Hopfield Networks and Boltzmann Machines

Hopfield Networks

A **Hopfield network** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions:

(i) $U_{\text{hidden}} = \emptyset$, $U_{\text{in}} = U_{\text{out}} = U$,

(ii) $C = U \times U - \{(u, u) \mid u \in U\}$.

- In a Hopfield network all neurons are input as well as output neurons.
- There are no hidden neurons.
- Each neuron receives input from all other neurons.
- A neuron is not connected to itself.

The connection weights are symmetric, that is,

$$\forall u, v \in U, u \neq v : \qquad w_{uv} = w_{vu}.$$

Hopfield Networks

The network input function of each neuron is the weighted sum of the outputs of all other neurons, that is,

$$\forall u \in U: \quad f_{\text{net}}^{(u)}(\vec{w}_u, \vec{\text{in}}_u) = \vec{w}_u^\top \vec{\text{in}}_u = \sum_{v \in U - \{u\}} w_{uv} \, \text{out}_v \, .$$

The activation function of each neuron is a threshold function, that is,

$$\forall u \in U: \quad f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \begin{cases} 1, & \text{if} \quad \text{net}_u \geq \theta, \\ -1, & \text{otherwise.} \end{cases}$$

The output function of each neuron is the identity, that is,

$$\forall u \in U: \quad f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u \, .$$

Alternative activation function

$$\forall u \in U: \quad f_{\text{act}}^{(u)}(\text{net}_u, \theta_u, \text{act}_u) = \begin{cases} 1, & \text{if} \quad \text{net}_u > \theta, \\ -1, & \text{if} \quad \text{net}_u < \theta, \\ \text{act}_u, & \text{if} \quad \text{net}_u = \theta. \end{cases}$$

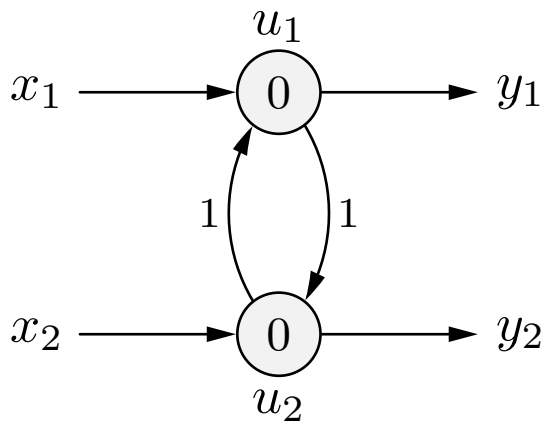
This activation function has advantages w.r.t. the physical interpretation of a Hopfield network.

General weight matrix of a Hopfield network

$$\mathbf{W} = \begin{pmatrix} 0 & w_{u_1u_2} & \dots & w_{u_1u_n} \\ w_{u_1u_2} & 0 & \dots & w_{u_2u_n} \\ \vdots & \vdots & & \vdots \\ w_{u_1u_n} & w_{u_1u_n} & \dots & 0 \end{pmatrix}$$

Hopfield Networks: Examples

Very simple Hopfield network



$$\mathbf{W} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The behavior of a Hopfield network can depend on the update order.

- Computations can oscillate if neurons are updated in parallel.
- Computations always converge if neurons are updated sequentially.

Hopfield Networks: Examples

Parallel update of neuron activations

	u_1	u_2
input phase	$\mathbf{-1}$	$\mathbf{1}$
work phase	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$

- The computations oscillate, no stable state is reached.
- Output depends on when the computations are terminated.

Hopfield Networks: Examples

Sequential update of neuron activations

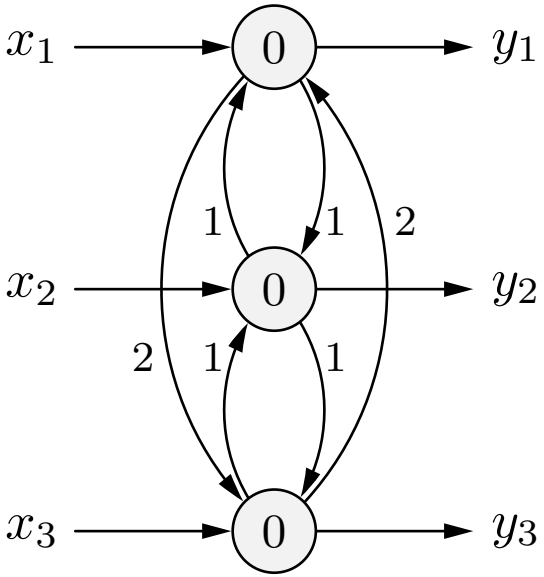
	u_1	u_2
input phase	$\mathbf{-1}$	$\mathbf{1}$
work phase	$\mathbf{1}$	1
	1	$\mathbf{1}$
	$\mathbf{1}$	1
	1	$\mathbf{1}$

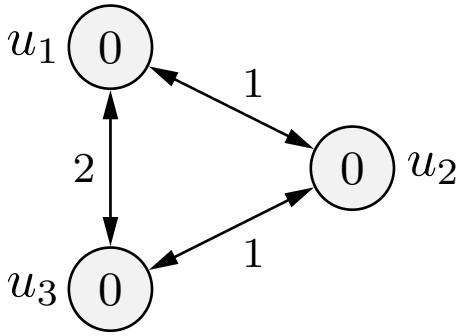
	u_1	u_2
input phase	$\mathbf{-1}$	$\mathbf{1}$
work phase	-1	$\mathbf{-1}$
	$\mathbf{-1}$	-1
	-1	$\mathbf{-1}$
	$\mathbf{-1}$	-1

- Update order u_1, u_2, u_1, u_2, ... (left) or u_2, u_1, u_2, u_1, ... (right)
- Regardless of the update order a stable state is reached.
- However, *which* state is reached depends on the update order.

Hopfield Networks: Examples

Simplified representation of a Hopfield network



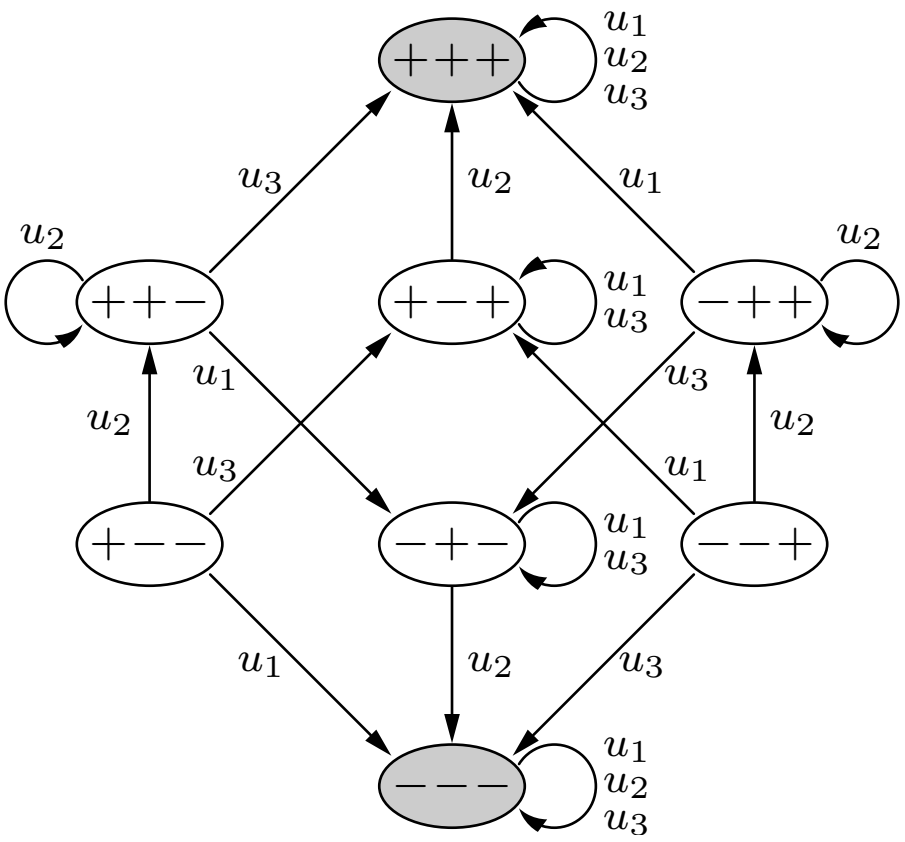


$$\mathbf{W} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

- Symmetric connections between neurons are combined.
- Inputs and outputs are not explicitely represented.

Hopfield Networks: State Graph

Graph of activation states and transitions:
(for the Hopfield network shown on the preceding slide)



"+"/"−" encode the neuron activations:
"+" means +1 and "−" means −1.

Labels on arrows indicate the neurons, whose updates (activation changes) lead to the corresponding state transitions.

States shown in gray:
stable states, cannot be left again
States shown in white:
unstable states, may be left again.

Such a state graph captures all imaginable update orders.

Hopfield Networks: Convergence

Convergence Theorem: If the activations of the neurons of a Hopfield network are updated sequentially (asynchronously), then a stable state is reached in a finite number of steps.

If the neurons are traversed cyclically in an arbitrary, but fixed order, at most $n \cdot 2^n$ steps (updates of individual neurons) are needed, where n is the number of neurons of the Hopfield network.

The proof is carried out with the help of an **energy function**.
The energy function of a Hopfield network with n neurons $u_1, \ldots, u_n$ is defined as

$$\begin{aligned} E &= -\frac{1}{2} \vec{\text{act}}^{\top} \mathbf{W} \vec{\text{act}} + \vec{\theta}^{\top} \vec{\text{act}} \\ &= -\frac{1}{2} \sum_{u,v \in U, u \neq v} w_{uv} \, \text{act}_u \, \text{act}_v + \sum_{u \in U} \theta_u \, \text{act}_u \, . \end{aligned}$$

Hopfield Networks: Convergence

Consider the energy change resulting from an update that changes an activation:

$$\begin{aligned}
\Delta E = E^{(\text{new})} - E^{(\text{old})} &= \Big(- \sum_{v \in U - \{u\}} w_{uv}\, \text{act}_u^{(\text{new})}\, \text{act}_v + \theta_u\, \text{act}_u^{(\text{new})} \Big) \\
&- \Big(- \sum_{v \in U - \{u\}} w_{uv}\, \text{act}_u^{(\text{old})}\, \text{act}_v + \theta_u\, \text{act}_u^{(\text{old})} \Big) \\
&= \Big(\text{act}_u^{(\text{old})} - \text{act}_u^{(\text{new})} \Big) \Big(\underbrace{\sum_{v \in U - \{u\}} w_{uv}\, \text{act}_v}_{= \text{net}_u} - \theta_u \Big).
\end{aligned}$$

- $\text{net}_u < \theta_u$: Second factor is less than 0.
  $\text{act}_u^{(\text{new})} = -1$ and $\text{act}_u^{(\text{old})} = 1$, therefore first factor greater than 0.
  Result: $\Delta E < 0$.
- $\text{net}_u \geq \theta_u$: Second factor greater than or equal to 0.
  $\text{act}_u^{(\text{new})} = 1$ and $\text{act}_u^{(\text{old})} = -1$, therefore first factor less than 0.
  Result: $\Delta E \leq 0$.

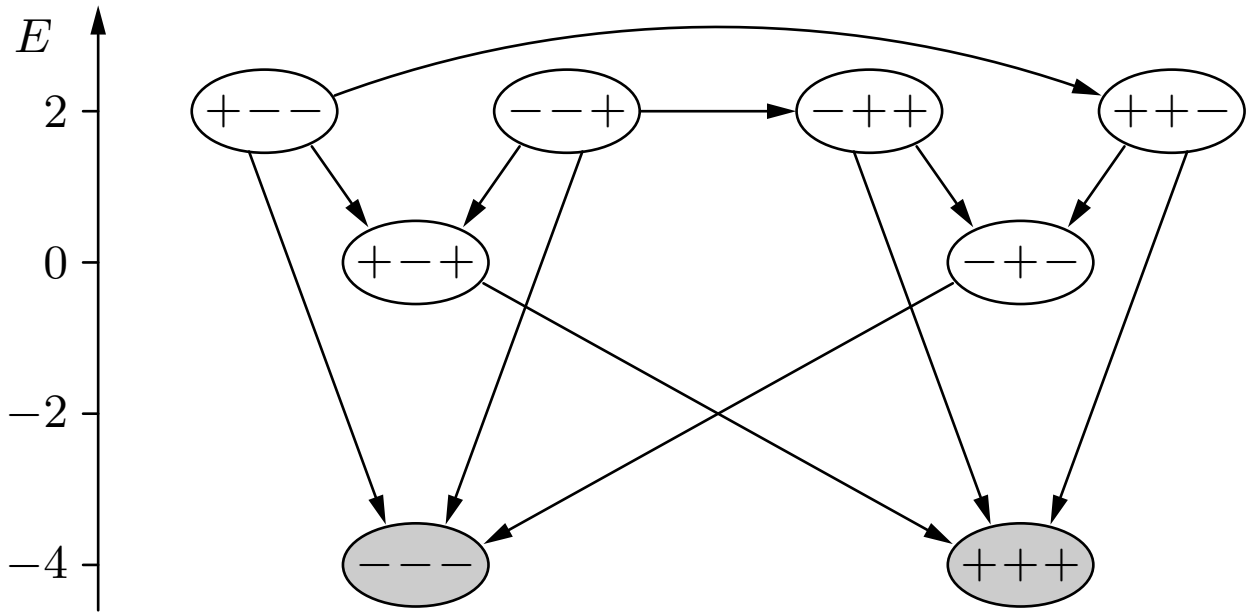
Hopfield Networks: Convergence

It takes at most $n \cdot 2^n$ update steps to reach convergence.

- Provided that the neurons are updated in an arbitrary, but fixed order, since this guarantees that the neurons are traversed cyclically, and therefore each neuron is updated every n steps.
- If in a traversal of all n neurons no activation changes: **a stable state has been reached**.
- If in a traversal of all n neurons at least one activation changes: **the previous state cannot be reached again**, because
  - either the new state has a smaller energy than the old (no way back: updates cannot increase the network energy)
  - or the number of +1 activations has increased (no way back: equal energy is possible only for $\text{net}_u \geq \theta_u$).
- The number of possible states of the Hopfield network is 2^n, at least one of which must be rendered unreachable in each traversal of the n neurons.

Hopfield Networks: Examples

Arrange states in state graph according to their energy

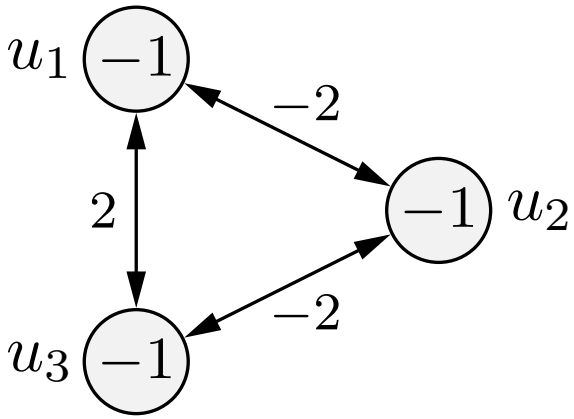


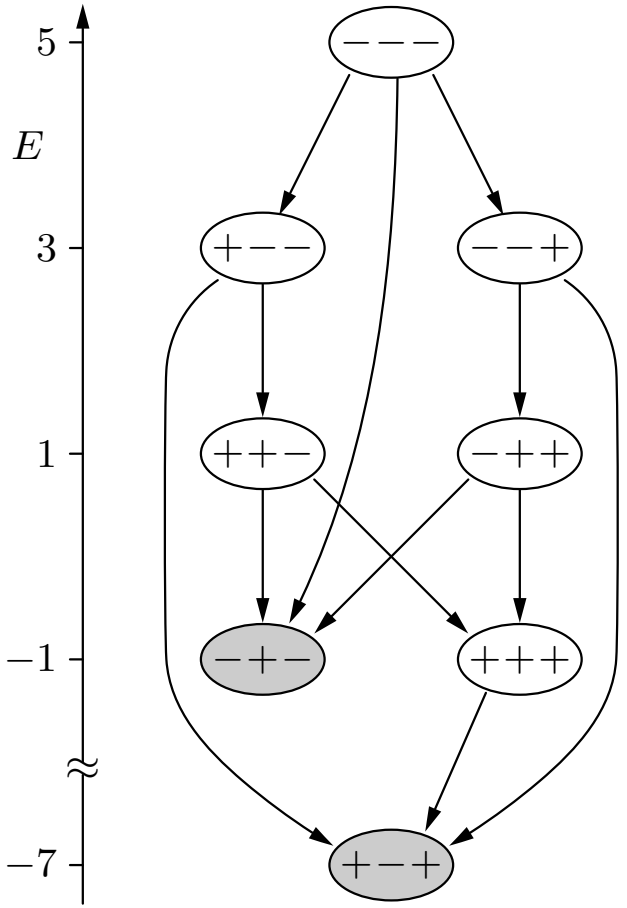
Energy function for example Hopfield network:

$$E = -\operatorname{act}_{u_1} \operatorname{act}_{u_2} - 2 \operatorname{act}_{u_1} \operatorname{act}_{u_3} - \operatorname{act}_{u_2} \operatorname{act}_{u_3} .$$

Hopfield Networks: Examples

The state graph need not be symmetric





Hopfield Networks: Physical Interpretation

Physical interpretation: Magnetism

A Hopfield network can be seen as a (microscopic) model of magnetism (so-called Ising model, [Ising 1925]).

physical	neural
atom	neuron
magnetic moment (spin)	activation state
strength of outer magnetic field	threshold value
magnetic coupling of the atoms	connection weights
Hamilton operator of the magnetic field	energy function

Idea: Use stable states to store patterns

First: Store only one pattern $\vec{x} = (\text{act}_{u_1}^{(l)}, \ldots, \text{act}_{u_n}^{(l)})^\top \in \{-1, 1\}^n$, $n \geq 2$, that is, find weights, so that pattern is a stable state.

Necessary and sufficient condition:

$$S(\mathbf{W}\vec{x} - \vec{\theta}\,) = \vec{x},$$

where

$$\begin{array}{rcl} S : \mathbb{R}^n & \to & \{-1, 1\}^n, \\ \vec{x} & \mapsto & \vec{y} \end{array}$$

with

$$\forall i \in \{1, \ldots, n\} : \quad y_i = \begin{cases} 1, & \text{if } x_i \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

Hopfield Networks: Associative Memory

If $\vec{\theta} = \vec{0}$ an appropriate matrix $\mathbf{W}$ can easily be found. It suffices

$$\mathbf{W}\vec{x} = c\vec{x} \qquad \text{with } c \in \mathbb{R}^+.$$

Algebraically: Find a matrix $\mathbf{W}$ that has a positive eigenvalue w.r.t. $\vec{x}$.

Choose

$$\mathbf{W} = \vec{x}\vec{x}^\top - \mathbf{E}$$

where $\vec{x}\vec{x}^\top$ is the so-called **outer product** of $\vec{x}$ with itself.

With this matrix we have

$$\begin{aligned} \mathbf{W}\vec{x} &= (\vec{x}\vec{x}^\top)\vec{x} - \underbrace{\mathbf{E}\vec{x}}_{=\vec{x}} \overset{(*)}{=} \vec{x}\,\underbrace{(\vec{x}^\top\vec{x})}_{=|\vec{x}\,|^2=n} - \vec{x} \\ &= n\vec{x} - \vec{x} = (n-1)\vec{x}. \end{aligned}$$

$(*)$ holds, because vector/matrix multiplication is associative.

Hebbian learning rule [Hebb 1949]

Written in individual weights the computation of the weight matrix reads:

$$w_{uv} = \begin{cases} 0, & \text{if } u = v, \\ 1, & \text{if } u \neq v, \text{act}_u^{(p)} = \text{act}_u^{(v)}, \\ -1, & \text{otherwise.} \end{cases}$$

- Originally derived from a biological analogy.
- Strengthen connection between neurons that are active at the same time.

Note that this learning rule also stores the complement of the pattern:

$$\text{With} \qquad \mathbf{W}\vec{x} = (n-1)\vec{x} \qquad \text{it is also} \qquad \mathbf{W}(-\vec{x}) = (n-1)(-\vec{x}).$$

Storing several patterns

Choose

$$\begin{aligned}\mathbf{W}\vec{x}_j &= \sum_{i=1}^{m}\mathbf{W}_i\vec{x}_j = \left(\sum_{i=1}^{m}(\vec{x}_i\vec{x}_i^\top)\vec{x}_j\right) - m\underbrace{\mathbf{E}\vec{x}_j}_{=\vec{x}_j} \\ &= \left(\sum_{i=1}^{m}\vec{x}_i(\vec{x}_i^\top\vec{x}_j)\right) - m\vec{x}_j\end{aligned}$$

If the patterns are orthogonal, we have

$$\vec{x}_i^\top\vec{x}_j = \begin{cases} 0, & \text{if} \quad i \neq j, \\ n, & \text{if} \quad i = j, \end{cases}$$

and therefore

$$\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j.$$

Storing several patterns

$$\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j.$$

Result: As long as $m < n$, $\vec{x}$ is a stable state of the Hopfield network.

Note that the complements of the patterns are also stored.

With $\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j$ it is also $\mathbf{W}(-\vec{x}_j) = (n-m)(-\vec{x}_j)$.

But: Capacity is very small compared to the number of possible states (2^n),
since at most $m = n-1$ ortogonal patterns can be stored (so that $n-m > 0$).

Furthermore, the requirement that the patterns must be orthogonal
is a strong limitation of the usefulness of this result.

Hopfield Networks: Associative Memory

Non-orthogonal patterns:

$$\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j + \underbrace{\sum_{\substack{i=1 \\ i \neq j}}^{m} \vec{x}_i(\vec{x}_i^{\top}\vec{x}_j)}_{\text{“disturbance term”}} .$$

- The “disturbance term” need not make it impossible to store the patterns.
- The states corresponding to the patterns $\vec{x}_j$ may still be stable, if the “disturbance term” is sufficiently small.
- For this term to be sufficiently small, the patterns must be “almost” orthogonal.
- The larger the number of patterns to be stored (that is, the smaller $n - m$), the smaller the “disturbance term” must be.
- The theoretically possible maximal capacity of a Hopfield network (that is, $m = n - 1$) is hardly ever reached in practice.

Associative Memory: Example

Example: Store patterns $\vec{x}_1 = (+1, +1, -1, -1)^\top$ and $\vec{x}_2 = (-1, +1, -1, +1)^\top$.

$$\mathbf{W} = \mathbf{W}_1 + \mathbf{W}_2 = \vec{x}_1 \vec{x}_1^\top + \vec{x}_2 \vec{x}_2^\top - 2\mathbf{E}$$

where

$$\mathbf{W}_1 = \begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{pmatrix}, \qquad \mathbf{W}_2 = \begin{pmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{pmatrix}.$$

The full weight matrix is:

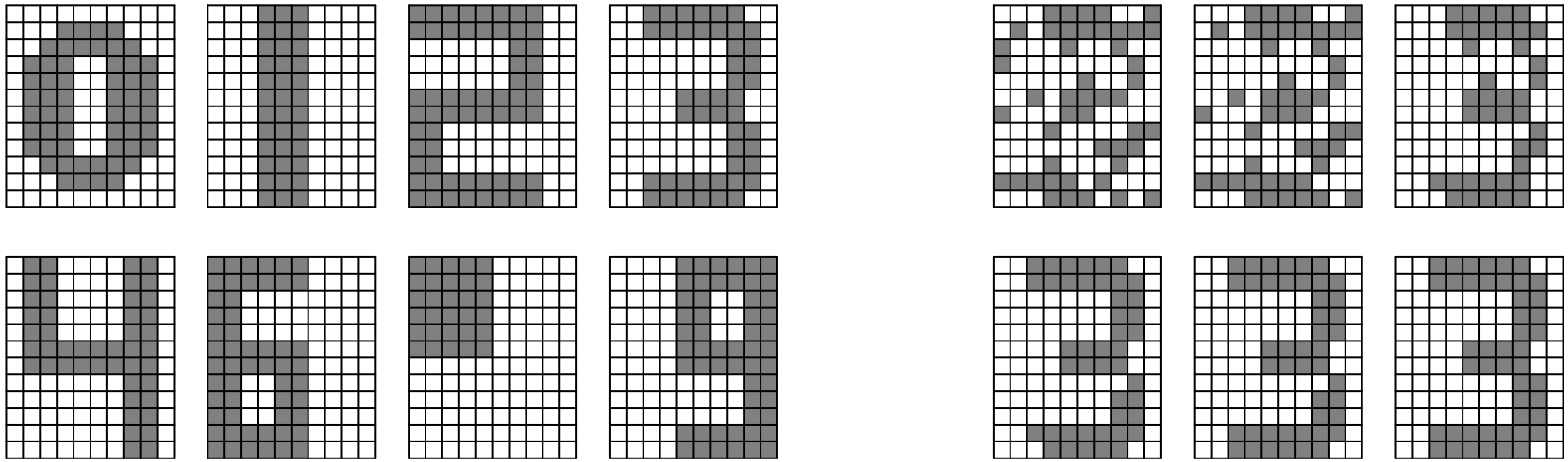
$$\mathbf{W} = \begin{pmatrix} 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 \\ 0 & -2 & 0 & 0 \\ -2 & 0 & 0 & 0 \end{pmatrix}.$$

Therefore it is

$$\mathbf{W}\vec{x}_1 = (+2, +2, -2, -2)^\top \qquad \text{and} \qquad \mathbf{W}\vec{x}_1 = (-2, +2, -2, +2)^\top.$$

Associative Memory: Examples

Example: Storing bit maps of numbers



- Left: Bit maps stored in a Hopfield network.
- Right: Reconstruction of a pattern from a random input.

Training a Hopfield network with the Delta rule

Necessary condition for pattern $\vec{x}$ being a stable state:

$$\begin{array}{llllll}
s(0 & + w_{u_1u_2}\,\mathrm{act}_{u_2}^{(p)} & + \ldots + w_{u_1u_n}\,\mathrm{act}_{u_n}^{(p)} & - \theta_{u_1}) = \mathrm{act}_{u_1}^{(p)}, \\
s(w_{u_2u_1}\,\mathrm{act}_{u_1}^{(p)} & + 0 & + \ldots + w_{u_2u_n}\,\mathrm{act}_{u_n}^{(p)} & - \theta_{u_2}) = \mathrm{act}_{u_2}^{(p)}, \\
\vdots & \vdots & \vdots & \vdots \quad \vdots \\
s(w_{u_nu_1}\,\mathrm{act}_{u_1}^{(p)} & + w_{u_nu_2}\,\mathrm{act}_{u_2}^{(p)} & + \ldots + 0 & - \theta_{u_n}) = \mathrm{act}_{u_n}^{(p)}.
\end{array}$$

with the standard threshold function

$$s(x) = \begin{cases} 1, & \text{if} \quad x \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

Training a Hopfield network with the Delta rule

Turn weight matrix into a weight vector:

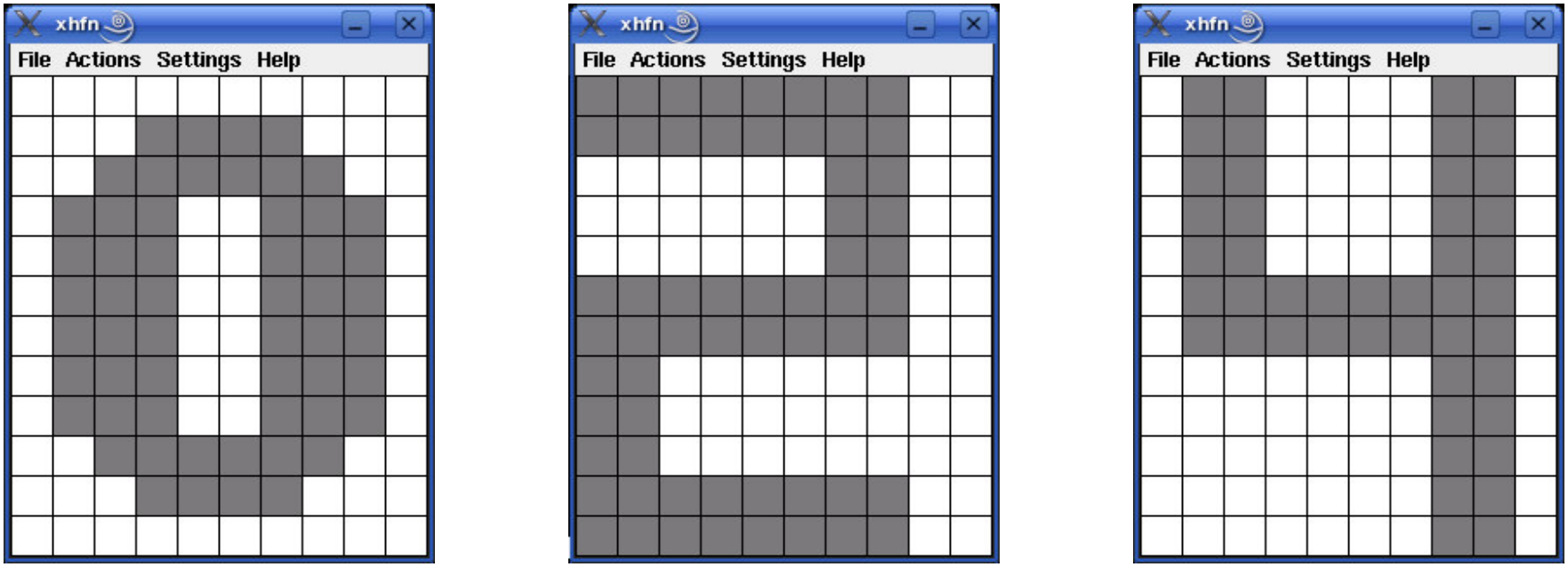
$$\begin{aligned}\vec{w} = (\quad & w_{u_1u_2}, & w_{u_1u_3}, & \ldots, & w_{u_1u_n}, \\ & & w_{u_2u_3}, & \ldots, & w_{u_2u_n}, \\ & & & \ddots & \vdots \\ & & & & w_{u_{n-1}u_n}, \\ & -\theta_{u_1}, & -\theta_{u_2}, & \ldots, & -\theta_{u_n} \qquad).\end{aligned}$$

Construct input vectors for a threshold logic unit

$$\vec{z}_2 = (\mathrm{act}_{u_1}^{(p)}, \underbrace{0, \ldots, 0,}_{n-2 \text{ zeros}} \mathrm{act}_{u_3}^{(p)}, \ldots, \mathrm{act}_{u_n}^{(p)}, \ldots 0, 1, \underbrace{0, \ldots, 0}_{n-2 \text{ zeros}}).$$

Apply Delta rule training / Widrow–Hoff procedure until convergence.

Demonstration Software: xhfn/whfn



Demonstration of Hopfield networks as associative memory:

- Visualization of the association/recognition process
- Two-dimensional networks of arbitrary size
- `http://www.borgelt.net/hfnd.html`

Hopfield Networks: Solving Optimization Problems

Use energy minimization to solve optimization problems

General procedure:

- Transform function to optimize into a function to minimize.
- Transform function into the form of an energy function of a Hopfield network.
- Read the weights and threshold values from the energy function.
- Construct the corresponding Hopfield network.
- Initialize Hopfield network randomly and update until convergence.
- Read solution from the stable state reached.
- Repeat several times and use best solution found.

Hopfield Networks: Activation Transformation

A Hopfield network may be defined either with activations -1 and 1 or with activations 0 and 1. The networks can be transformed into each other.

From $\text{act}_u \in \{-1, 1\}$ to $\text{act}_u \in \{0, 1\}$:

$$\begin{aligned} w^0_{uv} &= 2w^-_{uv} && \text{and} \\ \theta^0_u &= \theta^-_u + \sum_{v \in U - \{u\}} w^-_{uv} \end{aligned}$$

From $\text{act}_u \in \{0, 1\}$ to $\text{act}_u \in \{-1, 1\}$:

$$\begin{aligned} w^-_{uv} &= \frac{1}{2} w^0_{uv} && \text{and} \\ \theta^-_u &= \theta^0_u - \frac{1}{2} \sum_{v \in U - \{u\}} w^0_{uv}. \end{aligned}$$

Hopfield Networks: Solving Optimization Problems

Combination lemma: Let two Hopfield networks on the same set U of neurons with weights $w_{uv}^{(i)}$, threshold values $\theta_u^{(i)}$ and energy functions

$$E_i = -\frac{1}{2} \sum_{u \in U} \sum_{v \in U - \{u\}} w_{uv}^{(i)} \operatorname{act}_u \operatorname{act}_v + \sum_{u \in U} \theta_u^{(i)} \operatorname{act}_u,$$

$i = 1, 2$, be given. Furthermore let $a, b \in \mathbb{R}$. Then $E = aE_1 + bE_2$ is the energy function of the Hopfield network on the neurons in U that has the weights $w_{uv} = aw_{uv}^{(1)} + bw_{uv}^{(2)}$ and the threshold values $\theta_u = a\theta_u^{(1)} + b\theta_u^{(2)}$.

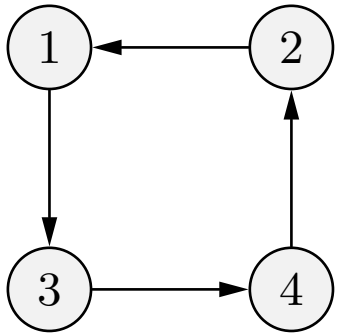
Proof: Just do the computations.

Idea: Additional conditions can be formalized separately and incorporated later.
(One energy function per condition, then apply combination lemma.)

Hopfield Networks: Solving Optimization Problems

Example: Traveling salesman problem

Idea: Represent tour by a matrix.



$$
\begin{array}{c}
\text{city} \\
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\left(\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0
\end{array} \right)
\end{array}
\begin{array}{l}
\\
1. \\
2. \\
3. \\
4.
\end{array}
\quad \text{step}
$$

An element m_{ij} of the matrix is 1 if the i-th city is visited in the j-th step and 0 otherwise.

Each matrix element will be represented by a neuron.

Minimization of the tour length

$$E_1 = \sum_{j_1=1}^{n} \sum_{j_2=1}^{n} \sum_{i=1}^{n} d_{j_1 j_2} \cdot m_{i j_1} \cdot m_{(i \bmod n)+1, j_2}.$$

Double summation over steps (index i) needed:

$$E_1 = \sum_{(i_1,j_1) \in \{1,\ldots,n\}^2} \sum_{(i_2,j_2) \in \{1,\ldots,n\}^2} d_{j_1 j_2} \cdot \delta_{(i_1 \bmod n)+1, i_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2},$$

where

$$\delta_{ab} = \begin{cases} 1, & \text{if} \quad a = b, \\ 0, & \text{otherwise.} \end{cases}$$

Symmetric version of the energy function:

$$E_1 = -\frac{1}{2} \sum_{\substack{(i_1,j_1) \in \{1,\ldots,n\}^2 \\ (i_2,j_2) \in \{1,\ldots,n\}^2}} -d_{j_1 j_2} \cdot (\delta_{(i_1 \bmod n)+1, i_2} + \delta_{i_1, (i_2 \bmod n)+1}) \cdot m_{i_1 j_1} \cdot m_{i_2 j_2}.$$

Hopfield Networks: Solving Optimization Problems

Additional conditions that have to be satisfied:

- Each city is visited on exactly one step of the tour:

$$\forall j \in \{1, \ldots, n\} : \qquad \sum_{i=1}^{n} m_{ij} = 1,$$

  that is, each column of the matrix contains exactly one 1.

- On each step of the tour exactly one city is visited:

$$\forall i \in \{1, \ldots, n\} : \qquad \sum_{j=1}^{n} m_{ij} = 1,$$

  that is, each row of the matrix contains exactly one 1.

These conditions are incorporated by finding additional functions to optimize.

Hopfield Networks: Solving Optimization Problems

Formalization of first condition as a minimization problem:

$$\begin{aligned} E_2^* &= \sum_{j=1}^{n} \left(\left(\sum_{i=1}^{n} m_{ij} \right)^2 - 2 \sum_{i=1}^{n} m_{ij} + 1 \right) \\ &= \sum_{j=1}^{n} \left(\left(\sum_{i_1=1}^{n} m_{i_1 j} \right) \left(\sum_{i_2=1}^{n} m_{i_2 j} \right) - 2 \sum_{i=1}^{n} m_{ij} + 1 \right) \\ &= \sum_{j=1}^{n} \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} m_{i_1 j} m_{i_2 j} - 2 \sum_{j=1}^{n} \sum_{i=1}^{n} m_{ij} + n. \end{aligned}$$

Double summation over cities (index i) needed:

$$E_2 = \sum_{(i_1,j_1) \in \{1,\ldots,n\}^2} \sum_{(i_2,j_2) \in \{1,\ldots,n\}^2} \delta_{j_1 j_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} - 2 \sum_{(i,j) \in \{1,\ldots,n\}^2} m_{ij}.$$

Hopfield Networks: Solving Optimization Problems

Resulting energy function:

$$E_2 = -\frac{1}{2} \sum_{\substack{(i_1,j_1)\in\{1,\ldots,n\}^2 \\ (i_2,j_2)\in\{1,\ldots,n\}^2}} -2\delta_{j_1 j_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} + \sum_{(i,j)\in\{1,\ldots,n\}^2} -2m_{ij}$$

Second additional condition is handled in a completely analogous way:

$$E_3 = -\frac{1}{2} \sum_{\substack{(i_1,j_1)\in\{1,\ldots,n\}^2 \\ (i_2,j_2)\in\{1,\ldots,n\}^2}} -2\delta_{i_1 i_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} + \sum_{(i,j)\in\{1,\ldots,n\}^2} -2m_{ij}.$$

Combining the energy functions:

$$E = aE_1 + bE_2 + cE_3 \qquad \text{where} \qquad \frac{b}{a} = \frac{c}{a} > 2 \max_{(j_1,j_2)\in\{1,\ldots,n\}^2} d_{j_1 j_2}.$$

Hopfield Networks: Solving Optimization Problems

From the resulting energy function we can read the weights

$$w_{(i_1,j_1)(i_2,j_2)} = \underbrace{-ad_{j_1j_2} \cdot (\delta_{(i_1 \bmod n)+1,i_2} + \delta_{i_1,(i_2 \bmod n)+1})}_{\text{from } E_1} \underbrace{-2b\delta_{j_1j_2}}_{\text{from } E_2} \underbrace{-2c\delta_{i_1i_2}}_{\text{from } E_3}$$

and the threshold values:

$$\theta_{(i,j)} = \underbrace{0a}_{\text{from } E_1} \quad \underbrace{-2b}_{\text{from } E_2} \quad \underbrace{-2c}_{\text{from } E_3} = -2(b+c).$$

Problem: Random initialization and update until convergence
not always leads to a matrix that represents a tour, let alone an optimal one.

Hopfield Networks: Reasons for Failure

Hopfield network only rarely finds a tour, let alone an optimal one.

- One of the main problems is that the Hopfield network is unable to switch from a found tour to another with a lower total length.
- The reason is that transforming a matrix that represents a tour into another matrix that represents a different tour requires that at least four neurons (matrix elements) change their activations.
- However, each of these changes, if carried out individually, violates at least one of the constraints and thus increases the energy.
- Only all four changes together can result in a smaller energy, but cannot be executed together due to the asynchronous update.
- Therefore the normal activation updates can never change an already found tour into another, even if this requires only a marginal change of the tour.

Hopfield Networks: Local Optima

- Results can be somewhat improved if instead of
  discrete Hopfield networks (activations in $\{-1, 1\}$ (or $\{0, 1\}$)) one uses
  continuous Hopfield networks (activations in $[-1, 1]$ (or $[0, 1]$)).
  However, the fundamental problem is not solved in this way.

- More generally, the reason for the difficulties that are encountered
  if an optimization problem is to be solved with a Hopfield network is:

  The update procedure may get stuck in a local optimum.

- The problem of local optima occurs also with many other optimization methods,
  for example, gradient descent, hill climbing, alternating optimization etc.

- Ideas to overcome this difficulty for other optimization methods
  may be transferred to Hopfield networks.

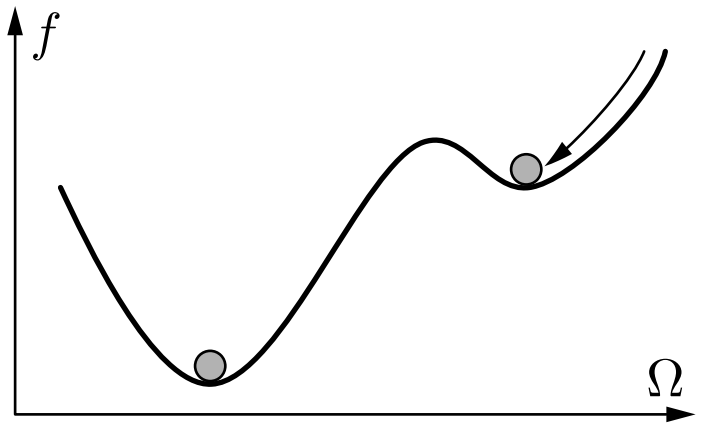
- One such method, which is very popular, is **simulated annealing**.

Simulated Annealing

May be seen as an extension of random or gradient descent that tries to avoid getting stuck.

Idea: transitions from higher to lower (local) minima should be more probable than *vice versa*.

[Metropolis *et al.* 1953; Kirkpatrik *et al.* 1983]



Principle of Simulated Annealing:

- Random variants of the current solution (candidate) are created.
- Better solution (candidates) are always accepted.
- Worse solution (candidates) are accepted with a probability that depends on
  - the quality difference between the new and the old solution (candidate) and
  - a temperature parameter that is decreased with time.

Simulated Annealing

- **Motivation:**
  - Physical minimization of the energy (more precisely: atom lattice energy) if a heated piece of metal is cooled slowly.
  - This process is called **annealing**.
  - It serves the purpose to make the metal easier to work or to machine by relieving tensions and correcting lattice malformations.
- **Alternative Motivation:**
  - A ball rolls around on an (irregularly) curved surface; minimization of the potential energy of the ball.
  - In the beginning the ball is endowed with a certain kinetic energy, which enables it to roll up some slopes of the surface.
  - In the course of time, friction reduces the kinetic energy of the ball, so that it finally comes to a rest in a valley of the surface.
- Attention: **There is no guarantee that the global optimum is found!**

Simulated Annealing: Procedure

1. Choose a (random) starting point $s_0 \in \Omega$ (Ω is the search space).
2. Choose a point $s' \in \Omega$ "in the vicinity" of s_i
   (for example, by a small random variation of s_i).
3. Set
$$s_{i+1} = \begin{cases} s' & \text{if } f(s') \geq f(s_i), \\ s' & \text{with probability } p = e^{-\frac{\Delta f}{kT}} \text{ and} \\ s_i & \text{with probability } 1 - p \text{ otherwise.} \end{cases}$$
$\Delta f = f(s_i) - f(s')$	quality difference of the solution (candidates)
$k = \Delta f_{\max}$	(estimation of the) range of quality values
T	temperature parameter (is (slowly) decreased over time)
4. Repeat steps 2 and 3 until some termination criterion is fulfilled.

- For (very) small T the method approaches a pure random descent.

Hopfield Networks: Simulated Annealing

Applying simulated annealing to Hopfield networks is very simple:

- All neuron activations are initialized randomly.
- The neurons of the Hopfield network are traversed repeatedly (for example, in some random order).
- For each neuron, it is determined whether an activation change leads to a reduction of the network energy or not.
- An activation change that reduces the network energy is always accepted (in the normal update process, only such changes occur).
- However, if an activation change increases the network energy, it is accepted with a certain probability (see preceding slide).
- Note that in this case we have simply

$$\Delta f = \Delta E = |\operatorname{net}_u - \theta_u|$$

Hopfield Networks: Summary

- Hopfield networks are **restricted recurrent neural networks** (full pairwise connections, symmetric connection weights).
- Synchronous update of the neurons may lead to oscillations, but **asynchronous update is guaranteed to reach a stable state** (asynchronous updates either reduce the energy of the network or increase the number of +1 activations).
- Hopfield networks can be used as **associative memory**, that is, as memory that is addressed by its contents, by using the stable states to store desired patterns.
- Hopfield networks can be used to **solve optimization problems**, if the function to optimize can be reformulated as an energy function (stable states are (local) minima of the energy function).
- Approaches like **simulated annealing** may be needed to prevent that the update gets stuck in a local optimum.

Boltzmann Machines

- Boltzmann machines are closely related to Hopfield networks.
- They differ from Hopfield networks mainly in how the neurons are updated.
- They also rely on the fact that one can define an **energy function** that assigns a numeric value (an *energy*) to each state of the network.
- With the help of this energy function a probability distribution over the states of the network is defined based on the **Boltzmann distribution** (also known as **Gibbs distribution**) of statistical mechanics, namely

$$P(\vec{s}) = \frac{1}{c}\, e^{-\frac{E(\vec{s})}{kT}}.$$

$\vec{s}$	describes the (discrete) state of the system,
c	is a normalization constant,
E	is the function that yields the energy of a state $\vec{s}$,
T	is the thermodynamic temperature of the system,
k	is Boltzmann's constant ($k \approx 1.38 \cdot 10^{-23}\, J/K$).

Boltzmann Machines

- For Boltzmann machines the product kT is often replaced by merely T, combining the temperature and Boltzmann's constant into a single parameter.
- The state $\vec{s}$ consists of the vector $\vec{\text{act}}$ of the neuron activations.
- The energy function of a Boltzmann machine is

$$E(\vec{\text{act}}) = -\frac{1}{2}\,\vec{\text{act}}^{\top}\mathbf{W}\vec{\text{act}} + \vec{\theta}^{\top}\vec{\text{act}},$$

  where $\mathbf{W}$: matrix of connection weights; $\vec{\theta}$: vector of threshold values.
- Consider the energy change resulting from the change of a single neuron u:

$$\Delta E_u = E_{\text{act}_u=1} - E_{\text{act}_u=0} = \sum_{v \in U-\{u\}} w_{uv}\,\text{act}_v - \theta_u$$

- Writing the energies in terms of the Boltzmann distribution yields

$$\Delta E_u = -kT\ln(P(\text{act}_u = 1) - (-kT\ln(P(\text{act}_u = 0)).$$

Boltzmann Machines

- Rewrite as
$$\begin{aligned}\frac{\Delta E_u}{kT} &= \ln(P(\text{act}_u = 1)) - \ln(P(\text{act}_u = 0)) \\ &= \ln(P(\text{act}_u = 1)) - \ln(1 - P(\text{act}_u = 1))\end{aligned}$$
(since obviously $P(\text{act}_u = 0) + P(\text{act}_u = 1) = 1$).

- Solving this equation for $P(\text{act}_u = 1)$ finally yields
$$P(\text{act}_u = 1) = \frac{1}{1 + e^{-\frac{\Delta E_u}{kT}}}.$$

- That is: the probability of a neuron being active is a logistic function of the (scaled) energy difference between its active and inactive state.

- Since the energy difference is closely related to the network input, namely as
$$\Delta E_u = \sum_{v \in U - \{u\}} w_{uv}\, \text{act}_v - \theta_u = \text{net}_u - \theta_u,$$
this formula suggests a stochastic update procedure for the network.

Boltzmann Machines: Update Procedure

- A neuron u is chosen (randomly), its network input, from it the energy difference ΔE_u and finally the probability of the neuron having activation 1 is computed. The neuron is set to activation 1 with this probability and to 0 otherwise.
- This update is repeated many times for randomly chosen neurons.
- Simulated annealing is carried out by slowly lowering the temperature T.
- This update process is a **Markov Chain Monte Carlo** (**MCMC**) procedure.
- After sufficiently many steps, the probability that the network is in a specific activation state depends only on the energy of that state.
  It is independent of the initial activation state the process was started with.
- This final situation is also referred to as **thermal equlibrium**.
- Therefore: Boltzmann machines are representations of and sampling mechanisms for the Boltzmann distributions defined by their weights and threshold values.

Boltzmann Machines: Training

- **Idea of Training:**
  Develop a training procedure with which the probability distribution represented by a Boltzmann machine via its energy function can be adapted to a given sample of data points, in order to obtain a **probabilistic model of the data**.
- This objective can only be achieved sufficiently well
  if the data points are actually a sample from a Boltzmann distribution.
  (Otherwise the model cannot, in principle, be made to fit the sample data well.)
- In order to mitigate this restriction to Boltzmann distributions,
  a deviation from the structure of Hopfield networks is introduced.
- The neurons are divided into
  - **visible neurons**, which receive the data points as input, and
  - **hidden neurons**, which are not fixed by the data points.

  (Reminder: Hopfield networks have only visible neurons.)

Boltzmann Machines: Training

- **Objective of Training:**
  Adapt the connection weights and threshold values in such a way that the true distribution underlying a given data sample is approximated well by the probability distribution represented by the Boltzmann machine on its visible neurons.

- **Natural Approach to Training:**
  - Choose a measure for the difference between two probability distributions.
  - Carry out a gradient descent in order to minimize this difference measure.

- Well-known measure: **Kullback–Leibler information divergence**.
  For two probability distributions p_1 and p_2 defined over the same sample space Ω:

$$KL(p_1, p_2) = \sum_{\omega \in \Omega} p_1(\omega) \ln \frac{p_1(\omega)}{p_2(\omega)}.$$

  Applied to Boltzmann machines: p_1 refers to the data sample,
  p_2 to the visible neurons of the Boltzmann machine.

Boltzmann Machines: Training

- In each training step the Boltzmann machine is ran twice (two "phases").
- **"Positive Phase"**: Visible neurons are fixed to a randomly chosen data point; only the hidden neurons are updated until thermal equilibrium is reached.
- **"Negative Phase"**: All units are updated until thermal equilibrium is reached.
- In the two phases statistics about individual neurons and pairs of neurons (both visible and hidden) being activated (simultaneously) are collected.
- Then update is performed according to the following two equations:

$$\Delta w_{uv} = \frac{1}{\eta}(p_{uv}^{+} - p_{uv}^{-}) \qquad \text{and} \qquad \Delta\theta_u = -\frac{1}{\eta}(p_u^{+} - p_u^{-}).$$

p_u	probability that neuron u is active,
p_{uv}	probability that neurons u and v are both active simultaneously,
$+\ -$	as upper indices indicate the phase referred to.

(All probabilities are estimated from observed relative frequencies.)

Boltzmann Machines: Training

- Intuitive explanation of the update rule:
  - If a neuron is more often active when a data sample is presented
    than when the network is allowed to run freely,
    the probability of the neuron being active is too low,
    so the threshold should be reduced.
  - If neurons are more often active together when a data sample is presented
    than when the network is allowed to run freely,
    the connection weight between them should be increased,
    so that they become more likely to be active together.
- This training method is very similar to the **Hebbian learning rule**.

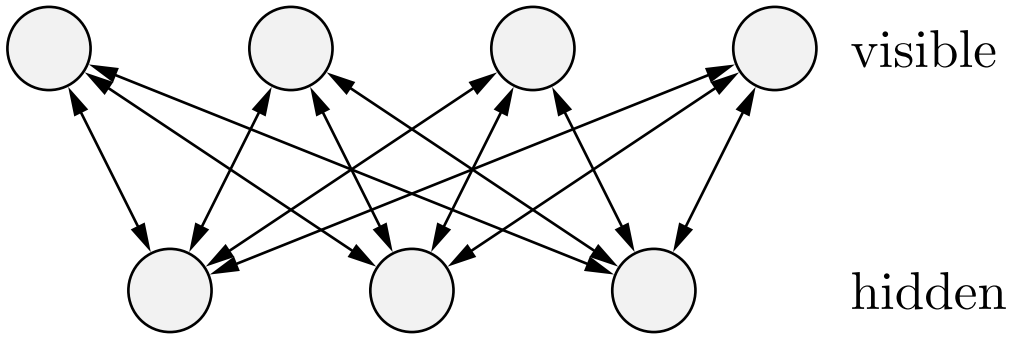
  Derived from a biological analogy it says:

  connections between two neurons that are synchronously active are strengthened

  ("cells that fire together, wire together").

Boltzmann Machines: Training

- Unfortunately, this procedure is impractical unless the networks are very small.
- The main reason is the fact that the larger the network, the more update steps need to be carried out in order to obtain sufficiently reliable statistics for the neuron activation (pairs) needed in the update formulas.
- Efficient training is possible for the **restricted Boltzmann machine**.
- Restriction consists in using a bipartite graph instead of a fully connected graph:
  - Vertices are split into two groups, the visible and the hidden neurons;
  - Connections only exist between neurons from different groups.



Restricted Boltzmann Machines

A **restricted Boltzmann Machine (RBM)** or **Harmonium** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions:

(i) $U = U_{\text{in}} \cup U_{\text{hidden}}, \quad U_{\text{in}} \cap U_{\text{hidden}} = \emptyset, \quad U_{\text{out}} = U_{\text{in}},$

(ii) $C = U_{\text{in}} \times U_{\text{hidden}}.$

- In a restricted Boltzmann machine, all input neurons are also output neurons and *vice versa* (all output neurons are also input neurons).
- There are hidden neurons, which are different from input and output neurons.
- Each input/output neuron receives input from all hidden neurons; each hidden neuron receives input from all input/output neurons.

The connection weights between input and hidden neurons are symmetric, that is,

$$\forall u \in U_{\text{in}}, v \in U_{\text{hidden}} : \qquad w_{uv} = w_{vu}.$$

Restricted Boltzmann Machines: Training

Due to the lack of connections within the visible units and within the hidden units, training can proceed by repeating the following three steps:

- Visible units are fixed to a randomly chosen data sample $\vec{x}$;
  hidden units are updated once and in parallel (result: $\vec{y}$).

  $\vec{x}\vec{y}^\top$ is called the **positive gradient** for the weight matrix.

- Hidden neurons are fixed to the computed vector $\vec{y}$;
  visible units are updated once and in parallel (result: $\vec{x}^*$).

  Visible neurons are fixed to "reconstruction" $\vec{x}^*$;
  hidden neurons are update once more (result: $\vec{y}^*$).

  $\vec{x}^*\vec{y}^{*\top}$ is called the **negative gradient** for the weight matrix.

- Connection weights are updated with difference of positive and negative gradient:

$$\Delta w_{uv} = \eta(\vec{x}_u\vec{y}_v^\top - \vec{x}_u^*\vec{y}_v^{*\top}) \qquad \text{where } \eta \text{ is a learning rate.}$$

Restriced Boltzmann Machines: Training and Deep Learning

- Many **improvements of this basic procedure** exist [Hinton 2010]:
  - use a momentum term for the training,
  - use actual probabilities instead of binary reconstructions,
  - use online-like training based on small batches etc.
- Restricted Boltzmann machines have also been used to build **deep networks** in a fashion similar to stacked auto-encoders for multi-layer perceptrons.
- Idea: train a restricted Boltzmann machine, then create a data set of hidden neuron activations by sampling from the trained Boltzmann machine, and build another restricted Boltzmann machine from the obtained data set.
- This procedure can be repeated several times and the resulting Boltzmann machines can then easily be stacked.
- The obtained stack is fine-tuned with a procedure similar to back-propagation [Hinton *et al.* 2006].