Lecture 5

- ❖ **Multi-layer Perceptrons**
  - ✓ **Chain rule and Computation graphs**
  - ✓ **Backpropogation**
  - ✓ **Gradient Descent Variants**
  - ✓ **Radial Basis Function Neural Networks**
  - ✓ **Self Organizing Maps**

CENG 632- Computational Intelligence, 2024-2025, Spring
Assist. Prof. Dr. Osman GÖKALP

Adapted from lecture notes of Christian Borgelt (https://computational-intelligence.eu/Lecture+Material.html)

Summary of Key Takeaways from Last Week

- **Linear regression** can fit an $a + bx$ model for a given input/output data using normal equations.
- **Polynomial regression** can fit an $a_0 + a_1 x + \cdots + a_m x^m$ model for a given input/output data using norma equations.
- **Multilinear regression** can fit an $a_0 + \sum_{k=1}^{m} a_k x_k$ model (allows multiple input data) for a given input/output data using norma equations.
- For functions that are neither linear nor polynomial, if we can find a suitable **transformation** (e.g. logit transformation), we can **approximate** a solution using available regression methods.

Summary of Key Takeaways from Last Week

- The **Regression approach** using normal equations are **limited to two-layer perceptrons** without any hidden layer (the delta rule in TLUs also suffers from the same limitation).
- To **train neural networks** without this limitation, **gradient descent** algorithm can be used.
- The **learning rate (step size)** parameter in gradient descent is crucial for convergence—it affects speed, risks of overshooting, and getting stuck in local optima.
- Gradient descent requires **partial derivatives** for each parameter.
- **Backpropagation** efficiently computes these derivatives for all layers, **including hidden layers**, using the **chain rule**.

Overview of the chain rule

- The **chain rule** is a fundamental differentiation rule used to compute the derivative of a composite function. It states that if a function $\boldsymbol{y}$ depends on an intermediate variable $\boldsymbol{u}$, which in turn depends on $\boldsymbol{x}$, then the derivative of $\boldsymbol{y}$ with respect to $\boldsymbol{x}$ is:

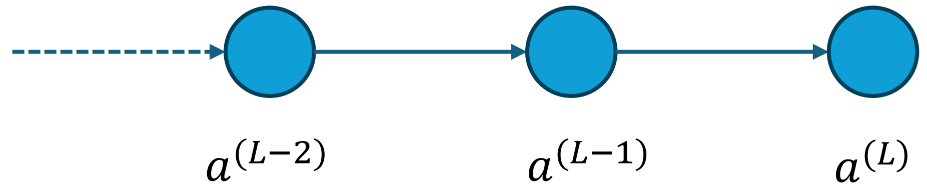
$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

- In **neural networks**, the chain rule is **essential** for computing **gradients during backpropagation**. Since each layer's output depends on the previous layer's output, derivatives are calculated layer by layer from the final layer (output) back to the initial layers (input). This allows efficient computation of **weight updates in deep networks**.

Computation graph for derivatives

- A **computation graph** is a directed acyclic graph (DAG) that represents mathematical operations as nodes and their dependencies as edges.
- In the context of **neural networks** and **backpropagation**, the graph illustrates how inputs are transformed into outputs through a **series of operations**, layer by layer.

Computation graph for derivatives – A simple example with one neuron per layer



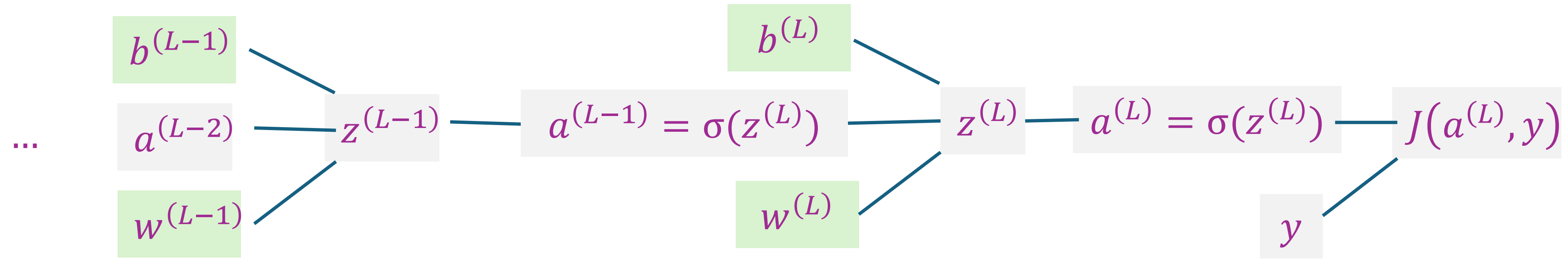
$$z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z)$$

$$J = Loss\left(a^{(L)}, y\right) = -(ylog_{a^{(L)}} + (1-y)log_{(1-a)^{(L)}})$$

- $\boldsymbol{z^{(L)}}$ is the **product** of **weights** and **inputs** of the previous layer. The **Bias** term $b^{(L)}$ is also added ($\boldsymbol{z} = f_{net}$).
- $\boldsymbol{\sigma}$ is the sigmoid activation function ($\boldsymbol{\sigma} = f_{act}$).
- $\boldsymbol{a^{(L)}}$ is the output of the neuron.
- $\boldsymbol{Loss}$ is a **binary cross entropy loss** (log loss) function to measure cost of binary classification problems. $\boldsymbol{J}$ is the cost value.
- Note that In this example f_{out} function is not used, e.g. $f_{out} = \boldsymbol{\sigma}$.

Computation graph for derivatives – A simple example with one neuron per layer



- Parameters $\boldsymbol{b}$ and $\boldsymbol{w}$ are to be updated using **gradient descent:**
  - $w^{(L)} = w^{(L)} - \eta . \partial_{w^{(L)}}$
  - $b^{(L)} = b^{(L)} - \eta . \partial_{b^{(L)}}$
  - $w^{(L-1)} = w^{(L-1)} - \eta . \partial_{w^{(L-1)}}$
  - $b^{(L-1)} = b^{(L-1)} - \eta . \partial_{b^{(L-1)}}$

  …

So, how to calculate these partial derivatives of ***J* (cost func.)** w.r.t. each parameter?

Computation graph for derivatives – A simple example with one neuron per layer

- $\partial_{w^{(L)}} = \frac{\partial_J}{\partial_{w^{(L)}}} = \frac{\partial_{z^{(L)}}}{\partial_{w^{(L)}}} \cdot \frac{\partial_{a^{(L)}}}{\partial_{z^{(L)}}} \cdot \frac{\partial_J}{\partial_{a^{(L)}}}$

- $\partial_{b^{(L)}} = \frac{\partial_J}{\partial_{b^{(L)}}} = \frac{\partial_{z^{(L)}}}{\partial_{b^{(L)}}} \cdot \frac{\partial_{a^{(L)}}}{\partial_{z^{(L)}}} \cdot \frac{\partial_J}{\partial_{a^{(L)}}}$

- $\partial_{w^{(L-1)}} = \frac{\partial_J}{\partial_{w^{(L-1)}}} = \frac{\partial_{z^{(L-1)}}}{\partial_{w^{(L-1)}}} \cdot \frac{\partial_{a^{(L-1)}}}{\partial_{z^{(L-1)}}} \cdot \frac{\partial_{z^{(L)}}}{\partial_{a^{(L-1)}}} \cdot \frac{\partial_{a^{(L)}}}{\partial_{z^{(L)}}} \cdot \frac{\partial_J}{\partial_{a^{(L)}}}$

- $\partial_{b^{(L-1)}} = \frac{\partial_J}{\partial_{b^{(L-1)}}} = \frac{\partial_{z^{(L-1)}}}{\partial_{b^{(L-1)}}} \cdot \frac{\partial_{a^{(L-1)}}}{\partial_{z^{(L-1)}}} \cdot \frac{\partial_{z^{(L)}}}{\partial_{a^{(L-1)}}} \cdot \frac{\partial_{a^{(L)}}}{\partial_{z^{(L)}}} \cdot \frac{\partial_J}{\partial_{a^{(L)}}}$

Computation graph for derivatives – A simple example with one neuron per layer

- $\frac{\partial_J}{\partial_{a^{(L)}}} = -\frac{y}{a^{(L)}} + \frac{1-y}{1-a^{(L)}} = \frac{\boldsymbol{a^{(L)}-y}}{\boldsymbol{a^{(L)}(1-a^{(L)})}}$ Note that $\frac{d}{d_a}\log(a) = \frac{1}{a}$
- $\frac{\partial_{a^{(L)}}}{\partial_{z^{(L)}}} = \sigma'\left(z^{(L)}\right) = \sigma\left(z^{(L)}\right)\left(1-\sigma\left(z^{(L)}\right)\right) = \boldsymbol{a^{(L)}(1-a^{(L)})}$ Note that $\sigma'(a) = \sigma(a)(1-\sigma(a))$
- $\frac{\partial_{z^{(L)}}}{\partial_{w^{(L)}}} = \boldsymbol{a^{(L-1)}}$
- $\frac{\partial_{z^{(L)}}}{\partial_{b^{(L)}}} = \boldsymbol{1}$
- $\frac{\partial_{z^{(L)}}}{\partial_{a^{(L-1)}}} = \boldsymbol{w^{(L)}}$
- …

$$\frac{\partial_{a^{(L)}}}{\partial_{z^{(L)}}} \cdot \frac{\partial_J}{\partial_{a^{(L)}}} = a^{(L)}(1-a^{(L)})\frac{a^{(L)}-y}{a^{(L)}(1-a^{(L)})} = \boldsymbol{a^{(L)}-y}$$

$$\frac{\partial_{a^{(L-1)}}}{\partial_{z^{(L-1)}}} \cdot \frac{\partial_{z^{(L)}}}{\partial_{a^{(L-1)}}} = \boldsymbol{a^{(L-1)}(1-a^{(L-1)})w^{(L)}}$$

Handling multiple neurons

- If there are **multiple output neurons**, calculate the sum (or avg.) of losses for each output.

$$J = \sum_{j=0}^{n_L - 1} Loss_j$$

- If there are **multiple neurons in the hidden layers**, use subscript notation to differentiate between them.

❑**Handling the weight from neuron k in layer $L-1$ to neuron k in layer L:**

Describes how sensitive the cost is to a weight $w_{jk}^{(L)}$.

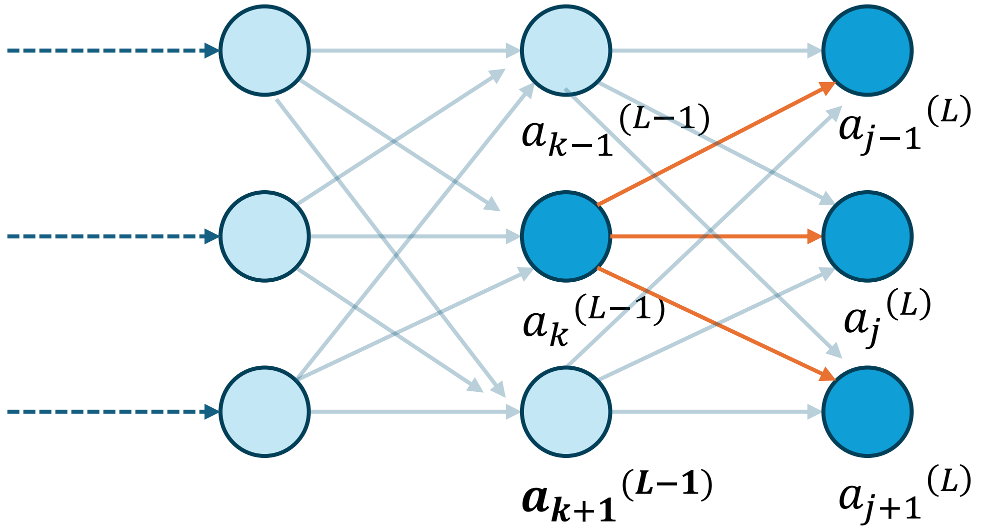
$$\frac{\partial_J}{\partial_{w_{jk}^{(L)}}} = \frac{\partial_{z_j^{(L)}}}{\partial_{w_{jk}^{(L)}}} \cdot \frac{\partial_{a_j^{(L)}}}{\partial_{z_j^{(L)}}} \cdot \frac{\partial_J}{\partial_{a_j^{(L)}}}$$

Handling multiple neurons

- **Handling the activation of neuron k in layer $L-1$**: We need sum operation, because the activation k affects all the neurons at the next layer.

$$\frac{\partial_J}{\partial_{a_k{}^{(L-1)}}} = \sum_{j=0}^{n_L-1} \frac{\partial_{z_j{}^{(L)}}}{\partial_{a_k{}^{(L-1)}}} \cdot \frac{\partial_{a_j{}^{(L)}}}{\partial_{z_j{}^{(L)}}} \cdot \frac{\partial_J}{\partial_{a_j{}^{(L)}}}$$

Describes how sensitive the cost is to an activation $a_k{}^{(L-1)}$.



Initializing parameters in a neural network

- Gradient descent can update current parameters but requires a starting point. How to determine **initial values of parameters**?
- Zero initialization will not work!
  - Each neuron in the same layer will have the same gradients during backpropagation.
  - They will update identically and remain symmetrical throughout training.
  - The network loses the ability to learn diverse features.
- Therefore, we use random initialization, He initialization, ...
- Random initialization:
  - Weights are initialized randomly with small values (often from a normal or uniform distribution).
  - Biases are often initialized to zero or small values.

Gradient Descent Variants

According to batch sizes:

- **Batch gradient descent**
    - uses all training samples before update
- **Stochastic gradient descent**
    - uses 1 training sample before update
- **Minibatch gradient descent**
    - uses training samples between 1 and the total number of training samples before each update.

Gradient Descent Variants

Indirect usage of current gradients

- **Momentum-based gradient descent**
  - a velocity term takes into past gradients account to smooth out the update.
- **RMSProp**
  - uses an exponentially weighted average of squared gradients to scale the learning rate for each parameter, allowing for more stable and faster convergence.
- **Adam**
  - adaptively updates the learning rate for each parameter based on both first moment (mean gradient) and second moment (variance of gradients).
- **Learning Rate Decay**
  - reduce the learning rate over time, ensuring faster exploration early on and finer adjustments later. There are several strategies for decay, including step decay, exponential decay, inverse time decay, polynomial decay, and cosine annealing.

Radial Basis Function Neural Networks

- Like multi-layer perceptrons, *radial basis function* networks are **feed-forward** neural networks with a **strictly layered structure**.
- The number of layers is always **three**, that is, there is **exactly one hidden layer**.
- Differs from multi-layer perceptrons in the network input and activation functions, especially in the hidden layer.
- Hidden layer **radial basis functions** are employed.

Radial Basis Function Networks

A **radial basis function network (RBFN)** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions

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

(ii) $C = (U_{\text{in}} \times U_{\text{hidden}}) \cup C', \quad C' \subseteq (U_{\text{hidden}} \times U_{\text{out}})$

The network input function of each hidden neuron is a **distance function** of the input vector and the weight vector, that is,

$$\forall u \in U_{\text{hidden}}: \qquad f_{\text{net}}^{(u)}(\vec{w}_u, \vec{\text{in}}_u) = d(\vec{w}_u, \vec{\text{in}}_u),$$

where $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_0^+$ is a function satisfying $\forall \vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n :$

$$\begin{array}{rlr} (i) & d(\vec{x}, \vec{y}) = 0 \;\Leftrightarrow\; \vec{x} = \vec{y}, & \\ (ii) & d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x}) & \text{(symmetry)}, \\ (iii) & d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z}) & \text{(triangle inequality)}. \end{array}$$

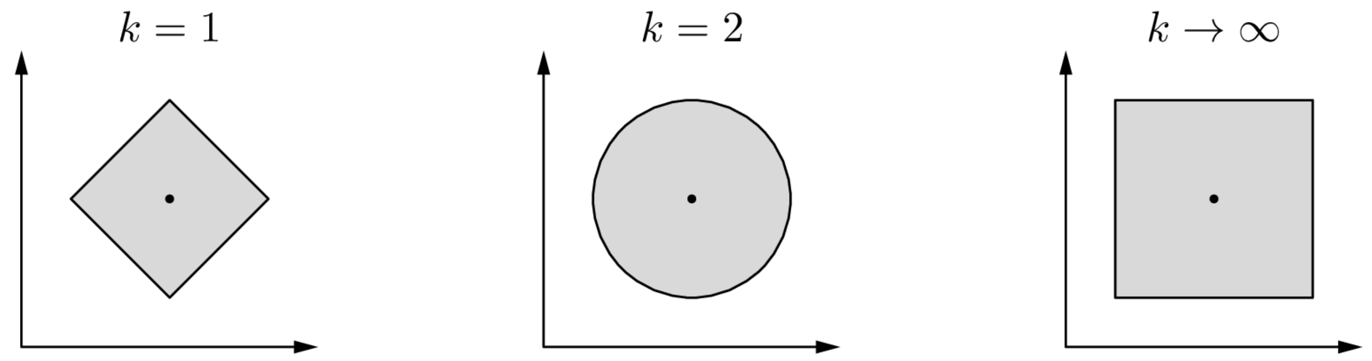
Distance Functions

Illustration of distance functions: Minkowski Family

$$d_k(\vec{x},\vec{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^k\right)^{\frac{1}{k}}$$

Well-known special cases from this family are:

$k = 1:$	Manhattan or city block distance,
$k = 2:$	Euclidean distance,
$k \to \infty:$	maximum distance, that is, $d_\infty(\vec{x},\vec{y}) = \max_{i=1}^{n} |x_i - y_i|$.



https://en.wikipedia.org/wiki/Taxicab_geometry

Radial Basis Function Networks

The network input function of the output neurons is the weighted sum of their inputs:

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

The activation function of each hidden neuron is a so-called **radial function**, that is, a monotonically decreasing function

$$f: \mathbb{R}_0^+ \to [0,1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

The activation function of each output neuron is a linear function, namely

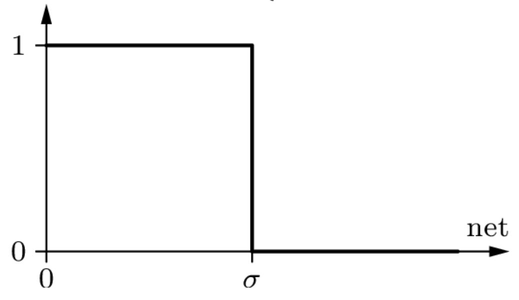
$$f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \text{net}_u - \theta_u.$$

(The linear activation function is important for the initialization.)

Radial Activation Functions

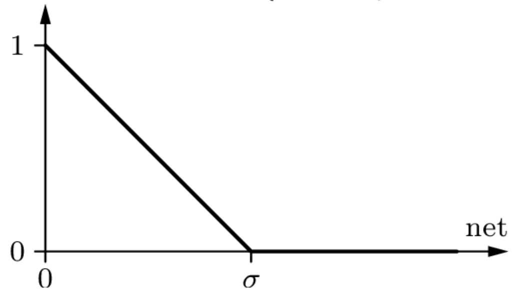
rectangle function:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{if net} > \sigma, \\ 1, & \text{otherwise.} \end{cases}$$



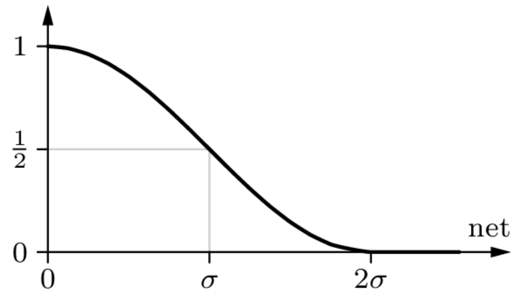
triangle function:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{if net} > \sigma, \\ 1 - \frac{\text{net}}{\sigma}, & \text{otherwise.} \end{cases}$$



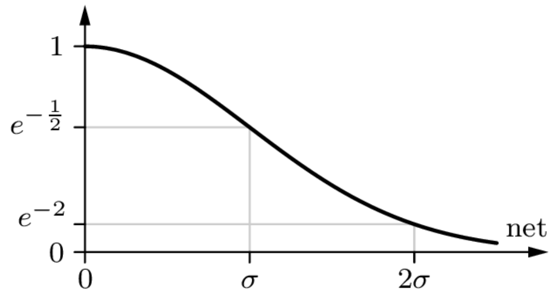
cosine until zero:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{if net} > 2\sigma, \\ \frac{\cos\left(\frac{\pi}{2\sigma}\text{net}\right)+1}{2}, & \text{otherwise.} \end{cases}$$



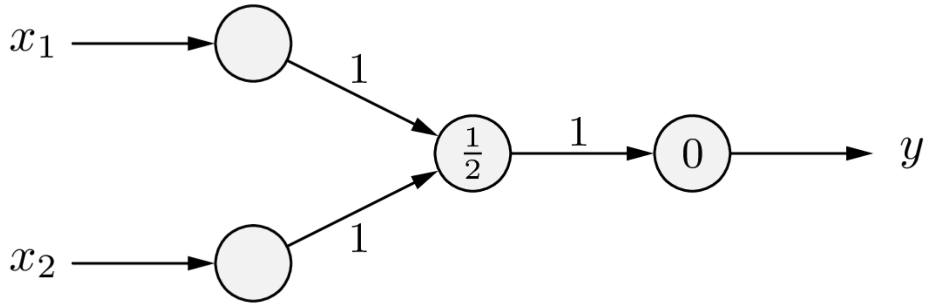
Gaussian function:

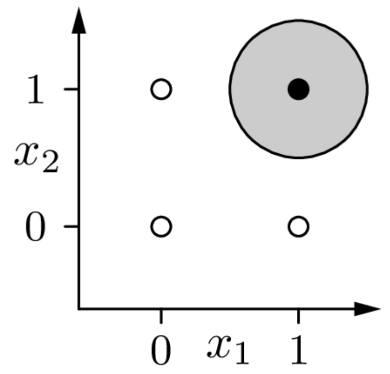
$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$

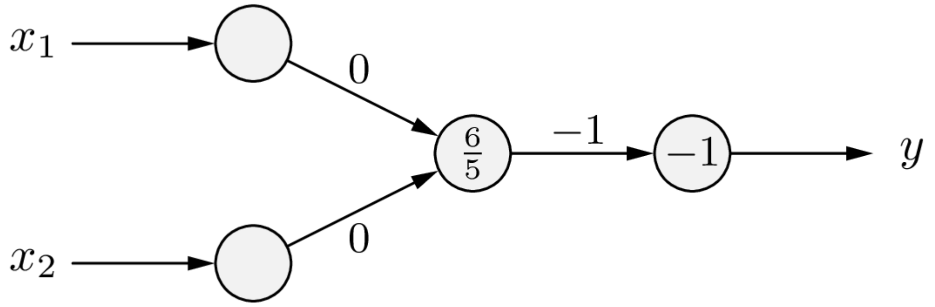


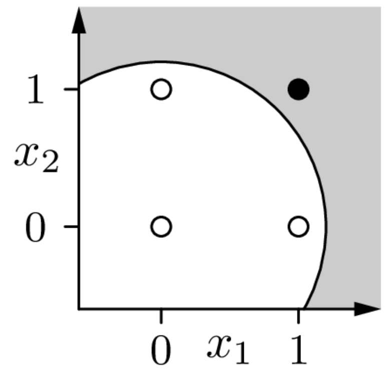
Radial Basis Function Networks: Examples

Radial basis function networks for the conjunction $x_1 \wedge x_2$







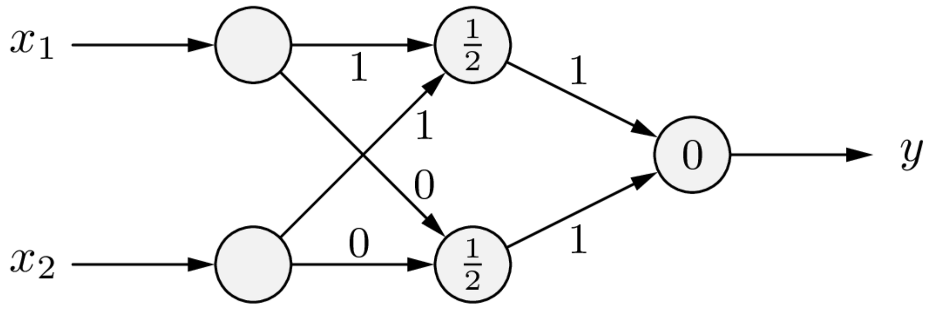


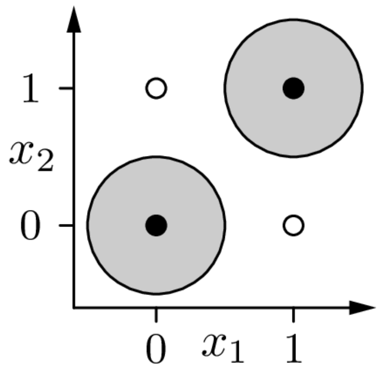
Radial Basis Function Networks: Examples

Radial basis function networks for the biimplication $x_1 \leftrightarrow x_2$

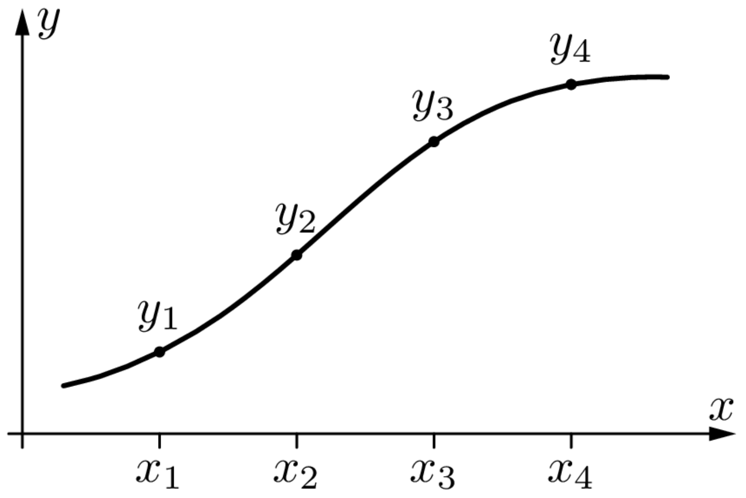
Idea: logical decomposition

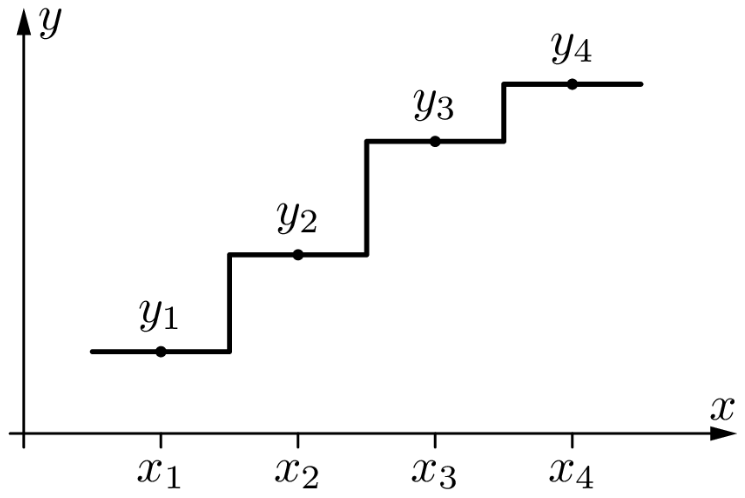
$$x_1 \leftrightarrow x_2 \quad \equiv \quad (x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)$$



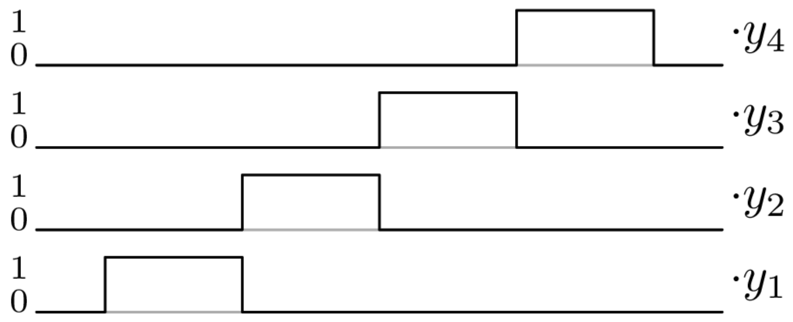


Radial Basis Function Networks: Function Approximation

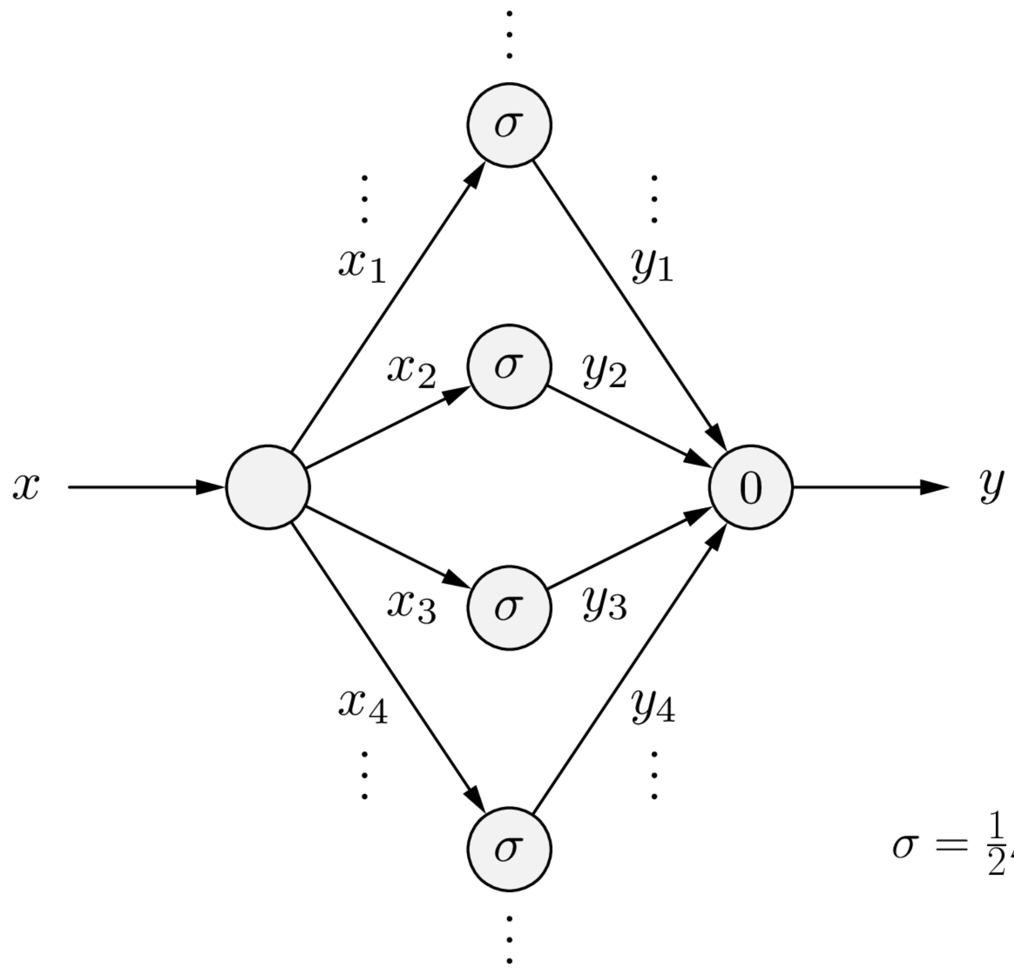




Approximation of a function by rectangular pulses, each of which can be represented by a neuron of an radial basis function network.



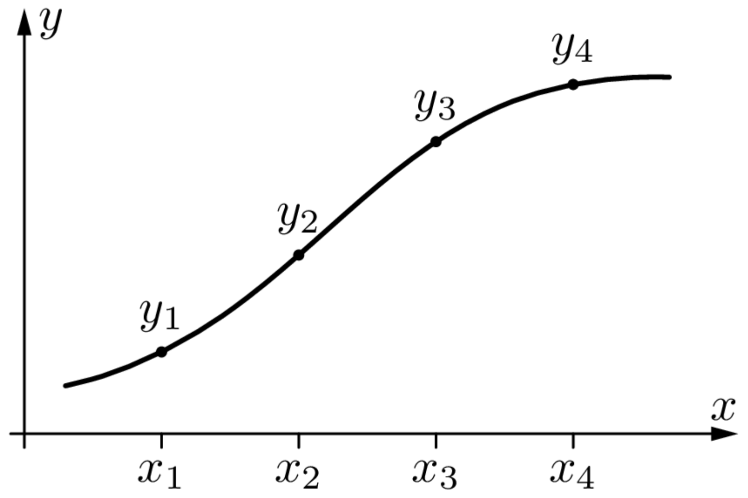
Radial Basis Function Networks: Function Approximation

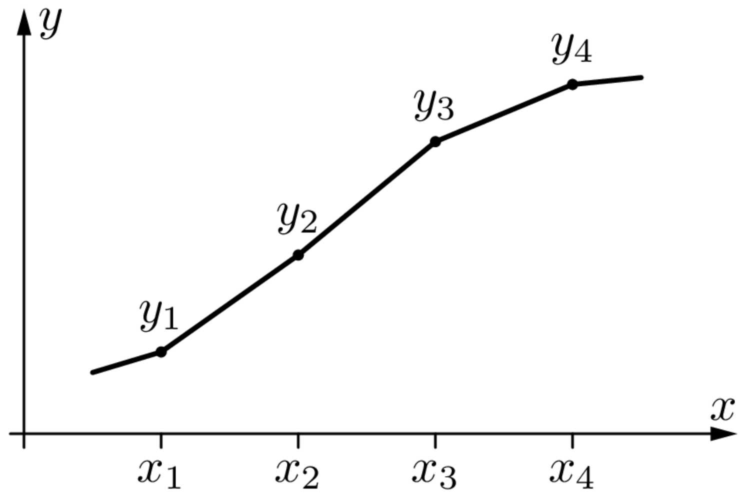


$$\sigma = \tfrac{1}{2}\Delta x = \tfrac{1}{2}(x_{i+1} - x_i)$$

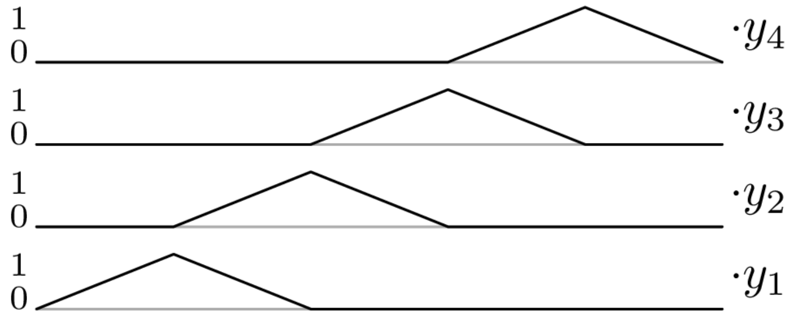
A radial basis function network that computes the step function on the preceding slide and the piecewise linear function on the next slide (depends on activation function).

Radial Basis Function Networks: Function Approximation

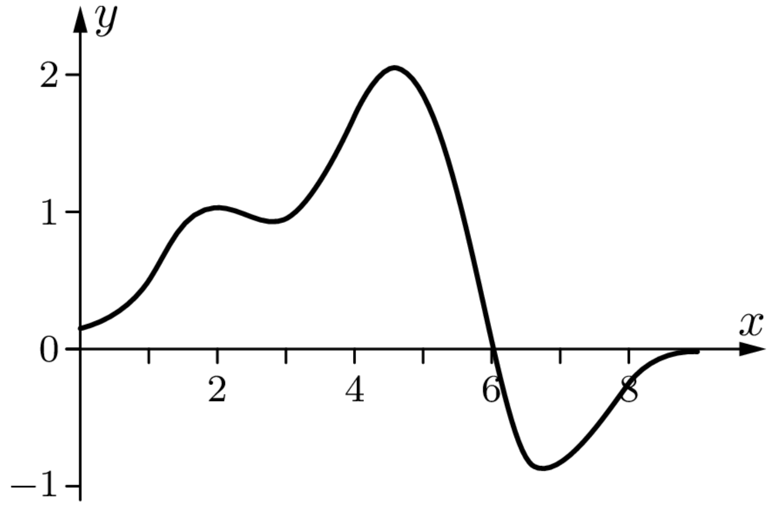


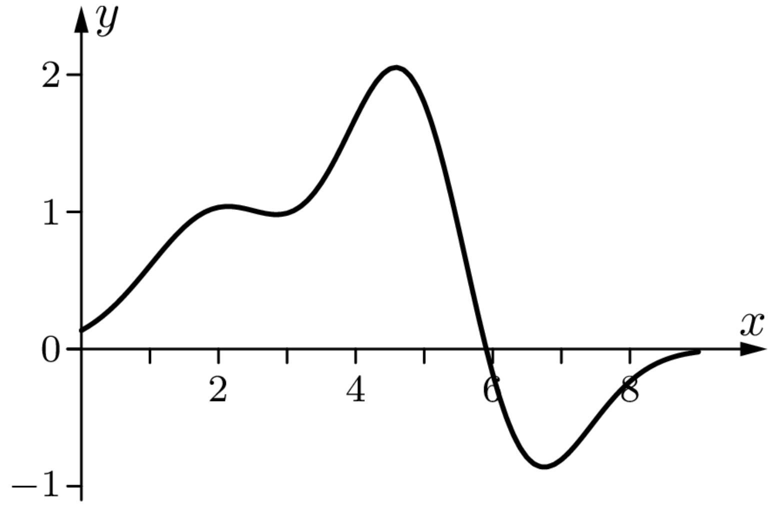


Approximation of a function by triangular pulses, each of which can be represented by a neuron of an radial basis function network.

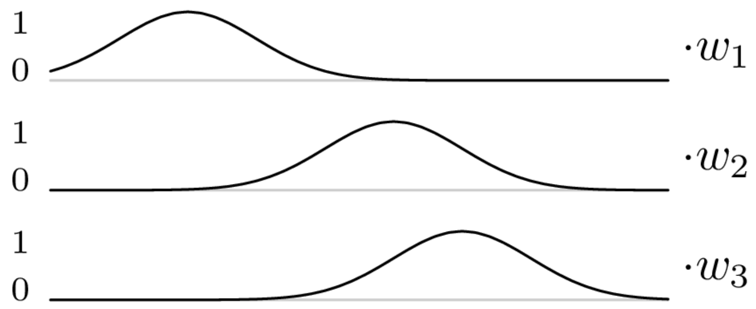


Radial Basis Function Networks: Function Approximation



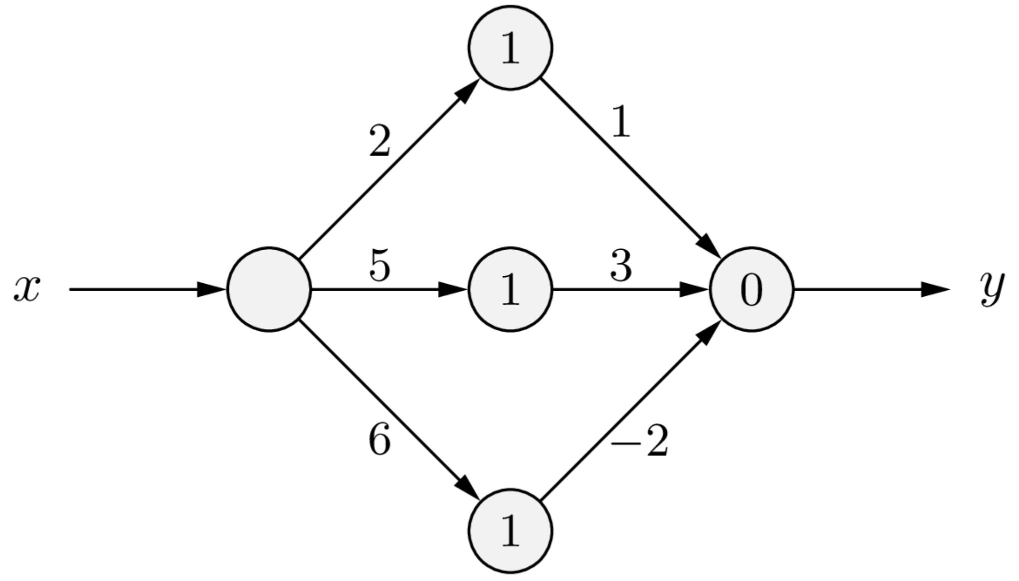


Approximation of a function by Gaussian functions with radius $\sigma = 1$. It is $w_1 = 1$, $w_2 = 3$ and $w_3 = -2$.



Radial Basis Function Networks: Function Approximation

Radial basis function network for a sum of three Gaussian functions



- The weights of the connections from the input neuron to the hidden neurons determine the locations of the Gaussian functions.
- The weights of the connections from the hidden neurons to the output neuron determine the height/direction (upward or downward) of the Gaussian functions.

Self-Organizing Maps

Self-Organizing Maps

A **self-organizing map** or **Kohonen feature map** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions

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

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

The network input function of each output neuron is a **distance function** of input and weight vector. The activation function of each output neuron is a **radial function**, that is, a monotonically decreasing function

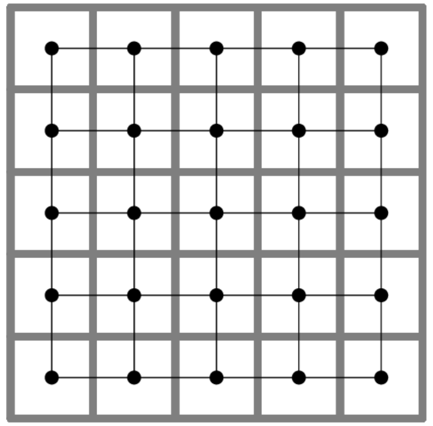
$$f : \mathbb{R}_0^+ \to [0,1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

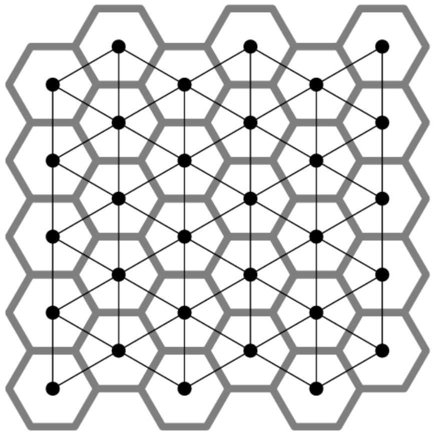
The output function of each output neuron is the identity.
The output is often discretized according to the "**winner takes all**" principle.
On the output neurons a **neighborhood relationship** is defined:

$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \to \mathbb{R}_0^+ .$$

Self-Organizing Maps: Neighborhood

Neighborhood of the output neurons: neurons form a grid


quadratic grid

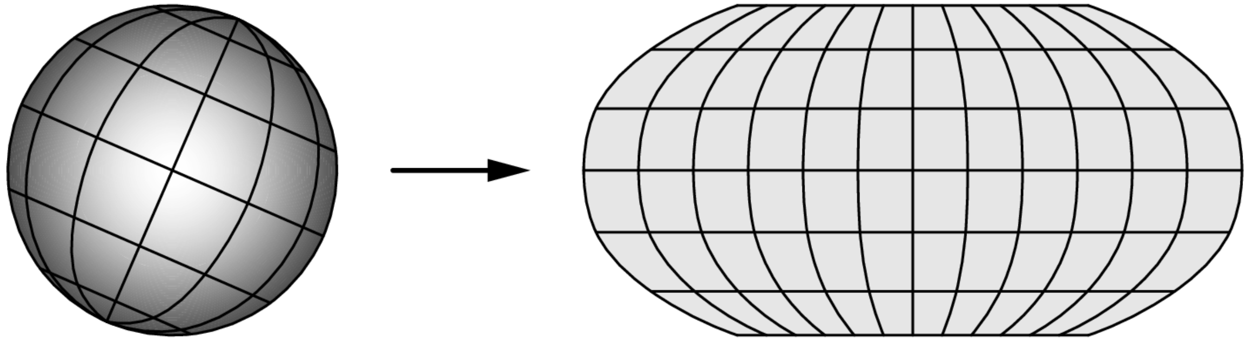

hexagonal grid

- Thin black lines: Indicate nearest neighbors of a neuron.
- Thick gray lines: Indicate regions assigned to a neuron for visualization.
- Usually two-dimensional grids are used to be able to draw the map easily.

Topology Preserving Mapping

Images of points close to each other in the original space should be close to each other in the image space.

Example: **Robinson projection** of the surface of a sphere
(maps from 3 dimensions to 2 dimensions)



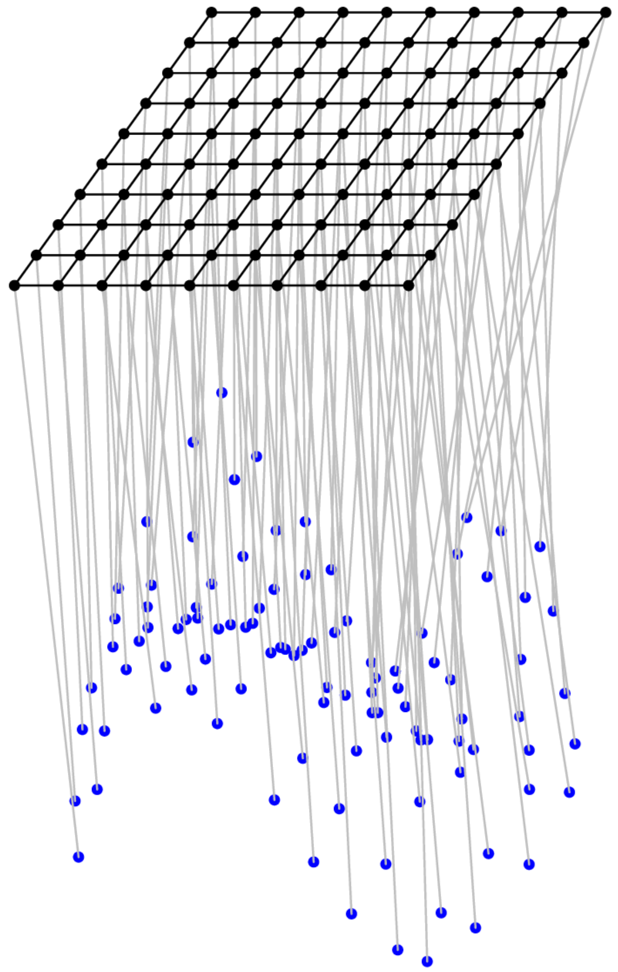
- Robinson projection is/was frequently used for world maps.
- The topology is preserved, although distances, angles, areas may be distorted.

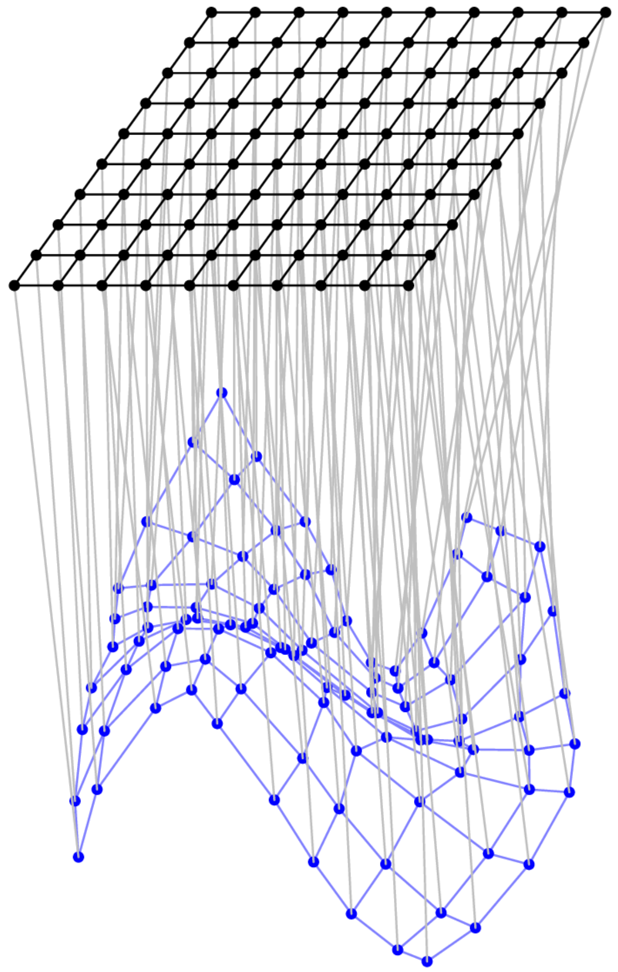
Self-Organizing Maps: Topology Preserving Mapping

neuron space/grid
usually 2-dimensional
quadratic or
hexagonal grid

input/data space
usually high-dim.
(here: only 3-dim.)
blue: ref. vectors

Connections may
be drawn between
vectors corresponding
to adjacent neurons.





Self-Organizing Maps: Neighborhood

Find topology preserving mapping by respecting the neighborhood

Reference vector update rule:

$$\vec{r}_u^{\,(\text{new})} = \vec{r}_u^{\,(\text{old})} + \eta(t) \cdot f_{\text{nb}}(d_{\text{neurons}}(u, u_*), \varrho(t)) \cdot (\vec{x} - \vec{r}_u^{\,(\text{old})}),$$

- u_* is the winner neuron (reference vector closest to data point).
- The neighborhood function f_{nb} is a radial function.

Time dependent learning rate

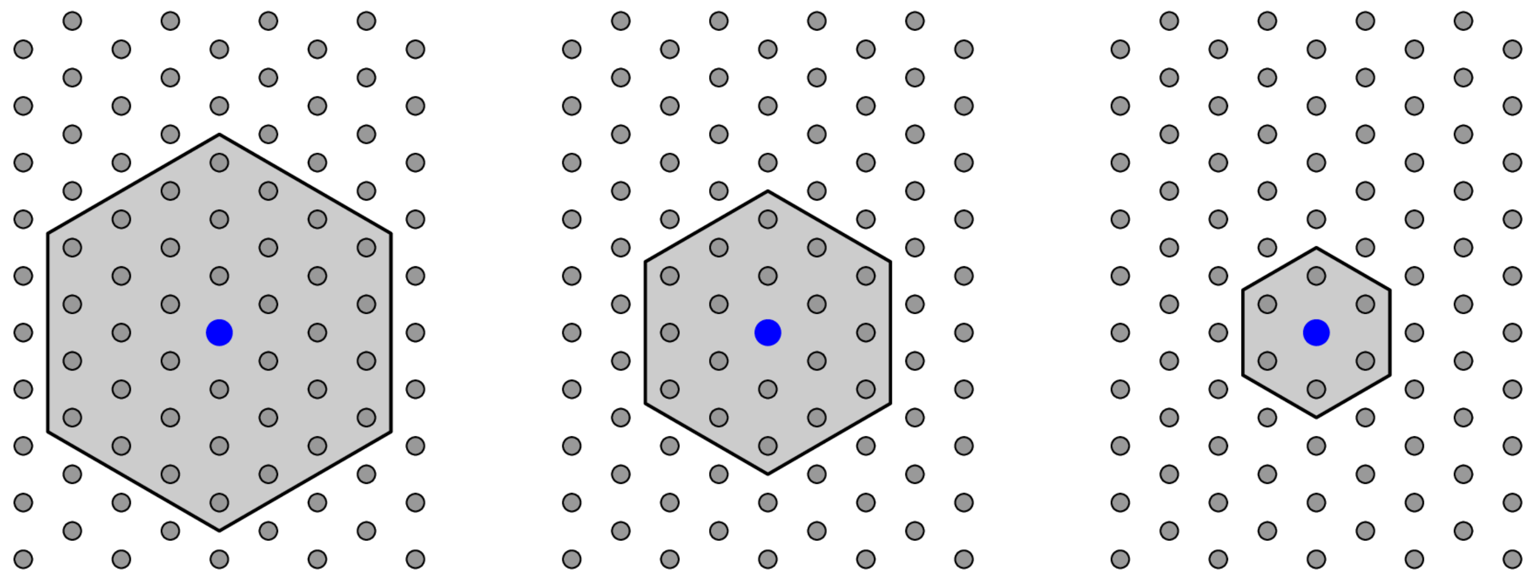
$$\eta(t) = \eta_0 \alpha_\eta^t, \quad 0 < \alpha_\eta < 1, \qquad \text{or} \qquad \eta(t) = \eta_0 t^{\kappa_\eta}, \quad \kappa_\eta < 0.$$

Time dependent neighborhood radius

$$\varrho(t) = \varrho_0 \alpha_\varrho^t, \quad 0 < \alpha_\varrho < 1, \qquad \text{or} \qquad \varrho(t) = \varrho_0 t^{\kappa_\varrho}, \quad \kappa_\varrho < 0.$$

Self-Organizing Maps: Neighborhood

The neighborhood size is reduced over time: (here: step function)



Note that a neighborhood function that is not a step function has a "soft" border and thus allows for a smooth reduction of the neighborhood size (larger changes of the reference vectors are restricted more and more to the close neighborhood).

SOM summary

Unsupervised Learning Algorithm

- Used for clustering and dimensionality reduction by mapping high-dimensional data onto a lower-dimensional grid.

Grid of Output Neurons with Weights

- Each neuron has a weight vector representing a prototype of the input data.

Best Matching Unit (BMU) Selection

- For each input, the neuron with the closest weight (smallest Euclidean distance) is chosen as the BMU.

Neighborhood-Based Weight Update

- The BMU and its neighboring neurons adjust their weights to become more like the input, preserving topological relationships.

Sigma (σ) Controls Influence Spread

- Larger σ affects more neurons; smaller σ focuses on local fine-tuning.

Applications: Clustering & Visualization

- Used in image segmentation, anomaly detection, customer segmentation, and data visualization.