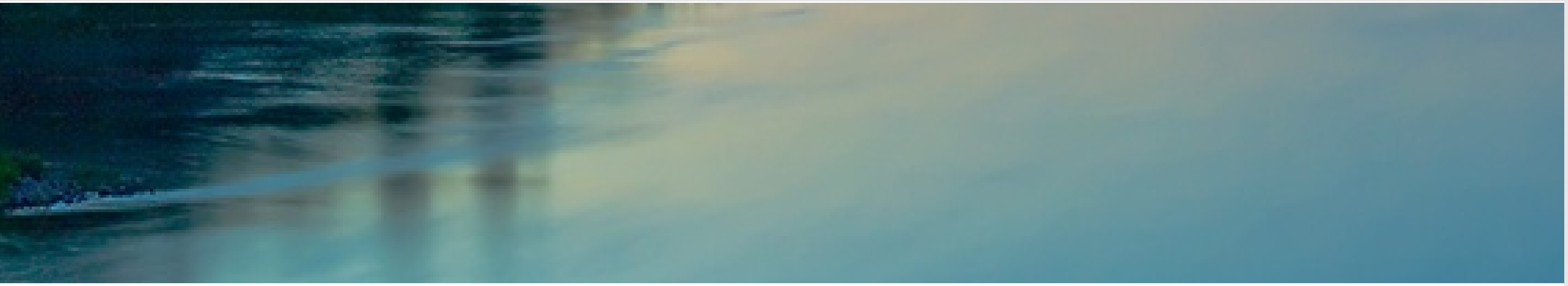


Neural Networks

Prof. Dr. Rudolf Kruse

Computational Intelligence Group
Faculty for Computer Science
kruse@iws.cs.uni-magdeburg.de

Training Multilayer Perceptrons

Training Multilayer Perceptrons: Gradient Descent

- Problem of logistic regression: Works only for two-layer perceptrons.
- More general approach: **gradient descent**.
- Necessary condition: **differentiable activation and output functions**.

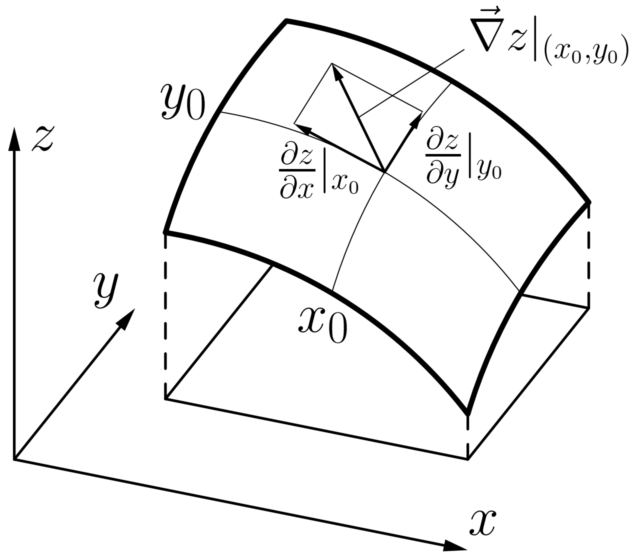


Illustration of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0).
It is $\vec{\nabla} z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|_{x_0}, \frac{\partial z}{\partial y}|_{y_0}\right)$.

Gradient Descent: Formal Approach

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

Error function:

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

Form gradient to determine the direction of the step:

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

Exploit the sum over the training patterns:

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

Gradient Descent: Formal Approach

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

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

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

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

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

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

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

Gradient Descent: Formal Approach

Therefore we have

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

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

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

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

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

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

Therefore we have for the gradient

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

and thus for the weight change

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

Gradient Descent: Formal Approach

Exact formulae depend on choice of activation and output function, since it is

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

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

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

Gradient Descent: Formal Approach

For a logistic activation function we have

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

and therefore

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

The resulting weight change is therefore

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

which makes the computations very simple.

Consider now: The neuron u is a **hidden neuron**, i.e. $u \in U_k$, $0 < k < r-1$.

The output $\mathrm{out}_v^{(l)}$ of an output neuron v depends on the network input $\mathrm{net}_u^{(l)}$ only indirectly through its successor neurons $\mathrm{succ}(u) = \{s \in U \mid (u,s) \in C\} = \{s_1, \ldots, s_m\} \subseteq U_{k+1}$, namely through their network inputs $\mathrm{net}_s^{(l)}$.

We apply the chain rule to obtain

$$\delta_u^{(l)} = \sum_{v \in U_{\mathrm{out}}} \sum_{s \in \mathrm{succ}(u)} (o_v^{(l)} - \mathrm{out}_v^{(l)}) \frac{\partial\, \mathrm{out}_v^{(l)}}{\partial\, \mathrm{net}_s^{(l)}} \frac{\partial\, \mathrm{net}_s^{(l)}}{\partial\, \mathrm{net}_u^{(l)}}.$$

Exchanging the sums yields

$$\delta_u^{(l)} = \sum_{s \in \mathrm{succ}(u)} \left(\sum_{v \in U_{\mathrm{out}}} (o_v^{(l)} - \mathrm{out}_v^{(l)}) \frac{\partial\, \mathrm{out}_v^{(l)}}{\partial\, \mathrm{net}_s^{(l)}} \right) \frac{\partial\, \mathrm{net}_s^{(l)}}{\partial\, \mathrm{net}_u^{(l)}} = \sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} \; \frac{\partial\, \mathrm{net}_s^{(l)}}{\partial\, \mathrm{net}_u^{(l)}}.$$

Error Backpropagation

Consider the network input

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

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

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

The result is the recursive equation (error backpropagation)

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

Error Backpropagation

The resulting formula for the weight change is

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

Consider again the special case with

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

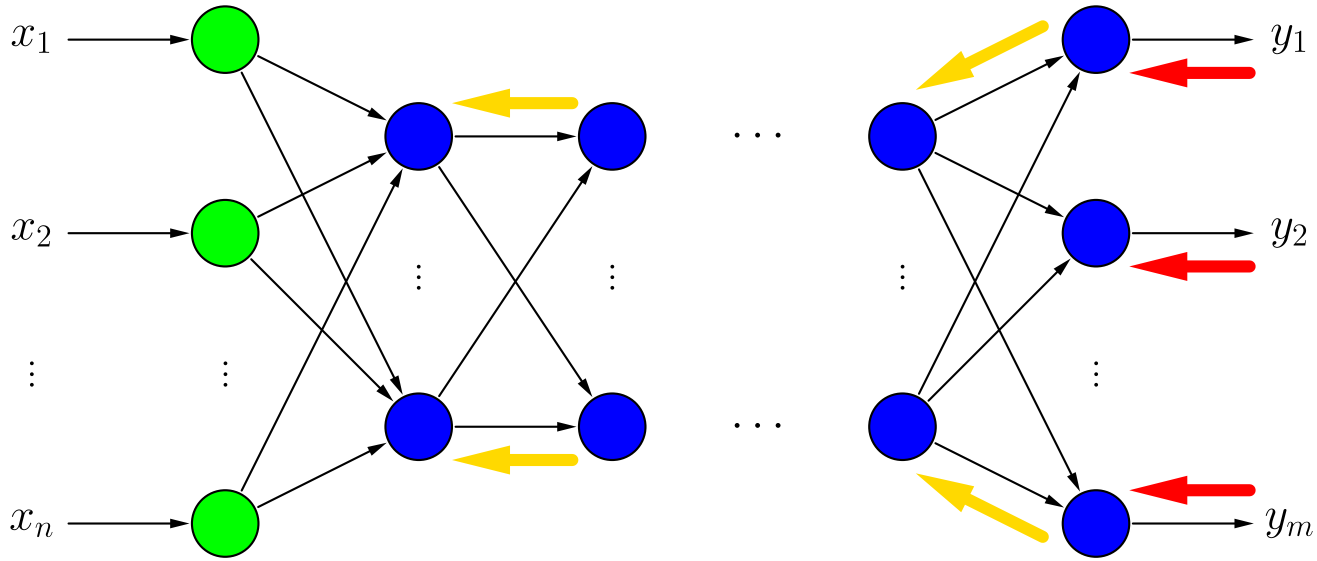
The resulting formula for the weight change is then

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

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

forward propagation:

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



- logistic activation function
- implicit bias value

error factor:

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

backward propagation:

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

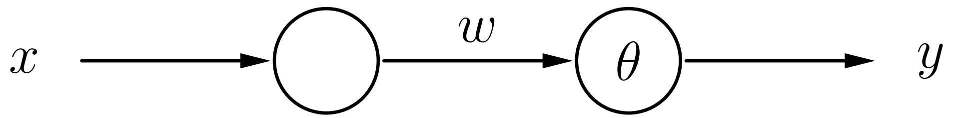
activation derivative:

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

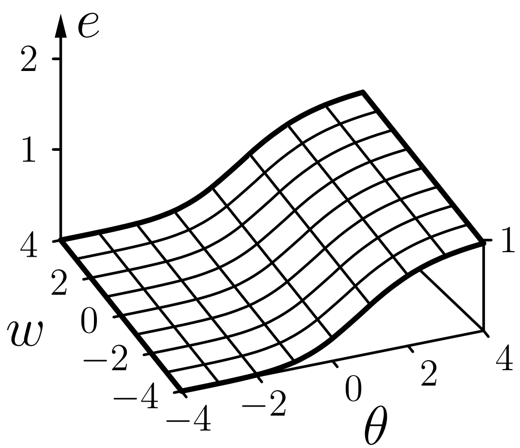
weight change:

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

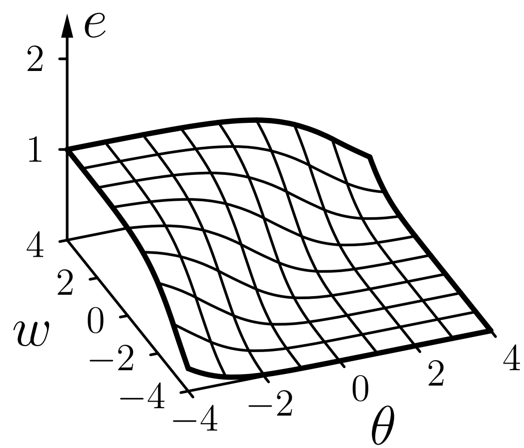
Gradient descent training for the negation $\neg x$



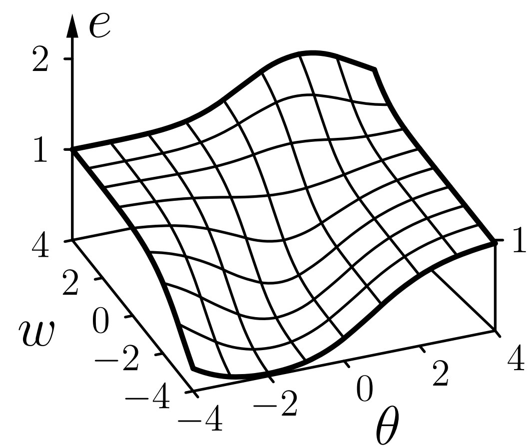
x	y
0	1
1	0



error for $x = 0$



error for $x = 1$



sum of errors

Gradient Descent: Examples

epoch	θ	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

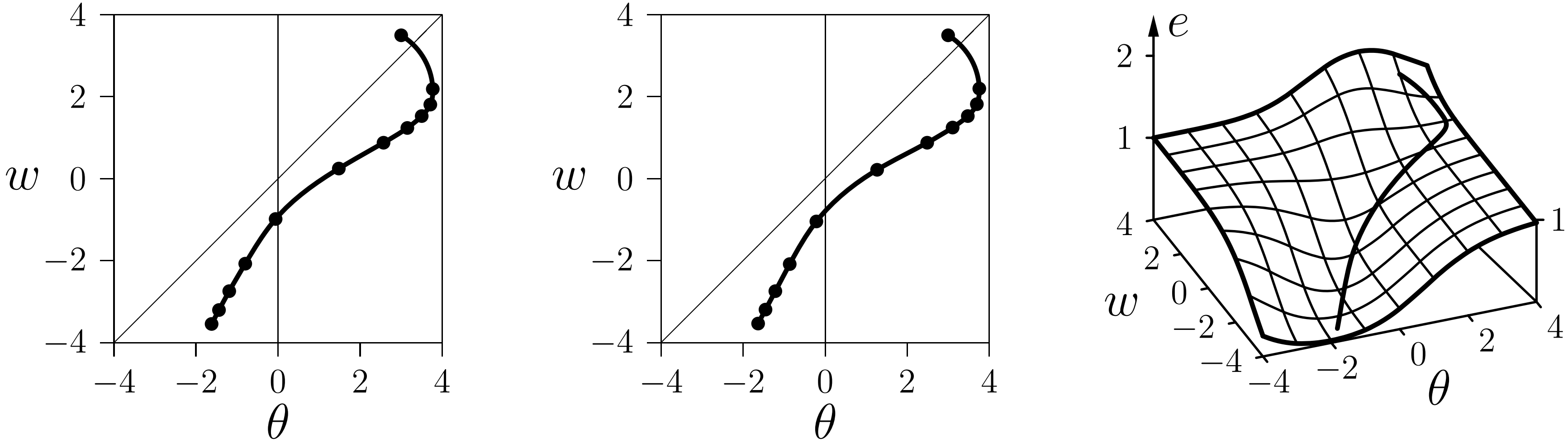
Online Training

epoch	θ	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

Visualization of gradient descent for the negation $\neg x$



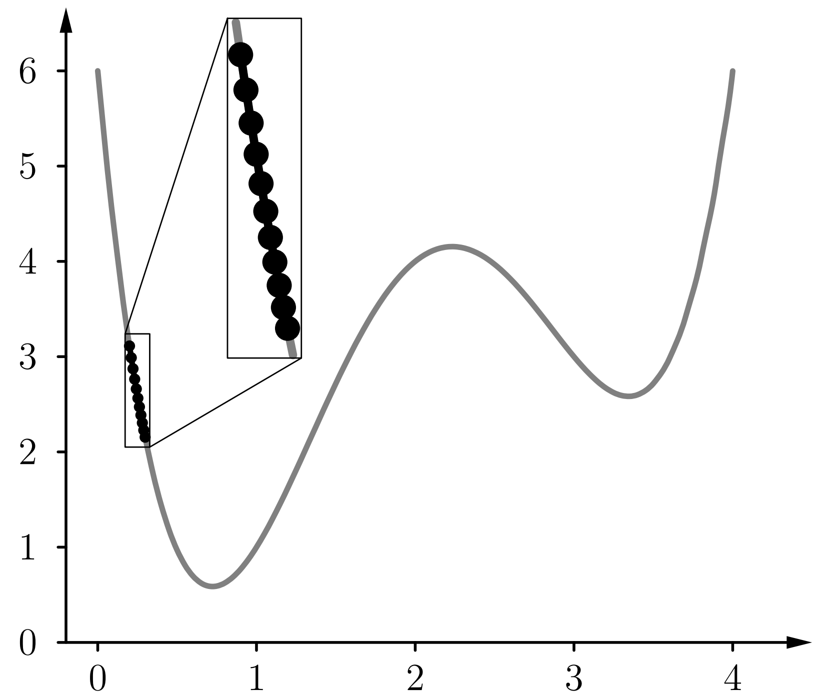
Online Training   Batch Training   Batch Training

- Training is obviously successful.
- Error cannot vanish completely due to the properties of the logistic function.

Gradient Descent: Examples

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

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_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		

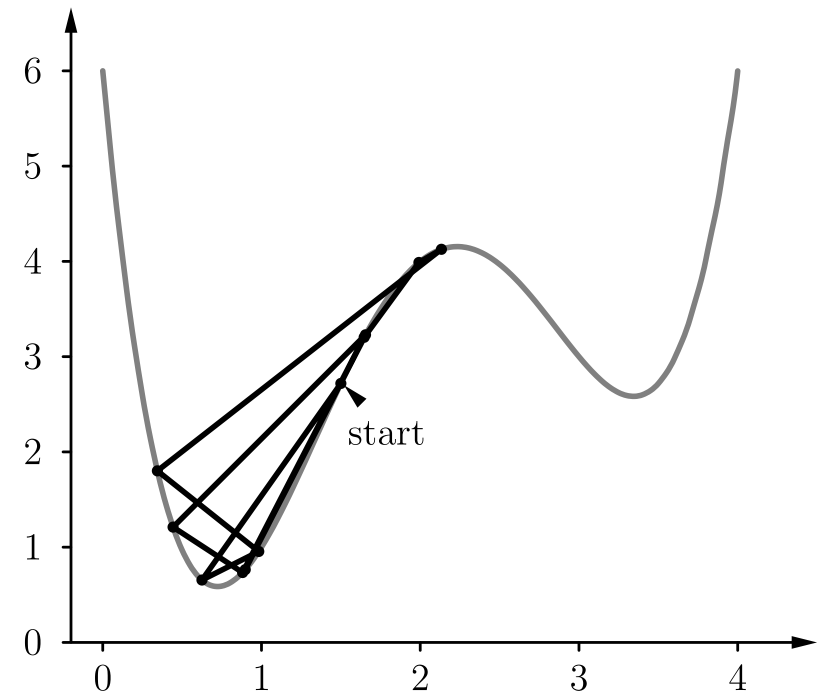


Gradient descent with initial value 0.2 and learning rate 0.001.

Gradient Descent: Examples

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

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_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		

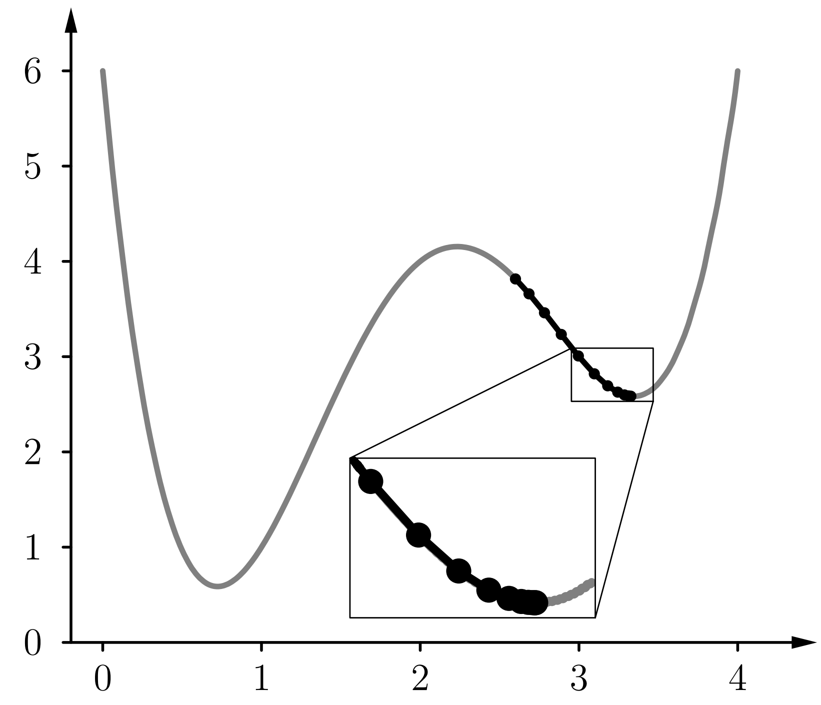


Gradient descent with initial value 1.5 and learning rate 0.25.

Gradient Descent: Examples

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

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_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.

Gradient Descent: Variants

Weight update rule:

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

Standard backpropagation:

$$\Delta w(t) = -\frac{\eta}{2} \nabla_w e(t)$$

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:

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

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

Self-adaptive error backpropagation:

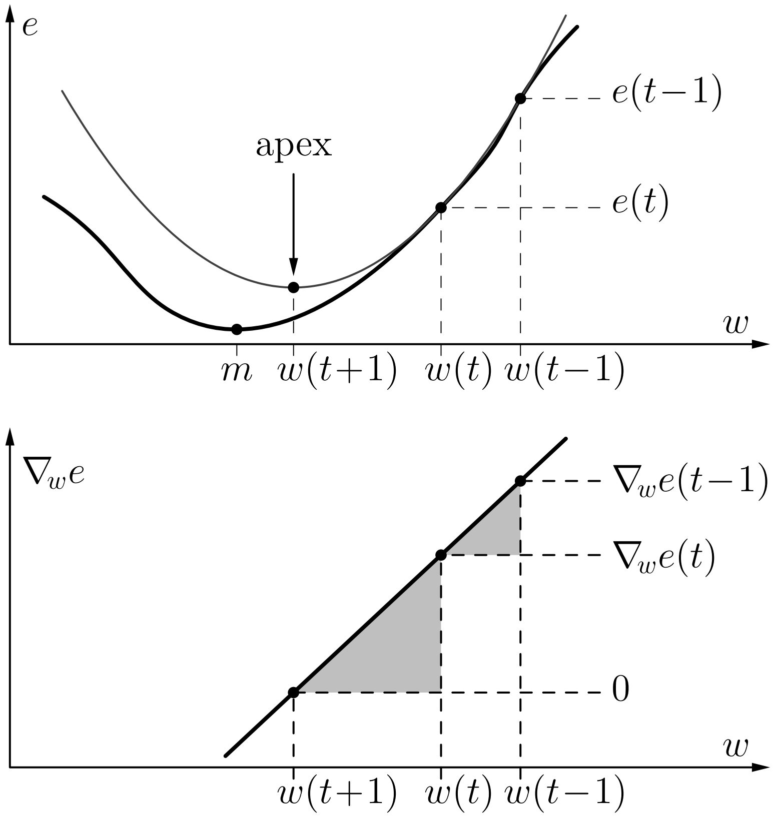
$$\eta_w(t) = \begin{cases} c^- \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \wedge\ \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \eta_w(t-1), & \text{otherwise.} \end{cases}$$

Resilient error backpropagation:

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

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

Quickpropagation



The weight update rule can be derived from the triangles:

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

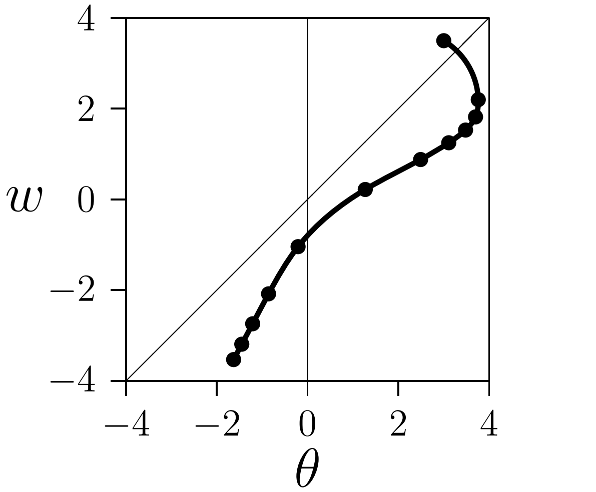
Gradient Descent: Examples

epoch	θ	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

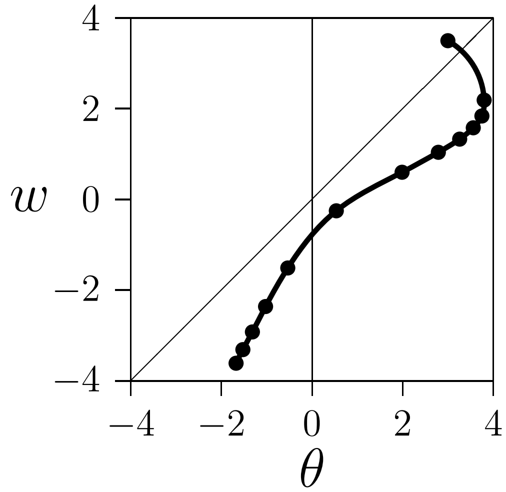
without momentum term

epoch	θ	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

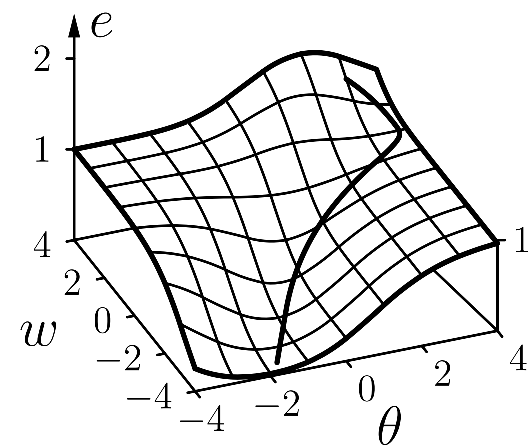
with momentum term



without momentum term



with momentum term



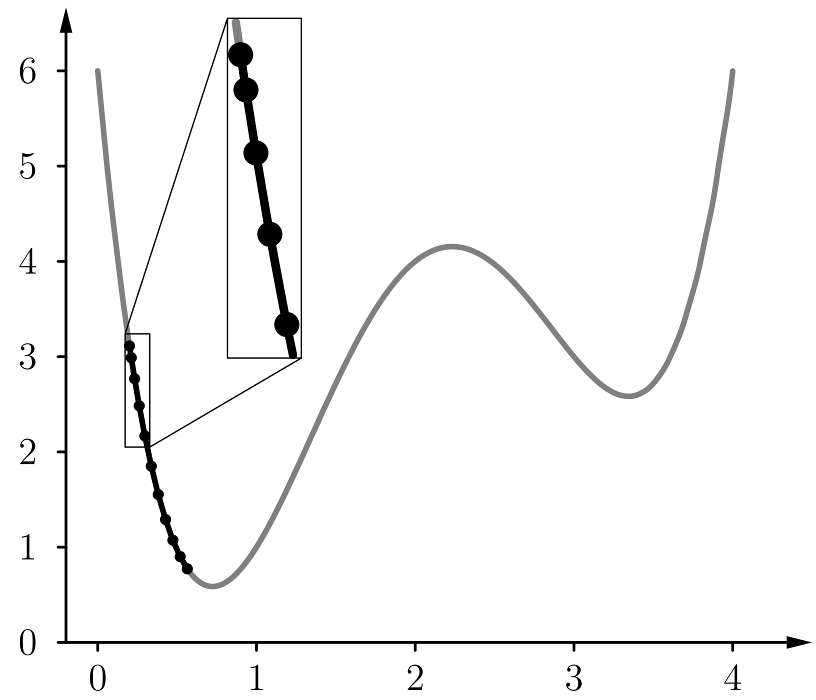
with momentum term

- 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: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_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		

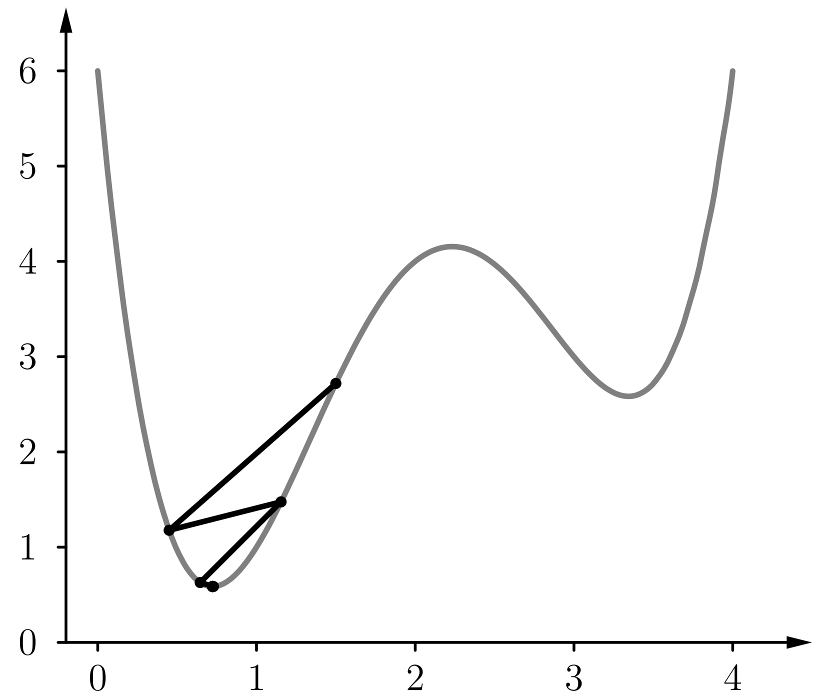


gradient descent with momentum term ($\beta = 0.9$)

Gradient Descent: Examples

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

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_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 ($c^+ = 1.2$, $c^- = 0.5$).

Flat Spot Elimination:

$$\Delta w(t) = -\frac{\eta}{2}\nabla_w e(t) + \zeta$$

- Eliminates slow learning in saturation region of logistic function.
- Counteracts the decay of the error signals over the layers.

Weight Decay:

$$\Delta w(t) = -\frac{\eta}{2}\nabla_w e(t) - \xi\, w(t),$$

- Helps to improve the robustness of the training results.
- Can be derived from an extended error function penalizing large weights:

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

Sensitivity Analysis

Sensitivity Analysis

Problem: the knowledge stored in a neural network is difficult to understand:

- Geometrical (or other) interpretation only feasible for simple networks, but not for complex practical problems
- Difficulties in imagining higher dimensional spaces.
- The neural network becomes a *black box* calculating inputs for outputs in a magical way.

Idea: Determine the influence of single input values on the output of the network

$\rightarrow$ Sensitivity Analysis

Question: How important are different inputs to the network?

Idea: Determine change of output relative to change of input.

$$\forall u \in U_{\text{in}}: \qquad s(u) = \frac{1}{|L_{\text{fixed}}|} \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \frac{\partial \operatorname{out}_v^{(l)}}{\partial \operatorname{ex}_u^{(l)}}.$$

Formal derivation: Apply chain rule.

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{ex}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} \frac{\partial \operatorname{out}_u}{\partial \operatorname{ex}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} \frac{\partial \operatorname{out}_u}{\partial \operatorname{ex}_u}.$$

Simplification: Assume that the output function is the identity.

$$\frac{\partial \operatorname{out}_u}{\partial \operatorname{ex}_u} = 1.$$

Sensitivity Analysis

For the second factor we get the general result:

$$\frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = \frac{\partial}{\partial \operatorname{out}_u} \sum_{p \in \operatorname{pred}(v)} w_{vp} \operatorname{out}_p = \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}.$$

This leads to the recursion formula

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}.$$

However, for the first hidden layer we get

$$\frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = w_{vu}, \qquad \text{therefore} \qquad \frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} w_{vu}.$$

This formula marks the start of the recursion.

Sensitivity Analysis

Consider as usual the special case with

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

The recursion formula is in this case

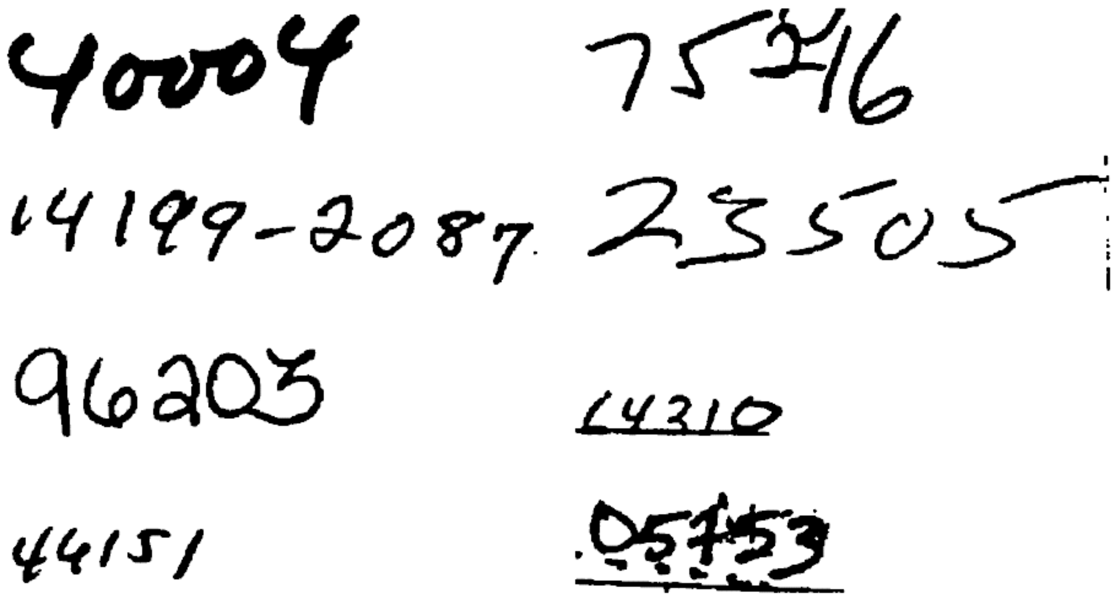
$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \operatorname{out}_v(1 - \operatorname{out}_v) \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}$$

and the anchor of the recursion is

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \operatorname{out}_v(1 - \operatorname{out}_v) w_{vu}.$$

Example: Recognizing handwritten zip codes

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

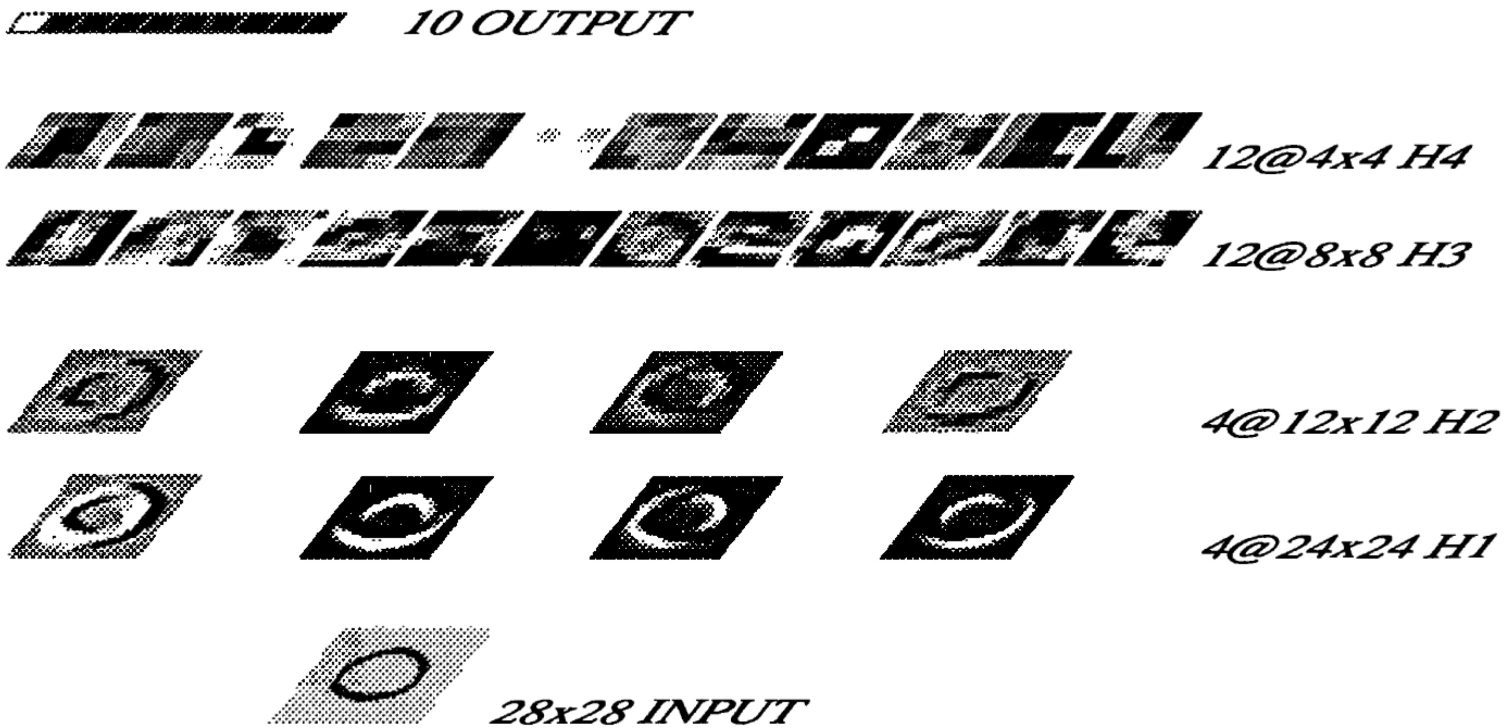
source: Schölkopf und Smola (2002)

- 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 left: 100 validation set digits

Example: Recognizing handwritten zip codes

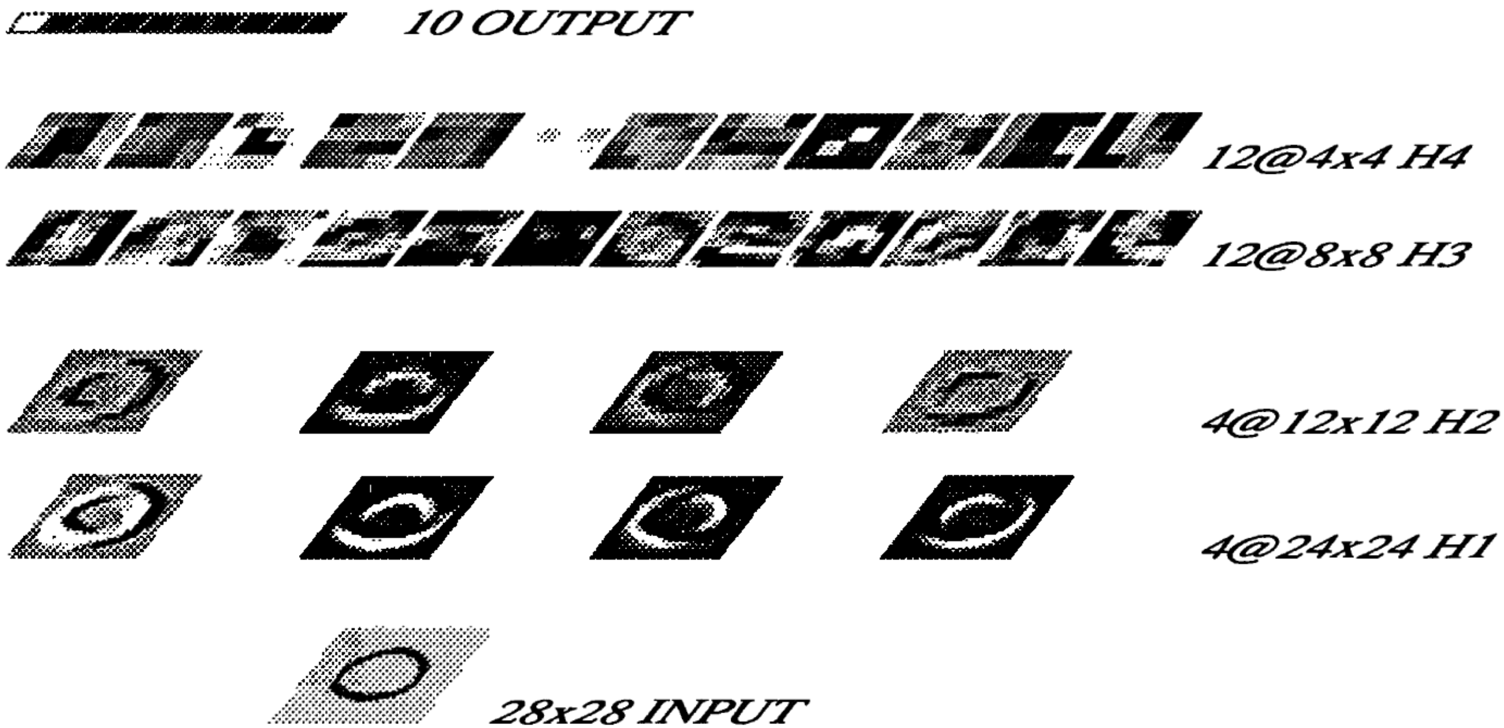
- challenge: every connection ought to be adaptabe (though with strong limitations)
- training by error backpropagation
- input: 16×16 pixel pattern of the normalized digit
- output: 10 neurons, 1 per class
  If a pattern is associated with a class, the output neuron i shall output $+1$ while all other 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

Zip codes: network architecture



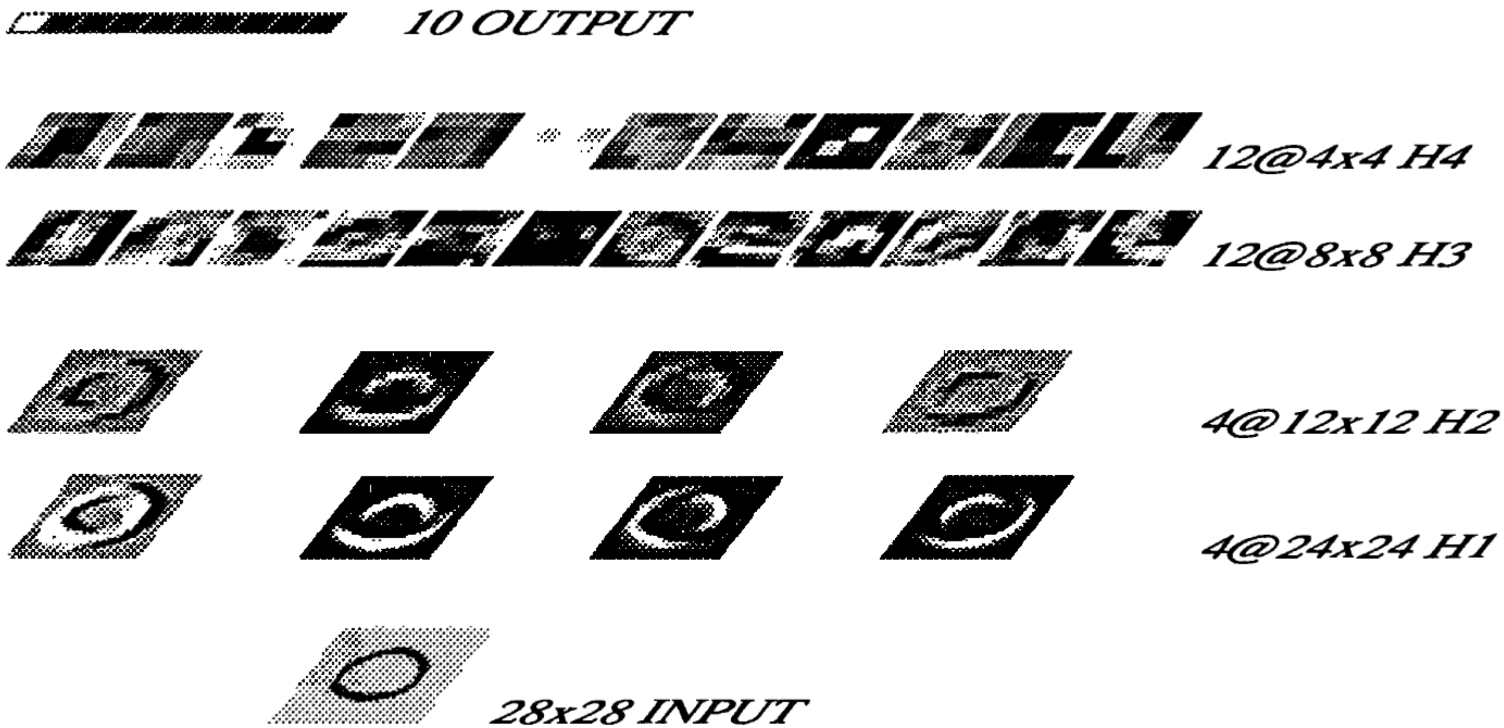
- 4 hiden layers H1, H2, H3 und H4
- neuron groups in H1, H3 share the same weights → less parameters
- neurons in H2, H4 calculate average values → Input values for upper layers
- input layer: enlarge from 16×16 to 28×28 pixel for considerig thresholds

Zip codes: Layer H1



- 4 groups of $24 \times 24 = 576$ neurons assembled as 4 independent feature maps.
- every neuron in a feature map takes a 5×5-Input
- every neuron in a feature map hold the same parameters
- these parameters may differ between the feature maps

Zip codes: Layer H2



- H2 is for forwarding: 4 maps of $12 \times 12 = 144$ 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
- conclusion: H2 is only for forwarding

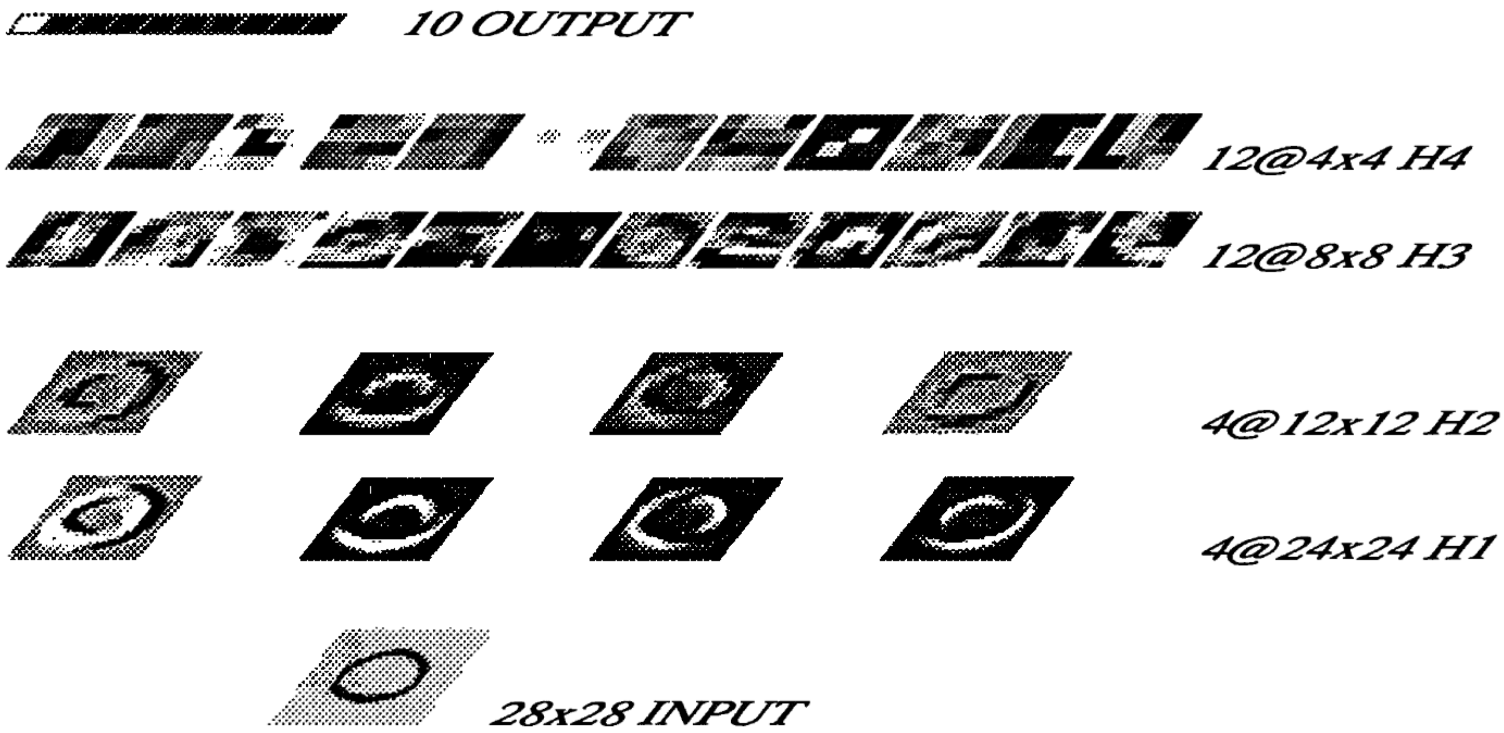
Zip codes: Layer H3

- in H3: 12 feature maps with $8 \times 8 = 64$ neurons each
- connection pattern between H2 and H3 is similar to the pattern between the Input and H1, but with more 2D-maps in H3.
- each neuron consists of one or two 5×5 neighbors (centered around neurons at identical positions of each H2-map)
- maps in H2 that serve as input for h3 are interconnected as follows:

	1	2	3	4	5	6	7	8	9	10	11	12
1	X	X	X		X	X						
2		X	X	X	X	X						
3							X	X	X		X	X
4								X	X	X	X	X

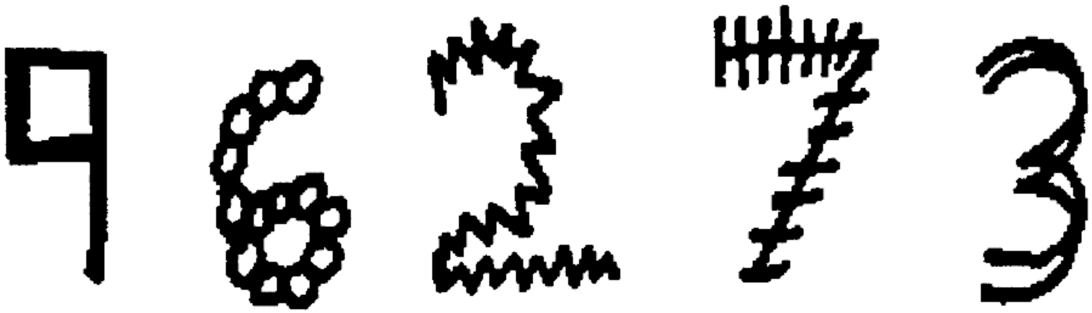
- therefore the network consists of two almost independent modules

Zip codes: Layer H4 and output



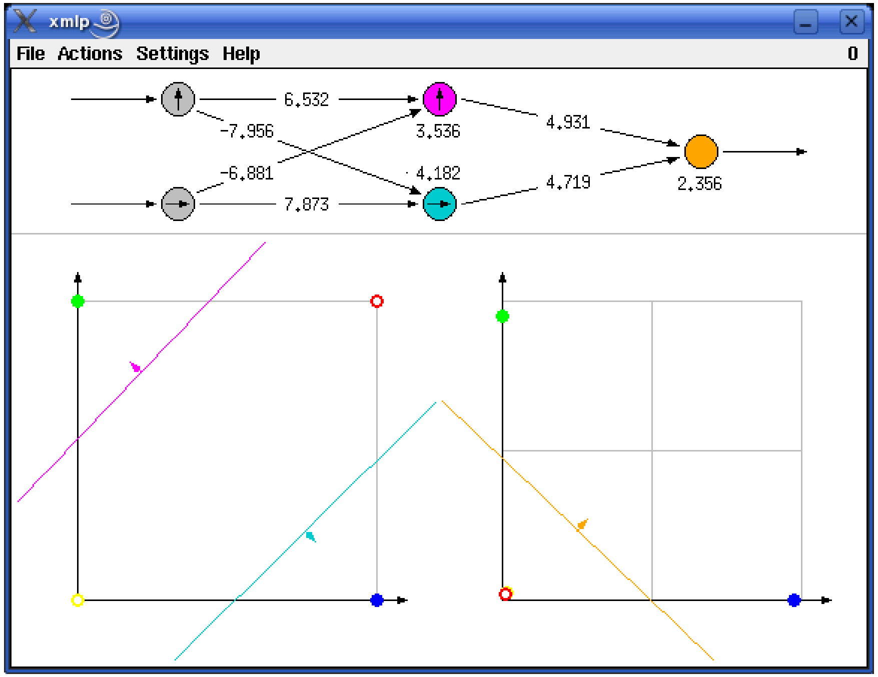
- layer H4 serves the same purpose as H2
- it consists of 12 maps of $4 \times 4 = 16$ neurons each
- the output layer contains 10 neurons and is fully interconnected with H4
- in total: 4635 neurons, 98442 interconnections, 2578 independent parameters
- this structure has been designed with geometrical background knowledge of digit pattern recognition

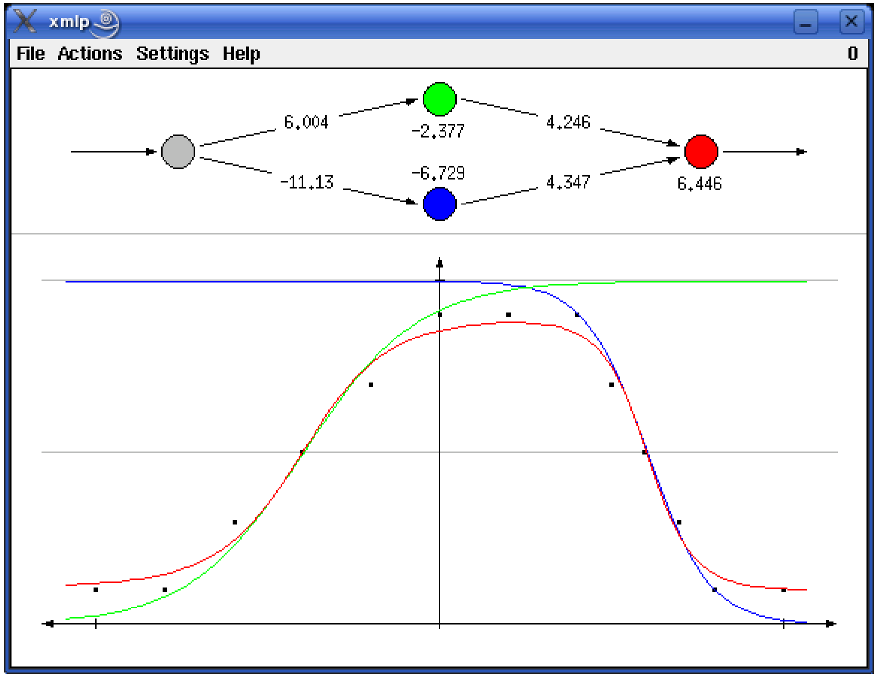
Zip codes: Results



atypical digits that have been recognized correctly

- training error after 30 training iterations: 1,1%
- validation error: 3,4%
- all classification errors occur with handwritten digits
- compare human error: 2,5%
- training on a SUN SPARC machine in 1989 took three days
- the trained network was implemented on a hardware chip
- a coprocessor in a computer with video camera is able to classify more than 10 digits per second
- or 30 classifications per second on normalized digits

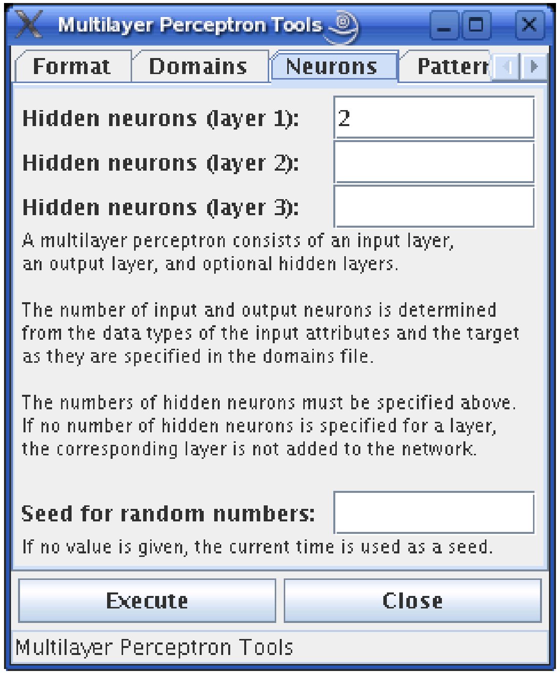


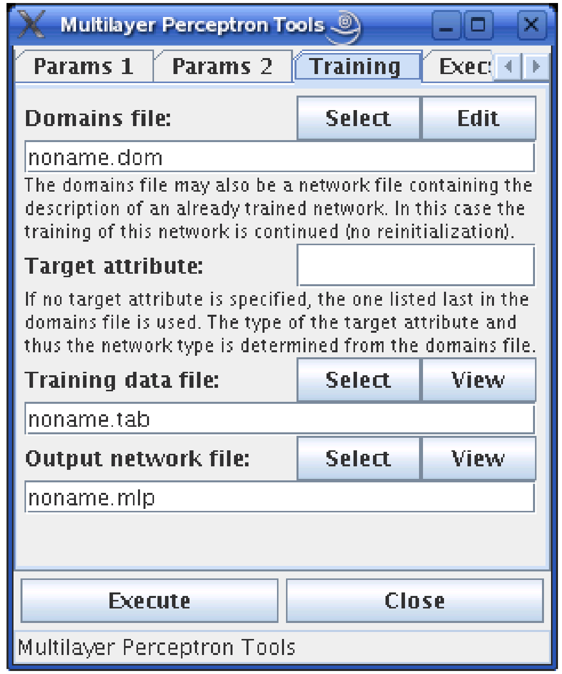


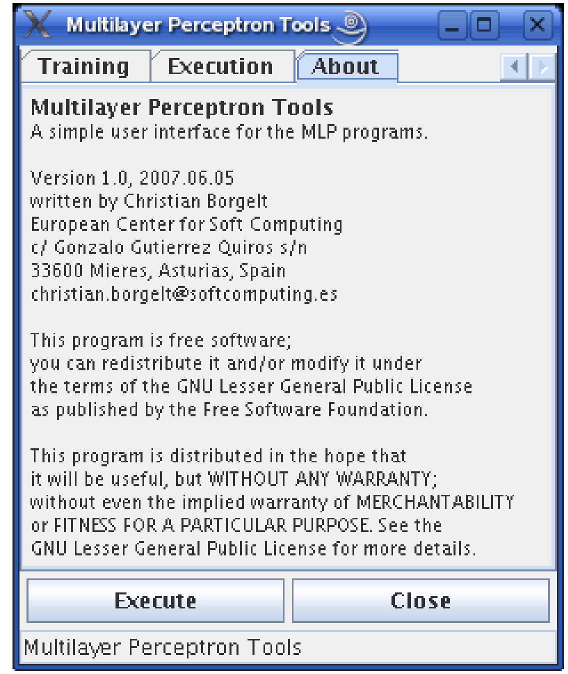
Demonstration of multilayer perceptron training:

- Visualization of the training process
- Biimplication and Exclusive Or, two continuous functions
- http://www.borgelt.net/mlpd.html

Multilayer Perceptron Software: mlp/mlpgui







Software for training general multilayer perceptrons:

- Command line version written in C, fast training
- Graphical user interface in Java, easy to use
- http://www.borgelt.net/mlp.html, http://www.borgelt.net/mlpgui.html