## Restricted Boltzmann machine (RBM)

**Architecture:**

- Neural network with cycles and symmetric connections, neurons divided into two disjoint sets:
  - $V$ - visible
  - $H$ - hidden

  Connections: $V \times S$ (complete bipartite graph)
- $N$ is a *set of all neurons*.
- Denote by $\xi_j$ the inner potential and by $y_j$ the output (i.e. state) of neuron $j$.
  *State of the machine*: $\vec{y} \in \{0, 1\}^{|N|}$.
- Denote by $w_{ji} \in \mathbb{R}$ the weight of the connection from $i$ to $j$ (and thus also from $j$ to $i$).
- Consider bias: $w_{j0}$ is the weight between $j$ and a neuron 0 whose value $y_0$ is always 1.

## RBM – activity

**Activity:** States of neurons initially set to values of $\{0, 1\}$, i.e. $y_j^{(0)} \in \{0, 1\}$ for $j \in N$.

In the step $t + 1$ do the following:

- $t$ even: randomly choose new values of all hidden neurons, for every $j \in H$

$$\mathbf{P}\left[y_j^{(t+1)} = 1\right] = 1 \Big/ \left(1 + \exp\left(-w_{j0} - \sum_{i \in V} w_{ji} y_i^{(t)}\right)\right)$$

- $t$ odd: randomly choose new values of all visible neurons, for every $j \in V$

$$\mathbf{P}\left[y_j^{(t+1)} = 1\right] = 1 \Big/ \left(1 + \exp\left(-w_{j0} - \sum_{i \in H} w_{ji} y_i^{(t)}\right)\right)$$

## Thermal equilibrium

Fix a temperature $T$ (i.e. $T(t) = T$ for $t = 1, 2, \ldots$).

**Theorem**

*For every $\gamma^* \in \{0,1\}^{|N|}$ we have that*

$$\lim_{t\to\infty} \mathbf{P}\left[\vec{y}^{(t)} = \gamma^*\right] = \frac{1}{Z} e^{-E(\gamma^*)/T}$$

*where*

$$Z = \sum_{\gamma\in\{0,1\}^{|N|}} e^{-E(\gamma)/T}$$

*and*

$$E(\gamma) = -\sum_{i\in V, j\in H} w_{ji} y_j^\gamma y_i^\gamma - \sum_{i\in V} w_{i0} y_i^\gamma - \sum_{j\in H} w_{j0} y_j^\gamma$$

Define $p_N(\gamma^*) := \lim_{t\to\infty} \mathbf{P}\left[\vec{y}^{(t)} = \gamma^*\right]$ for every $\gamma^* \in \{0,1\}^{|N|}$.

# RBM – learning

**Learning:**
Let $p_d$ be a probability distribution on states of visible neurons, i.e. on $\{0,1\}^{|V|}$.

Our goal is to find a configuration of the network $W$ such that $p_V \approx p_d$.

A suitable measure of difference between probability distributions $p_V$ and $p_d$ is relative entropy weighted by probabilities of states (Kullback-Leibler divergence):

$$\mathcal{E}(W) = \sum_{\alpha \in \{0,1\}^{|V|}} p_d(\alpha) \ln \frac{p_d(\alpha)}{p_V(\alpha)}$$

## RBM – learning

Minimize $\mathcal{E}(\vec{w})$ using gradient descent, i.e. compute a sequence of weight matrices: $W^{(0)}, W^{(1)}, \ldots$

- initialise $W^{(0)}$ randomly, close to 0
- in step $t+1$ compute $W^{(t+1)}$ as follows:

  $$W_{ji}^{(t+1)} = W_{ji}^{(t)} + \Delta W_{ji}^{(t)}$$

  where

  $$\Delta W_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial \mathcal{E}}{\partial w_{ji}}(W^{(t)})$$

  is the update of the weight $w_{ji}$ in the step $t+1$ and $0 < \varepsilon(t) \leq 1$ is the learning rate in the step $t+1$.

It remains to compute $\frac{\partial \mathcal{E}}{\partial w_{ji}}(W)$.

## RBM – learning

For sufficiently large $t^*$ (i.e. in thermal equilibrium) we have

$$\frac{\partial \mathcal{E}}{\partial w_{ji}} \approx -\frac{1}{T}\left(\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed} - \left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}\right)$$

- $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed}$ is the expected value of $y_j^{(t^*)} y_i^{(t^*)}$ in the thermal equilibrium assuming that values of visible neurons are **fixed** at the beginning of computation according to $p_d$.
- $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ is the expected value of $y_j^{(t^*)} y_i^{(t^*)}$ in the thermal equilibrium (no values fixed).

**Problem:** Computation of $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ takes long time.

$\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ can be estimated with $\left\langle y_j y_i \right\rangle_{recon}$, the expectation of $y_j^{(3)} y_i^{(3)}$ when values of visible neurons chosen by $p_d$.

## RBM – learning

Thus

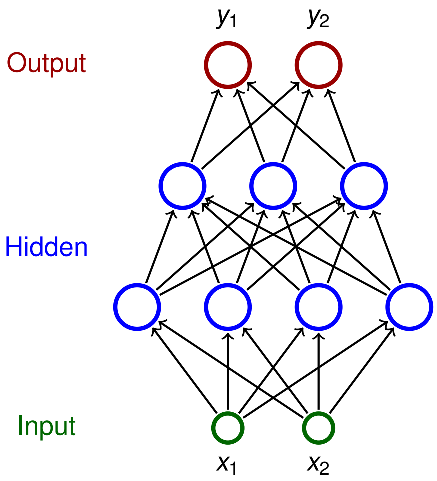
$$\Delta w_{ji}^{(t)} = \varepsilon(t) \cdot \left( \left\langle y_j y_i \right\rangle_{fixed} - \left\langle y_j y_i \right\rangle_{recon} \right)$$

- Compute $\left\langle y_j y_i \right\rangle_{fixed}$ as follows: Let $\mathcal{Y} := 0$ and repeat the following $q$ times:
  - **fix** values of visible neurons randomly by $p_d$
  - simulate one step of computation, add $y_j y_i$ to $\mathcal{Y}$

  For a suitable $q$ we have that $\mathcal{Y}/q$ estimates $\left\langle y_j y_i \right\rangle_{fixed}$ well.
- Compute $\left\langle y_j y_i \right\rangle_{recon}$ as follows: Let $\mathcal{Y} := 0$ and repeat $q$ times:
  - choose initial values of visible neurons by $p_d$
  - simuate three steps, add $y_j y_i$ to $\mathcal{Y}$
    (i.e. compute values of hidden neurons, then of visible ones (reconstruction of the input) and then of hidden neurons)

  For a suitable $q$ we have that $\mathcal{Y}/q$ estimates $\left\langle y_j y_i \right\rangle_{recon}$ well.

## Deep MLP



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the *i*-th layer are connected with all neurons in the $i+1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

## Deep MLP

- Denote
  - $X$ a set of *input* neurons
  - $Y$ a set of *output* neurons
  - $Z$ a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices $i, j$ etc.
  - $\xi_j$ is the inner potential of the neuron $j$ *after the computation stops*
  - $y_j$ is the output of the neuron $j$ *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- $w_{ji}$ is the weight of the connection **from** $i$ **to** $j$

  (in particular, $w_{j0}$ is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where $b_j$ is the bias of the neuron $j$)
- $j_{\leftarrow}$ is a set of all $i$ such that $j$ is adjacent from $i$
  (i.e. there is an arc **to** $j$ from $i$)
- $j^{\rightarrow}$ is a set of all $i$ such that $j$ is adjacent to $i$
  (i.e. there is an arc **from** $j$ to $i$)

## Deep MLP – activity

- inner potential of neuron $j$:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- activation function $\sigma_j$ for neuron $j$ (arbitrary differentiable):

$$y_j = \sigma_j(\xi_j)$$

A deep MLP is evaluated layer-wise, for each $j \in Y$ we have that $y_j(\vec{w}, \vec{x})$ is the value of the output neuron $j$ after evaluating the network with weights $\vec{w}$ and input $\vec{x}$.

## Deep MLP – learning

- Given a **training set** $\mathcal{T}$ of the form

  $$\left\{\left(\vec{x}_k, \vec{d}_k\right) \;\middle|\; k = 1, \ldots, p\right\}$$

  Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by $d_{kj}$ the desired output of the neuron $j$ for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function – mean square error (for example):**

  $$E(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} E_k(\vec{w})$$

  where

  $$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

  Other errors such as cross-entropy possible.

## Convolutional networks – SGD

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

    $$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

    where

    $$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \frac{1}{|T|} \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

Here $T$ is a *minibatch* (of a fixed size),

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example $k$

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially. **Epoch** consists of one round through all data.

## Why deep networks

... if one hidden layer is able to represent an arbitrary (reasonable) function?

- One hidden layer may be very inefficient, i.e. huge amount of neurons may be needed. One can show that
  - the number of hidden neurons may be exponential w.r.t. the dimension of the input,
  - networks with multiple layers may be exponentially more succinct as opposed to single hidden layer.

... ok, so let's try to teach deep networks ... using backpropagation?

**Problems:**

- Gradient may vanish/explode when backpropagated through many layers.
- Deep networks (with many neurons) overfit very easily.

## Deep MLP - vanishing gradient

For every $w_{ji}$ we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

and for every $j \in Z \smallsetminus X$ holds

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{pro } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{pro } j \in Z \smallsetminus (Y \cup X)$$

$\sigma_r'(\xi_r) \cdot w_{rj}$ **is less than one** for standard logistic sigmoid and weights initialized close to 0.

## Deep MLP – pretraining

Assume $k$ layers. Denote

- $W_i$ the weight matrix between layers $i-1$ and $i$
- $F_i$ function computed by the "lower" part of the MLP consisting of layers $0, 1, \ldots, i$

  $F_1$ is a function which consists of the input and the first hidden layer (which is now considered as the output layer).

**Crucial observation:** For every $i$, the layers $i-1$ and $i$ together with the matrix $W_i$ can be considered as a RBM (assume $T = 1$).

Denote such a RBM as $B_i$.

## Deep MLP – pretraining

For now, consider only input vectors $\vec{x}_1, \ldots, \vec{x}_p$ where $\vec{x}_k \in \{0,1\}^n$ for all $k = 1, \ldots, p$.

- **unsupervised pretraining:** Gradually, for every $i = 1, \ldots, k$, train RBM $B_i$ on randomly selected inputs from the training set:

  $$F_{i-1}(\vec{x}_1), \ldots, F_{i-1}(\vec{x}_p)$$

  using the training algorithm for RBM (here $F_0(\vec{x}_i) = \vec{x}_i$).

  (Thus $B_i$ learns from training samples **transformed by the already pretrained layers** $0, \ldots, i-1$**)**

We obtain a *deep belief network* $\mathcal{D}$ representing a distribution given by $\vec{x}_1, \ldots, \vec{x}_p$.
(Recall that in such a distribution the probability of a given $\vec{x}$ is equal to the relative frequency of $\vec{x}$ in $\vec{x}_1, \ldots, \vec{x}_p$.)

## Deep belief network

The network $\mathcal{D}$ can be used to sample from the distribution as follows:

- Simulate the topmost RBM for some steps (ideally to thermal equilibrium), this gives values of neurons in the two topmost layers.
- Propagate the values downwards by always simulating one step of the corresponding RBM. That is,
  - you have already computed values of neurons in layers $k$ and $k-1$.
  - To compute values of neurons in the layer $k-2$, simulate one step of RBM $B_{k-1}$, that is sample values of neurons in the layer $k-2$ using RBM dynamics of $B_{k-1}$ with values of the layer $k-1$ fixed.
  - Similarly, compute values of $k-3$ by simulating $B_{k-2}$ ... etc.
  - ... finally obtain values of input neurons.
- Probability with which a concrete input $\vec{x}$ is sampled by the above procedure is the probability of $\vec{x}$ in the distribution represented by $\mathcal{D}$.

## Deep MLP – training with pretraining

Now consider supervised learning with a training set:
$\mathcal{T} = \left\{ \left( \vec{x}_k, \vec{d}_k \right) \;\middle|\; k = 1, \ldots, p \right\}$.
Still assume that $\vec{x}_k \in \{0,1\}^n$.

- **unsupervised pretraining:** Gradually, for every $i = 1, \ldots, k$, train RBM $B_i$ on randomly selected inputs from the training set:

  $$F_{i-1}(\vec{x}_1), \ldots, F_{i-1}(\vec{x}_p)$$

  using the training algorithm for RBM (here $F_0(\vec{x}_i) = \vec{x}_i$).

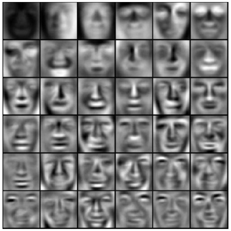
  (Thus $B_i$ learns from training samples **transformed by the already pretrained layers** $0, \ldots, i-1$**)**

  Obtain $\mathcal{D}$.
- Add one (or more) layer to the top of $\mathcal{D}$ and consider the result to be MLP.

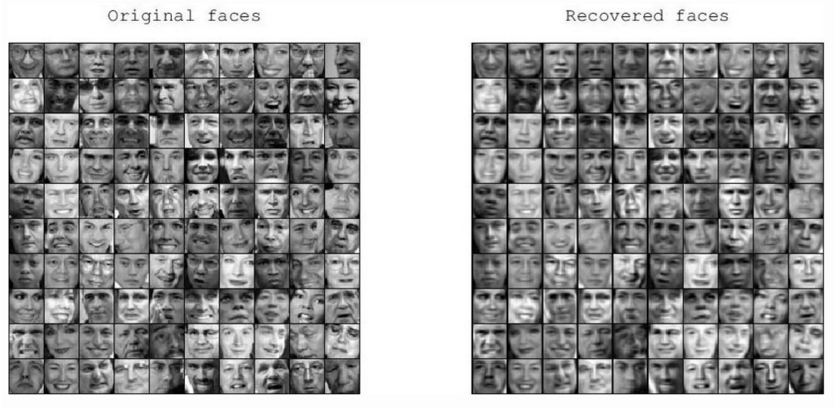
  (i.e. forget the RBM dynamics and start considering the network as MLP with sigmoidal activations).
- **supervised fine-tuning:** Train in supervised mode (on the training set $\mathcal{T}$) using e.g. gradient descent + backprop.

## Application – dimensionality reduction

- Dimensionality reduction: A mapping $R$ from $\mathbb{R}^n$ to $\mathbb{R}^m$ where
  - $m < n$,
  - for every example $\vec{x}$ we have that $\vec{x}$ can be "reconstructed" from $R(\vec{x})$.
- Standard method: PCA (there are many linear as well as non-linear variants)



## Reconstruction – PCA



1024 pixels compressed to 100 dimensions (i.e. 100 numbers).

## Autoencoders

Dimensionality reduction using MLP.

- Consider MLP $n - m - n$ where $m << n$.
- The same vector on the input as well as output.
- Dimensionality reduction:
  - Encoding: Compute values of hidden neurons.
  - Reconstruction: Compute values of output neurons given values of hidden neurons.

Can also be used for compression (in communication).

One can show that if linear neurons are used, the method implements PCA.

## Autoencoder – historical implementation

**Architecture:** MLP 64 – 16 – 64

**Activity:** activation function: hyperbolic tangens with limits $-1$ and 1

**Learning:**
Data:

- Images $256 \times 256$, 8 bits per pixel.
- Samples: input and output is a frame $8 \times 8$, randomly selected in the image.
- Inputs normalized to $[-1, 1]$.

Learning:

- Backpropagation
- Learning rate: 0.01 for hidden, 0.1 pro output
- trained in 50 000 - 100 000 iterations

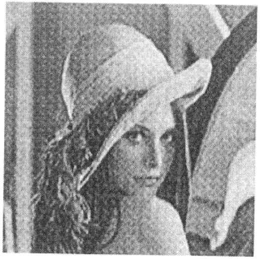
The goal was to compress images to smaller data size.

# Dimensionality reduction – compression

(A)

(B)

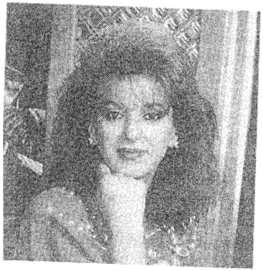
(C)

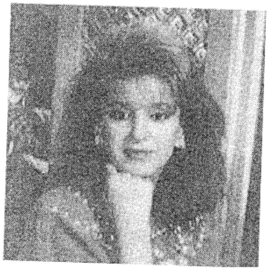

(D)

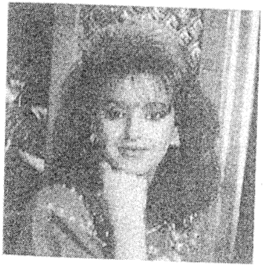
A frame $8 \times 8$ passes through the image $256 \times 256$ (no overlap)

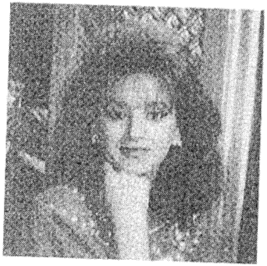
- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

# Dimensionality reduction – compression


(A)


(B)


(C)


(D)

New image (trained on the previous one):

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

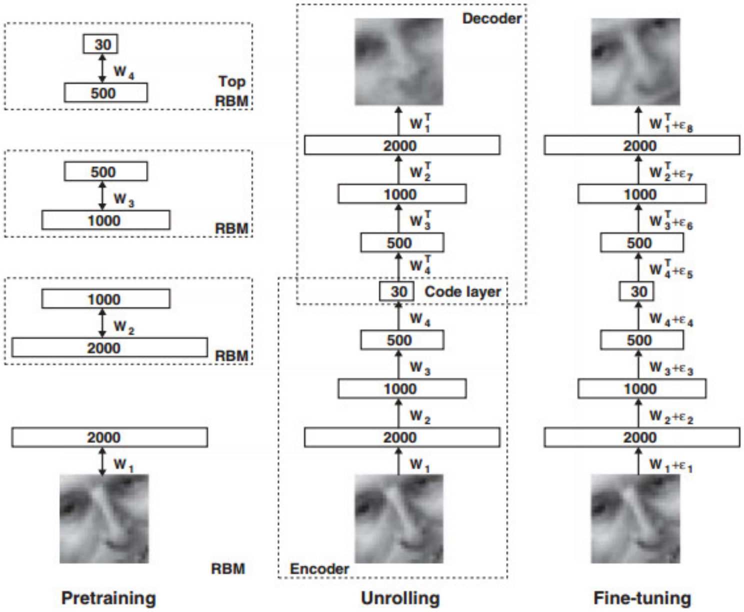
## Deep MLP – dimensionality reduction

Hinton, G. E., Osindero, S. and Teh, Y. (2006)
A fast learning algorithm for deep belief nets.
Neural Computation, 18, pp 1527-1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006)
Reducing the dimensionality of data with neural networks.
Science, Vol. 313. no. 5786, pp. 504 - 507, 28 July 2006.

This basically started all the deep learning craze ...

# Deep MLP – dimensionality reduction



## Images – pretraining

- **Data:** 165 600 black-white images, $25 \times 25$, mean intensity 0, variance 1.

  Images obtained from Olivetti Faces database of images $64 \times 64$ using standard transformations.
- 103 500 training set, 20 700 validation, 41 400 test
- **Network:** 2000-100-500-30, training using layered RBM.

Notes:

Training of the lowest layer (2000 neurons): Values of pixels distorted using Gaussian noise, low learning rate: 0.001, 200 iterations

Training all hidden layers: Values of neurons are binary.

Training of output layer: Values computed directly using the sigmoid activation functions + noise. That is, values of output neurons are from the interval $[0, 1]$.

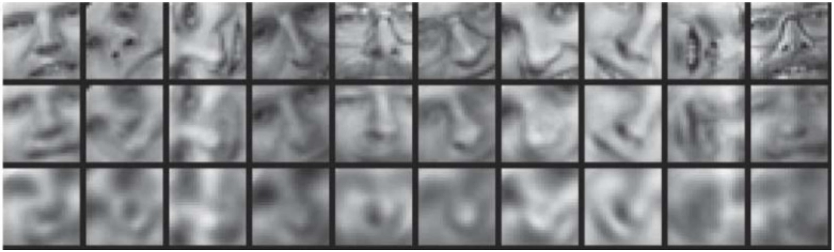
## Images – fine-tuning

- Stochastic activation substituted with deterministic.
  That is the value of hidden neurons is not chosen randomly but directly computed by application of sigmoid on the inner potential (this gives the mean activation).
- Backpropagation.
- Error function: cross-entropy

$$-\sum_i p_i \ln \hat{p}_i - \sum_i (1 - p_i) \ln(1 - \hat{p}_i)$$

  here $p_i$ is the intensity of $i$-th pixel of the input and $\hat{p}_i$ of the reconstruction.

## Results



1. Original
2. Reconstruction using deep networks (reduction to 30-dim)
3. Reconstruction using PCA (reduction to 30-dim)