Chapter 9:
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.

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

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

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

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

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

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

\[ f_{\text{act}}(u; \text{net}_u, \theta, \text{act}_u) = \begin{cases} 
1, & \text{if } \text{net}_u > \theta, \\
-1, & \text{if } \text{net}_u < \theta, \\
\text{act}_u, & \text{if } \text{net}_u = \theta.
\end{cases} \]

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

General weight matrix of a Hopfield network

\[
W = \begin{pmatrix}
0 & w_{u_1 u_2} & \cdots & w_{u_1 u_n} \\
w_{u_1 u_2} & 0 & \cdots & w_{u_2 u_n} \\
\vdots & \vdots & \ddots & \vdots \\
w_{u_1 u_n} & w_{u_1 u_n} & \cdots & 0
\end{pmatrix}
\]
Hopfield Networks: Examples

Very simple Hopfield network

\[ W = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \]

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

<table>
<thead>
<tr>
<th>( u_1 )</th>
<th>( u_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Sequential update of neuron activations

<table>
<thead>
<tr>
<th>u₁</th>
<th>u₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u₁</th>
<th>u₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

- 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

\[
W = \begin{pmatrix}
0 & 1 & 2 \\
1 & 0 & 1 \\
2 & 1 & 0 \\
\end{pmatrix}
\]

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

$$E = -\frac{1}{2} \overrightarrow{\text{act}}^T \mathbf{W} \overrightarrow{\text{act}} + \overrightarrow{\theta}^T \overrightarrow{\text{act}}$$

$$= -\frac{1}{2} \sum_{u,v \in U, u \neq v} w_{uv} \text{act}_u \text{act}_v + \sum_{u \in U} \theta_u \text{act}_u.$$
Hopfield Networks: Convergence

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

$$\Delta E = E^{(\text{new})} - E^{(\text{old})} = \left( - \sum_{v \in U \setminus \{u\}} w_{uv} \text{act}_u^{(\text{new})} \text{act}_v + \theta_u \text{act}_u^{(\text{new})} \right)$$
$$- \left( - \sum_{v \in U \setminus \{u\}} w_{uv} \text{act}_u^{(\text{old})} \text{act}_v + \theta_u \text{act}_u^{(\text{old})} \right)$$
$$= \left( \text{act}_u^{(\text{old})} - \text{act}_u^{(\text{new})} \right) \left( \sum_{v \in U \setminus \{u\}} w_{uv} \text{act}_v - \theta_u \right).$$

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

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

Arrange states in state graph according to their energy

Energy function for example Hopfield network:

\[ E = -\text{act}_{u_1}\text{act}_{u_2} - 2\text{act}_{u_1}\text{act}_{u_3} - \text{act}_{u_2}\text{act}_{u_3}. \]
Hopfield Networks: Examples

The state graph need not be symmetric

\[ u_1 \overset{2}{\rightarrow} -1 \overset{2}{\rightarrow} -1 \quad u_2 \]

\[ u_3 \overset{-2}{\rightarrow} -1 \]

\[ +-- \]

\[ --- \]

\[ +-- \]

\[ ++- \]

\[ +++ \]

\[ --- \]

\[ +-- \]

\[ +-- \]
Physical interpretation: Magnetism

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

<table>
<thead>
<tr>
<th>physical</th>
<th>neural</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>neuron</td>
</tr>
<tr>
<td>magnetic moment (spin)</td>
<td>activation state</td>
</tr>
<tr>
<td>strength of outer magnetic field</td>
<td>threshold value</td>
</tr>
<tr>
<td>magnetic coupling of the atoms</td>
<td>connection weights</td>
</tr>
<tr>
<td>Hamilton operator of the magnetic field</td>
<td>energy function</td>
</tr>
</tbody>
</table>

Prof. Dr. Rudolf Kruse
Hopfield Networks: Associative Memory

Idea: Use stable states to store patterns

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

Necessary and sufficient condition:

\[
S(W \vec{p} - \vec{\theta}) = \vec{p},
\]

where

\[
S : \mathbb{R}^n \rightarrow \{-1, 1\}^n,
\]

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 $\theta = 0$ an appropriate matrix $W$ can easily be found. It suffices

$$W\bar{p} = c\bar{p} \quad \text{with} \quad c \in \mathbb{R}^+.$$ 

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

Choose

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

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

With this matrix we have

$$W\bar{p} = (\bar{p}\bar{p}^T)\bar{p} - \frac{1}{2}\mathbf{E}\bar{p} = \bar{p} (\bar{p}^T \bar{p}) - \bar{p} \\ = n\bar{p} - \bar{p} = (n - 1)\bar{p}.$$
Hopfield Networks: Associative Memory

Hebbian learning rule  [Hebb 1949]

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

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

- 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:

With \( W\tilde{p} = (n - 1)\tilde{p} \) it is also \( W(-\tilde{p}) = (n - 1)(-\tilde{p}) \).
Storing several patterns

Choose

\[ W \vec{p}_j = \sum_{i=1}^{m} W_i \vec{p}_j = \left( \sum_{i=1}^{m} (\vec{p}_i \vec{p}_i^T) \vec{p}_j \right) - m \overline{\vec{p}_j} = \overline{\vec{p}_j} \]

\[ = \left( \sum_{i=1}^{m} \vec{p}_i (\vec{p}_i^T \vec{p}_j) \right) - m \overline{\vec{p}_j} \]

If patterns are orthogonal, we have

\[ p_i^T \vec{p}_j = \begin{cases} 0, & \text{if } i \neq j, \\ n, & \text{if } i = j, \end{cases} \]

and therefore

\[ W \vec{p}_j = (n - m) \overline{\vec{p}_j}. \]
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 \( W \vec{p}_j = (n - m)\vec{p}_j \) it is also \( W(-\vec{p}_j) = (n - m)(-\vec{p}_j) \).

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

Non-orthogonal patterns:

\[
W \vec{p}_j = (n - m)\vec{p}_j + \sum_{\substack{i=1 \atop i \neq j}}^{m} \vec{p}_i (\vec{p}_i^T \vec{p}_j) \\
\text{“disturbance term”}
\]
Associative Memory: Example

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

$$W = W_1 + W_2 = \vec{p}_1 \vec{p}_1^T + \vec{p}_2 \vec{p}_2^T - 2E$$

where

$$W_1 = \begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{pmatrix}, \quad W_2 = \begin{pmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{pmatrix}.$$ 

The full weight matrix is:

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

Therefore it is

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

\[
\begin{align*}
     s(0) + w_{u_1 u_2} \text{act}^{(p)}_{u_2} + \ldots + w_{u_1 u_n} \text{act}^{(p)}_{u_n} - \theta_{u_1} &= \text{act}^{(p)}_{u_1}, \\
     s(w_{u_2 u_1} \text{act}^{(p)}_{u_1} + 0) + \ldots + w_{u_2 u_n} \text{act}^{(p)}_{u_n} - \theta_{u_2} &= \text{act}^{(p)}_{u_2}, \\
     \vdots & \quad \vdots \quad \vdots \\
     s(w_{u_n u_1} \text{act}^{(p)}_{u_1} + w_{u_n u_2} \text{act}^{(p)}_{u_2} + \ldots + 0 - \theta_{u_n}) &= \text{act}^{(p)}_{u_n}.
\end{align*}
\]

with the standard threshold function

\[
s(x) = \begin{cases} 
1, & \text{if } x \geq 0, \\
-1, & \text{otherwise.}
\end{cases}
\]
Hopfield Networks: Associative Memory

Training a Hopfield network with the Delta rule

Turn weight matrix into a weight vector:

\[ \tilde{w} = (w_{u_1u_2}, w_{u_1u_3}, \ldots, w_{u_1u_n}, w_{u_2u_3}, \ldots, w_{u_2u_n}, \ldots, w_{u_{n-1}u_n}, -\theta_{u_1}, -\theta_{u_2}, \ldots, -\theta_{u_n}) \].

Construct input vectors for a threshold logic unit

\[ \tilde{z}_2 = (\text{act}_{u_1}^{(p)}, 0, \ldots, 0, \text{act}_{u_3}^{(p)}, \ldots, \text{act}_{u_n}^{(p)}, 0, 1, 0, \ldots, 0) \text{ with } n - 2 \text{ zeros}. \]

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.
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 $act_u \in \{-1, 1\}$ to $act_u \in \{0, 1\}$:

$$w_{uv}^0 = 2w_{uv}^-$$
$$\theta_u^0 = \theta_u^- + \sum_{v \in U \setminus \{u\}} w_{uv}^-$$

From $act_u \in \{0, 1\}$ to $act_u \in \{-1, 1\}$:

$$w_{uv}^- = \frac{1}{2} w_{uv}^0$$
$$\theta_u^- = \theta_u^0 - \frac{1}{2} \sum_{v \in U \setminus \{u\}} w_{uv}^0.$$
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

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

$i = 1, 2$, 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.

![Graph of a graph with nodes 1, 2, 3, 4 and edges connecting them.]

<table>
<thead>
<tr>
<th>city</th>
<th>1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0</td>
<td>1 0 0</td>
</tr>
</tbody>
</table>

1. 2. step
2. 3.
3. 4.

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

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

Double summation over steps (index \( i \)) needed:

\[ E_1 = \sum_{(i_1,j_1)\in\{1,...,n\}^2} \sum_{(i_2,j_2)\in\{1,...,n\}^2} d_{j_1, j_2} \cdot \delta_{(i_1 \mod n) + 1, i_2} \cdot m_{i_1,j_1} \cdot m_{i_2,j_2}. \]

where

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

Symmetric version of the energy function:

\[ E_1 = -\frac{1}{2} \sum_{(i_1,j_1)\in\{1,...,n\}^2} \sum_{(i_2,j_2)\in\{1,...,n\}^2} -d_{j_1, j_2} \cdot (\delta_{(i_1 \mod n) + 1, i_2} + \delta_{1, (i_2 \mod n) + 1}) \cdot m_{i_1,j_1} \cdot m_{i_2,j_2}. \]
Hopfield Networks: Solving Optimization Problems

Additional conditions that have to be satisfied:

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

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

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

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

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

  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:

\[
E^*_2 = \sum_{j=1}^{n} \left( \left( \sum_{i=1}^{n} m_{ij} \right)^2 - 2 \sum_{i=1}^{n} m_{ij} + 1 \right)
= \sum_{j=1}^{n} \left( \left( \sum_{i_1=1}^{n} m_{i_1j} \right) \left( \sum_{i_2=1}^{n} m_{i_2j} \right) - 2 \sum_{i=1}^{n} m_{ij} + 1 \right)
= \sum_{j=1}^{n} \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} m_{i_1j}m_{i_2j} - 2 \sum_{j=1}^{n} \sum_{i=1}^{n} m_{ij} + n.
\]

Double summation over cities (index \(i\)) needed:

\[
E_2 = \sum_{(i_1,j_1)\in\{1,\ldots,n\}^2} \sum_{(i_2,j_2)\in\{1,\ldots,n\}^2} \delta_{j_1j_2} \cdot m_{i_1j_1} \cdot m_{i_2j_2} - 2 \sum_{(i,j)\in\{1,\ldots,n\}^2} m_{ij}.
\]
Hopfield Networks: Solving Optimization Problems

Resulting energy function:

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

Second additional condition is handled in a completely analogous way:

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

Combining the energy functions:

\[ E = aE_1 + bE_2 + cE_3 \quad \text{where} \quad \frac{b}{a} = \frac{c}{a} > 2, \quad \max_{(j_1,j_2) \in \{1,\ldots,n\}^2} d_{j_1,j_2}. \]
Hopfield Networks: Solving Optimization Problems

From the resulting energy function we can read the weights

\[ w_{ij} = -ad_{ij} \cdot (\delta_{i_1 \mod n+1, j} + \delta_{i_1, (i_2 \mod n+1)} - 2b\delta_{j_1 j_2} - 2c\delta_{i_1 i_2} ) \]

from \( E_1 \) to \( E_2 \) from \( E_3 \)

and the threshold values:

\[ \theta_{ij} = 0a - 2b - 2c = -2(b + c). \]

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 \(\vartheta_0\) that is placed into an environment with temperature \(\vartheta_A\).

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

\[
\frac{d\vartheta}{dt} = \dot{\vartheta} = -k(\vartheta - \vartheta_A).
\]

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:

\[
\begin{align*}
\vartheta(t_0) & \rightarrow -k\vartheta_A \Delta t \\
\rightarrow (t) & \rightarrow -k\vartheta A \Delta t \\
\end{align*}
\]
More formal derivation of the recursive formula:

Replace differential quotient by **forward difference**

\[
\frac{dθ(t)}{dt} \approx \frac{Δθ(t)}{Δt} = \frac{θ(t + Δt) - θ(t)}{Δt}
\]

with sufficiently small \(Δt\). Then it is

\[
θ(t + Δt) - θ(t) = Δθ(t) \approx -k(θ(t) - θ_A)Δt,
\]

\[
θ(t + Δt) - θ(t) = Δθ(t) \approx -kΔtθ(t) + kθ_AΔt
\]

and therefore

\[
θ_i \approx θ_{i-1} - kΔtθ_{i-1} + kθ_AΔt.
\]
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:

\[
m\dddot{x} = -cx \quad \text{or} \quad \dddot{x} = -\frac{c}{m}x.
\]
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

\[
\omega = \sqrt{\frac{c}{m}}, \quad a = x(t_0) \sin(\omega t_0) + v(t_0) \cos(\omega t_0), \\
b = x(t_0) \cos(\omega t_0) - v(t_0) \sin(\omega t_0).
\]

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:

\[ \dot{x} = v \quad \text{and} \quad \dot{v} = -\frac{c}{m}x. \]

Approximate differential quotient by forward difference:

\[ \frac{\Delta x}{\Delta t} = \frac{x(t + \Delta t) - x(t)}{\Delta t} = v \quad \text{and} \quad \frac{\Delta v}{\Delta t} = \frac{v(t + \Delta t) - v(t)}{\Delta t} = -\frac{c}{m}x. \]

Resulting recursive equations:

\[ x(t_i) = x(t_{i-1}) + \Delta x(t_{i-1}) = x(t_{i-1}) + \Delta t \cdot v(t_{i-1}) \quad \text{and} \]
\[ v(t_i) = v(t_{i-1}) + \Delta v(t_{i-1}) = v(t_{i-1}) - \frac{c}{m} \Delta t \cdot x(t_{i-1}). \]
Recurrent Networks: Mass on a Spring

Neuron $u_1$:

$$f_{\text{net}}^{(u_1)}(v, w_{u_1u_2}) = w_{u_1u_2}v = -\frac{c}{m} \Delta t v$$

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 $u_2$:

$$f_{\text{net}}^{(u_2)}(x, w_{u_2u_1}) = w_{u_2u_1}x = \Delta t x$$

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:

<table>
<thead>
<tr>
<th>$t$</th>
<th>$v$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>0.1</td>
<td>−0.5000</td>
<td>0.9500</td>
</tr>
<tr>
<td>0.2</td>
<td>−0.9750</td>
<td>0.8525</td>
</tr>
<tr>
<td>0.3</td>
<td>−1.4012</td>
<td>0.7124</td>
</tr>
<tr>
<td>0.4</td>
<td>−1.7574</td>
<td>0.5366</td>
</tr>
<tr>
<td>0.5</td>
<td>−2.0258</td>
<td>0.3341</td>
</tr>
<tr>
<td>0.6</td>
<td>−2.1928</td>
<td>0.1148</td>
</tr>
</tbody>
</table>

- 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:

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

Introduce $n - 1$ intermediary quantities

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

to obtain the system

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

of $n$ coupled first order differential equations.
Recurrent Networks: Differential Equations

Replace differential quotient by forward distance to obtain the recursive equations

\[ x(t_i) = x(t_{i-1}) + \Delta t \cdot y_1(t_{i-1}), \]
\[ y_1(t_i) = y_1(t_{i-1}) + \Delta t \cdot y_2(t_{i-1}), \]
\[ \vdots \]
\[ y_{n-2}(t_i) = y_{n-2}(t_{i-1}) + \Delta t \cdot y_{n-3}(t_{i-1}), \]
\[ y_{n-1}(t_i) = y_{n-1}(t_{i-1}) + f(t_{i-1}, x(t_{i-1}), y_1(t_{i-1}), \ldots, y_{n-1}(t_{i-1})). \]

- 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

![Diagram of diagonal throw of a body](image)

Diagonal throw of a body.

Two differential equations (one for each coordinate):

\[ \ddot{x} = 0 \quad \text{and} \quad \ddot{y} = -g, \]

where \( g = 9.81 \text{ ms}^{-2} \).

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

\[ v_x = \dot{x} \quad \text{and} \quad v_y = \dot{y} \]

to reach the system of differential equations:

\[ \dot{x} = v_x, \quad \dot{v}_x = 0, \]
\[ \dot{y} = v_y, \quad \dot{v}_y = -g, \]

from which we get the system of recursive update formulae

\[ x(t_i) = x(t_{i-1}) + \Delta t \ v_x(t_{i-1}), \]
\[ v_x(t_i) = v_x(t_{i-1}), \]
\[ y(t_i) = y(t_{i-1}) + \Delta t \ v_y(t_{i-1}), \]
\[ v_y(t_i) = v_y(t_{i-1}) - \Delta t \ g. \]
Recurrent Networks: Diagonal Throw

Better description: Use vectors as inputs and outputs

\[ \ddot{r} = -g \bar{e}_y, \]

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

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

Introduce one vector-valued intermediary quantity \( \bar{u} = \dot{r} \) to obtain

\[ \dot{r} = \bar{u}, \quad \dot{v} = -g \bar{e}_y \]

This leads to the recursive update rules

\[ \begin{align*}
    \bar{r}(t_i) &= \bar{r}(t_{i-1}) + \Delta t \bar{u}(t_{i-1}), \\
    \bar{u}(t_i) &= \bar{u}(t_{i-1}) - \Delta t \ g \bar{e}_y
\end{align*} \]
Recurrent Networks: Diagonal Throw

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

\[ \ddot{a} = -\beta \dot{v} = -\beta \ddot{r} \]

\( \beta \) is a constant that depends on the size and the shape of the body. This leads to the differential equation

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

Introduce the intermediary quantity \( \vec{v} = \vec{r} \) to obtain

\[ \dot{r} = \vec{v}, \quad \vec{v} = -\beta \vec{v} - g \vec{e}_y, \]

from which we obtain the recursive update formulae

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

\[ \ddot{\mathbf{r}} = -\gamma m \frac{\mathbf{r}}{|\mathbf{r}|^3}, \quad \Rightarrow \quad \mathbf{r} = \mathbf{v}, \quad \mathbf{\dot{v}} = -\gamma m \frac{\mathbf{r}}{|\mathbf{r}|^3}. \]

Recursive update rules:

\[
\begin{align*}
\mathbf{r}(t_i) & = \mathbf{r}(t_{i-1}) + \Delta t \mathbf{v}(t_{i-1}), \\
\mathbf{v}(t_i) & = \mathbf{v}(t_{i-1}) - \Delta t \gamma m \frac{\mathbf{r}(t_{i-1})}{|\mathbf{r}(t_{i-1})|^3};
\end{align*}
\]
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

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

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.