Chapter 9:
Hopfield Networks

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, i.e.

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

Hopfield Networks

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

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

The activation function of each neuron is a threshold function, i.e.

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

The output function of each neuron is the identity, i.e.

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

Hopfield Networks

Alternative activation function

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

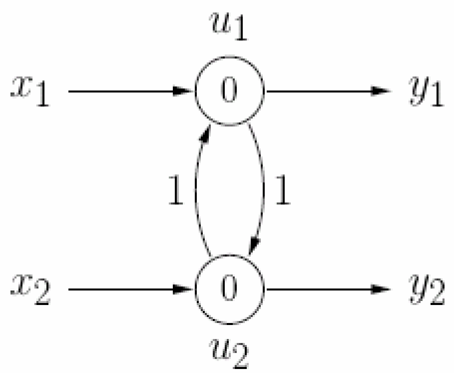
This activation function has advantages w.r.t. the physical interpretation of a Hopfield network.

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	-1	1
work phase	1	-1
	-1	1
	1	-1
	-1	1
	1	-1
	-1	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

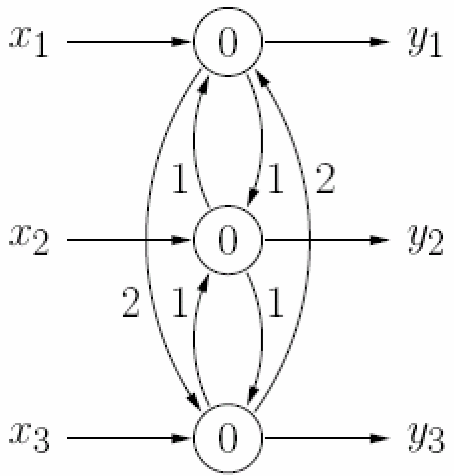
	u_1	u_2
input phase	$\mathbf{-1}$	$\mathbf{1}$
work phase	$\mathbf{1}$	1
	1	$\mathbf{1}$
	$\mathbf{1}$	1
	1	$\mathbf{1}$

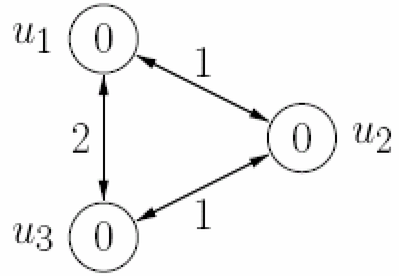
	u_1	u_2
input phase	$\mathbf{-1}$	$\mathbf{1}$
work phase	-1	$\mathbf{-1}$
	$\mathbf{-1}$	-1
	-1	$\mathbf{-1}$
	$\mathbf{-1}$	-1

- Regardless of the update order a stable state is reached.
- Which state is reached depends on the update order.

Hopfield Networks: Examples

Simplified representation of a Hopfield network



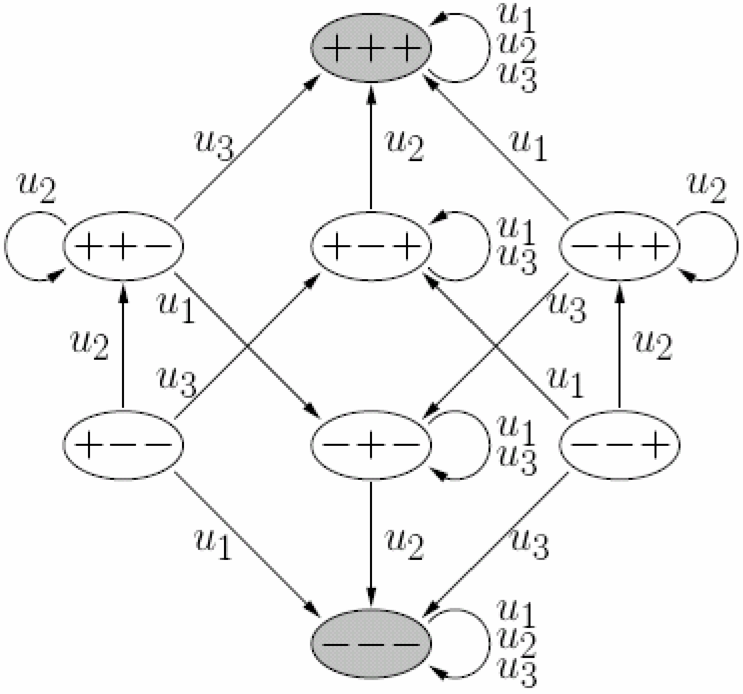


$$\mathbf{W} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

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

Hopfield Networks: State Graph

Graph of activation states and transitions



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

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

Hopfield Networks: Convergence

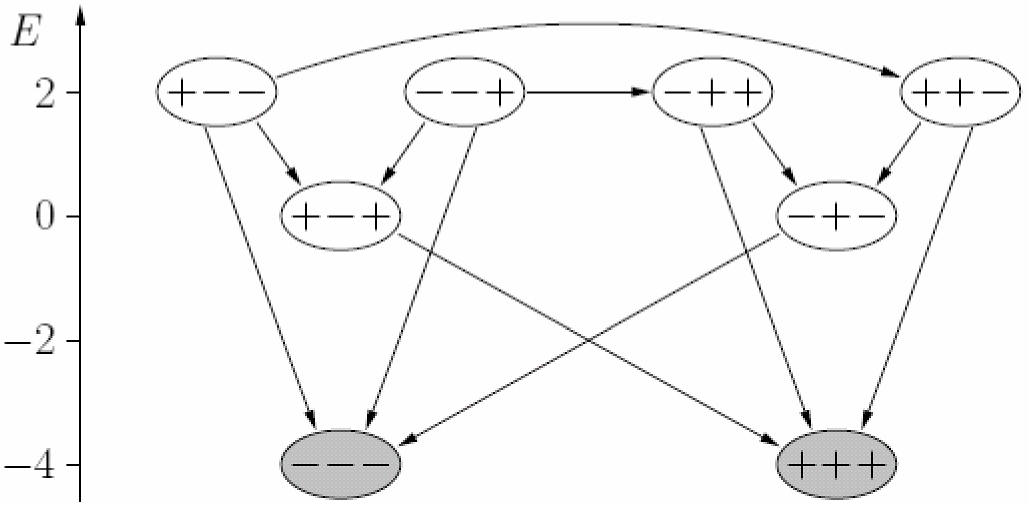
Consider the energy change resulting from an update that changes an activation:

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

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

Hopfield Networks: Examples

Arrange states in state graph according to their energy

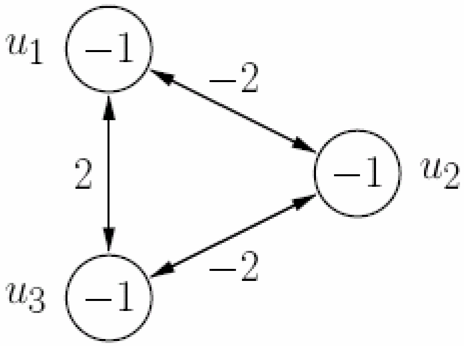


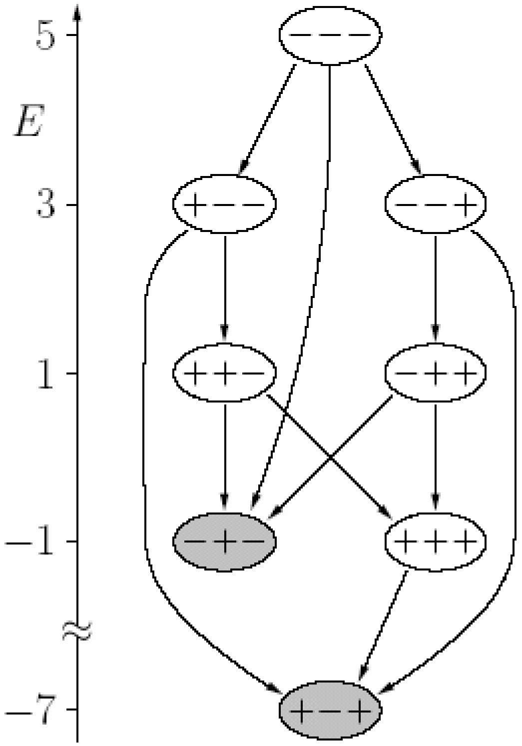
Energy function for example Hopfield network:

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

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

Hopfield Networks: Associative Memory

Idea: Use stable states to store patterns

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

Necessary and sufficient condition:

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

where

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

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

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

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

Choose

$$\mathbf{W} = \vec{p}\vec{p}^{\,T} - \mathbf{E}$$

where $\vec{p}\vec{p}^{\,T}$ is the so-called **outer product**.

With this matrix we have

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

Hopfield Networks: Associative Memory

Hebbian learning rule [Hebb 1949]

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

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

- 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:

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

Hopfield Networks: Associative Memory

Storing several patterns

Choose

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

If patterns are orthogonal, we have

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

and therefore

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

Hopfield Networks: Associative Memory

Storing several patterns

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

Note that the complements of the patterns are also stored.

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

But: Capacity is very small compared to the number of possible states (2^n).

Non-orthogonal patterns:

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

Associative Memory: Example

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

$$\mathbf{W} = \mathbf{W}_1 + \mathbf{W}_2 = \vec{p}_1\vec{p}_1^{\,T} + \vec{p}_2\vec{p}_2^{\,T} - 2\mathbf{E}$$

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:

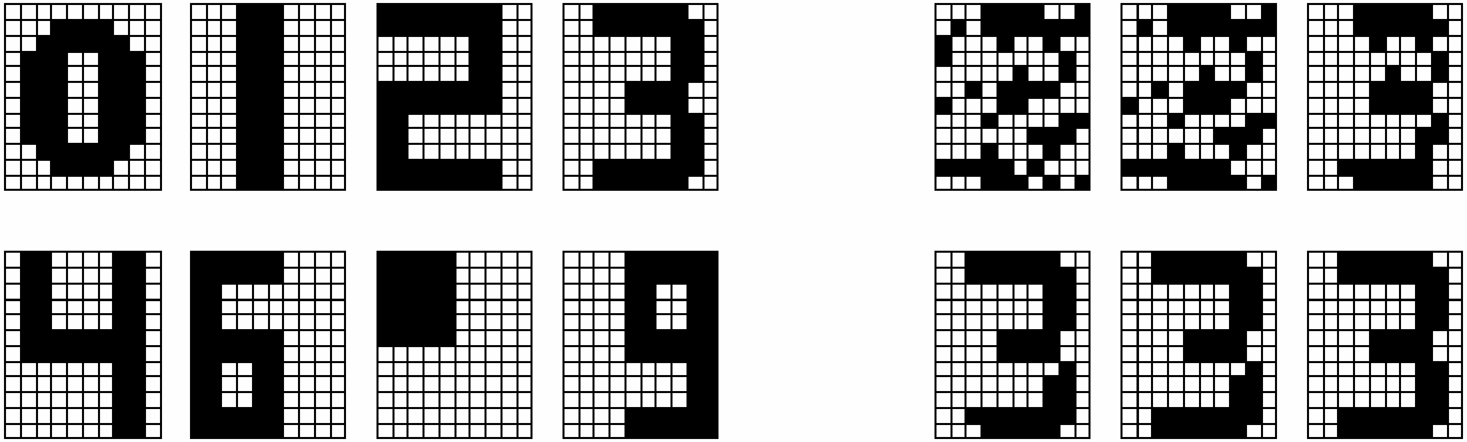
$$\mathbf{W} = \begin{pmatrix} 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 \\ 0 & -2 & 0 & 0 \\ -2 & 0 & 0 & 0 \end{pmatrix}.$$

Therefore it is

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

Associative Memory: Example

Example: Storing bit maps of numbers



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

Hopfield Networks: Associative Memory

Training a Hopfield network with the Delta rule

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

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

with the standard threshold function

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

Hopfield Networks: Associative Memory

Training a Hopfield network with the Delta rule

Turn weight matrix into a weight vector:

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

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 until convergence.

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

$$
\begin{aligned}
w_{uv}^0 &= 2w_{uv}^- \qquad \text{and} \\
\theta_u^0 &= \theta_u^- + \sum_{v \in U - \{u\}} w_{uv}^-
\end{aligned}
$$

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

$$
\begin{aligned}
w_{uv}^- &= \frac{1}{2} w_{uv}^0 \qquad \text{and} \\
\theta_u^- &= \theta_u^0 - \frac{1}{2} \sum_{v \in U - \{u\}} w_{uv}^0.
\end{aligned}
$$

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 $\theta_u = a\theta_u^{(1)} + b\theta_u^{(2)}$.

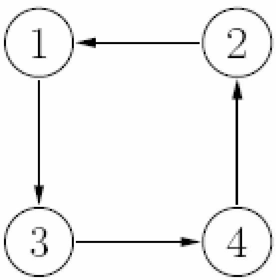
Proof: Just do the computations.

Idea: Additional conditions can be formalized separately and incorporated later.

Hopfield Networks: Solving Optimization Problems

Example: Traveling salesman problem

Idea: Represent tour by a matrix.



$$
\begin{array}{c c c}
 & \text{city} & \\
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} & \\
 & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \begin{array}{l} 1. \\ 2. \\ 3. \\ 4. \end{array} \ \text{step}
\end{array}
$$

An element a_{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.

Hopfield Networks: Solving Optimization Problems

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:

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

Hopfield Networks: Solving Optimization Problems

Additional conditions that have to be satisfied:

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

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

  i.e., each column of the matrix contains exactly one 1.

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

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

  i.e., 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

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

and the threshold values:

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

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

Chapter 10:

Recurrent Networks

Recurrent Networks: Cooling Law

A body of temperature ϑ_0 that is placed into an environment with temperature ϑ_A.

The cooling/heating of the body can be described by **Newton's cooling law**:

$$\frac{\mathrm{d}\vartheta}{\mathrm{d}t} = \dot{\vartheta} = -k(\vartheta - \vartheta_A).$$

Exact analytical solution:

$$\vartheta(t) = \vartheta_A + (\vartheta_0 - \vartheta_A)e^{-k(t-t_0)}$$

Approximate solution with **Euler-Cauchy polygon courses**:

$$\vartheta_1 = \vartheta(t_1) = \vartheta(t_0) + \dot{\vartheta}(t_0)\Delta t = \vartheta_0 - k(\vartheta_0 - \vartheta_A)\Delta t.$$

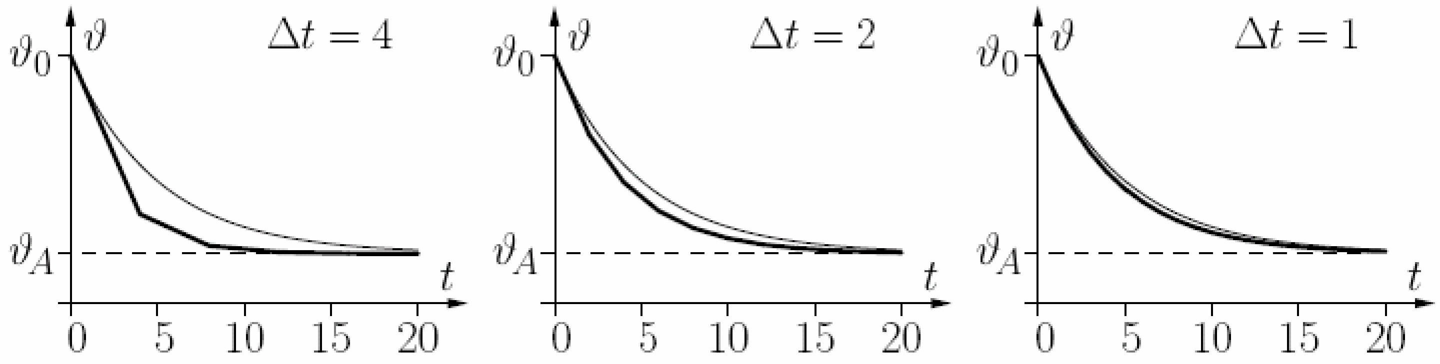
$$\vartheta_2 = \vartheta(t_2) = \vartheta(t_1) + \dot{\vartheta}(t_1)\Delta t = \vartheta_1 - k(\vartheta_1 - \vartheta_A)\Delta t.$$

General recursive formula:

$$\vartheta_i = \vartheta(t_i) = \vartheta(t_{i-1}) + \dot{\vartheta}(t_{i-1})\Delta t = \vartheta_{i-1} - k(\vartheta_{i-1} - \vartheta_A)\Delta t$$

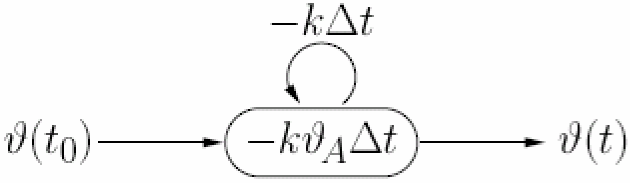
Recurrent Networks: Cooling Law

Euler–Cauchy polygon courses for different step widths:



The thin curve is the exact analytical solution.

Recurrent neural network:



Recurrent Networks: Cooling Law

More formal derivation of the recursive formula:

Replace differential quotient by **forward difference**

$$\frac{\mathrm{d}\vartheta(t)}{\mathrm{d}t} \approx \frac{\Delta\vartheta(t)}{\Delta t} = \frac{\vartheta(t+\Delta t) - \vartheta(t)}{\Delta t}$$

with sufficiently small Δt. Then it is

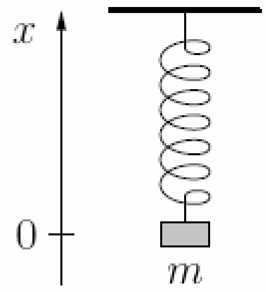
$$\vartheta(t+\Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k(\vartheta(t) - \vartheta_A)\Delta t,$$

$$\vartheta(t+\Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k\Delta t\vartheta(t) + k\vartheta_A\Delta t$$

and therefore

$$\vartheta_i \approx \vartheta_{i-1} - k\Delta t\vartheta_{i-1} + k\vartheta_A\Delta t.$$

Recurrent Networks: Mass on a Spring



Governing physical laws:

- **Hooke's law**: $F = c\Delta l = -cx$ (c is a spring dependent constant)
- **Newton's second law**: $F = ma = m\ddot{x}$ (force causes an acceleration)

Resulting differential equation:

$$m\ddot{x} = -cx \qquad \text{or} \qquad \ddot{x} = -\frac{c}{m}x.$$

Recurrent Networks: Mass on a Spring

General analytical solution of the differential equation:

$$x(t) = a\sin(\omega t) + b\cos(\omega t)$$

with the parameters

$$\omega = \sqrt{\frac{c}{m}}, \qquad \begin{aligned} a &= x(t_0)\sin(\omega t_0) \; + v(t_0)\cos(\omega t_0), \\ b &= x(t_0)\cos(\omega t_0) \; - v(t_0)\sin(\omega t_0). \end{aligned}$$

With given initial values $x(t_0) = x_0$ and $v(t_0) = 0$ and the additional assumption $t_0 = 0$ we get the simple expression

$$x(t) = x_0 \cos\left(\sqrt{\frac{c}{m}}\, t\right).$$

Recurrent Networks: Mass on a Spring

Turn differential equation into two coupled equations:

$$\dot{x} = v \qquad \text{and} \qquad \dot{v} = -\frac{c}{m}x.$$

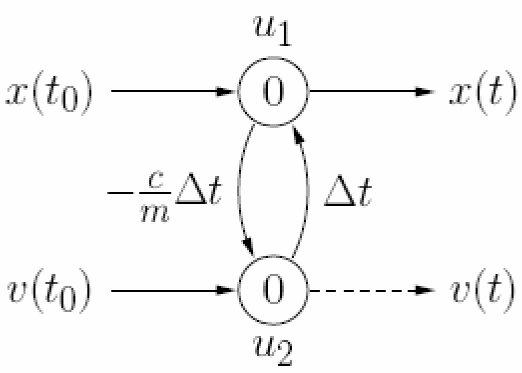
Approximate differential quotient by forward difference:

$$\frac{\Delta x}{\Delta t} = \frac{x(t+\Delta t) - x(t)}{\Delta t} = v \qquad \text{and} \qquad \frac{\Delta v}{\Delta t} = \frac{v(t+\Delta t) - v(t)}{\Delta t} = -\frac{c}{m}x$$

Resulting recursive equations:

$$\begin{aligned} x(t_i) &= x(t_{i-1}) + \Delta x(t_{i-1}) &= x(t_{i-1}) + \Delta t \cdot v(t_{i-1}) \qquad \text{and} \\ v(t_i) &= v(t_{i-1}) + \Delta v(t_{i-1}) &= v(t_{i-1}) - \frac{c}{m}\Delta t \cdot x(t_{i-1}). \end{aligned}$$

Recurrent Networks: Mass on a Spring



Neuron u_1: $\quad f_{\text{net}}^{(u_1)}(v, w_{u_1u_2}) = w_{u_1u_2} v = -\dfrac{c}{m}\Delta t\ v \qquad$ and

$$f_{\text{act}}^{(u_1)}(\text{act}_{u_1}, \text{net}_{u_1}, \theta_{u_1}) = \text{act}_{u_1} + \text{net}_{u_1} - \theta_{u_1},$$

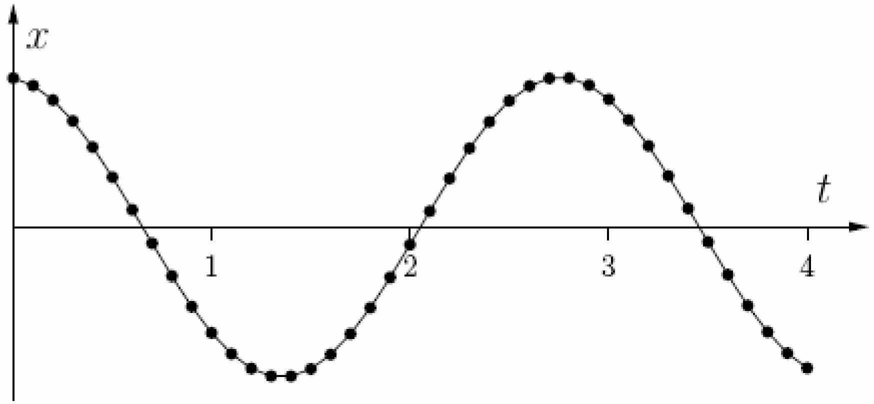
Neuron u_2: $\quad f_{\text{net}}^{(u_2)}(x, w_{u_2u_1}) = w_{u_2u_1} x = \Delta t\ x \qquad$ and

$$f_{\text{act}}^{(u_2)}(\text{act}_{u_2}, \text{net}_{u_2}, \theta_{u_2}) = \text{act}_{u_2} + \text{net}_{u_2} - \theta_{u_2}.$$

Recurrent Networks: Mass on a Spring

Some computation steps of the neural network:

t	v	x
0.0	0.0000	1.0000
0.1	−0.5000	0.9500
0.2	−0.9750	0.8525
0.3	−1.4012	0.7124
0.4	−1.7574	0.5366
0.5	−2.0258	0.3341
0.6	−2.1928	0.1148



- The resulting curve is close to the analytical solution.
- The approximation gets better with smaller step width.

Recurrent Networks: Differential Equations

General representation of explicit n-th order differential equation:

$$x^{(n)} = f(t, x, \dot{x}, \ddot{x}, \ldots, x^{(n-1)})$$

Introduce $n-1$ intermediary quantities

$$y_1 = \dot{x}, \qquad y_2 = \ddot{x}, \qquad \ldots \qquad y_{n-1} = x^{(n-1)}$$

to obtain the system

$$\begin{aligned}
\dot{x} &= y_1, \\
\dot{y}_1 &= y_2, \\
&\vdots \\
\dot{y}_{n-2} &= y_{n-1}, \\
\dot{y}_{n-1} &= f(t, x, y_1, y_2, \ldots, y_{n-1})
\end{aligned}$$

of n coupled first order differential equations.

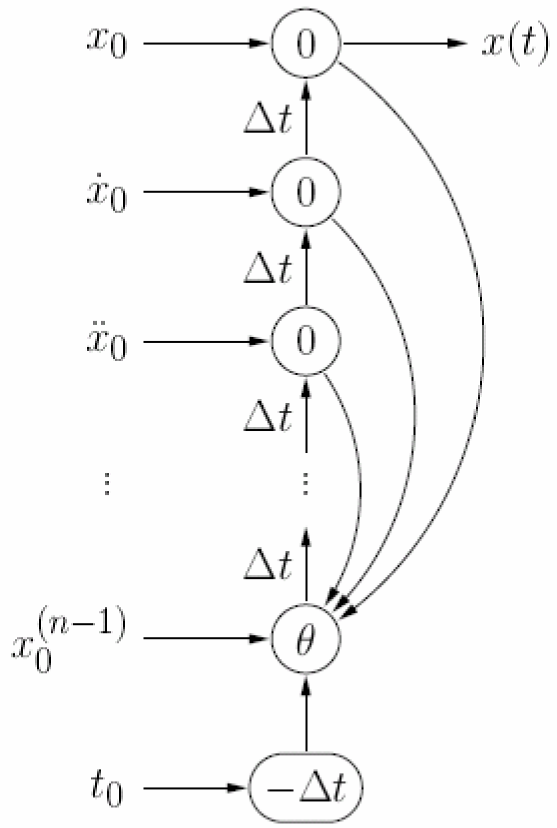
Recurrent Networks: Differential Equations

Replace differential quotient by forward distance to obtain the recursive equations

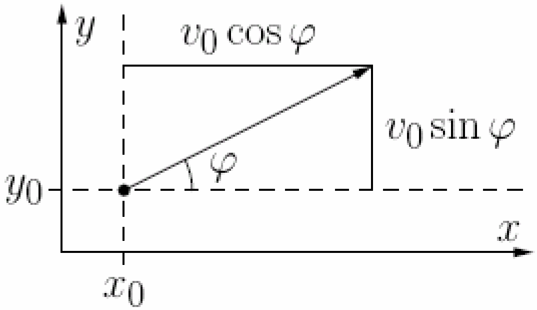
$$
\begin{aligned}
x(t_i) &= \quad x(t_{i-1}) + \Delta t \cdot y_1(t_{i-1}), \\
y_1(t_i) &= \quad y_1(t_{i-1}) + \Delta t \cdot y_2(t_{i-1}), \\
&\vdots \\
y_{n-2}(t_i) &= y_{n-2}(t_{i-1}) + \Delta t \cdot y_{n-3}(t_{i-1}), \\
y_{n-1}(t_i) &= y_{n-1}(t_{i-1}) + f(t_{i-1}, x(t_{i-1}), y_1(t_{i-1}), \ldots, y_{n-1}(t_{i-1}))
\end{aligned}
$$

- Each of these equations describes the update of one neuron.
- The last neuron needs a special activation function.

Recurrent Networks: Differential Equations



Recurrent Networks: Diagonal Throw



Diagonal throw of a body.

Two differential equations (one for each coordinate):

$$\ddot{x} = 0 \qquad \text{and} \qquad \ddot{y} = -g,$$

where $g = 9.81\,\mathrm{ms}^{-2}$.

Initial conditions $x(t_0) = x_0$, $y(t_0) = y_0$, $\dot{x}(t_0) = v_0 \cos\varphi$ and $\dot{y}(t_0) = v_0 \sin\varphi$.

Recurrent Networks: Diagonal Throw

Introduce intermediary quantities

$$v_x = \dot{x} \qquad \text{and} \qquad v_y = \dot{y}$$

to reach the system of differential equations:

$$\begin{aligned} \dot{x} &= v_x, & \dot{v}_x &= 0, \\ \dot{y} &= v_y, & \dot{v}_y &= -g, \end{aligned}$$

from which we get the system of recursive update formulae

$$\begin{aligned} x(t_i) &= x(t_{i-1}) + \Delta t \; v_x(t_{i-1}), & v_x(t_i) &= v_x(t_{i-1}), \\ y(t_i) &= y(t_{i-1}) + \Delta t \; v_y(t_{i-1}), & v_y(t_i) &= v_y(t_{i-1}) - \Delta t \; g. \end{aligned}$$

Recurrent Networks: Diagonal Throw

Better description: Use **vectors** as inputs and outputs

$$\ddot{\vec{r}} = -g\vec{e}_y,$$

where $\vec{e}_y = (0, 1)$.

Initial conditions are $\vec{r}(t_0) = \vec{r}_0 = (x_0, y_0)$ and $\dot{\vec{r}}(t_0) = \vec{v}_0 = (v_0 \cos\varphi, v_0 \sin\varphi)$.

Introduce one **vector-valued** intermediary quantity $\vec{v} = \dot{\vec{r}}$ to obtain

$$\dot{\vec{r}} = \vec{v}, \qquad \dot{\vec{v}} = -g\vec{e}_y$$

This leads to the recursive update rules

$$\begin{aligned} \vec{r}(t_i) &= \vec{r}(t_{i-1}) + \Delta t\ \vec{v}(t_{i-1}), \\ \vec{v}(t_i) &= \vec{v}(t_{i-1}) - \Delta t\ g\vec{e}_y \end{aligned}$$

Recurrent Networks: Diagonal Throw

Advantage of vector networks becomes obvious if friction is taken into account:

$$\vec{a} = -\beta\vec{v} = -\beta\dot{\vec{r}}$$

β is a constant that depends on the size and the shape of the body.
This leads to the differential equation

$$\ddot{\vec{r}} = -\beta\dot{\vec{r}} - g\vec{e}_y.$$

Introduce the intermediary quantity $\vec{v} = \dot{\vec{r}}$ to obtain

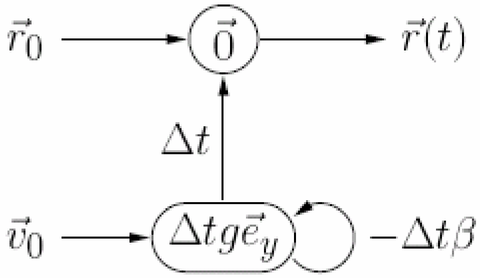
$$\dot{\vec{r}} = \vec{v}, \qquad \dot{\vec{v}} = -\beta\vec{v} - g\vec{e}_y,$$

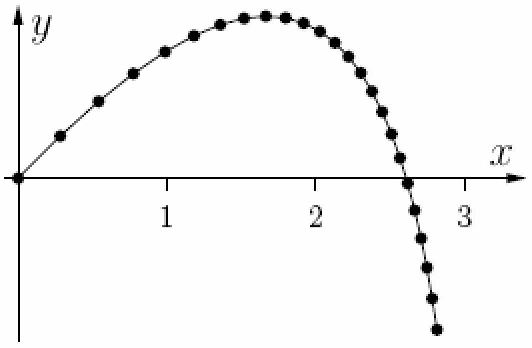
from which we obtain the recursive update formulae

$$\begin{aligned}\vec{r}(t_i) &= \vec{r}(t_{i-1}) + \Delta t\ \vec{v}(t_{i-1}),\\ \vec{v}(t_i) &= \vec{v}(t_{i-1}) - \Delta t\ \beta\ \vec{v}(t_{i-1}) - \Delta t\ g\vec{e}_y.\end{aligned}$$

Recurrent Networks: Diagonal Throw

Resulting recurrent neural network:





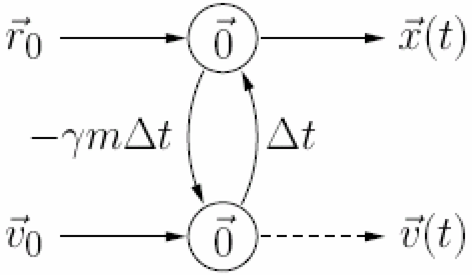
- There are no strange couplings as there would be in a non-vector network.
- Note the deviation from a parabola that is due to the friction.

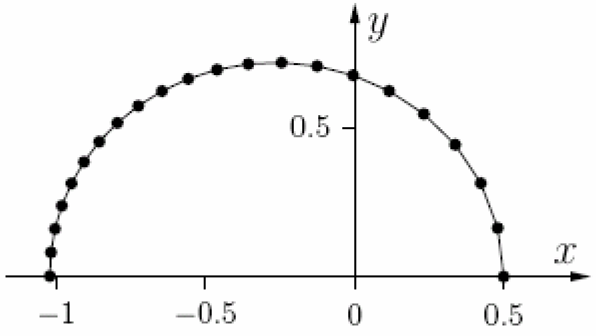
Recurrent Networks: Planet Orbit

$$\ddot{\vec{r}} = -\gamma m \frac{\vec{r}}{|\vec{r}\,|^3}, \qquad \Rightarrow \qquad \dot{\vec{r}} = \vec{v}, \qquad \dot{\vec{v}} = -\gamma m \frac{\vec{r}}{|\vec{r}\,|^3}.$$

Recursive update rules:

$$\begin{aligned} \vec{r}(t_i) &= \vec{r}(t_{i-1}) + \Delta t\ \vec{v}(t_{i-1}), \\ \vec{v}(t_i) &= \vec{v}(t_{i-1}) - \Delta t\ \gamma m \frac{\vec{r}(t_{i-1})}{|\vec{r}(t_{i-1})|^3}, \end{aligned}$$





Recurrent Networks: Backpropagation through Time

Idea: Unfold the network between training patterns,
i.e., create one neuron for each point in time.

Example: **Newton's cooling law**

$$\vartheta(t_0) \longrightarrow \bigcirc \xrightarrow{1-k\Delta t} \theta \xrightarrow{1-k\Delta t} \theta \xrightarrow{1-k\Delta t} \theta \xrightarrow{1-k\Delta t} \theta \longrightarrow \vartheta(t)$$

Unfolding into four steps. It is $\theta = -k\vartheta_A \Delta t$.

- Training is standard backpropagation on unfolded network.
- All updates refer to the same weight.
- updates are carried out after first neuron is reached.