# EE496 : COMPUTATIONAL INTELLINGENCE
# NN03 : MULTI LAYER PERCEPTRONS (MLPs)

UGUR HALICI
METU: Department of Electrical and Electronics Engineering (EEE)
METU-Hacettepe U: Neuroscience and Neurotechnology (NSNT)

# Multilayer 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) $U_{\text{hidden}} = U^{(1)}_{\text{hidden}} \cup \cdots \cup U^{(r-2)}_{\text{hidden}}$

$$\forall\, 1 \le i < j \le r-2 : U^{(i)}_{\text{hidden}} \cap U^{(j)}_{\text{hidden}} = \emptyset$$

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

or, if there are no hidden neurons ($r = 2$, $U_{\text{hidden}} = \emptyset$), then $C \subseteq (U_{\text{in}} \times U_{\text{out}})$

- It is a feed-forward network with strictly layered structure.

# General definition of a neural network

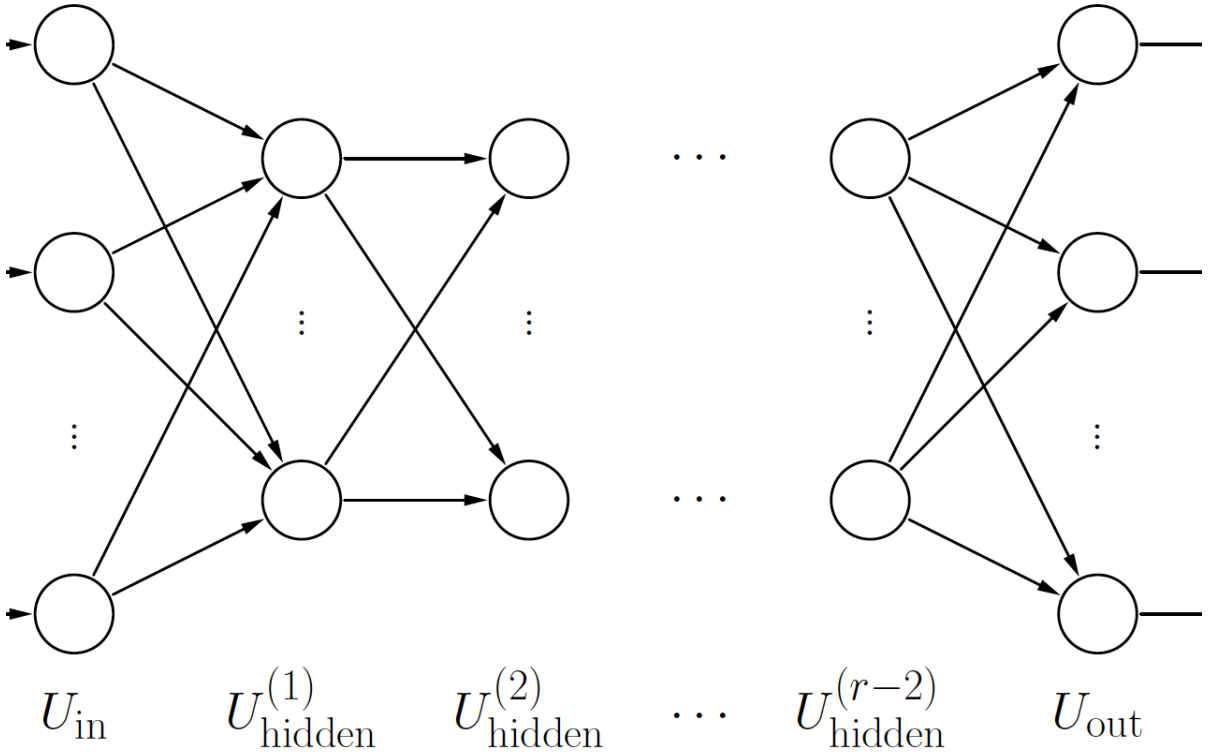
An (artificial) **neural network** is a (directed) graph $G = (U,C)$,
whose nodes $u \in U$ are called **neurons** or units and
whose edges $c \in C$ are called **connections**.
The set $U$ of nodes is partitioned into

- the set $U_{\text{in}}$ of **input neurons**,
- the set $U_{\text{out}}$ of **output neurons**, and
- the set $U_{hidden}$ of **hidden neurons**.

It is

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

# General structure of a multilayer perceptron



# General structure of a multilayer perceptron

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

  $\forall u \in U_{\text{hidden}} \cup U_{\text{out}} : \mathrm{f}^{(\mathrm{u})}_{\text{net}}(\vec{\mathbf{w}}_{\mathrm{u}}, \vec{\mathbf{in}}_{\mathrm{u}}) = \vec{\mathbf{w}}_u \vec{\mathbf{in}}_u = \sum_{v \in \text{pred}(u)} w_{uv}\, \text{out}_v$

- The activation function of each hidden neuron is a so-called **sigmoid function**, i.e. a monotonously increasing function

  $\mathrm{f} : \mathrm{R} \rightarrow [0, 1]$ with $\lim_{x \rightarrow -\infty} \mathrm{f}(\mathrm{x}) = 0$ and $\lim_{x \rightarrow \infty} \mathrm{f}(x) = 1$.

- The activation function of each output neuron is either also a sigmoid function

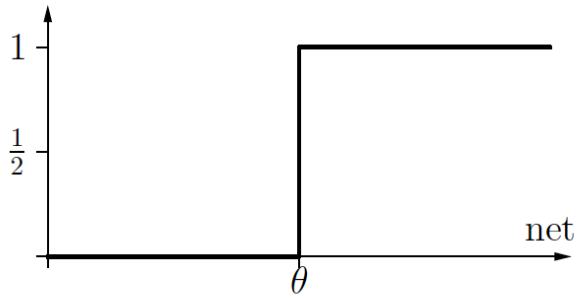
  or a linear function, i.e. $\mathrm{f}_{\text{act}}(\text{net}, \theta) = \alpha\, \text{net} - \theta$.

- Only the step function serves as a neurologically plausible activation function.

# Sigmoid Activation Functions

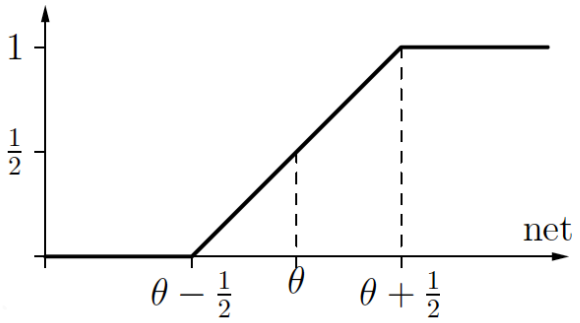
step function:

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



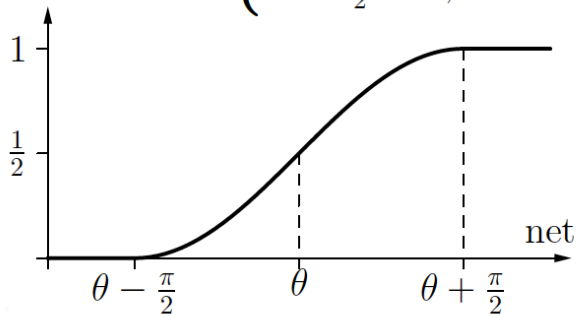

semi-linear function:

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



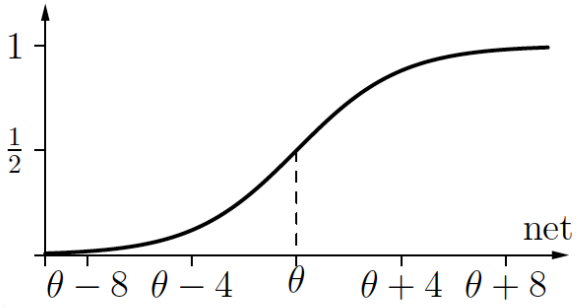

sine until saturation:

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




logistic function:

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



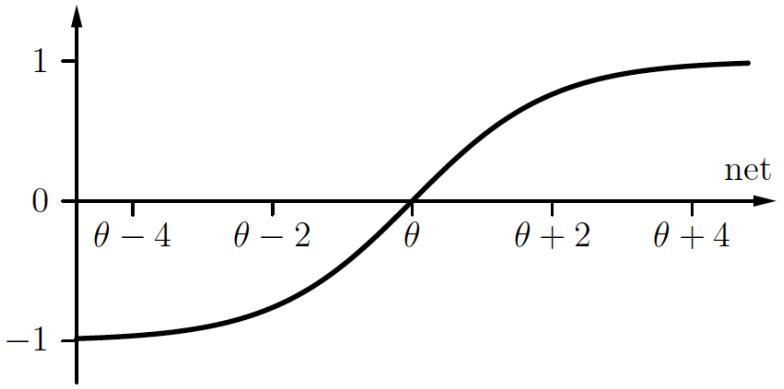
# Sigmoid Activation Functions

- All sigmoid functions on the previous slide are **unipolar**, i.e., they range from 0 to 1.
- Sometimes **bipolar** sigmoid functions are used, like the tangens hyperbolicus.

tangens hyperbolicus:

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

$$= \frac{2}{1 + e^{-2(\text{net} - \theta)}} - 1$$



# Sigmoid Activation Functions

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 multilayer perceptron.

Their connection weights are represented by an $n$ x $m$ matrix

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

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

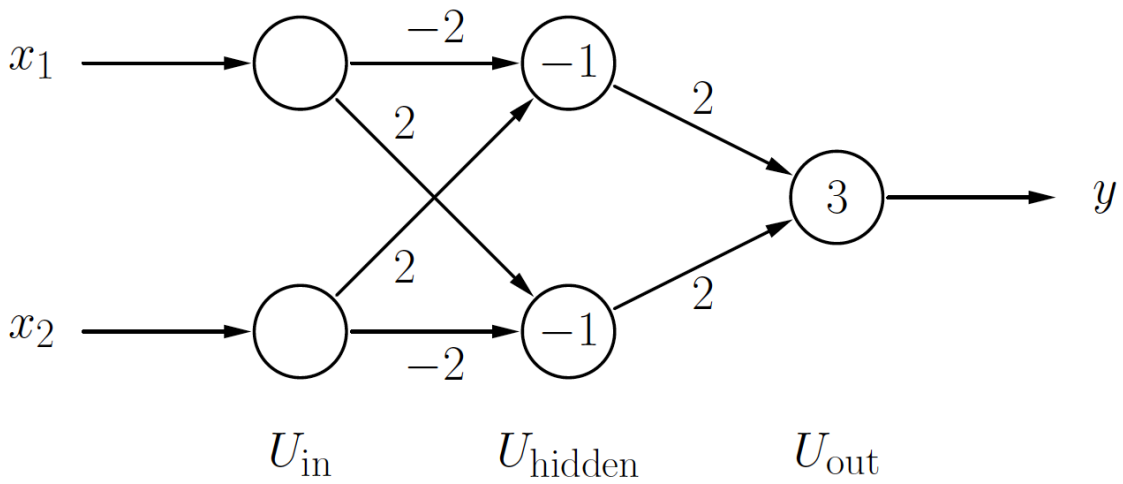
Advantage: The computation of the network input can be written as

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

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

# Multilayer Perceptrons: Biimplication

Solving the biimplication problem with a multilayer perceptron.




Note the additional input neurons compared to the TLU solution.

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

# Training Multilayer Perceptrons: Gradient Descent

- A general approach: gradient descent.
- Necessary condition: differentiable activation and output functions.

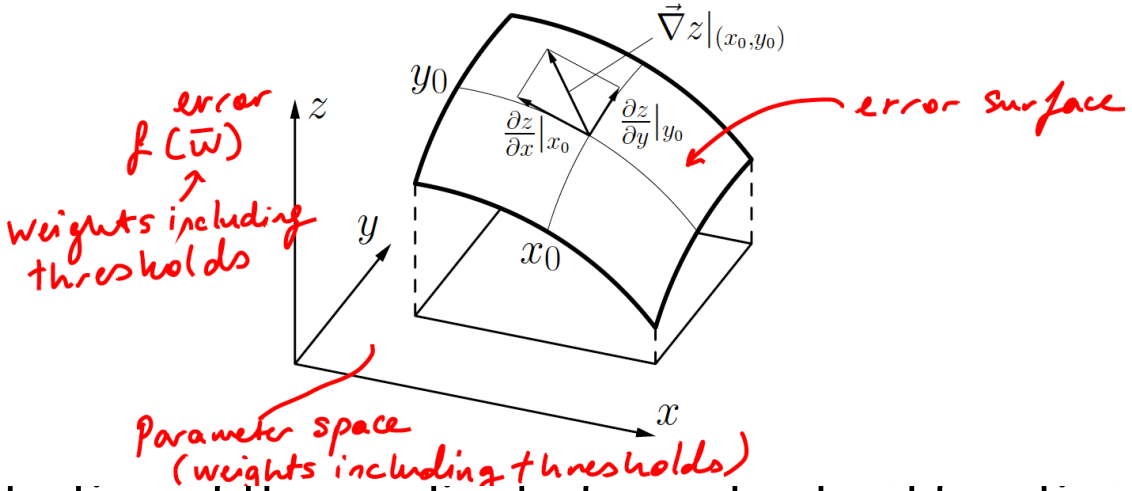



Illustration of the gradient of a real-valued function $z = \mathrm{f}(x, y)$ at a point $(x_0, y_0)$, which is

$$\vec{\nabla} z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|x_0, \frac{\partial z}{\partial y}|y_0\right)$$

which lies in the direction of the steepest ascend for $(x_0, y_0)$

# Training MLP: Gradient Descent: Formal Approach

**General Idea**: Approach the minimum of the error function in small steps.

Error function:

error for sample $l$ — for each sample in $L$ — for each network output

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{v \in U_{\text{out}}} e_v = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} e_v^{(l)}, \qquad (1)$$

Eq. No

a scalar function

error at output $v$ for sample $l$

Form gradient to determine the direction of the step:

$$\vec{\nabla}_{\vec{w}_u} e = \frac{\partial e}{\partial \vec{w}_u} = \left( -\frac{\partial e}{\partial \theta_u}, \frac{\partial e}{\partial w_{up_1}}, \ldots, \frac{\partial e}{\partial w_{up_n}} \right). \qquad (2)$$

weight vector of unit $u$

Exploit the sum over the training patterns:

$$\vec{\nabla}_{\vec{w}_u} e = \frac{\partial e}{\partial \vec{w}_u} = \frac{\partial}{\partial \vec{w}_u} \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}} \frac{\partial e^{(l)}}{\partial \vec{w}_u}. \qquad (3)$$

result is a vector

# Gradient Descent: Formal Approach

Single pattern error depends on weights only through the network input:

$$\vec{\nabla}_{\vec{w}_u} e^{(l)} = \frac{\partial e^{(l)}}{\partial \vec{w}_u} = \frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} \frac{\partial \operatorname{net}_u^{(l)}}{\partial \vec{w}_u}. \qquad (4)$$

Since $\operatorname{net}_u^{(l)} = \vec{w}_u \vec{\operatorname{in}}_u^{(l)}$ we have for the second factor

$$\frac{\partial \operatorname{net}_u^{(l)}}{\partial \vec{w}_u} = \vec{\operatorname{in}}_u^{(l)}. \qquad (5)$$

So $\vec{\nabla}_{\vec{w}_u} e^{(l)} = \frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} \vec{\operatorname{in}}_u^{(l)}$ (6)

For the first factor we consider the error $e^{(l)}$ for the training pattern $l = (\vec{\imath}^{(l)}, \vec{o}^{(l)})$:

$$e^{(l)} = \sum_{v \in U_{\text{out}}} e_u^{(l)} = \sum_{v \in U_{\text{out}}} \left( o_v^{(l)} - \operatorname{out}_v^{(l)} \right)^2, \qquad (7)$$

i.e. the sum of the errors over all output neurons.

↳ by definition

# Gradient Descent: Formal Approach

Therefore we have:

$$\frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} = \frac{\partial \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2}{\partial \operatorname{net}_u^{(l)}} = \sum_{v \in U_{\text{out}}} \frac{\partial \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2}{\partial \operatorname{net}_u^{(l)}}. \quad (8)$$

Constant

- Since only the actual output $\operatorname{out}^{(l)}_v$ of an output neuron v depends on the network input $\operatorname{net}^{(l)}_u$ of the neuron u we are considering, it is

$$\frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} = -2 \underbrace{\sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right) \frac{\partial \operatorname{out}_v^{(l)}}{\partial \operatorname{net}_u^{(l)}}}_{\delta_u^{(l)}}, \quad (10)$$

Definition: $\delta_u^{(l)}$ : error term at unit u

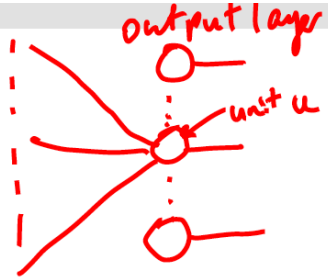
- which also introduces the abbreviation $\delta^{(l)}_u$ for the important sum appearing here.

(Eq 6) $\Rightarrow$ so $\vec{\nabla}_{\vec{w}_u} e^{(l)} = -2\, \delta_u^{(l)}\, \vec{in}_u^{(l)}$ (11)

# Gradient Descent: Formal Approach

Distinguish two cases:

- The neuron $u$ is an **output neuron**.
- The neuron $u$ is a **hidden neuron**.




In the first case we have

(Eq 9) definition of $\delta_u^{(l)} \Rightarrow$

$$\forall u \in U_{\text{out}}: \qquad \delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \qquad (11)$$

summation in formula drops since $\text{net}_u^{(l)}$ effects only $\text{out}_u^{(l)}$. The derivative for other outputs is zero

Therefore we have for the gradient

Eq. 10 $\Rightarrow$

$$\forall u \in U_{\text{out}}: \qquad \vec{\nabla}_{\vec{w}_u} e_u^{(l)} = \frac{\partial e_u^{(l)}}{\partial \vec{w}_u} = -2\left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \vec{\text{in}}_u^{(l)} \qquad (12)$$

and thus for the weight change

$$\forall u \in U_{\text{out}}: \qquad \Delta \vec{w}_u^{(l)} = -\frac{\eta}{2} \vec{\nabla}_{\vec{w}_u} e_u^{(l)} = \eta\left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \vec{\text{in}}_u^{(l)}. \qquad (13)$$

for simplification

# Gradient Descent: Formal Approach

Exact formulae depend on choice of activation and output function,

$$out^{(l)}_u = f_{out}(\,\mathrm{act}^{(l)}_u\,) = f_{out}(f_{act}(\,\mathrm{net}^{(l)}_u\,)).$$

Consider special case with

- output function is the identity,
- activation function is logistic, i.e.

$$f_{\mathrm{act}}(x) = \frac{1}{1+e^{-x}}.$$

- The first assumption yields

$$\frac{\partial\,\mathrm{out}^{(l)}_u}{\partial\,\mathrm{net}^{(l)}_u} = \frac{\partial\,\mathrm{act}^{(l)}_u}{\partial\,\mathrm{net}^{(l)}_u} = f'_{\mathrm{act}}(\,\mathrm{net}^{(l)}_u).$$

# Gradient Descent: Formal Approach

For a logistic activation function we have

$$\begin{aligned} f'_{\text{act}}(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\left(1+e^{-x}\right)^{-1} = -\left(1+e^{-x}\right)^{-2}\left(-e^{-x}\right) \\ &= \frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\ &= f_{\text{act}}(x)\cdot(1-f_{\text{act}}(x)), \end{aligned} \tag{17}$$

and therefore

$$f'_{\text{act}}(\,\text{net}_u^{(l)}) = f_{\text{act}}(\,\text{net}_u^{(l)}) \cdot \left(1 - f_{\text{act}}(\,\text{net}_u^{(l)})\right) = \text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right). \tag{18}$$

The resulting weight change is therefore (for output neuron)

$$\Delta\vec{w}_u^{(l)} = \eta\left(o_u^{(l)} - \text{out}_u^{(l)}\right)\text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right)\vec{\text{in}}_u^{(l)}, \tag{19}$$

which makes the computations very simple.

$f'_{\text{act}}(\text{net}_u^{(l)})$ in general

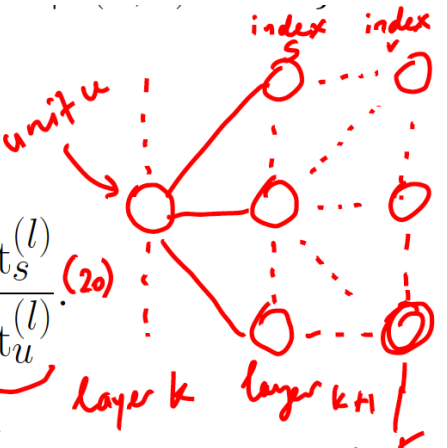
# Error Backpropagation

Consider now: The neuron u is a **hidden neuron**,

i.e. $u \in U_k$, $0 < k < r - 1$.

The output $\text{out}^{(l)}_v$ of an output neuron $v$ depends on the network input $\text{net}^{(l)}_u$ only indirectly through its successor neurons

$\text{succ}(u) = \{s \in U \mid (u, s) \in C\} = \{s_1, \ldots, s_m\} \subseteq U_{k+1}$, namely through their network inputs $\text{net}^{(l)}_s$ .




We apply the chain rule to obtain

summation through each neuron in the next layer

(Eq 9) Definition of $\delta^{(l)}_u$

$$\Rightarrow \delta^{(l)}_u = \sum_{v \in U_{\text{out}}} \sum_{s \in \text{succ}(u)} (o^{(l)}_v - \text{out}^{(l)}_v) \frac{\partial \text{out}^{(l)}_v}{\partial \text{net}^{(l)}_s} \frac{\partial \text{net}^{(l)}_s}{\partial \text{net}^{(l)}_u} \cdot \quad (20)$$

chain rule

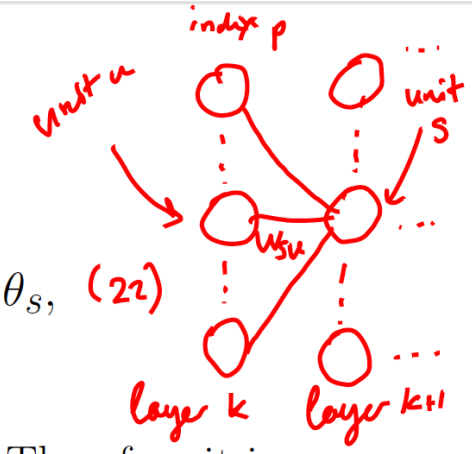
Exchanging the sums yields

$$\delta^{(l)}_u = \sum_{s \in \text{succ}(u)} \left( \sum_{v \in U_{\text{out}}} (o^{(l)}_v - \text{out}^{(l)}_v) \frac{\partial \text{out}^{(l)}_v}{\partial \text{net}^{(l)}_s} \right) \frac{\partial \text{net}^{(l)}_s}{\partial \text{net}^{(l)}_u} = \sum_{s \in \text{succ}(u)} \delta^{(l)}_s \frac{\partial \text{net}^{(l)}_s}{\partial \text{net}^{(l)}_u} \cdot \quad (21)$$

# Error Backpropagation

Consider the network input

$$\mathrm{net}_s^{(l)} = \vec{w}_s \vec{\mathrm{in}}_s^{(l)} = \left( \sum_{p \in \mathrm{pred}(s)} w_{sp} \, \mathrm{out}_p^{(l)} \right) - \theta_s, \qquad (22)$$




where one element of $\vec{\mathrm{in}}_s^{(l)}$ is the output $\mathrm{out}_u^{(l)}$ of the neuron $u$. Therefore it is

others except u is zero

$$\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \left( \sum_{p \in \mathrm{pred}(s)} w_{sp} \frac{\partial \mathrm{out}_p^{(l)}}{\partial \mathrm{net}_u^{(l)}} \right) - \frac{\partial \theta_s}{\partial \mathrm{net}_u^{(l)}} = w_{su} \frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}, \qquad (23)$$

zero

The result is the recursive equation (error backpropagation)

Eq 21 $\Rightarrow$

$$\delta_u^{(l)} = \left( \sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \right) \frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}. \qquad (24)$$

summation of error terms in next layer multiplied by connection weights

# Error Backpropagation

The resulting formula for the weight change is

$$\Delta \vec{w}_u^{(l)} = -\frac{\eta}{2} \vec{\nabla}_{\vec{w}_u} e^{(l)} = \eta \, \delta_u^{(l)} \, \vec{\text{in}}_u^{(l)} = \eta \left( \sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su} \right) \frac{\partial \, \text{out}_u^{(l)}}{\partial \, \text{net}_u^{(l)}} \, \vec{\text{in}}_u^{(l)}. \quad (25)$$

learning rate; for simplification; gradient descent; by Eq 10; by Eq 24

Consider again the special case with

- output function is the identity,
- activation function is logistic.

The resulting formula for the weight change is then (for hidden neurons)

$$\Delta \vec{w}_u^{(l)} = \eta \left( \sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su} \right) \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right) \vec{\text{in}}_u^{(l)}. \quad (26)$$

$f'_{\text{act}}(\text{net}_u^{(l)})$ in general

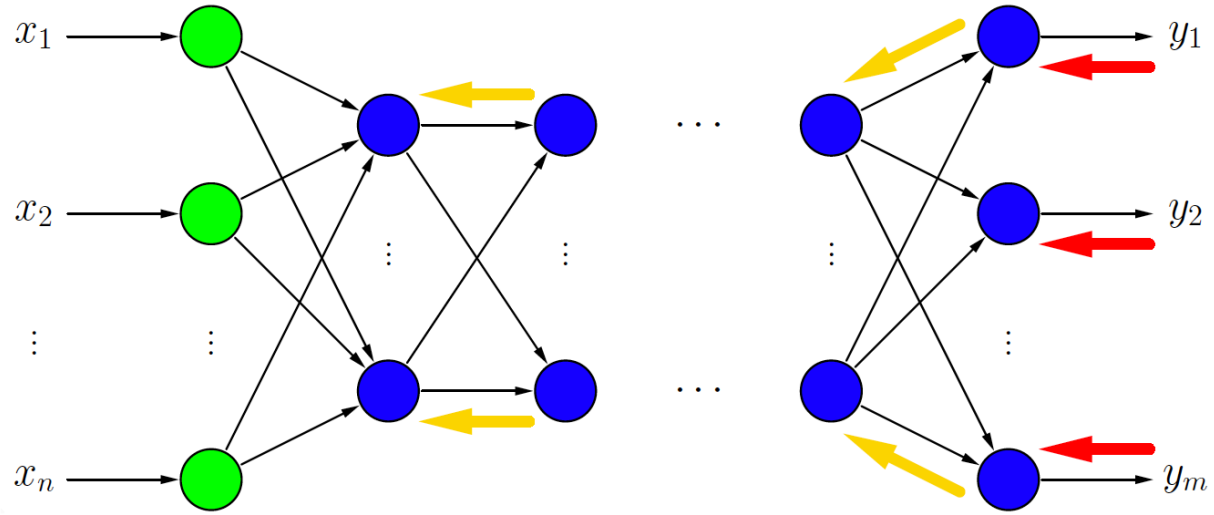
# Error Backpropagation : cookbook recipe

$\forall u \in U_{\text{in}}$ :
$\text{out}_u^{(l)} = \text{ex}_u^{(l)}$

forward propagation:

$\forall u \in U_{\text{hidden}} \cup U_{\text{out}}$ :
$\text{out}_u^{(l)} = \left(1 + \exp\left(-\sum_{p \in \text{pred}(u)} w_{up}\, \text{out}_p^{(l)}\right)\right)^{-1}$

$\text{net}_u^{(l)}$




- logistic activation function
- implicit bias value

backward propagation:

$\forall u \in U_{\text{hidden}}$ :
$\delta_u^{(l)} = \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su}\right) \lambda_u^{(l)}$

error factor:

$\forall u \in U_{\text{out}}$ :
$\delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \lambda_u^{(l)}$

activation derivative:

$\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$

$f'_{act}(\text{net}_u^{(l)})$

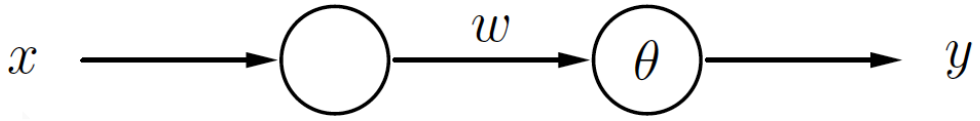
weight change:

$\Delta w_{up}^{(l)} = \eta\, \delta_u^{(l)}\, \text{out}_p^{(l)}$

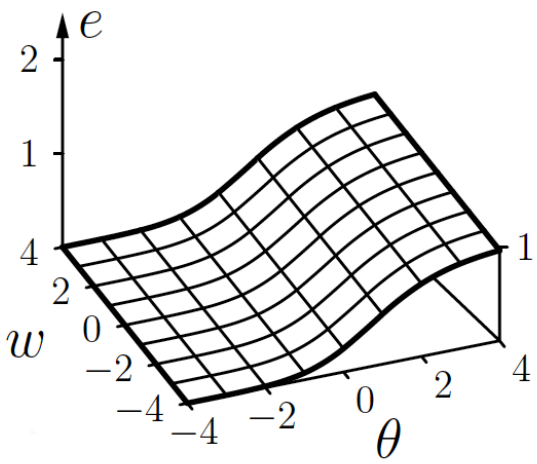
$\text{in}_{up}$

# Gradient Descent : Examples

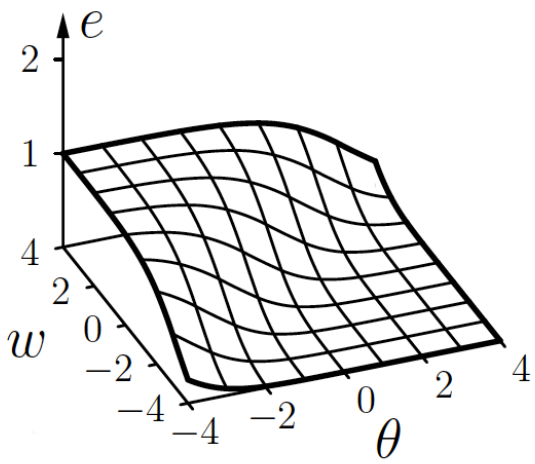
Gradient descent training for the negation ¬x



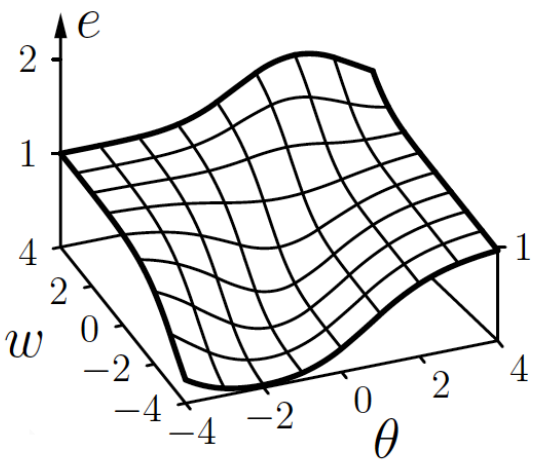

| $x$ | $y$ |
|---|---|
| 0 | 1 |
| 1 | 0 |




error for $x = 0$




error for $x = 1$




sum of errors

# Gradient Descent : Examples

$\eta$=1

| epoch | $\theta$ | $w$ | error |
|---|---|---|---|
| 0 | 3.00 | 3.50 | 1.307 |
| 20 | 3.77 | 2.19 | 0.986 |
| 40 | 3.71 | 1.81 | 0.970 |
| 60 | 3.50 | 1.53 | 0.958 |
| 80 | 3.15 | 1.24 | 0.937 |
| 100 | 2.57 | 0.88 | 0.890 |
| 120 | 1.48 | 0.25 | 0.725 |
| 140 | −0.06 | −0.98 | 0.331 |
| 160 | −0.80 | −2.07 | 0.149 |
| 180 | −1.19 | −2.74 | 0.087 |
| 200 | −1.44 | −3.20 | 0.059 |
| 220 | −1.62 | −3.54 | 0.044 |

Online Training

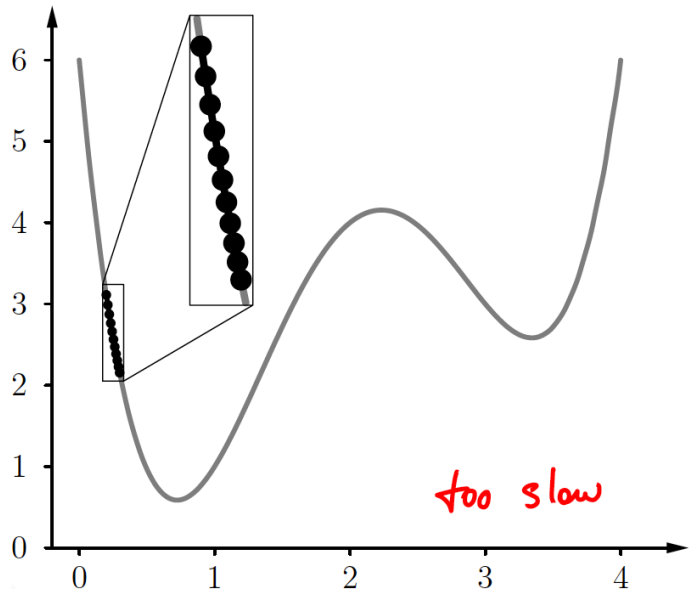
| epoch | $\theta$ | $w$ | error |
|---|---|---|---|
| 0 | 3.00 | 3.50 | 1.295 |
| 20 | 3.76 | 2.20 | 0.985 |
| 40 | 3.70 | 1.82 | 0.970 |
| 60 | 3.48 | 1.53 | 0.957 |
| 80 | 3.11 | 1.25 | 0.934 |
| 100 | 2.49 | 0.88 | 0.880 |
| 120 | 1.27 | 0.22 | 0.676 |
| 140 | −0.21 | −1.04 | 0.292 |
| 160 | −0.86 | −2.08 | 0.140 |
| 180 | −1.21 | −2.74 | 0.084 |
| 200 | −1.45 | −3.19 | 0.058 |
| 220 | −1.63 | −3.53 | 0.044 |

Batch Training

# Gradient Descent : Examples

- Example function: $f(x) = (5/4)\,x^4 - 7\,x^3 + (115/6)\,x^2 - 18x + 6$

| $i$ | $x_i$ | $f(x_i)$ | $f'(x_i)$ | $\Delta x_i$ |
|---|---|---|---|---|
| 0 | 0.200 | 3.112 | −11.147 | 0.011 |
| 1 | 0.211 | 2.990 | −10.811 | 0.011 |
| 2 | 0.222 | 2.874 | −10.490 | 0.010 |
| 3 | 0.232 | 2.766 | −10.182 | 0.010 |
| 4 | 0.243 | 2.664 | −9.888 | 0.010 |
| 5 | 0.253 | 2.568 | −9.606 | 0.010 |
| 6 | 0.262 | 2.477 | −9.335 | 0.009 |
| 7 | 0.271 | 2.391 | −9.075 | 0.009 |
| 8 | 0.281 | 2.309 | −8.825 | 0.009 |
| 9 | 0.289 | 2.233 | −8.585 | 0.009 |
| 10 | 0.298 | 2.160 | | |

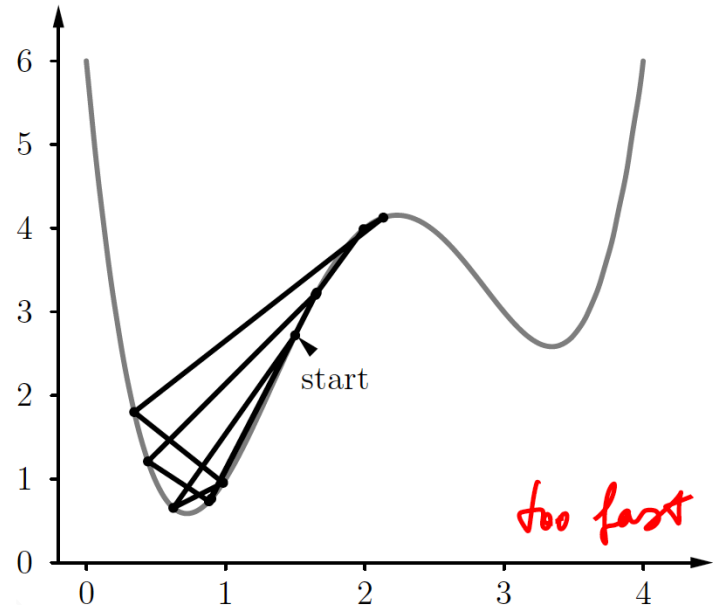



- Gradient descent with initial value 0.2 and learning rate 0.001.

# Gradient Descent : Examples

- Example function: $f(x) = (5/4)\,x^4 - 7\,x^3 + (115/6)\,x^2 - 18x + 6$

| $i$ | $x_i$ | $f(x_i)$ | $f'(x_i)$ | $\Delta x_i$ |
|---|---|---|---|---|
| 0 | 1.500 | 2.719 | 3.500 | −0.875 |
| 1 | 0.625 | 0.655 | −1.431 | 0.358 |
| 2 | 0.983 | 0.955 | 2.554 | −0.639 |
| 3 | 0.344 | 1.801 | −7.157 | 1.789 |
| 4 | 2.134 | 4.127 | 0.567 | −0.142 |
| 5 | 1.992 | 3.989 | 1.380 | −0.345 |
| 6 | 1.647 | 3.203 | 3.063 | −0.766 |
| 7 | 0.881 | 0.734 | 1.753 | −0.438 |
| 8 | 0.443 | 1.211 | −4.851 | 1.213 |
| 9 | 1.656 | 3.231 | 3.029 | −0.757 |
| 10 | 0.898 | 0.766 | | |

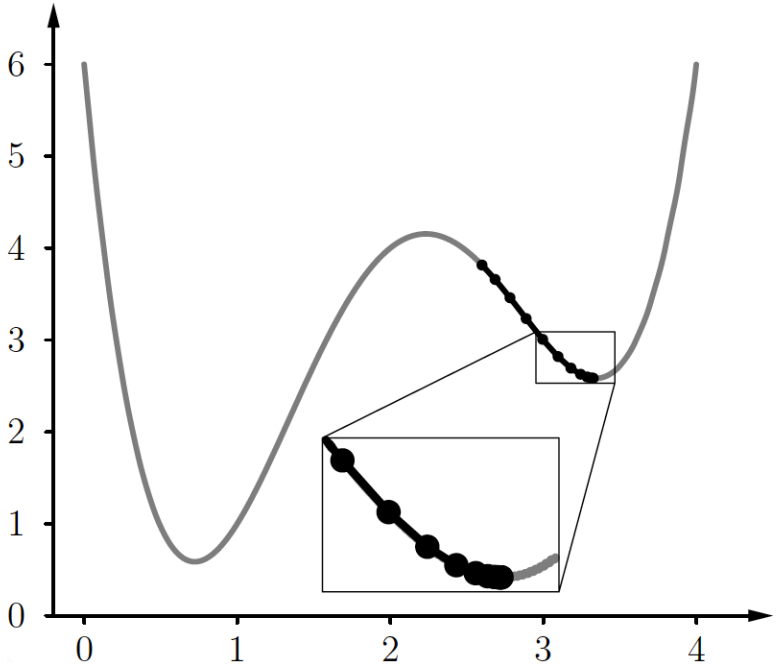



- Gradient descent with initial value 1.5 and learning rate 0.25.

# Gradient Descent : Examples

- Example function: $f(x) = (5/4)\,x^4 - 7\,x^3 + (115/6)\,x^2 - 18x + 6$

| $i$ | $x_i$ | $f(x_i)$ | $f'(x_i)$ | $\Delta x_i$ |
|---|---|---|---|---|
| 0 | 2.600 | 3.816 | −1.707 | 0.085 |
| 1 | 2.685 | 3.660 | −1.947 | 0.097 |
| 2 | 2.783 | 3.461 | −2.116 | 0.106 |
| 3 | 2.888 | 3.233 | −2.153 | 0.108 |
| 4 | 2.996 | 3.008 | −2.009 | 0.100 |
| 5 | 3.097 | 2.820 | −1.688 | 0.084 |
| 6 | 3.181 | 2.695 | −1.263 | 0.063 |
| 7 | 3.244 | 2.628 | −0.845 | 0.042 |
| 8 | 3.286 | 2.599 | −0.515 | 0.026 |
| 9 | 3.312 | 2.589 | −0.293 | 0.015 |
| 10 | 3.327 | 2.585 | | |




- Gradient descent with initial value 2.6 and learning rate 0.05.
- So another initial value may result in converging to another local minimum

# Gradient Descent : Variants

**Weight update rule:**

$$w(t+1) = w(t) + \Delta w(t)$$

**Standard backpropagation:**

$$\Delta w(t) = -(\eta/2)\nabla_w e(t)$$

**Manhattan training:**

$$\Delta w(t) = -\eta\ \mathrm{sgn}(\nabla_w e(t)).$$

i.e. considering only one direction (sign) and selecting a fixed increment

**Momentum term:**

$$\Delta w(t) = -(\eta/2)\nabla_w e(t) + \beta\, \Delta w(t-1),$$

i.e. every step is dependent on the previous change thus speeding up the process.

# Gradient Descent : Variants

**Self-adaptive error backpropagation:**

The related component of the gradient vector

$$\eta_w(t) = \begin{cases} c^- \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \wedge \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \eta_w(t-1), & \text{otherwise.} \end{cases}$$

a seperate learning rate for each parameter

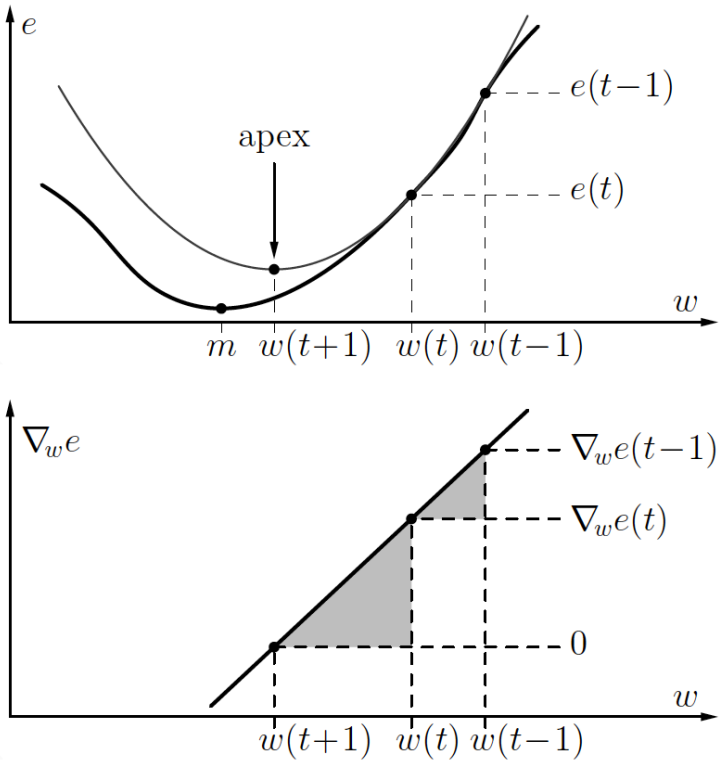
**Resilient error backpropagation:** combination of Manhattan training and self adaptive BP

$$\Delta w(t) = \begin{cases} c^- \cdot \Delta w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \Delta w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \wedge \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \Delta w(t-1), & \text{otherwise.} \end{cases}$$

Typical values: $c^- \in [0.5, 0.7]$ and $c^+ \in [1.05, 1.2]$.

# Gradient Descent : Variants

**Quickpropagation**




The weight update rule can be derived from the triangles:

$$\Delta w(t) = \frac{\nabla_w e(t)}{\nabla_w e(t-1) - \nabla_w e(t)} \cdot \Delta w(t-1).$$

check if $\frac{\nabla_w e(t-1) - \nabla_w e(t)}{\Delta w(t-1)} < 0$ to understand if parabola opens upward (i.e. minimum)

# Gradient Descent : Examples

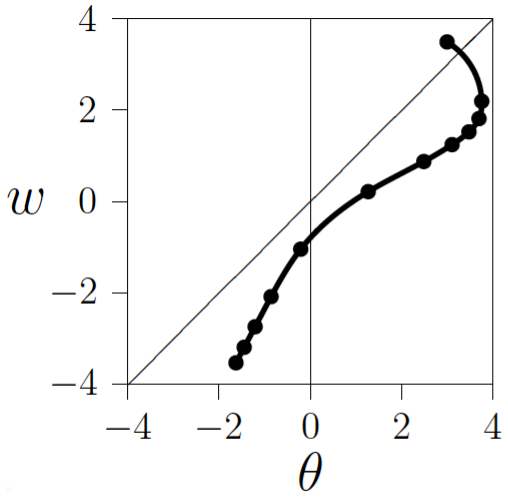
| epoch | $\theta$ | $w$ | error |
|---|---|---|---|
| 0 | 3.00 | 3.50 | 1.295 |
| 20 | 3.76 | 2.20 | 0.985 |
| 40 | 3.70 | 1.82 | 0.970 |
| 60 | 3.48 | 1.53 | 0.957 |
| 80 | 3.11 | 1.25 | 0.934 |
| 100 | 2.49 | 0.88 | 0.880 |
| 120 | 1.27 | 0.22 | 0.676 |
| 140 | −0.21 | −1.04 | 0.292 |
| 160 | −0.86 | −2.08 | 0.140 |
| 180 | −1.21 | −2.74 | 0.084 |
| 200 | −1.45 | −3.19 | 0.058 |
| 220 | −1.63 | −3.53 | 0.044 |

without momentum term

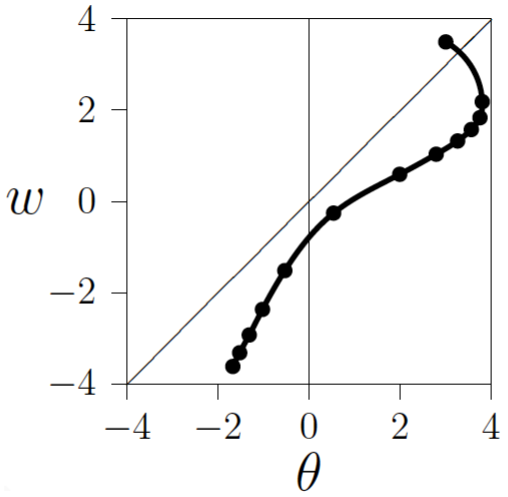
| epoch | $\theta$ | $w$ | error |
|---|---|---|---|
| 0 | 3.00 | 3.50 | 1.295 |
| 10 | 3.80 | 2.19 | 0.984 |
| 20 | 3.75 | 1.84 | 0.971 |
| 30 | 3.56 | 1.58 | 0.960 |
| 40 | 3.26 | 1.33 | 0.943 |
| 50 | 2.79 | 1.04 | 0.910 |
| 60 | 1.99 | 0.60 | 0.814 |
| 70 | 0.54 | −0.25 | 0.497 |
| 80 | −0.53 | −1.51 | 0.211 |
| 90 | −1.02 | −2.36 | 0.113 |
| 100 | −1.31 | −2.92 | 0.073 |
| 110 | −1.52 | −3.31 | 0.053 |
| 120 | −1.67 | −3.61 | 0.041 |

with momentum term

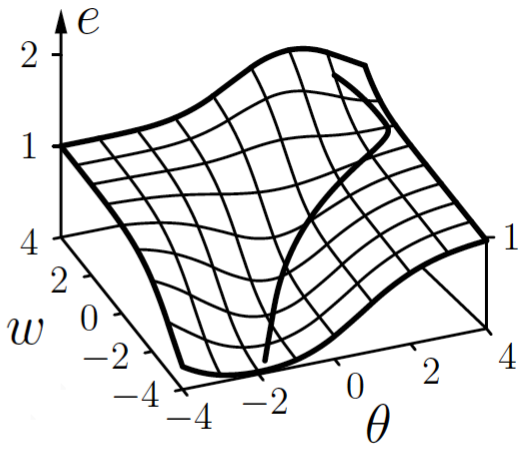
# Gradient Descent : Examples




without momentum term




with momentum term




with momentum term

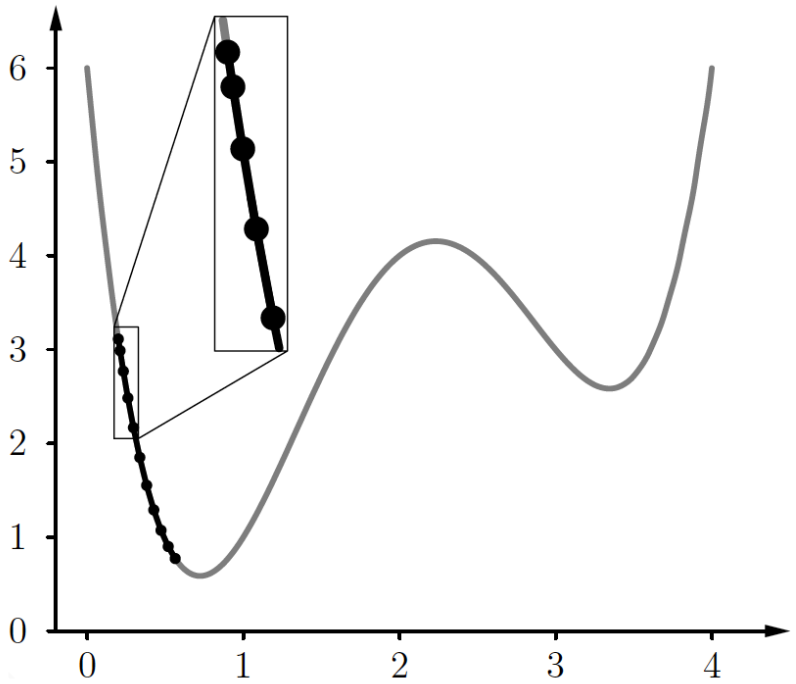
Dots show position every 20 (without momentum term) or every 10 epochs (with momentum term).

- Learning with a momentum term is about twice as fast.

# Gradient Descent : Examples

- Example function: $f(x) = (5/4)\,x^4 - 7\,x^3 + (115/6)\,x^2 - 18x + 6$

| $i$ | $x_i$ | $f(x_i)$ | $f'(x_i)$ | $\Delta x_i$ |
|---|---|---|---|---|
| 0 | 0.200 | 3.112 | −11.147 | 0.011 |
| 1 | 0.211 | 2.990 | −10.811 | 0.021 |
| 2 | 0.232 | 2.771 | −10.196 | 0.029 |
| 3 | 0.261 | 2.488 | −9.368 | 0.035 |
| 4 | 0.296 | 2.173 | −8.397 | 0.040 |
| 5 | 0.337 | 1.856 | −7.348 | 0.044 |
| 6 | 0.380 | 1.559 | −6.277 | 0.046 |
| 7 | 0.426 | 1.298 | −5.228 | 0.046 |
| 8 | 0.472 | 1.079 | −4.235 | 0.046 |
| 9 | 0.518 | 0.907 | −3.319 | 0.045 |
| 10 | 0.562 | 0.777 | | |

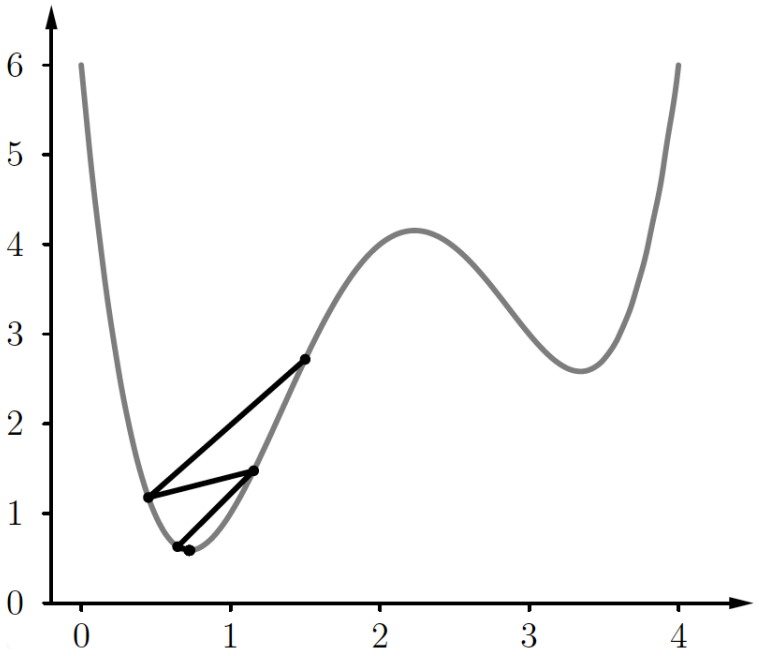



gradient descent with momentum term (β = 0.9)

# Gradient Descent : Examples

- Example function: $f(x) = (5/4)\,x^4 - 7\,x^3 + (115/6)\,x^2 - 18x + 6$

| $i$ | $x_i$ | $f(x_i)$ | $f'(x_i)$ | $\Delta x_i$ |
|---|---|---|---|---|
| 0 | 1.500 | 2.719 | 3.500 | −1.050 |
| 1 | 0.450 | 1.178 | −4.699 | 0.705 |
| 2 | 1.155 | 1.476 | 3.396 | −0.509 |
| 3 | 0.645 | 0.629 | −1.110 | 0.083 |
| 4 | 0.729 | 0.587 | 0.072 | −0.005 |
| 5 | 0.723 | 0.587 | 0.001 | 0.000 |
| 6 | 0.723 | 0.587 | 0.000 | 0.000 |
| 7 | 0.723 | 0.587 | 0.000 | 0.000 |
| 8 | 0.723 | 0.587 | 0.000 | 0.000 |
| 9 | 0.723 | 0.587 | 0.000 | 0.000 |
| 10 | 0.723 | 0.587 | | |




Gradient descent with self-adapting learning rate (c+ = 1.2, c− = 0.5).

# Other Extensions of Error Backpropagation

**Flat Spot Elimination:**

$$\Delta w(t) = -(\eta/2)\nabla_w e(t) + \zeta$$

- Eliminates slow learning in saturation region of logistic function.
- Counteracts the decay of the error signals over the layers.
- i.e. instead $f'_{act}$ use $f'_{act} + \alpha$ where $\alpha$ is a constant, say $\alpha = 0.1$

**Weight Decay:**

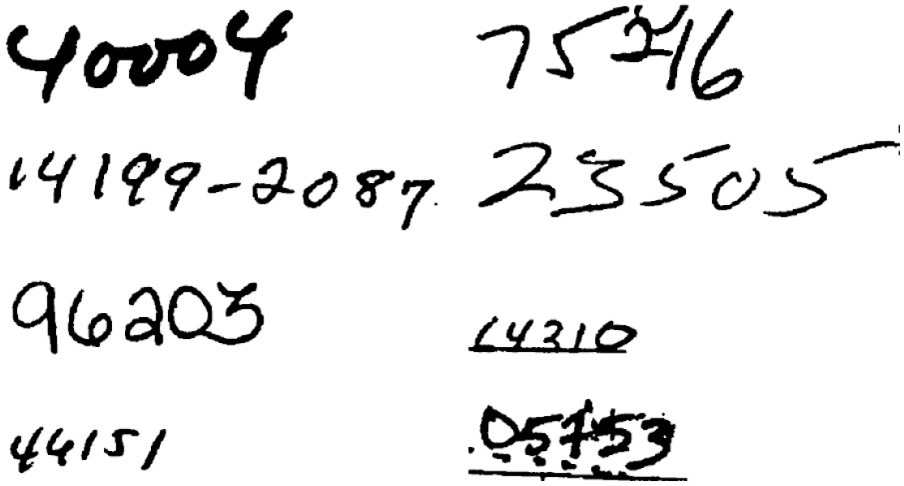
$$\Delta w(t) = -(\eta/2)\ \nabla_w e(t) - \xi\, w(t),$$

- choose $0.005 \leq \xi \leq 0.03$
- Helps to improve the robustness of the training results.
- Can be derived from an extended error function penalizing large weights:

$$e^* = e + (\xi/2) \sum_{u \in U\text{out} \cup U\text{hidden}} (\theta_u^2 + \sum_{p \in \text{pred}(u)} w_{up}^2)$$

# Example: Recognizing handwritten zip codes

source: Le Cun u.a. (1990) Advances in NIPS:2, 396–404



- 9298 segmented and digitized digits of handwritten postal codes
- survey: post office in Buffalo, NY, USA (U.S. Postal Service)
- digits were written by many different people: high variance in height, style of writing, writing tools and care.
- in addition 3349 printed digits in 35 different fonts

# Example: Recognizing handwritten zip codes

source: Schölkopf und Smola (2002)

- aim: Learning a MLP for recognizing zip codes
- training set size: 7291 handwritten and 2549 printed digits
- validation set size: 2007 handwritten and 700 printed digits
- both sets include several ambiguous, unclassified or misclassified samples
- shown right: 100 validation set digits

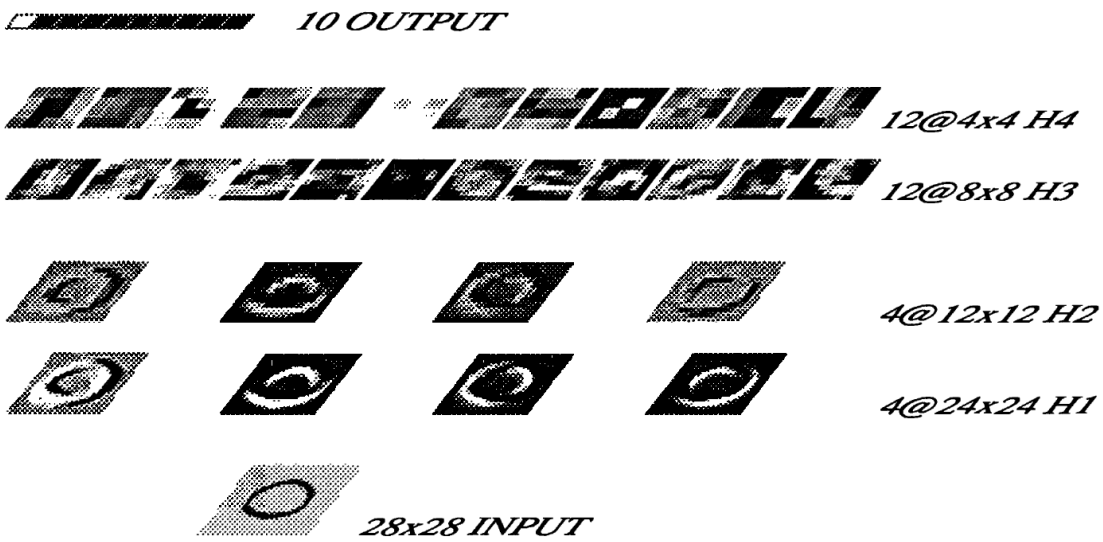
# Other Extensions of Error Backpropagation

challenge: every connection ought to be adaptabe (though with strong limitations)

- training by error backpropagation
- input: 16 x 16 pixel pattern of the normalized digit
- output: 10 neurons, 1 per class
- If a pattern is associated with a class, the output neuron i shall output +1 while all other neurons output −1
- problem: for fully interconnected neural network with multiple hidden layers there are too many parameters to be trained
- solution: restricted connection pattern

# Other Extensions of Error Backpropagation

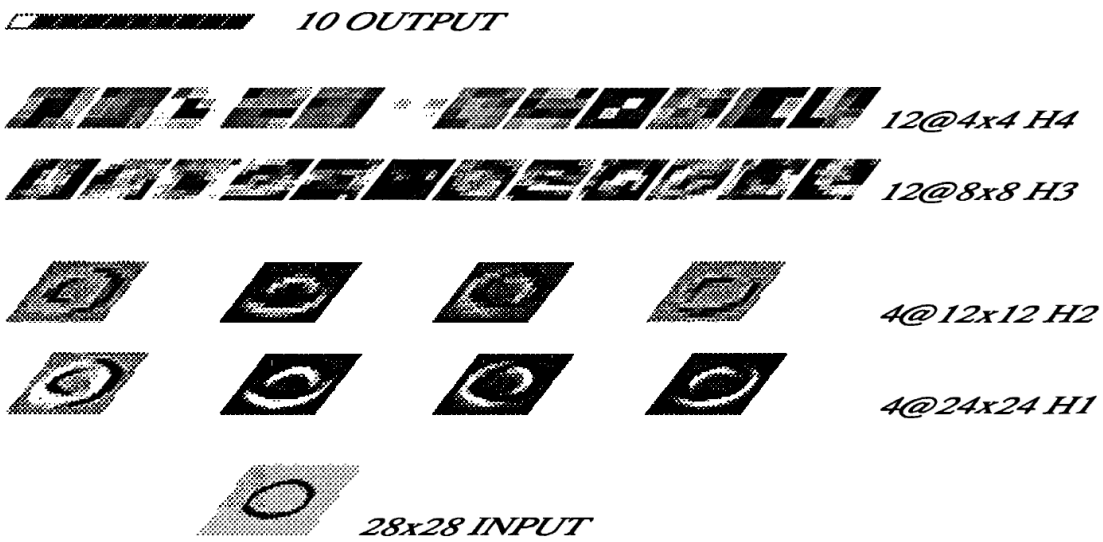
solution: restricted connection pattern




- 4 hiden layers H1, H2, H3 und H4
- neuron groups in H1, H3 share the same weights → less parameters
- neurons in H2, H4 calculate average values → Input values for upper layers
- input layer: enlarge from 16 x 16 to 28 x28 pixel for considering thresholds

# Other Extensions of Error Backpropagation

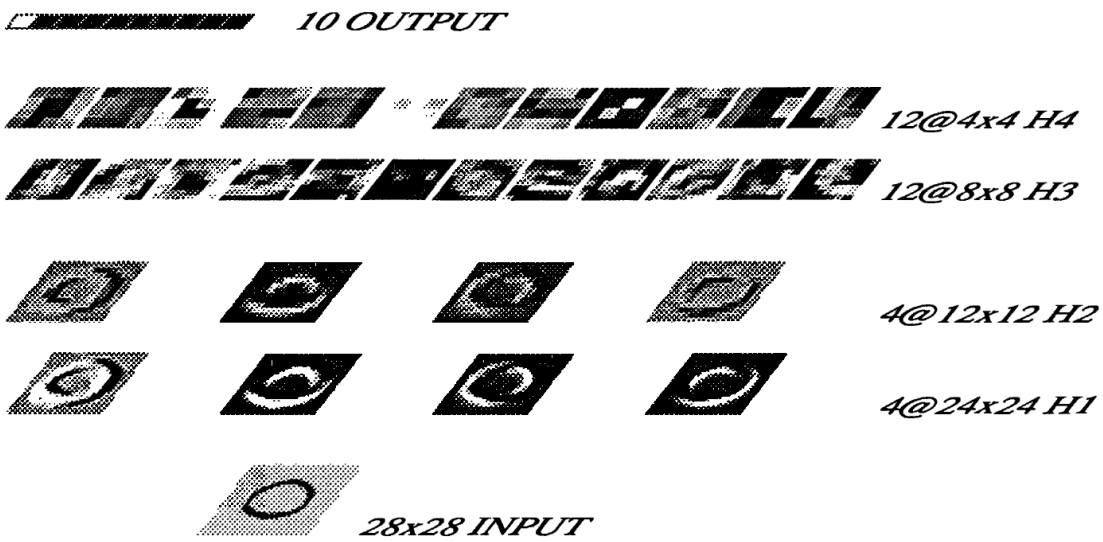
solution: restricted connection pattern




- 4 groups of 24x24 = 576 neurons assembled as 4 independent feature maps.
- every neuron in a feature map takes a 5 x 5 Input (convolution: the correponding weights are called filters (kernels) which are learned by Back Propagation (BP). Now mostly 3x3 filters are used to reduce the number of parameters)
- every neuron in a feature map hold the same parameters (weight sharing)
- these parameters may differ between the feature maps

# Other Extensions of Error Backpropagation

solution: restricted connection pattern




- H2 is for forwarding: 4 maps of 12 x 12 = 144 neurons each
- each neuron in these maps receives an input from 4 neurons of the corresponding map in H1
- all weights are equal, even within one single neuron
- so H2 is reducing the size while forwarding the information (pooling)

- NOW: this structure is used in Convolutional Neural Networks (CNN) which is network for Deep Learning.