of restricted Boltzmann machine is described by one of two equations:
  - If the states of the neurons correspond to $+1$ and -1, then the equation is as follows

$$E(x,h) = -\frac{1}{2}\sum_{i=0}^{N}\sum_{\substack{j=0\\ j\neq i}}^{K} w_{ji}h_j x_i$$

  - If the states correspond to $+1$ and 0, then the equation is as follows

$$E(x,h) = -\sum_{i=0}^{N}\sum_{\substack{j=0\\ j\neq i}}^{K} w_{ji}h_j x_i$$

The goal of training (2)

- The neural network simulates the joint probability density function:

$$p(x,h)=\frac{e^{-E(x,h)}}{Z},\qquad Z=\sum_{r=0}^{L-1}\sum_{t=0}^{S-1}e^{-E\left(x^{(r)},h^{(t)}\right)},$$

where Z is a normalizing factor,

L is an image number of the vector x,

S is an image number of the vector h,

$p(x,h)$ is the Gibbs distribution

- The task of the training network is to maximize the function:

$$p(x)=\sum_{t=0}^{S-1}p\left(x,h^{(t)}\right)=\frac{1}{Z}\sum_{t=0}^{S-1}e^{-E\left(x,h^{(t)}\right)}\rightarrow \max$$

Conditional probabilities in the case of binary states of the input vector (1)

- The probability that for a given input vector x one of the hidden states is activated ($h_k = 1$) is as follows:

$$E_1 = E(x, h_k = 1) = -\sum_{i=0}^{N} \sum_{\substack{j=0 \\ j \neq i \\ j \neq k}}^{K} w_{ji} h_j x_i - \sum_{\substack{i=0 \\ i \neq k}}^{N} w_{ki} x_i \,,$$

$$E_0 = E(x, h_k = 0) = -\sum_{i=0}^{N} \sum_{\substack{j=0 \\ j \neq i \\ j \neq k}}^{K} w_{ji} h_j x_i \,, \qquad E_1 - E_0 = -\sum_{\substack{i=0 \\ i \neq k}}^{N} w_{ki} x_i$$

$$p(h_k = 1|x) = \frac{e^{-E_1}}{e^{-E_1} + e^{-E_0}} = \frac{1}{1 + e^{-\sum_{\substack{i=0 \\ i \neq k}}^{N} w_{ki} x_i}} = sigm\left(\sum_{\substack{i=0 \\ i \neq k}}^{N} w_{ki} x_i\right),$$

Conditional probabilities in the case of binary states of the input vector (2)

- ❑ Since all the neurons of the hidden layer are independent, the conditional probability can be expressed as follows:

$$p(h|x) = \prod_{j=0}^{K} p\left(h_j|x\right)$$

- ❑ By analogy, we can derive a formula for calculating $p(x|h)$:

$$p(x_k = 1|h) = sigm\left(\sum_{\substack{j=0\\ j\neq k}}^{K} w_{jk}h_j\right),$$

$$p(x|h) = \prod_{i=0}^{N} p(x_i|h)$$

Conditional probabilities in the case of real states of the input vector

- ❑ The energy function has a modified form, and $p(x_k|h)$ is modeled by the normal distribution:

$$E(x,h) = -\frac{1}{\sigma}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{K} w_{ji}h_j x_i - \sum_{j=1}^{K} b_j h_j + \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i - c_i)^2 ,$$

$$p(x_k|h) = N\left(\sigma\sum_{j=1}^{K} w_{ji}h_j + c_k , \sigma^2\right),$$

where b_j, c_i are shifts, σ is a standard deviation

Calculation of the probability distribution function derivatives

- ***The goal of training:***

$$p(x) = \sum_{t=0}^{S-1} p\left(x, h^{(t)}\right) = \frac{1}{Z}\sum_{t=0}^{S-1} e^{-E\left(x,h^{(t)}\right)} \to \max$$

- The energy function derivatives by parameters:

$$\frac{\partial E(x,h)}{\partial w_{ji}} = -h_j x_i$$

- The derivative of the exponential $e^{-E(x,h)}$ and the normalizing factor Z:

$$\frac{\partial e^{-E(x,h)}}{\partial w_{ji}} = e^{-E(x,h)} \frac{\partial\left(-E(x,h)\right)}{\partial w_{ji}} = h_j x_i e^{-E(x,h)}$$

$$\frac{\partial Z}{\partial w_{ji}} = \sum_{r=0}^{L-1}\sum_{t=0}^{S-1} \frac{\partial e^{-E\left(x^{(r)},h^{(t)}\right)}}{\partial w_{ji}} = \sum_{r=0}^{L-1}\sum_{t=0}^{S-1} h_j^{(r)} x_i^{(t)} e^{-E\left(x^{(r)},h^{(t)}\right)}$$

The derivative of the probability density function

- The derivative of the probability density function $p(x)$ is as follows:

$$\frac{\partial p(x)}{\partial w_{ji}} = -\frac{1}{Z^2}\frac{\partial Z}{\partial w_{ji}}\sum_{t=0}^{S-1} e^{-E\left(x,h^{(t)}\right)} + \frac{1}{Z}\sum_{t=0}^{S-1}\frac{\partial e^{-E\left(x,h^{(t)}\right)}}{\partial w_{ji}}$$

$$\frac{d\big(f(x)\cdot g(x)\big)}{dx} = \frac{d\big(f(x)\big)}{dx}\cdot g(x) + f(x)\cdot\frac{d\big(g(x)\big)}{dx}$$

$$= -\frac{1}{Z^2}\left(\sum_{r=0}^{L-1}\sum_{t=0}^{S-1} h_j^{(r)} x_i^{(t)} e^{-E\left(x^{(r)},h^{(t)}\right)}\right)\left(\sum_{t=0}^{S-1} e^{-E\left(x,h^{(t)}\right)}\right)$$

$$+ \frac{1}{Z}\sum_{t=0}^{S-1} h_j^{(t)} x_i e^{-E\left(x,h^{(t)}\right)}$$

$$= -\frac{1}{Z}p(x)\sum_{r=0}^{L-1}\sum_{t=0}^{S-1} h_j^{(r)} x_i^{(t)} e^{-E\left(x^{(r)},h^{(t)}\right)} + \sum_{t=0}^{S-1} h_j^{(t)} x_i\, p\left(x,h^{(t)}\right)$$

The maximization of the logarithm of the probability distribution function

- Maximization of the probability is equivalent to maximizing the logarithm of the probability:

$$p(x) \to \max \quad \Longrightarrow \ \ln p(x) \to \max$$

The derivative of the logarithm with respect to the parameters of the system

- The derivative of the logarithm of the probability function with respect to the parameters of the system:

$$\frac{\partial \ln p(x)}{\partial w_{ji}} = \frac{1}{p(x)} \frac{\partial p(x)}{\partial w_{ji}}$$

$$= -\frac{1}{Z} \sum_{r=0}^{L-1} \sum_{t=0}^{S-1} h_j^{(r)} x_i^{(t)} e^{-E\left(x^{(r)}, h^{(t)}\right)} + \sum_{t=0}^{S-1} h_j^{(t)} x_i \frac{p\left(x, h^{(t)}\right)}{p(x)}$$

$$= -\sum_{r=0}^{L-1} \sum_{t=0}^{S-1} h_j^{(r)} x_i^{(t)} p\left(x^{(r)}, h^{(t)}\right) + \sum_{t=0}^{S-1} h_j^{(t)} x_i \, p\left(h^{(t)} | x\right)$$

The Boltzmann learning rule

- The rule of updating the network weights during training:

$$\Delta w_{ji} = \eta \frac{\partial \ln p\left(x^{(k)}\right)}{\partial w_{ji}} = \eta \left(M\left[x_i^{(k)} h_j\right] - M\left[x_i h_j\right]\right),$$

where $M[.]$ is a mean value of the discrete variable, η is a learning rate (a parameter of the method)

- This rule is called the ***Boltzmann learning rule***

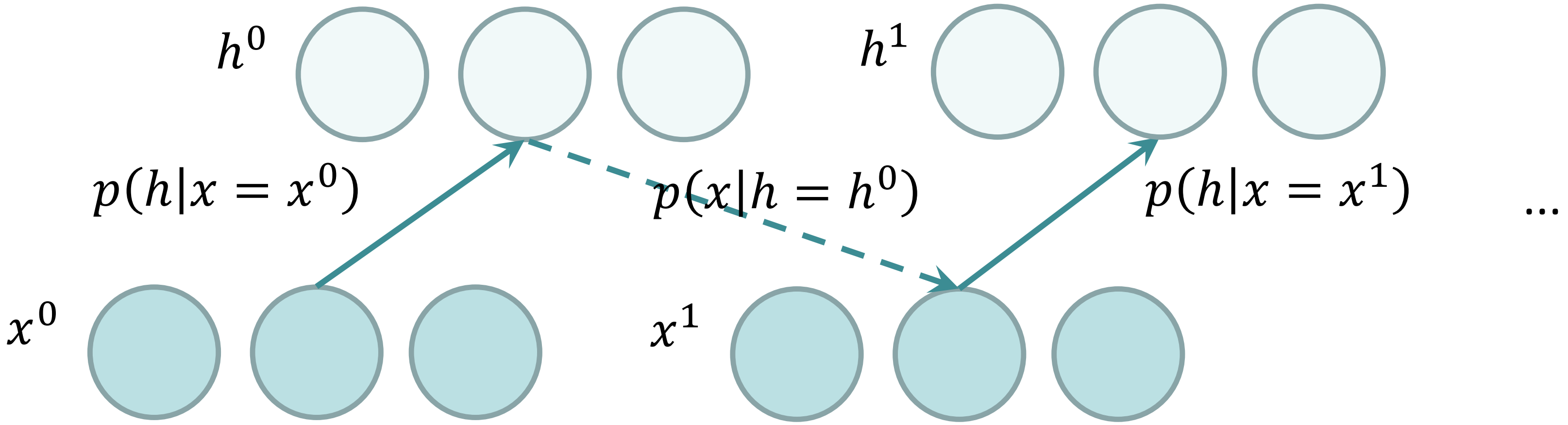
Training algorithm (1)

- The algorithm of training the Boltzmann machine was proposed by J. Hinton in 2002, and it is based on the ***Gibbs sampling*** procedure
- The training algorithm is called the ***contrastive divergence algorithm*** (CD-k)
- The algorithm describes the calculation scheme during the training parameters of the Boltzmann machine

Training algorithm (2)

1. Assumed the visible neuron state x equals the network input $x^0 = x^{(0)}$
2. Compute the probabilities if hidden layer states h^0 in accordance with the distribution $p(h|x = x^0)$
3. Loop by t consisting of k iterations. Collect statistics:
    1. Compute the probabilities of visible layer states x^t in accordance with distribution $p(x|h^{t-1})$
    2. Compute the probabilities of hidden layer states h^t in accordance with distribution $p(h|x = x^t)$
4. Update the network weights and go to the next input

Training algorithm (3)

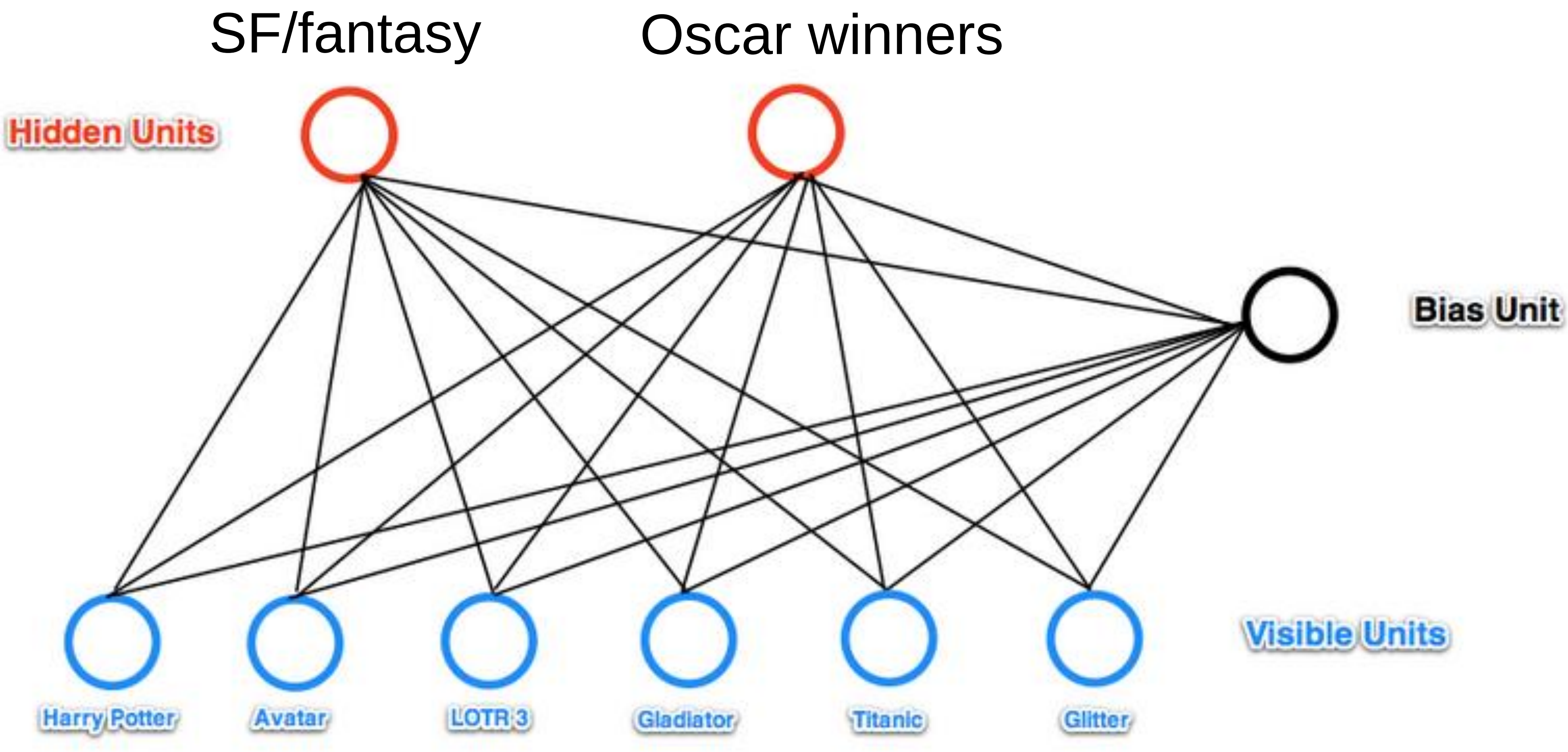


Interpretation of a restricted Boltzmann machine by an example (1)

- ❑ Visible neurons = movies
- ❑ Hidden neurons = film groups (fantasy, Oscar winners)

- ❑ A person chooses from 6 movies those that he likes => the input elements (binary values) are activated
- ❑ For each film, it is decided to belong to a certain film group => some hidden elements (binary states) are activated
- ❑ 6 visible elements send messages to hidden variables
- ❑ The activation energy of the hidden element is the weighted sum of messages coming from the input elements

Interpretation of a restricted Boltzmann machine by an example (2)



Interpretation of a restricted Boltzmann machine by an example (3)

- ❑ Bottom-up:
  - – RBM attempts to explain a person's preferences in terms of hidden variables, i.e. what kind of movies do people prefer?
  - – However, if a person chooses Harry Potter, Avatar, and LOTR 3 films, this does not mean that a hidden element that corresponds to the SF/fantasy group will be activated
  - – This means that the probability of activation of this hidden element will be increased due to the high activation energy

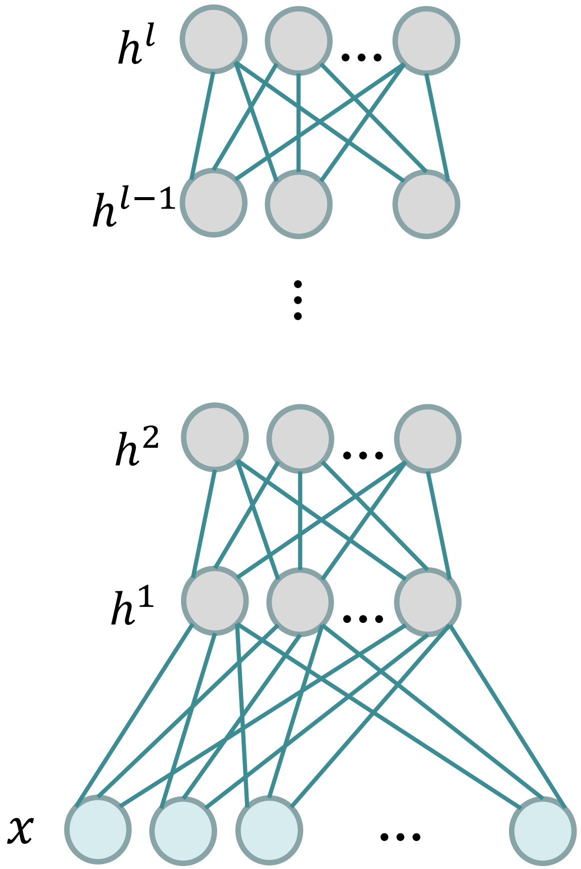
Interpretation of a restricted Boltzmann machine by an example (4)

- ❑ Top-bottom:
  - – If you know that some people prefer SF/fantasy, then you can ask RBM which films belong to the specified group and which films a person might like
  - – In this case, the hidden variables send messages to visible ones => the status of the visible variables is updated
  - – However, it can not be guaranteed that Harry Potter, Avatar, and LOTR 3 films will always be recommended to a person. Only a subset of them can be recommended

DEEP BOLTZMANN MACHINES AND DEEP BELIEF NETWORKS

Deep Boltzmann machine

- ❑ By analogy with the stack of autoencoders, it is possible to construct a stack of restricted Boltzmann machines
- ❑ If the states of the neurons of each hidden layer depend on the previous and the next ones (symmetric connections), then this model is called the ***deep Boltzmann machine*** (DBM)



Training a deep Boltzmann machine (1)

- ❑ The parameters of this model can be trained by analogy with a single-layer Boltzmann machine
- ❑ It is necessary to determine the joint distribution function through the energy of the system $E\left(x, h^1, h^2, \dots, h^l;\ \theta\right)$, and express the probability of visible neurons $p(x;\theta)$:

$$E\left(x, h^1, h^2, \dots, h^l;\ \theta\right) = -\sum_{i=1}^{l} \left(h^{i-1}\right)^T W^i h^i,$$

$$p(x;\theta) = \frac{1}{Z} \sum_{h^1,h^2,\dots h^l} e^{-E\left(x,h^1,h^2,\dots,h^l;\ \theta\right)},$$

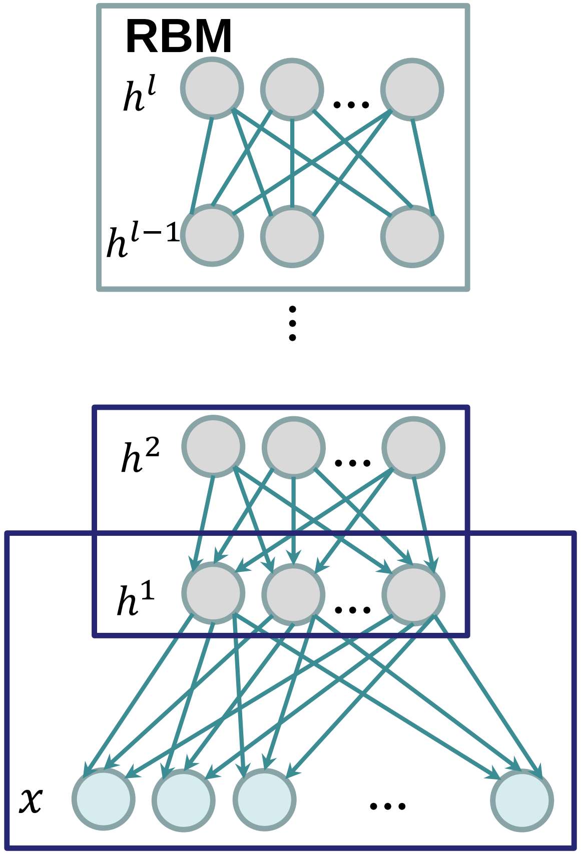
where $x = h^0$ is a vector of visible neurons, h^i is a vector of hidden neurons, $\theta = \{W^1, \dots, W^l\}$ is a set of parameters

Training a deep Boltzmann machine (2)

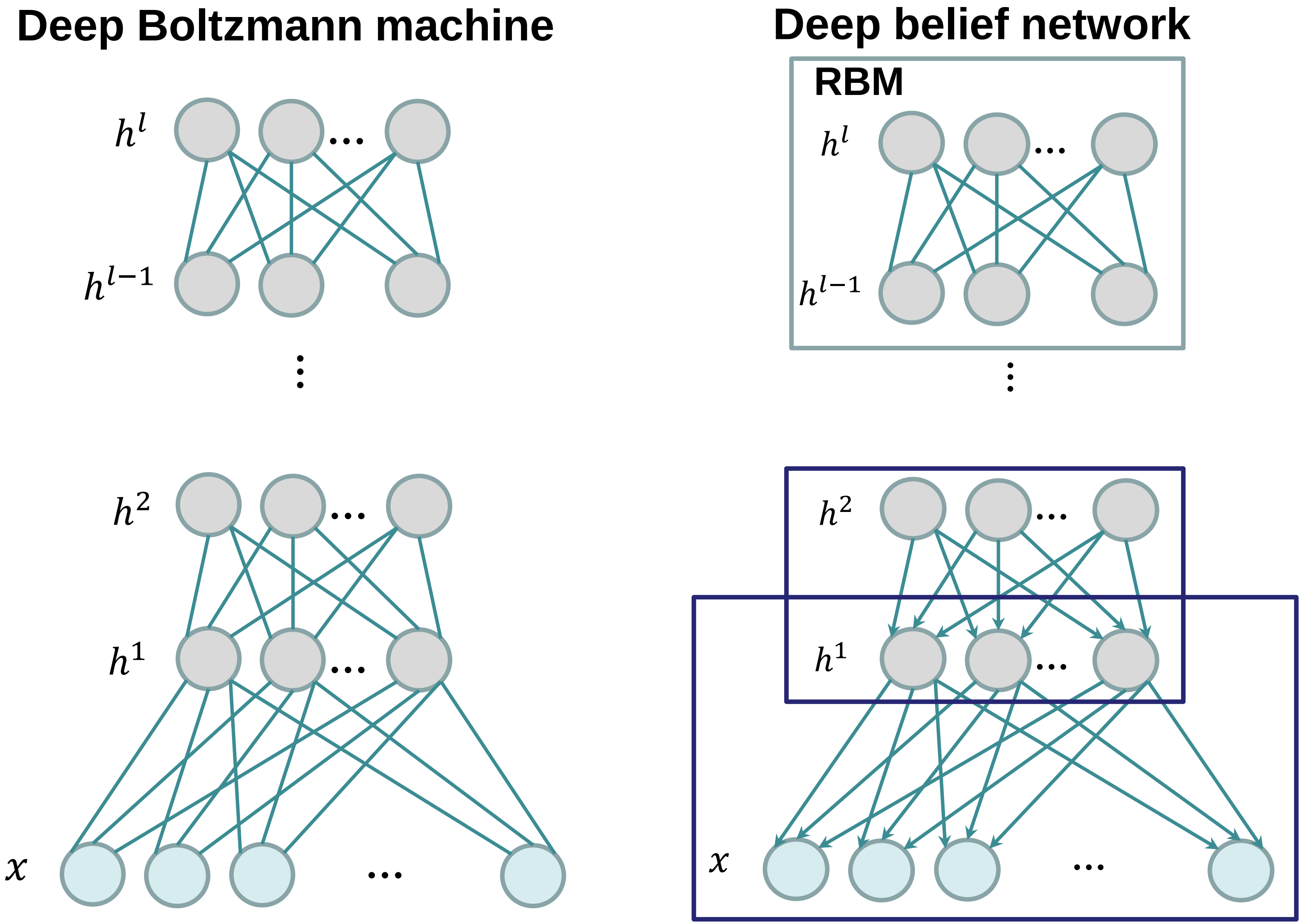
- Next, we need to obtain formulas for calculating conditional probabilities $p\left(x\middle|h^{(1)}\right)$, $p\left(h^k\middle|h^{k-1}\right)$, $p\left(h^k\middle|h^{k-1},h^{k+1}\right)$ and the derivatives of the probability distribution function with respect to each parameter
- The procedure of training weights goes the slower, the further hidden neurons are located from the layer of visible neurons

Deep belief network

- ❑ In 2006, J. Hinton proposed an approach that allows quick initialization of the model parameters
- ❑ This approach involves ***"greedy" layer-by-layer network training***
- ❑ Training is performed on the assumption that each layer of neurons does not depend on the next one
- ❑ After the stack of restricted Boltzmann machines is trained, the system can be viewed as a single "probabilistic model" called the ***deep belief network*** (DBN)



The difference between a deep Boltzmann machine and a deep belief network



Sigmoid belief network

- ***Sigmoid belief network*** consisting of l layers models the function of joint distribution $p\left(x, h^1, h^2, \dots, h^l; \theta\right)$
- The joint probability distribution function can be expressed as a product of conditional probabilities:

$$p\left(x, h^1, \dots, h^l; \theta\right) = p(x|h^1; \theta)\left(\prod_{k=1}^{l-2} p\left(h^k \middle| h^{k+1}; \theta\right)\right) p\left(h^{l-1}, h^l; \theta\right)$$

Conditional probability distribution functions for a sigmoid belief network

- Since the neurons on the layer are independent, and the activation function on each layer is a sigmoid, the distribution functions of the conditional probability are calculated as follows:

$$p\left(h^k\right) = \prod_i p\left(h_i^k\right),$$

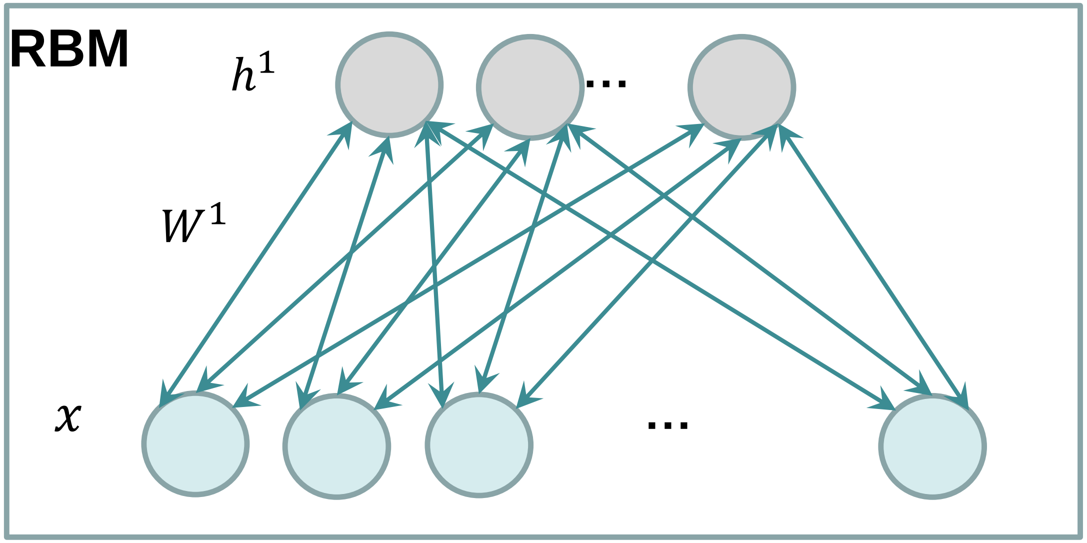
$$q\left(h^k \middle| h^{k-1};\theta\right) = p\left(h^k \middle| h^{k-1};\theta\right) = \prod_j p\left(h_j^k \middle| h^{k-1};\theta\right)$$

$$= \prod_j sigm\left(W_{j\cdot}^k h^{k-1}\right),$$

$$p\left(h^{k-1} \middle| h^k;\theta\right) = \prod_j p\left(h_j^{k-1} \middle| h^k;\theta\right) = \prod_j sigm\left(W_{\cdot j}^{k^T} h^k\right)$$

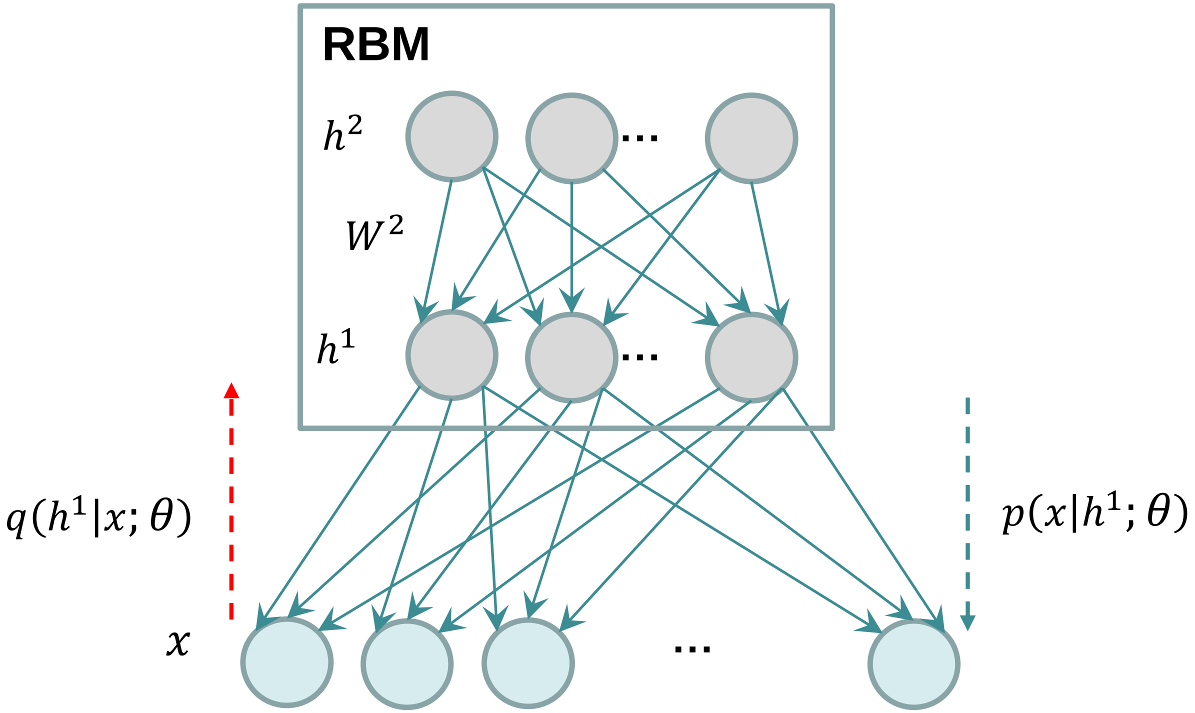
Greedy layer-by-layer training of a sigmoid belief network (1)

1. Construct a restricted Boltzmann machine based on the input layer x and the first layer of hidden neurons h^1, training the parameters of the first layer W^1



Greedy layer-by-layer training of a sigmoid belief network (2)

2. Construct a restricted Boltzmann machine based on the hidden layers h^1 и h^2 and training the second layer parameters W^2. The weight matrix W^1 is fixed, and the vector values h^1 are sampled from the distribution $q(h^1|x; W^1)$ obtained at the previous step



Greedy layer-by-layer training of a sigmoid belief network (3)

3. Construct restricted Boltzmann machines based on the following pair of hidden layers h^k and h^{k+1}, and train the corresponding parameters W^{k+1}. The weight matrix W^k is fixed, h^k are sampled from the distribution $q\left(h^k \middle| h^{k-1}; \theta\right)$

Note: at the end of the greedy layer-by-layer training of a deep belief network, it is possible to fine-tune the parameters, for example, using the “wake-sleep” algorithm*

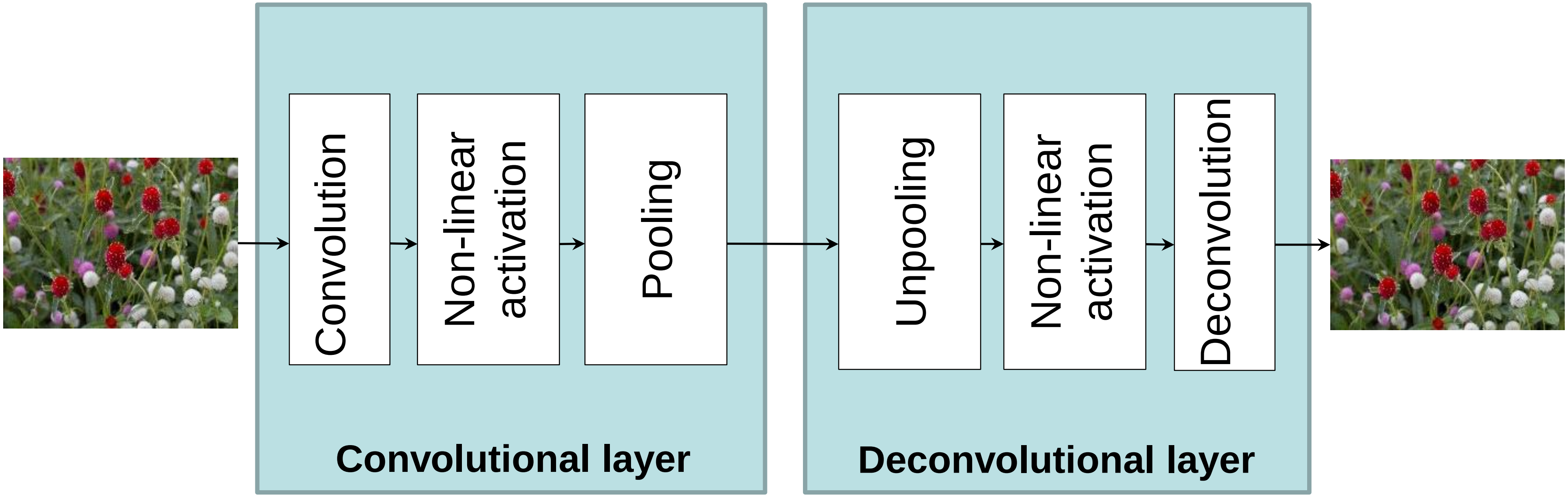
* Hinton G.E., Dayan P., Frey B. J., Neal R.M. The wake-sleep algorithm for unsupervised neural networks // Science. – 1995. – Vol. 268, pp. 1558–1161.

DECONVOLUTIONAL NEURAL NETWORKS

Deconvolutional neural networks (1)

- ***Deconvolutional neural networks*** proposed as a convolutional version of sparse autoencoders
- Deconvolutional neural networks are used for visualization of feature maps in convolutional networks
- Later, the idea of deconvolutional neural networks was widely used to solve the problem of semantic segmentation
- In the simplest case, these networks consist of two blocks:
  - Convolutional layer is a sequence of convolution, activation function and pooling
  - Deconvolutional layer is a sequence of reverse transforms (unpooling, activation function and deconvolution)

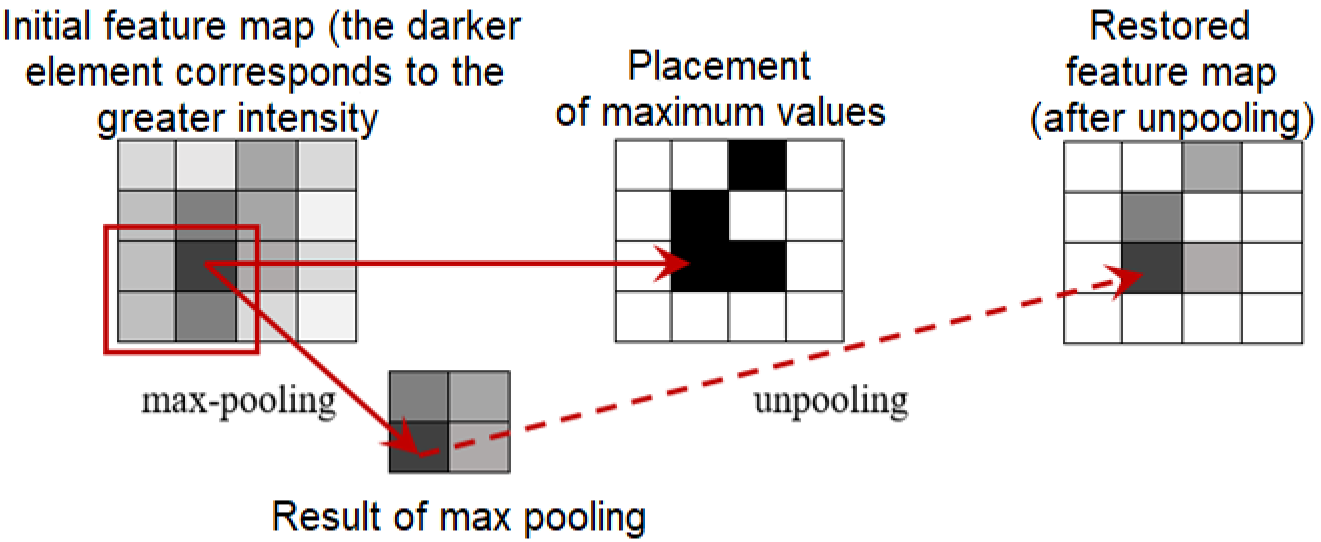
Deconvolutional neural networks (2)



- As an activation function, let us consider the ReLU function
- As a pooling operation, let us consider the max pooling

Reverse transforms (1)

- ***Unpooling*** is a reverse transform to pooling
  - The appeal is realized by storing the indices of the element of the feature map in which the maximum value
  - When deployed, the maximum value is placed in the corresponding cell, and the surrounding values are reset to zero


Initial feature map (the darker element corresponds to the greater intensity
Placement of maximum values
Restored feature map (after unpooling)
max-pooling
unpooling
Result of max pooling

Reverse transforms (2)

- ***Reverse non-linear activation***
  - The inverse function of ReLU is a ReLU function
  - The authors of the method argue the choice of such an inverse transformation in that the convolution is applied to the positive part during feed forward of the network, so, during the backward, the deconvolution should also be applied to the positive part

Reverse transforms (3)

- ***Deconvolution***
  - Deconvolution involves the use of the same filters as in the computation of a convolution
  - The only difference is that the filter kernels are transposed
  - An output image close to the original one

Training the deconvolutional neural network (1)

- ***The task of training the deconvolutional network*** is to obtain a feature map before the deconvolution operation, applying a set of filters to which we can obtain an output as close as possible to the input image

- Assumed that $y = (y_1, y_2, \dots, y_s)$ is an input image, $\tilde{y} = (\widetilde{y_1}, \widetilde{y_2}, \dots, \widetilde{y_s})$ is an output image, where s is a number of channels
- ***The goal of training*** is to restore the feature map $z = (z_1, z_2, \dots, z_K)$, which provides the best approximation to the input image. The network output is calculated by applying convolutional filters to the feature map z

Training the deconvolutional neural network (2)

- The network output is calculated by applying convolutions to the feature map z:

$$\widetilde{y_k} = \sum_i \langle z_i, f_{i,k} \rangle$$

- The cost function is the Euclidean norm

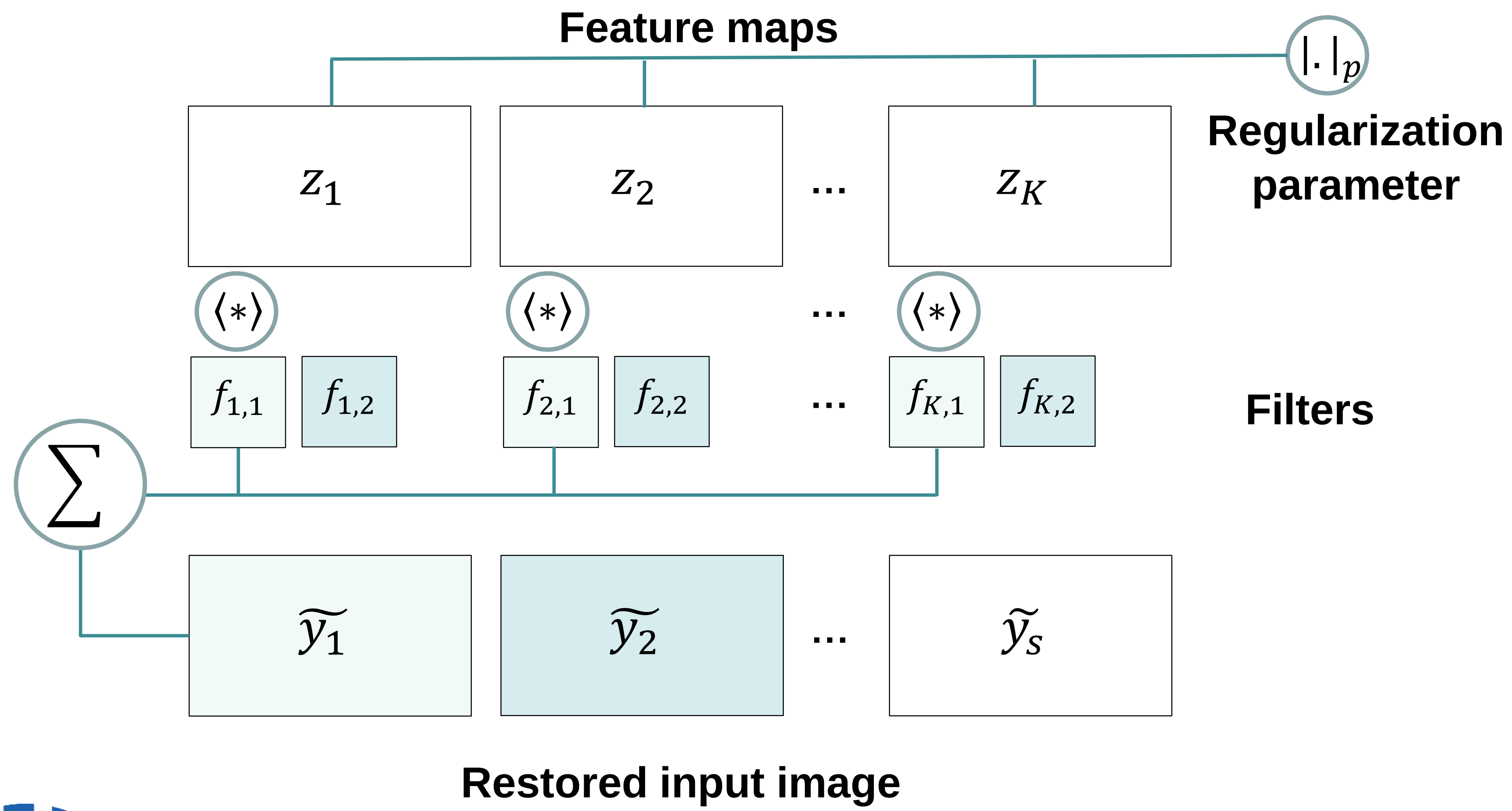
$$L(y, \tilde{y}) = \sum_k \left\| \sum_i \langle z_i, f_{i,k} \rangle - y_k \right\|_2^2 ,$$

- The minimized functional is as follows:

$$J(\theta) = \frac{\lambda}{2} \sum_{y \in I} \sum_k \left\| \sum_i \langle z_i, f_{i,k} \rangle - y_k \right\|_2^2 + \sum_{y \in I} \sum_i |z_i|^p \rightarrow \min_{z,\theta} \quad ,$$

where $\theta = \{f_{i,k}\}$ is a set of parameters, I is a set of input images

Training the deconvolutional neural network (3)



Training the deconvolutional neural network (4)

- ❑ The above optimization problem is solved using the ISTA iteration algorithm (Iterative Shrinkage-Thresholding Algorithm)*

* Beck A., Teboulle M. A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems [https://people.rennes.inria.fr/Cedric.Herzet/Cedric.Herzet/Sparse_Seminar/Entrees/2012/11/12_A_Fast_Iterative_Shrinkage-Thresholding_Algorithmfor_Linear_Inverse_Problems_(A._Beck,_M._Teboulle)_files/Breck_2009.pdf].

EXAMPLE OF UNSUPERVISED LEARNING APPLICATION FOR PRE-TRAINING PARAMETERS

Architecture of fully-connected neural network

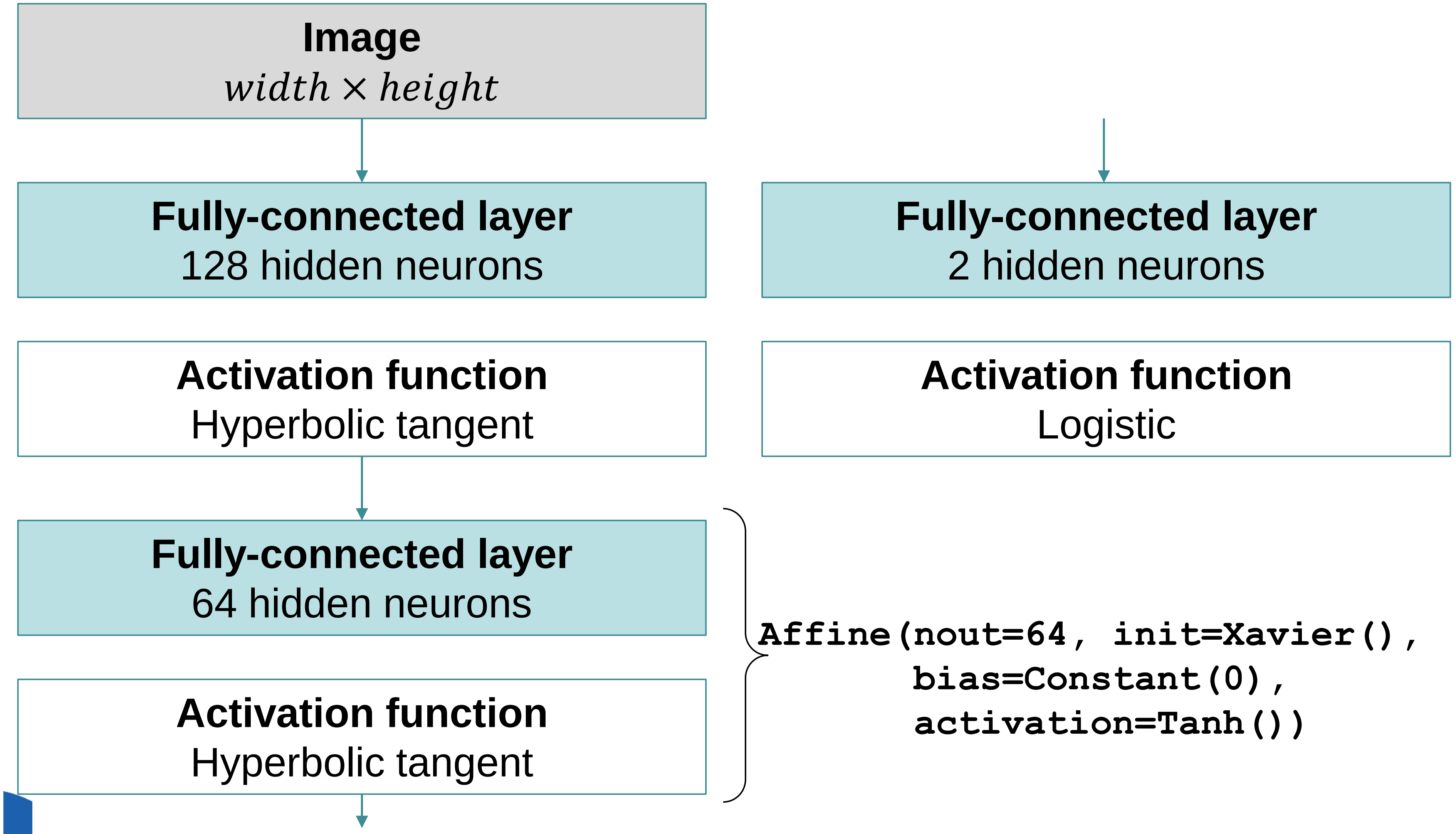


Image
$width \times height$

Fully-connected layer
128 hidden neurons

Activation function
Hyperbolic tangent

Fully-connected layer
64 hidden neurons

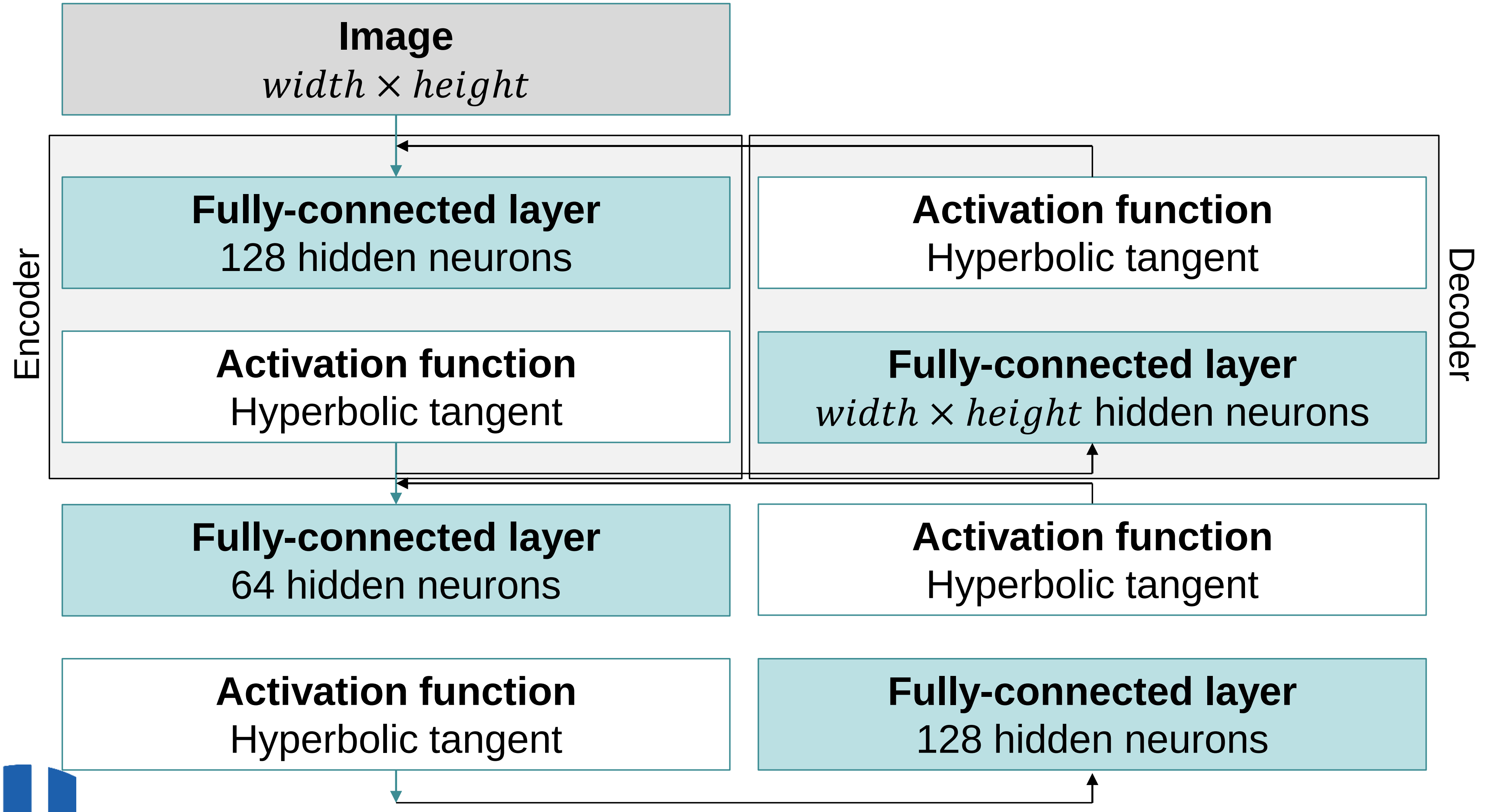
Activation function
Hyperbolic tangent

Fully-connected layer
2 hidden neurons

Activation function
Logistic

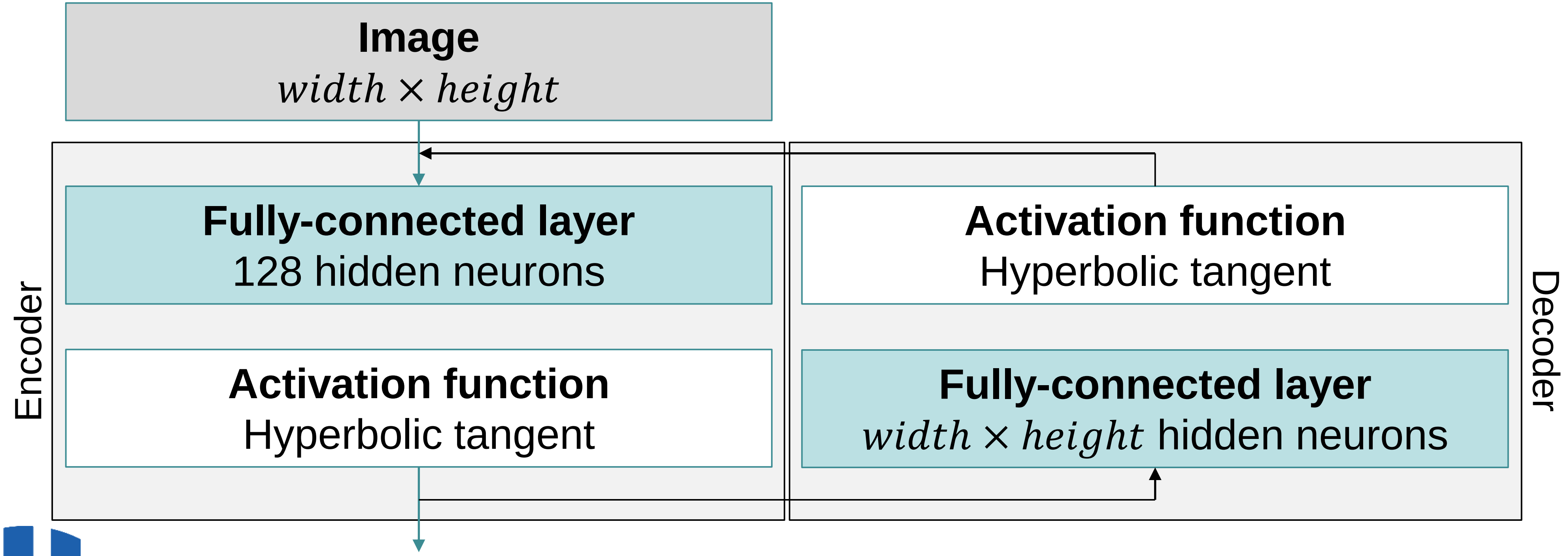
```
Affine(nout=64, init=Xavier(),
       bias=Constant(0),
       activation=Tanh())
```

Stack of autoencoders



Example of weight pre-training using Intel® neon™ Framework (1)

- ***The first autoencoder***
  - Input data of encoder are images
  - Output data of decoder are images
  - Labeled data: <image, image>



Example of weight pre-training using Intel® neon™ Framework (2)

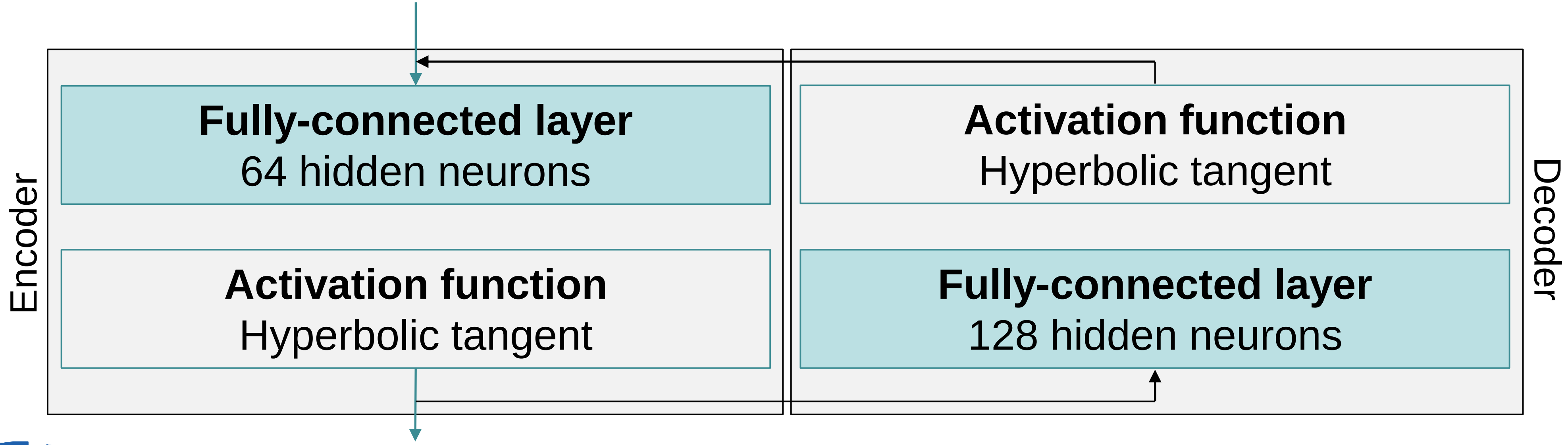
```
# the first autoencoder
def generate_mlp2b2_128_ae_stacked_step1_model(input_shape):
    output_size=reduce(lambda p, x:p*int(x), input_shape, 1)

    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),
        Affine(nout=128, init=Gaussian(scale=0.1),
          bias=Constant(0), activation=Tanh(), name='fc_1'),
        Affine(nout=output_size, init=Gaussian(scale=0.1),
          bias=Constant(0), activation=Tanh(), name='fc_-1')
    ]
    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

Example of weight pre-training using Intel® neon™ Framework (3)

- ***The second autoencoder***
  - How to construct?

Example of weight pre-training using Intel® neon™ Framework (4)

- ***The second autoencoder***
  - Input data is an encoder output of the previous autoencoder
  - Let we save the pre-trained encoder layer of the previous autoencoder in the second autoencoder model
  - Labeled data: <image, output of the previous autoencoder>



Example of weight pre-training using Intel® neon™ Framework (5)

```
# the second autoencoder
def generate_mlp2b2_128_ae_stacked_step2_model(input_shape):
    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),
        Affine(nout=128, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
        Affine(nout=64, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_2'),
        Affine(nout=128, init=Xavier(),
            bias=Constant(0),
            activation=Tanh(), name='fc_-2') ]
    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

Example of weight pre-training using Intel® neon™ Framework (6)

```
# the final network
def generate_mlp2b2_128_ae_stacked_model():
    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),
        Affine(nout=128, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
        Affine(nout=64, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_2'),
        Affine(nout=2, init=Xavier(), bias=Constant(0),
            activation=Logistic(shortcut=True), name='cls')
    ]
    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=CrossEntropyBinary())
    return (model, cost)
```

Example of weight pre-training using Intel® neon™ Framework (7)

- Training the stack of autoencoders:
  - Prepare training data for the first autoencoder (the training data consist of pairs <image, image>)
  - Train the first autoencoder
  - Remove the decoder from the model of the trained autoencoder
  - Prepare the training data for the second autoencoder (the training data consists of pairs <image, output of the previous encoder>)
  - Train the second autoencoder
  - Remove the decoder from the model of the trained autoencoder
  - Train the final network with pre-trained weights (use transfer learning)

Example of weight pre-training using Intel® neon™ Framework (8)

- The complete sources that implement the above sequence of steps are described in the relevant practice (**`Practice4_ae/main_train_stacked_autoencoder.py`**)

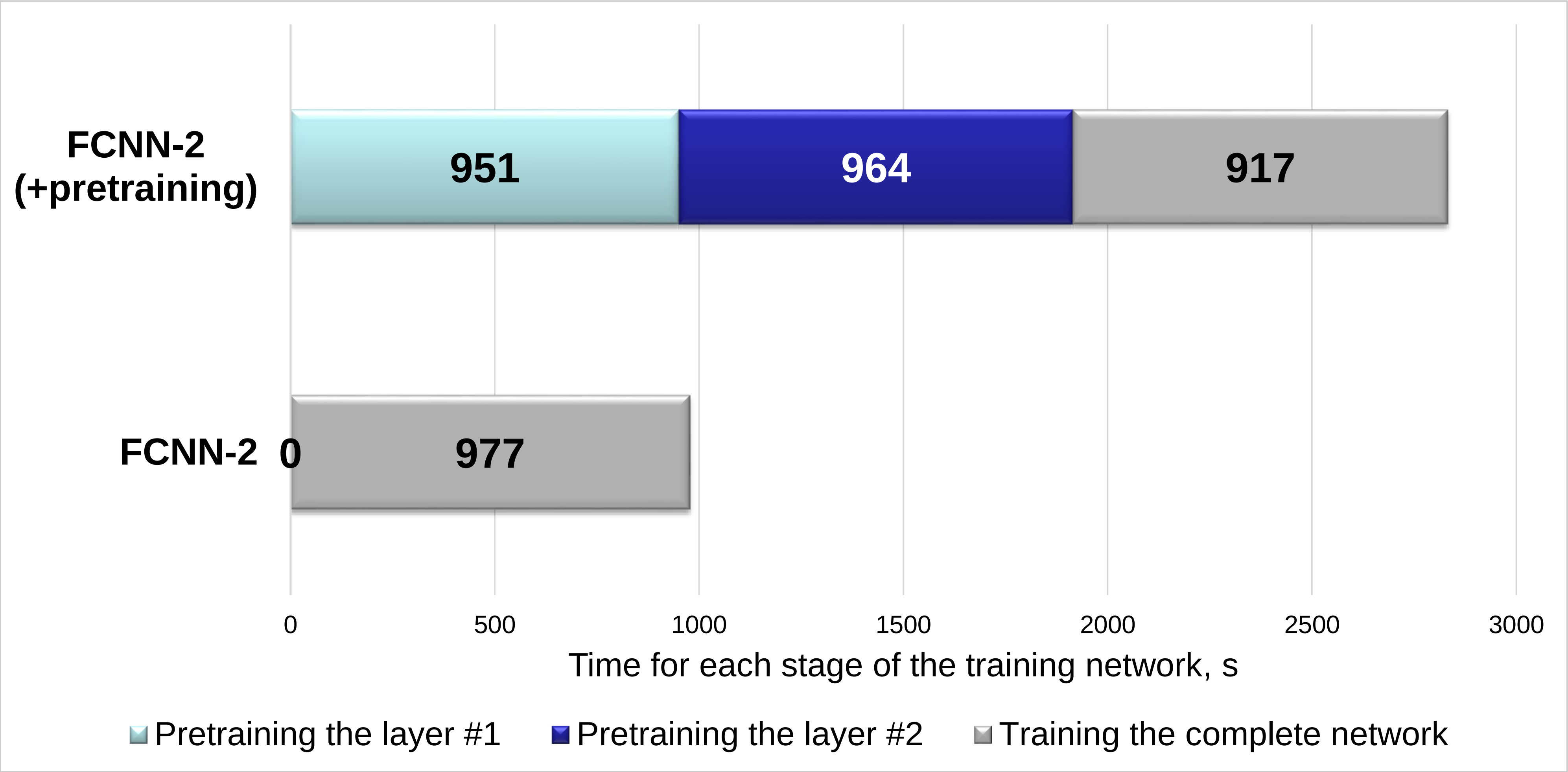
Infrastructure

- ❑ CPU: Intel® Xeon® CPU E5-2660 0 @ 2.20GHz
- ❑ GPU: Tesla K40s 11Gb
- ❑ OS: Ubuntu 16.04.4 LTS
- ❑ Frameworks:
  - Intel® neon™ Framework 2.6.0
  - CUDA 8.0
  - Python 3.5.2
  - Intel® Math Kernel Library 2017 (Intel® MKL)

Summary results

Network id	Accuracy, %	Training time, s
FCNN-1	71.2	932
FCNN-2	73.5	977
FCNN-3	**77.7**	1013
CNN-1	79.3	1582
CNN-2	**83.5**	2030
ResNet-18 (90 epochs)	**81.3**	15127
ResNet-50 (30 epochs)	80.9	11849
FCNN-2 (+pretraining, 30 epochs)	**78.9**	2832

Time distribution between pre-training network weights and training the complete network



Conclusion

- Pre-training of network parameters takes a long time, comparable with the training time of the network
- The developed methods of random parameter initialization make it possible to obtain a good initial approximation to optimize the cost function
- At present, methods of pre-training parameters are rarely used in solving practical problems because of the effective random generators existence

Literature

- Haykin S. Neural Networks: A Comprehensive Foundation. – Prentice Hall PTR Upper Saddle River, NJ, USA. – 1998.
- Osovsky S. Neural networks for information processing. – 2002.
- Goodfellow I., Bengio Y., Courville A. Deep Learning. – MIT Press. – 2016. – [http://www.deeplearningbook.org].

Authors

- **Kustikova Valentina Dmitrievna**
  Phd, lecturer, department of Computer software and supercomputer technologies, Institute of Information Technologies, Mathematics and Mechanics, Nizhny Novgorod State University
  valentina.kustikova@itmm.unn.ru
- **Zhiltsov Maxim Sergeevich**
  master of the 1st year training, Institute of Information Technology, Mathematics and Mechanics, Nizhny Novgorod State University
  zhiltsov.max35@gmail.com
- **Zolotykh Nikolai Yurievich**
  D.Sc., Prof., department of Algebra, geometry and discrete mathematics, Institute of Information Technologies, Mathematics and Mechanics, Nizhny Novgorod State University
  nikolai.zolotykh@itmm.unn.ru