Lecture 3

- ❖ **Multi-layer Perceptrons**
  - ✓ **MLP definition and examples**
  - ✓ **Non-linar activation functions**
  - ✓ **Universal approximation theorem**

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

- ANNs can be represented by a special data structure: **directed graph**
- Three types of neurons: **input, hidden, output**
- **Feed forward networks**: no cycles

  Recurrent networks: has cycle(s)
- Two **phases** of ANN calculation: (i) **input phase**, (ii) **work phase**
- **Ordering** of **calculations**:
  - Feed forward networks: **topological**
  - Recurrent networks: final output may **depend** on the order

Summary of Key Takeaways from Last Week

- Scale types of attributes: **nominal, ordinal, metric**
- How to use nominal and ordinal attributes are handled for ANNs: **1-hot encoding**
- How to handle unequal scaling of attribute data: **z-score normalization**.
- Error calculation for ANNs:
  - Regression tasks: **squared deviation**
  - Classification tasks: **cross entropy**

Multi-layer Perceptrons (MLPs)

Multi-layer Perceptrons

An **r-layer perceptron** 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) $U_\text{hidden} = U_\text{hidden}^{(1)} \cup \cdots \cup U_\text{hidden}^{(r-2)}$,

$\forall 1 \leq i < j \leq r-2: \quad U_\text{hidden}^{(i)} \cap U_\text{hidden}^{(j)} = \emptyset$,

(iii) $C \subseteq \left(U_\text{in} \times U_\text{hidden}^{(1)}\right) \cup \left(\bigcup_{i=1}^{r-3} U_\text{hidden}^{(i)} \times U_\text{hidden}^{(i+1)}\right) \cup \left(U_\text{hidden}^{(r-2)} \times U_\text{out}\right)$

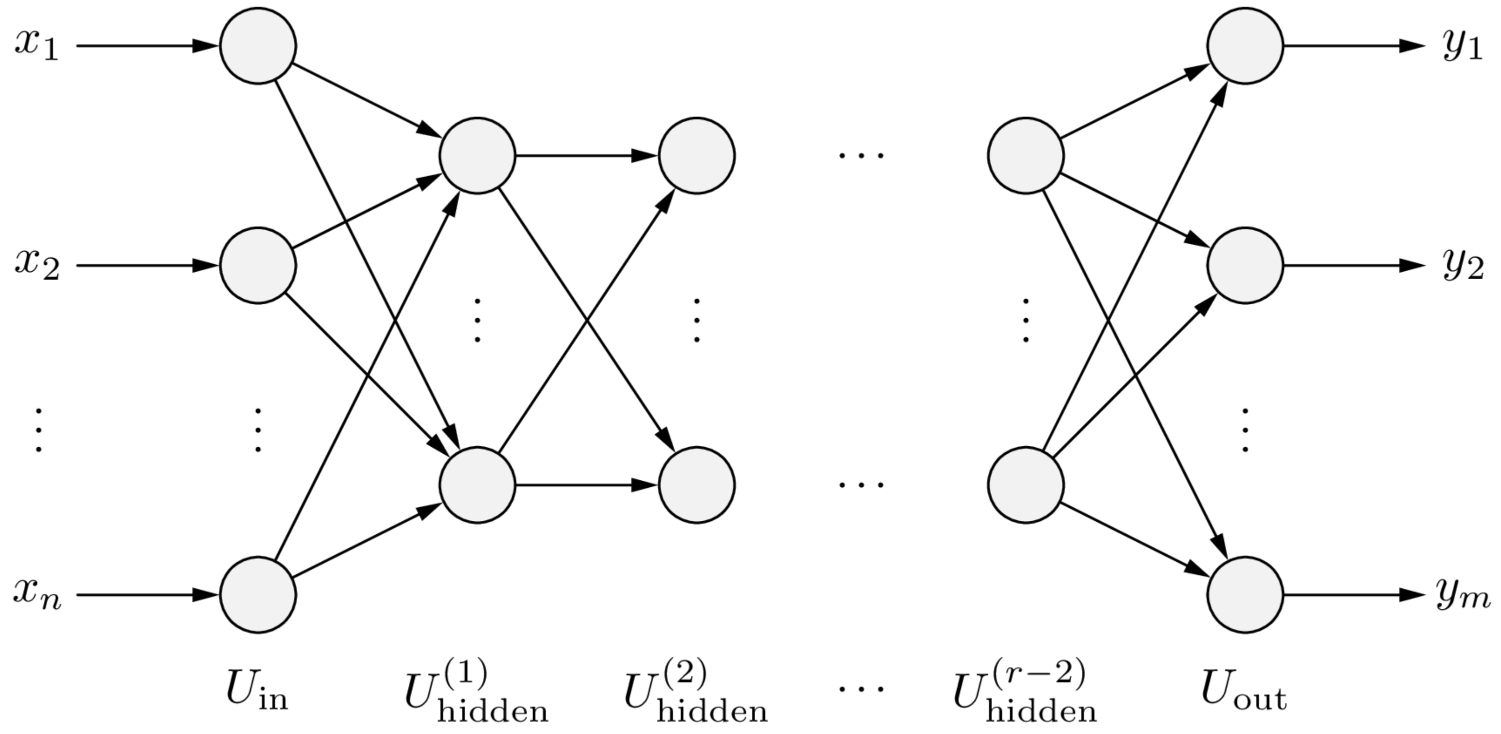
or, if there are no hidden neurons ($r = 2, U_\text{hidden} = \emptyset$),

$C \subseteq U_\text{in} \times U_\text{out}$.

- Feed-forward network with strictly layered structure.

Multi-layer Perceptrons

General structure of a multi-layer perceptron



x_1
x_2
x_n
y_1
y_2
y_m
U_{in}
$U_{\text{hidden}}^{(1)}$
$U_{\text{hidden}}^{(2)}$
$\cdots$
$U_{\text{hidden}}^{(r-2)}$
U_{out}

Multi-layer Perceptrons

‘T’ is a transpose operator:
Turns a row vector into a column vector.

- The network input function of each hidden neuron and of each output neuron is the **weighted sum** of its inputs, that is,

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

Weight from $v \to u$.

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

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

- The activation function of each output neuron is either also a sigmoid function or a **linear function**, that is,

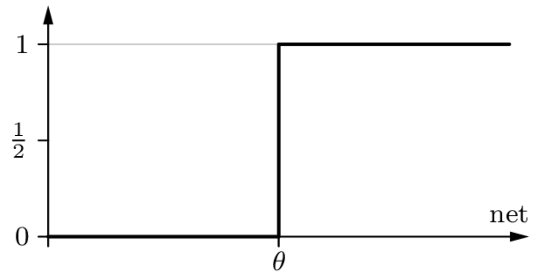
$$f_{\text{act}}(\text{net}, \theta) = \alpha \, \text{net} - \theta.$$

Only the step function is a neurobiologically plausible activation function.

Sigmoid Activation Functions

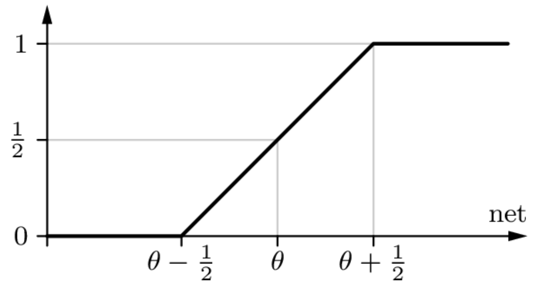
step function:

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



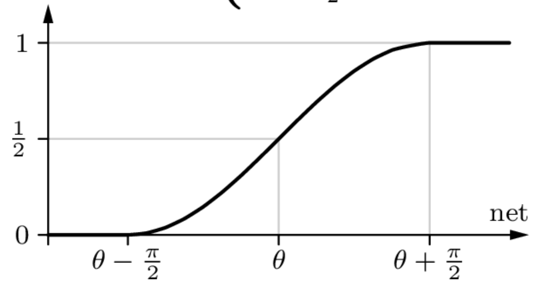
semi-linear function:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if net} > \theta + \frac{1}{2}, \\ 0, & \text{if net} < \theta - \frac{1}{2}, \\ (\text{net} - \theta) + \frac{1}{2}, & \text{otherwise.} \end{cases}$$



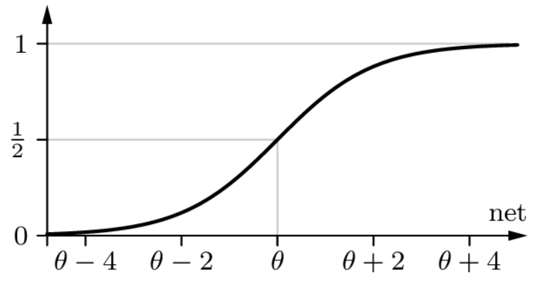
sine until saturation:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if net} > \theta + \frac{\pi}{2}, \\ 0, & \text{if net} < \theta - \frac{\pi}{2}, \\ \frac{\sin(\text{net} - \theta) + 1}{2}, & \text{otherwise.} \end{cases}$$



logistic function:

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$

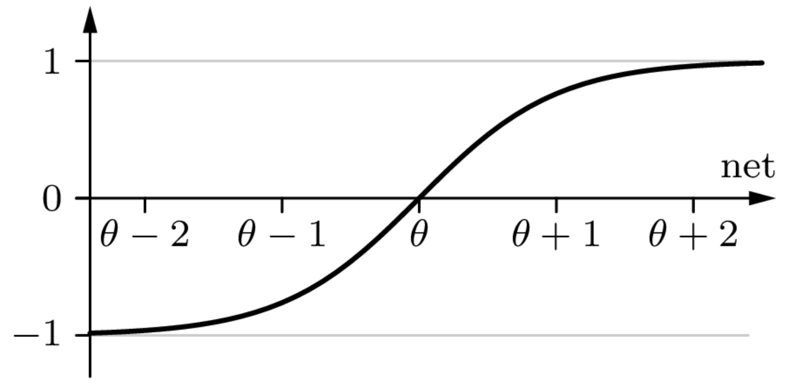


Sigmoid Activation Functions

- All sigmoid functions on the previous slide are **unipolar**, that is, they range from 0 to 1.
- Sometimes **bipolar** sigmoid functions are used (ranging from -1 to $+1$), like the hyperbolic tangent (*tangens hyperbolicus*).

hyperbolic tangent:

$$\begin{aligned} f_{\text{act}}(\text{net}, \theta) &= \tanh(\text{net} - \theta) \\ &= \frac{e^{(\text{net}-\theta)} - e^{-(\text{net}-\theta)}}{e^{(\text{net}-\theta)} + e^{-(\text{net}-\theta)}} \\ &= \frac{1 - e^{-2(\text{net}-\theta)}}{1 + e^{-2(\text{net}-\theta)}} \\ &= \frac{2}{1 + e^{-2(\text{net}-\theta)}} - 1 \end{aligned}$$

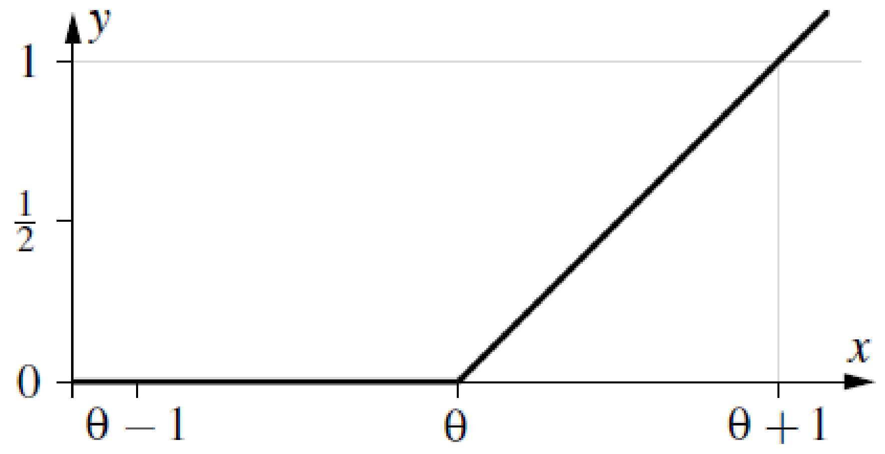


ReLU activation function

- Rectified Linear Unit (ReLU) is a popular activation function in deep learning.

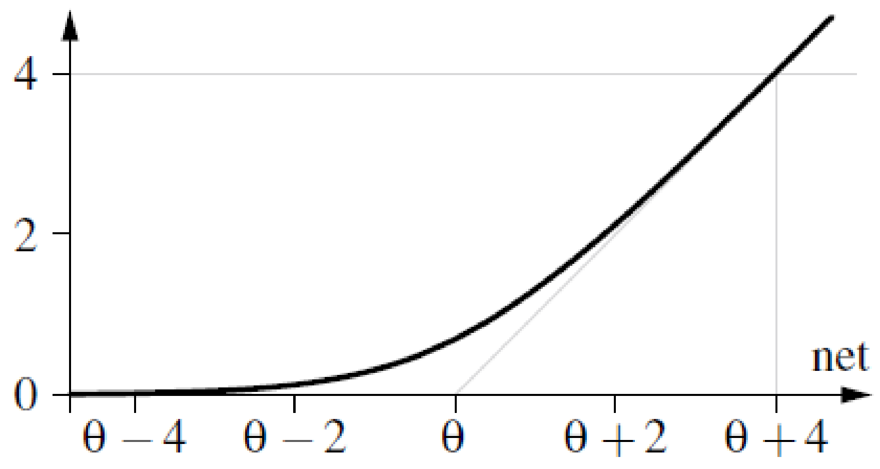
rectified maximum/ramp function:

$f_{\text{act}}(\text{net}, \theta) = \max\{0, \text{net} - \theta\}$



softplus function:

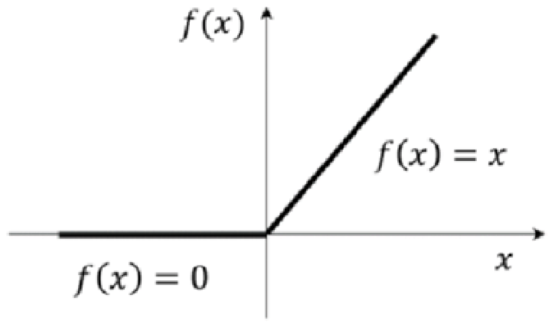
$f_{\text{act}}(\text{net}, \theta) = \ln(1 + e^{\text{net} - \theta})$



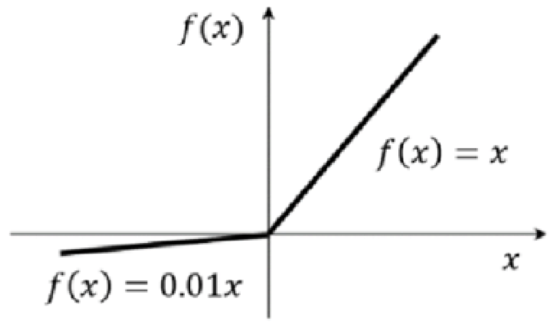
Leaky ReLU

- Solution to **'dying ReLU' problem**: when a ReLU neuron always outputs zero, effectively becoming inactive and never updating its weights during training.
- Leaky ReLU **adds a small slope for negative values** in the activation function, ensuring a small gradient for negative inputs and preventing complete inactivity.

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} \text{net} - \theta & \text{if net} \geq \theta, \\ \nu(\text{net} - \theta) & \text{otherwise,} \end{cases}$$



ReLU activation function



LeakyReLU activation function

Multi-layer Perceptrons: Weight Matrices

Let $U_1 = \{v_1, \ldots, v_m\}$ and $U_2 = \{u_1, \ldots, u_n\}$ be the neurons of two consecutive layers of a multi-layer perceptron.

Their connection weights are represented by an $n \times m$ matrix

$$\mathbf{W} = \begin{pmatrix} w_{u_1 v_1} & w_{u_1 v_2} & \ldots & w_{u_1 v_m} \\ w_{u_2 v_1} & w_{u_2 v_2} & \ldots & w_{u_2 v_m} \\ \vdots & \vdots & & \vdots \\ w_{u_n v_1} & w_{u_n v_2} & \ldots & w_{u_n v_m} \end{pmatrix},$$

This weight matrix is only for W: $U_1 \rightarrow U_2$

To represent the whole network, we need separate weigh matrices for each consecutive layers.

where $w_{u_i v_j} = 0$ if there is no connection from neuron v_j to neuron u_i.

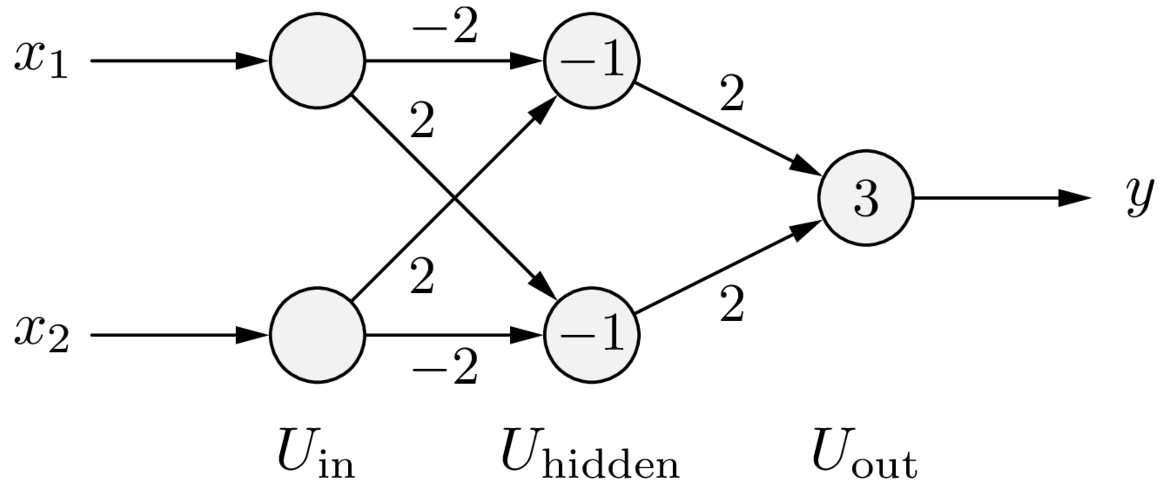
Advantage: The computation of the network input can be written as

$$\vec{\text{net}}_{U_2} = \mathbf{W} \cdot \vec{\text{in}}_{U_2} = \mathbf{W} \cdot \vec{\text{out}}_{U_1}$$

where $\vec{\text{net}}_{U_2} = (\text{net}_{u_1}, \ldots, \text{net}_{u_n})^\top$ and $\vec{\text{in}}_{U_2} = \vec{\text{out}}_{U_1} = (\text{out}_{v_1}, \ldots, \text{out}_{v_m})^\top$.

Multi-layer Perceptrons: Biimplication

Solving the biimplication problem with a multi-layer perceptron.

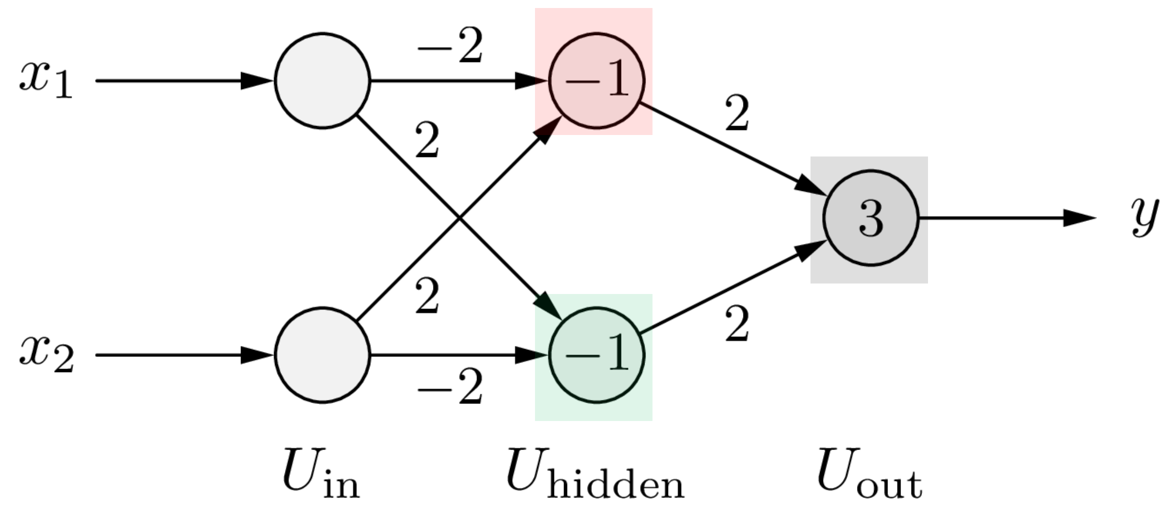


Note the additional input neurons compared to the TLU solution.

$$\mathbf{W}_1 = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \qquad \text{and} \qquad \mathbf{W}_2 = \begin{pmatrix} 2 & 2 \end{pmatrix}$$

Multi-layer Perceptrons: Biimplication

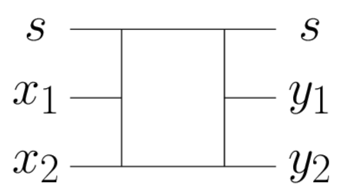
Solving the biimplication problem with a multi-layer perceptron.

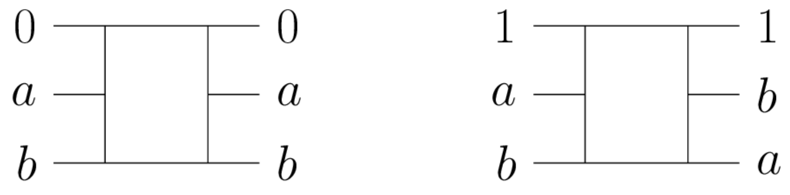


Note the additional input neurons compared to the TLU solution.

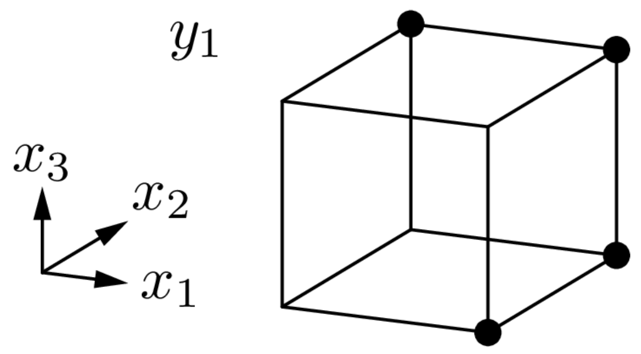
$$\mathbf{W}_1 = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \quad \text{and} \quad \mathbf{W}_2 = \begin{pmatrix} 2 & 2 \end{pmatrix}$$

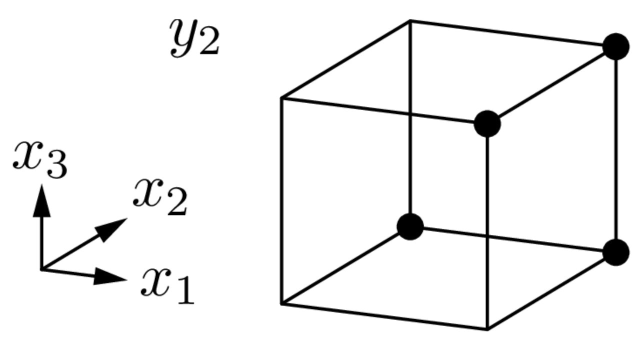
Multi-layer Perceptrons: Fredkin Gate





s	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1
x_2	0	1	0	1	0	1	0	1
y_1	0	0	1	1	0	1	0	1
y_2	0	1	0	1	0	0	1	1





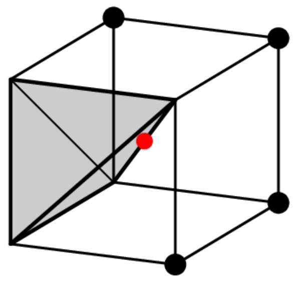
Multi-layer Perceptrons: Fredkin Gate

- The **Fredkin gate** (after Edward Fredkin *1934) or **controlled swap gate** (CSWAP) is a computational circuit that is used in **conservative logic** and **reversible computing**.
- Conservative logic is a model of computation that explicitly reflects the physical properties of computation, like the reversibility of the dynamical laws and the conservation of certain quantities (e.g. energy) [Fredkin and Toffoli 1982].
- The Fredkin gate is **reversible** in the sense that the inputs can be computed as functions of the outputs in the same way in which the outputs can be computed as functions of the inputs (no information loss, no entropy gain).
- The Fredkin gate is **universal** in the sense that all Boolean functions can be computed using only Fredkin gates.
- Note that both outputs, y_1 and y_2 are **not linearly separable**, because the convex hull of the points mapped to 0 and the convex hull of the points mapped to 1 share the point in the center of the cube.

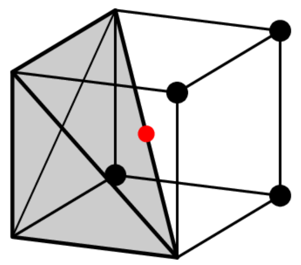
Reminder: Convex Hull Theorem

Theorem: Two sets of points in a Euclidean space are **linearly separable** if and only if their convex hulls are disjoint (that is, have no point in common).

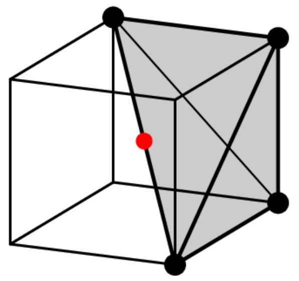
Both outputs y_1 and y_2 of a Fredkin gate are not linearly separable:



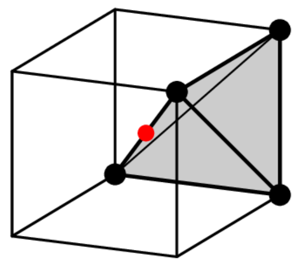
Convex hull of points with $y_1 = 0$



Convex hull of points with $y_2 = 0$

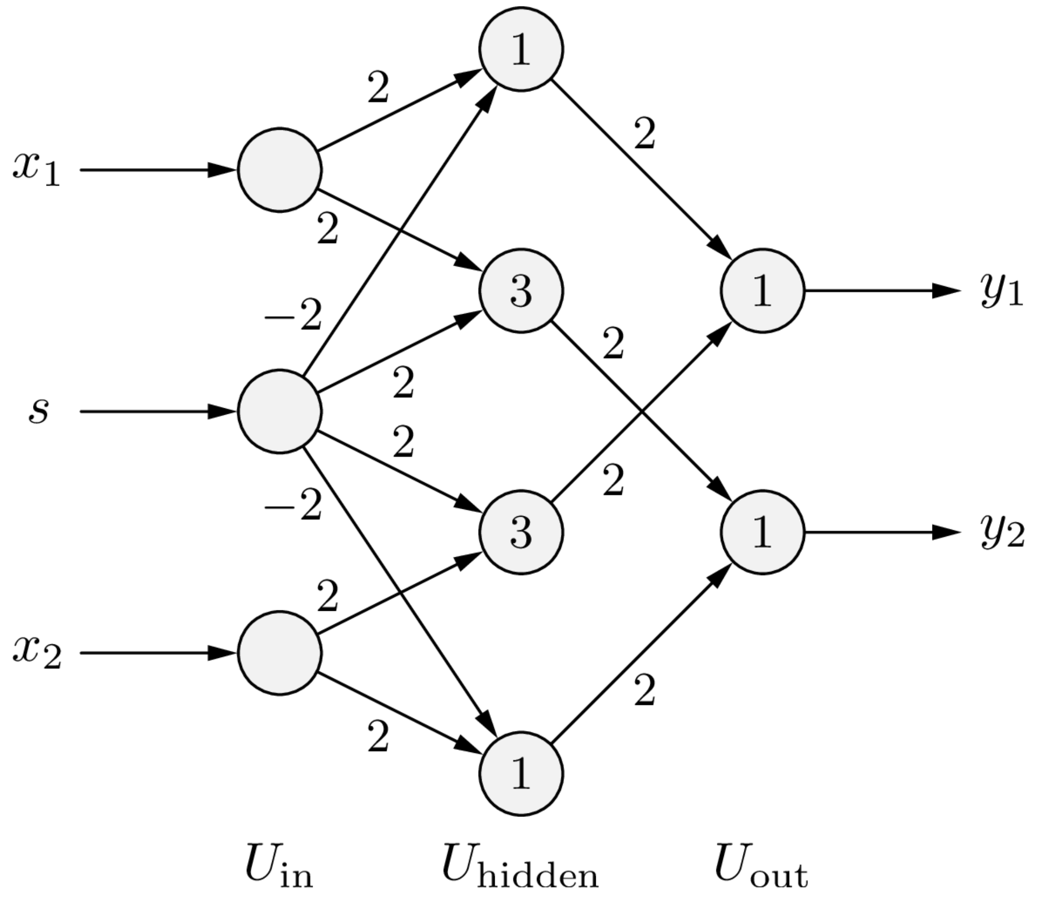


Convex hull of points with $y_1 = 1$



Convex hull of points with $y_2 = 1$

Multi-layer Perceptrons: Fredkin Gate



Connections (weights) from the input layer to the hidden layer. 0 means no connection.

$$\mathbf{W}_1 = \begin{pmatrix} 2 & -2 & 0 \\ 2 & 2 & 0 \\ 0 & 2 & 2 \\ 0 & -2 & 2 \end{pmatrix}$$

Connections from the hidden layer to the output layer.

$$\mathbf{W}_2 = \begin{pmatrix} 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{pmatrix}$$

Why Non-linear Activation Functions?

With weight matrices we have for two consecutive layers U_1 and U_2

$$\vec{\mathrm{net}}_{U_2} = \mathbf{W} \cdot \vec{\mathrm{in}}_{U_2} = \mathbf{W} \cdot \vec{\mathrm{out}}_{U_1}.$$

If the activation functions are linear, that is,

$$f_{\mathrm{act}}(\mathrm{net}, \theta) = \alpha\,\mathrm{net} - \theta.$$

the activations of the neurons in the layer U_2 can be computed as

$$\vec{\mathrm{act}}_{U_2} = \mathbf{D}_{\mathrm{act}} \cdot \vec{\mathrm{net}}_{U_2} - \vec{\theta},$$

where

- $\vec{\mathrm{act}}_{U_2} = (\mathrm{act}_{u_1}, \ldots, \mathrm{act}_{u_n})^\top$ is the activation vector,
- $\mathbf{D}_{\mathrm{act}}$ is an $n \times n$ diagonal matrix of the factors α_{u_i}, $i = 1, \ldots, n$, and
- $\vec{\theta} = (\theta_{u_1}, \ldots, \theta_{u_n})^\top$ is a bias vector.

Why Non-linear Activation Functions?

If the output function is also linear, it is analogously

$$\vec{\mathrm{out}}_{U_2} = \mathbf{D}_{\mathrm{out}} \cdot \vec{\mathrm{act}}_{U_2} - \vec{\xi},$$

where

- $\vec{\mathrm{out}}_{U_2} = (\mathrm{out}_{u_1}, \ldots, \mathrm{out}_{u_n})^\top$ is the output vector,
- $\mathbf{D}_{\mathrm{out}}$ is again an $n \times n$ diagonal matrix of factors, and
- $\vec{\xi} = (\xi_{u_1}, \ldots, \xi_{u_n})^\top$ a bias vector.

Combining these computations we get

$$\vec{\mathrm{out}}_{U_2} = \mathbf{D}_{\mathrm{out}} \cdot \left(\mathbf{D}_{\mathrm{act}} \cdot \left(\mathbf{W} \cdot \vec{\mathrm{out}}_{U_1}\right) - \vec{\theta}\right) - \vec{\xi}$$

and thus

$$\vec{\mathrm{out}}_{U_2} = \mathbf{A}_{12} \cdot \vec{\mathrm{out}}_{U_1} + \vec{b}_{12}$$

with an $n \times m$ matrix $\mathbf{A}_{12}$ and an n-dimensional vector $\vec{b}_{12}$.

Why Non-linear Activation Functions?

Therefore we have

$$\vec{\text{out}}_{U_2} = \mathbf{A}_{12} \cdot \vec{\text{out}}_{U_1} + \vec{b}_{12}$$

and

$$\vec{\text{out}}_{U_3} = \mathbf{A}_{23} \cdot \vec{\text{out}}_{U_2} + \vec{b}_{23}$$

for the computations of two consecutive layers U_2 and U_3.

These two computations can be combined into

$$\vec{\text{out}}_{U_3} = \mathbf{A}_{13} \cdot \vec{\text{out}}_{U_1} + \vec{b}_{13},$$

where $\mathbf{A}_{13} = \mathbf{A}_{23} \cdot \mathbf{A}_{12}$ and $\vec{b}_{13} = \mathbf{A}_{23} \cdot \vec{b}_{12} + \vec{b}_{23}$.

Result: With linear activation and output functions any multi-layer perceptron can be reduced to a two-layer perceptron.

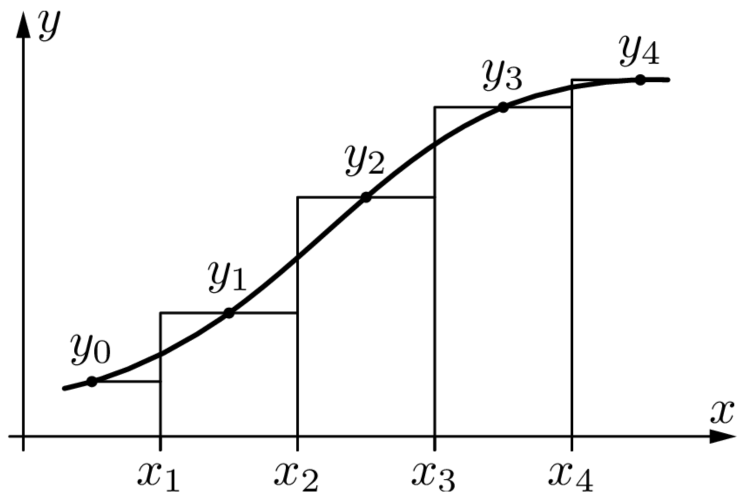
If activation and output functions all linear, MLP can only calculate linear problems. For more complex tasks non-linear activation functions are needed.

Multi-layer Perceptrons: Function Approximation

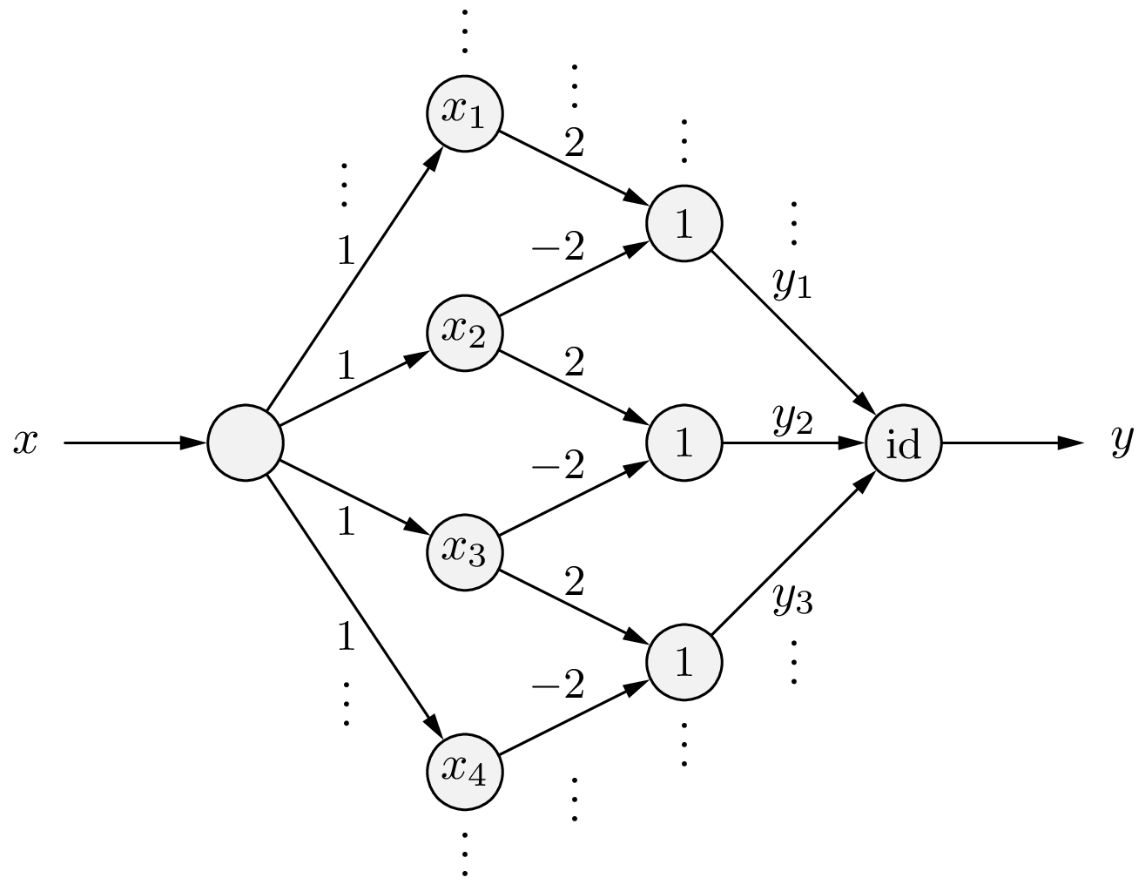
- Up to now: representing and learning Boolean functions $f : \{0,1\}^n \to \{0,1\}$.
- Now: representing and learning real-valued functions $f : \mathbb{R}^n \to \mathbb{R}$.

General idea of function approximation:

- Approximate a given function by a step function.
- Construct a neural network that computes the step function.

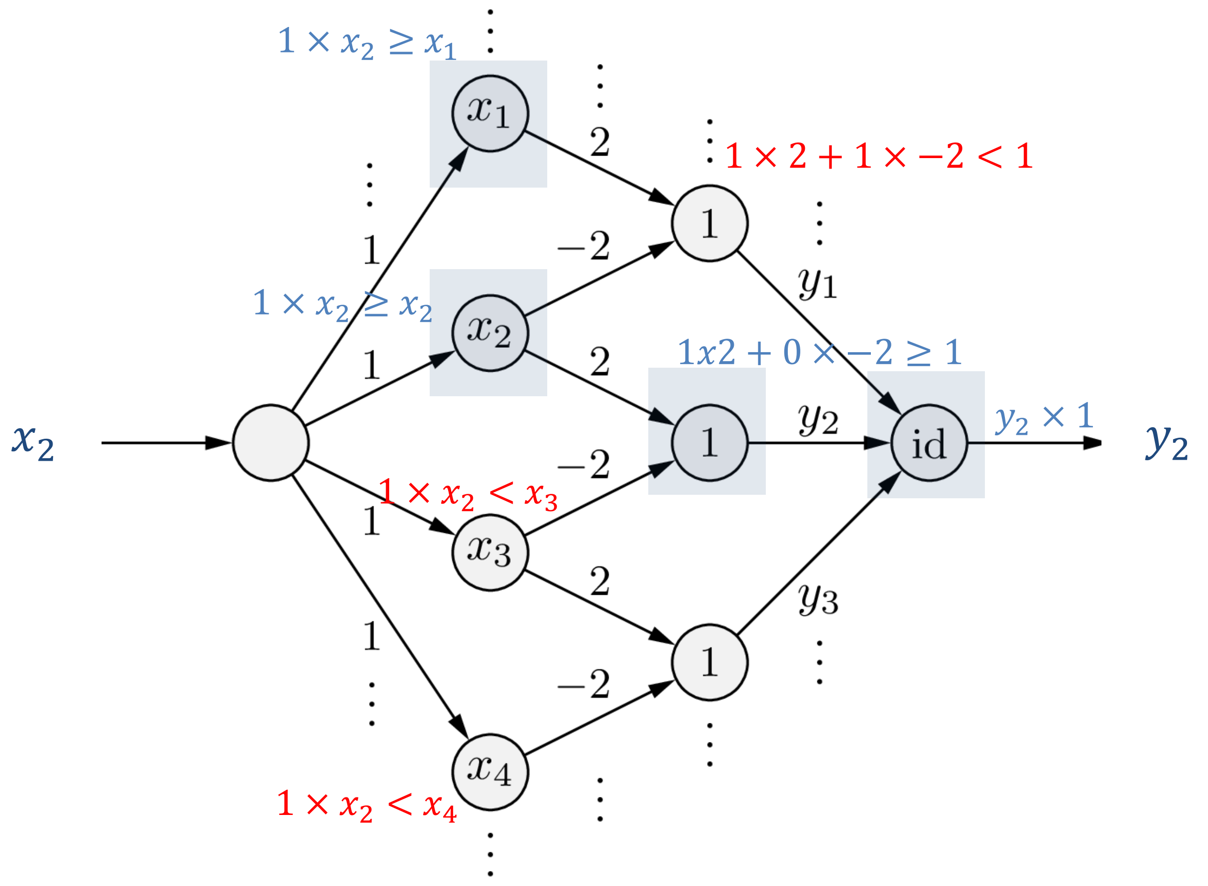


Multi-layer Perceptrons: Function Approximation



A neural network that computes the step function shown on the preceding slide.
According to the input value only one step is active at any time.
The output neuron has the identity as its activation and output functions.

Multi-layer Perceptrons: Function Approximation

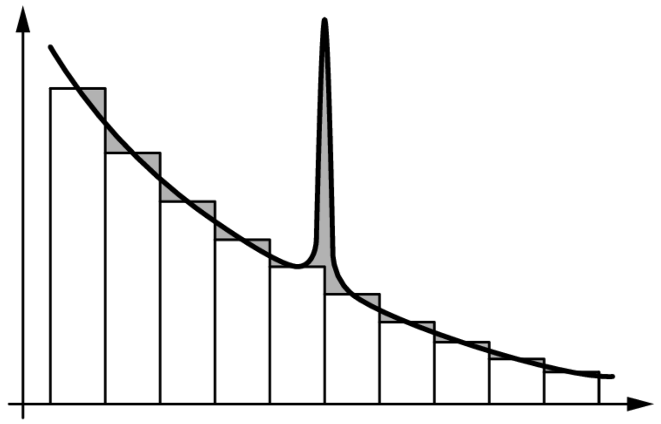


A neural network that computes the step function shown on the preceding slide.
According to the input value only one step is active at any time.
The output neuron has the identity as its activation and output functions.

Multi-layer Perceptrons: Function Approximation

Theorem: Any Riemann-integrable function can be approximated with arbitrary accuracy by a four-layer perceptron.

- But: Error is measured as the **area** between the functions.



- More sophisticated mathematical examination allows a stronger assertion: With a three-layer perceptron any continuous function can be approximated with arbitrary accuracy (error: maximum function value difference).

Multi-layer Perceptrons as Universal Approximators

Universal Approximation Theorem [Hornik 1991]:

Let $\varphi(\cdot)$ be a continuous, bounded and nonconstant function,
let X denote an arbitrary compact subset of $\mathbb{R}^m$, and
let $C(X)$ denote the space of continuous functions on X.

Given any function $f \in C(X)$ and $\varepsilon > 0$, there exists an integer N, real constants $v_i, \theta_i \in \mathbb{R}$ and real vectors $\vec{w}_i \in \mathbb{R}^m$, $i = 1, \ldots, N$, such that we may define

$$F(\vec{x}) = \sum_{i=1}^{N} v_i \, \varphi\Big(\vec{w}_i^\top \vec{x} - \theta_i\Big)$$

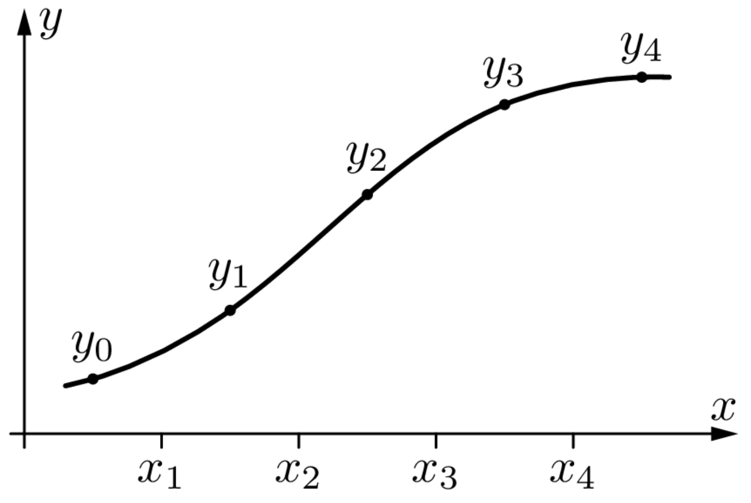
as an approximate realization of the function f where f is independent of φ. That is,

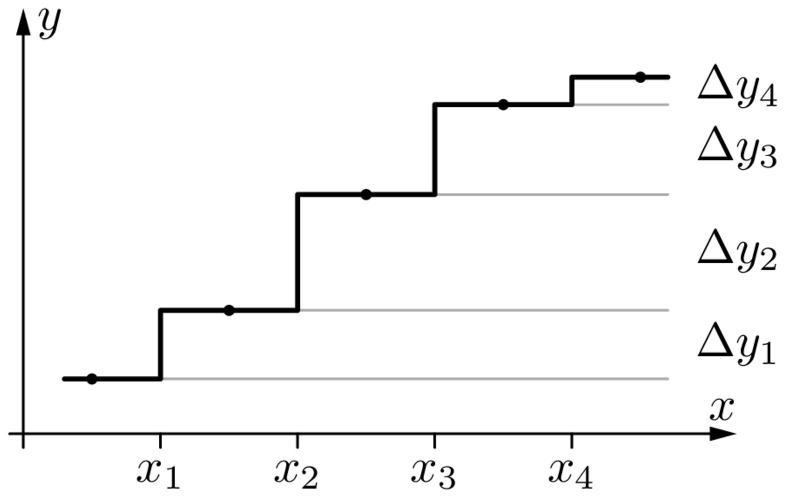
$$|F(\vec{x}) - f(\vec{x})| < \varepsilon$$

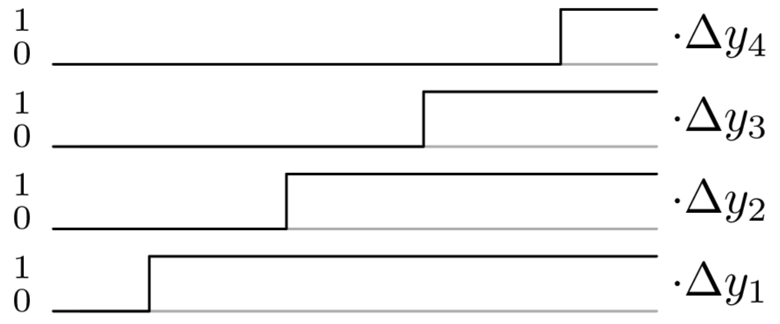
for all $\vec{x} \in X$. In other words, functions of the form $F(\vec{x})$ are dense in $C(X)$.

Note that it is *not* the shape of the activation function, but the layered structure of the feedforward network that renders multi-layer perceptrons universal approximators.

Multi-layer Perceptrons: Function Approximation

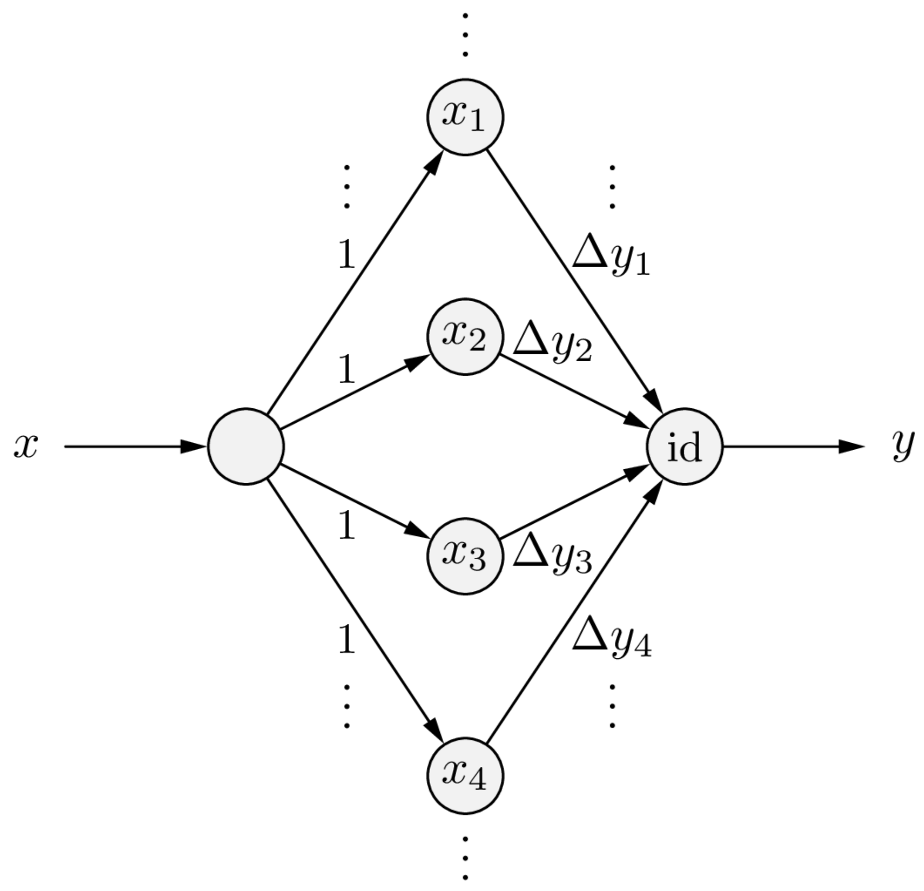






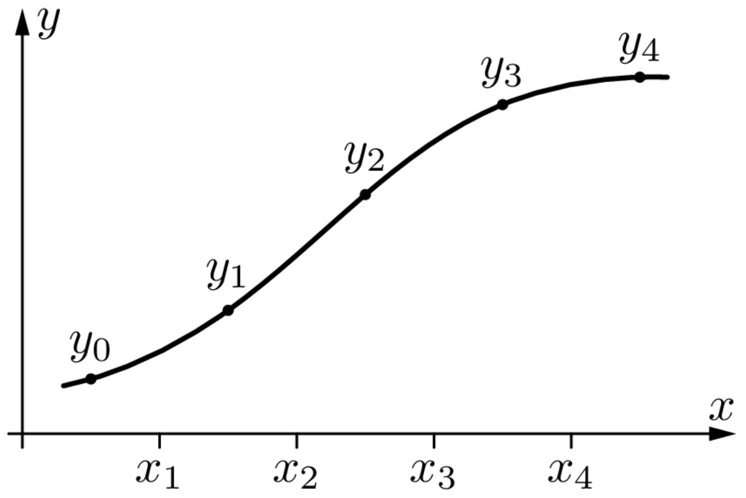
By using relative step heights
one layer can be saved.

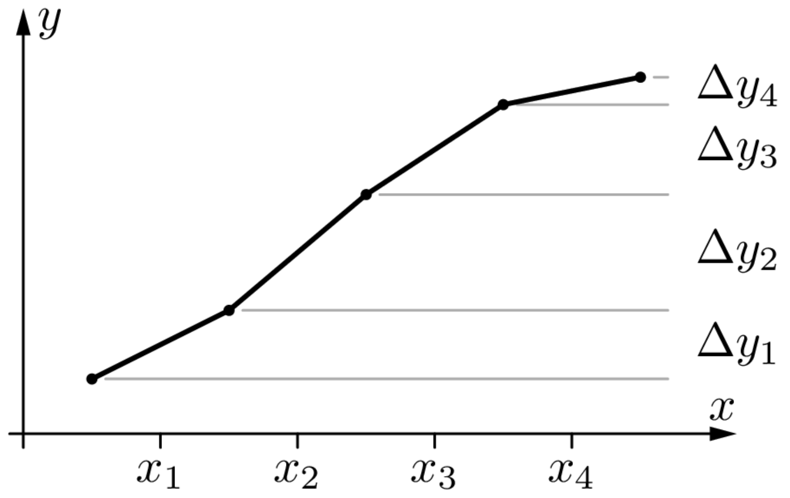
Multi-layer Perceptrons: Function Approximation



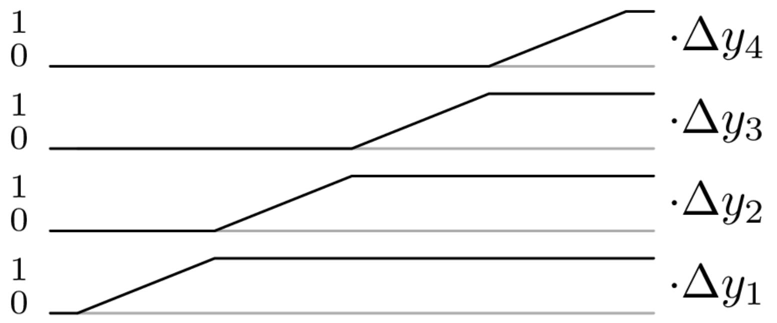
A neural network that computes the step function shown on the preceding slide.
The output neuron has the identity as its activation and output functions.

Multi-layer Perceptrons: Function Approximation

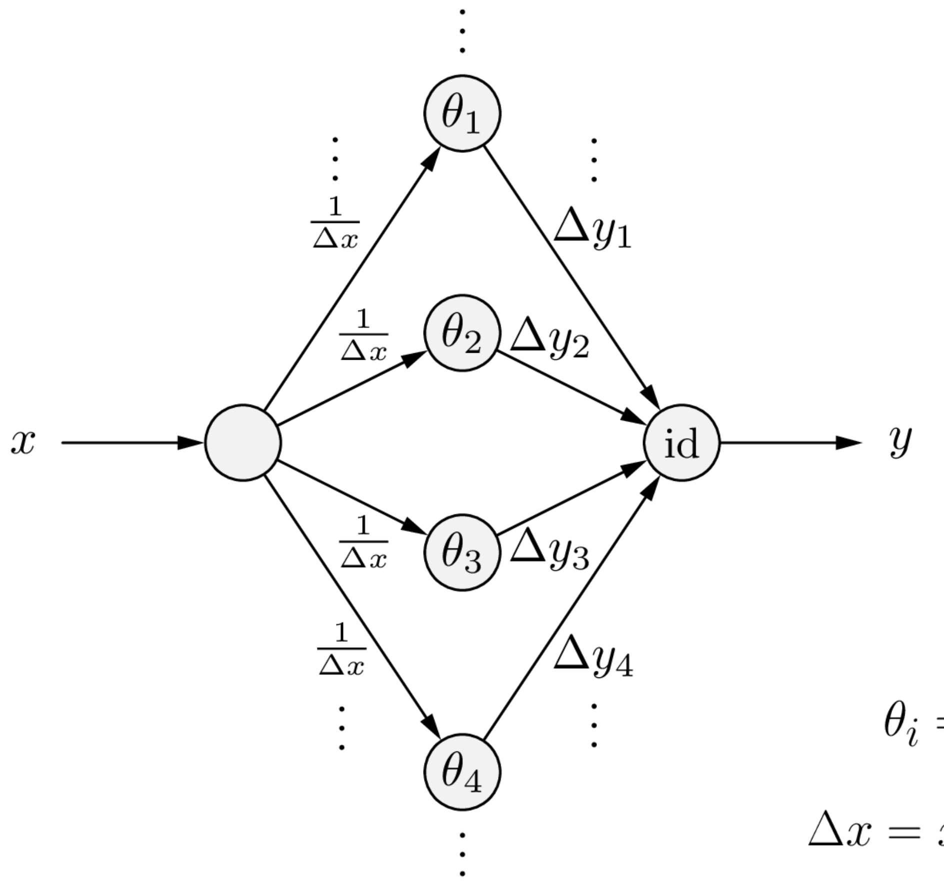




By using semi-linear functions the approximation can be improved.



Multi-layer Perceptrons: Function Approximation



$$\theta_i = \frac{x_i}{\Delta x}$$

$$\Delta x = x_{i+1} - x_i$$

A neural network that computes the step function shown on the preceding slide.
The output neuron has the identity as its activation and output functions.