

Training Multi-layer Perceptrons

Training Multi-layer 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**.

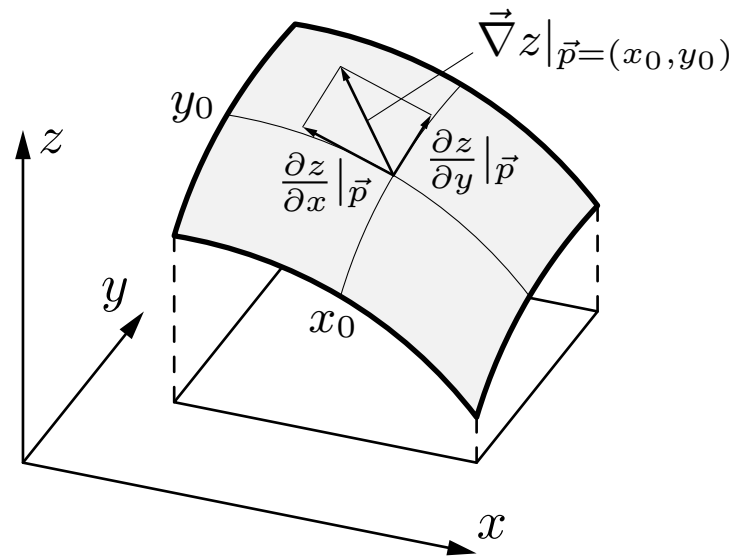


Illustration of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0) .
It is $\vec{\nabla} z |_{(x_0, y_0)} = \left(\frac{\partial z}{\partial x} |_{x_0}, \frac{\partial z}{\partial y} |_{y_0} \right)$. ($\vec{\nabla}$ is a differential operator called “nabla” or “del”.)

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

(here and in the following: extended weight vector $\vec{w}_u = (-\theta_u, w_{up_1}, \dots, w_{up_n})$):

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

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:

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

Since $\text{net}_u^{(l)} = \vec{w}_u^\top \vec{\text{in}}_u^{(l)}$ (note: extended input vector $\vec{\text{in}}_u^{(l)} = (1, \text{in}_{p_1 u}^{(l)}, \dots, \text{in}_{p_n u}^{(l)})$), we have for the second factor

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

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

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

that is, the sum of the errors over all output neurons.

Gradient Descent: Formal Approach

Therefore we have

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

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

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

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

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

Therefore we have for the gradient

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

and thus for the weight change

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

Gradient Descent: Formal Approach

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

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

Consider the special case with

- output function is the identity,
- activation function is logistic, that is, $f_{\text{act}}(x) = \frac{1}{1+e^{-x}}$.

The first assumption yields

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

Gradient Descent: Formal Approach

For a logistic activation function we have

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

and therefore

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

The resulting weight change is therefore

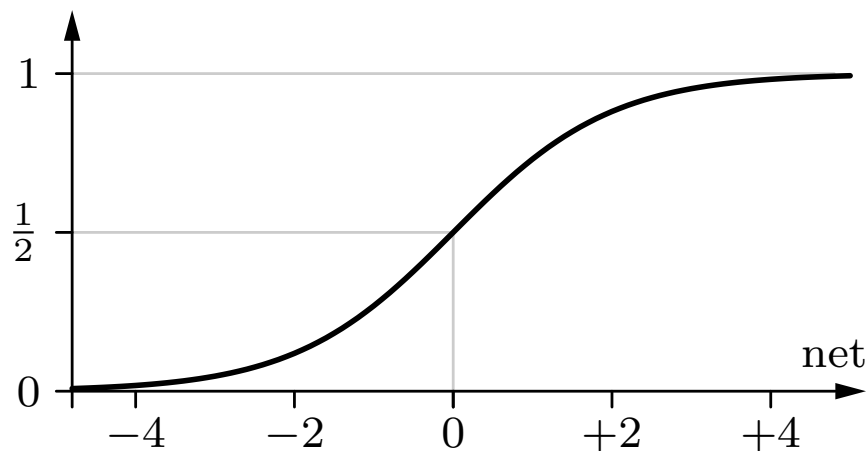
$$\Delta \vec{w}_u^{(l)} = \eta \left(o_u^{(l)} - \text{out}_u^{(l)} \right) \text{out}_u^{(l)} (1 - \text{out}_u^{(l)}) \vec{\text{in}}_u^{(l)},$$

which makes the computations very simple.

Gradient Descent: Formal Approach

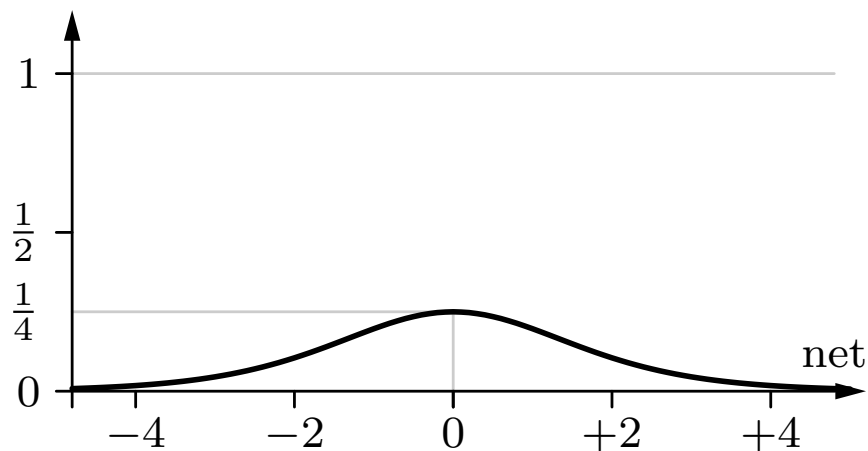
logistic activation function:

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



derivative of logistic function:

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

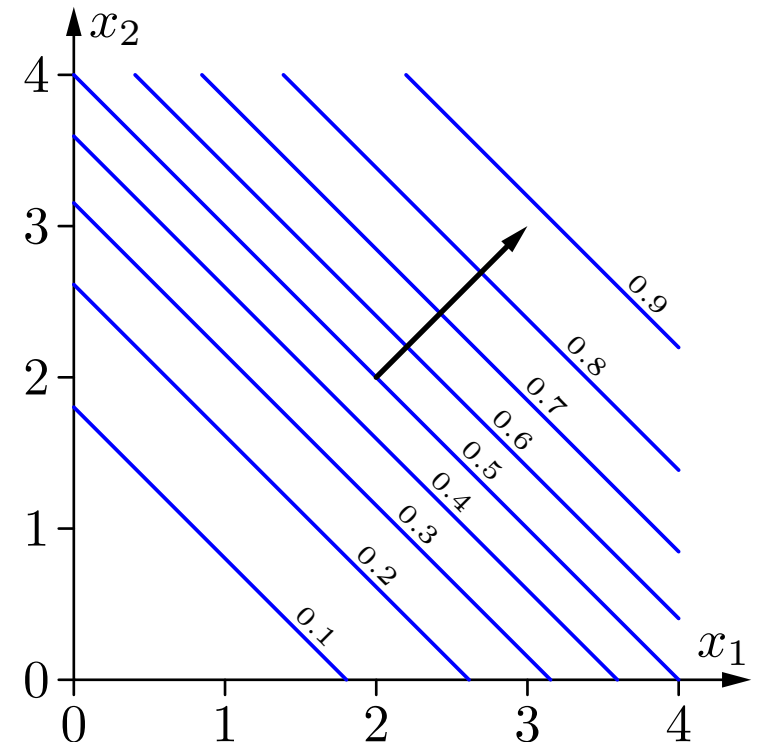
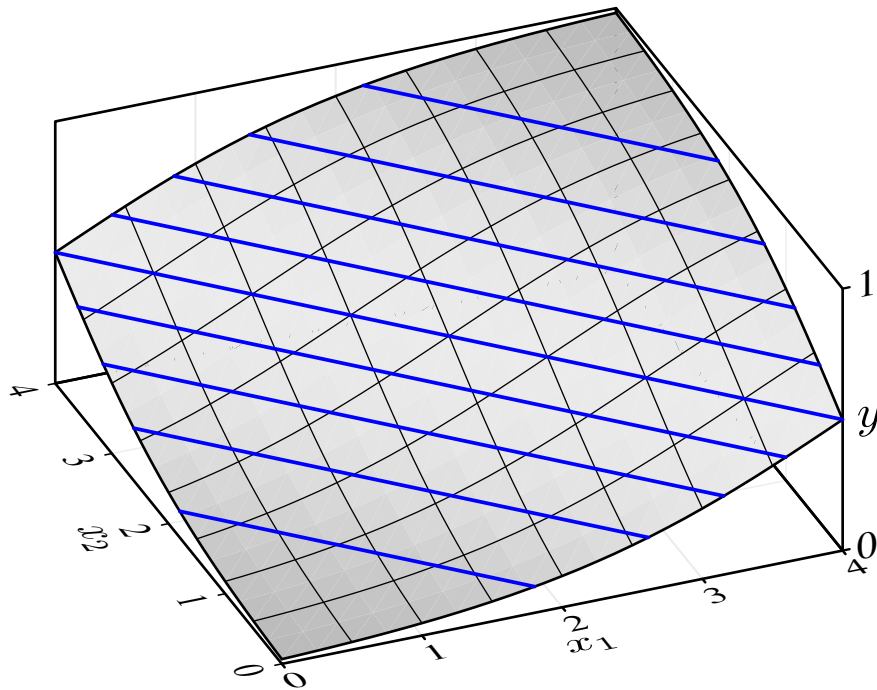


- If a logistic activation function is used (shown on left), the weight changes are proportional to $\lambda_u^{(l)} = \text{out}_u^{(l)} (1 - \text{out}_u^{(l)})$ (shown on right; see preceding slide).
- Weight changes are largest, and thus the training speed highest, in the vicinity of $\text{net}_u^{(l)} = 0$. Far away from $\text{net}_u^{(l)} = 0$, the gradient becomes (very) small (“saturation regions”) and thus training (very) slow.

Reminder: Two-dimensional Logistic Function

Example logistic function for two arguments x_1 and x_2 :

$$y = \frac{1}{1 + \exp(4 - x_1 - x_2)} = \frac{1}{1 + \exp(4 - (1, 1)^\top (x_1, x_2))}$$



The blue lines have show where the logistic function has a certain value in $\{0.1, \dots, 0.9\}$.

Error Backpropagation

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

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

We apply the chain rule to obtain

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

Exchanging the sums yields

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

Error Backpropagation

Consider the network input

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

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

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

The result is the recursive equation (error backpropagation)

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

Error Backpropagation

The resulting formula for the weight change is

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

Consider again the special case with

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

The resulting formula for the weight change is then

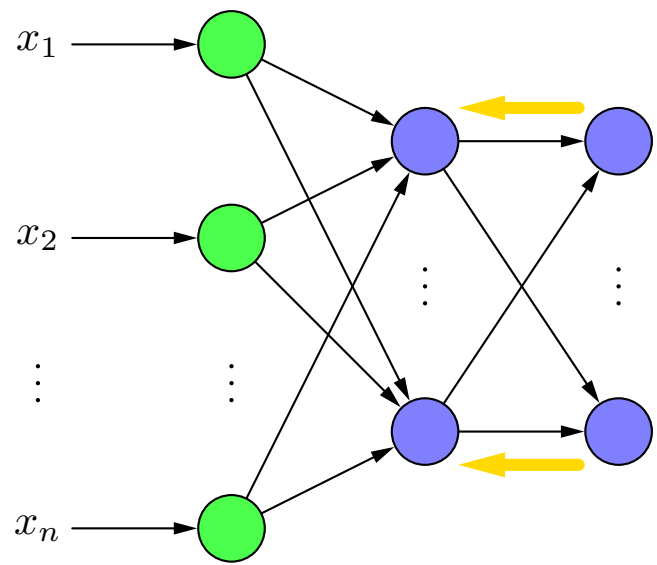
$$\Delta \vec{w}_u^{(l)} = \eta \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su} \right) \text{out}_u^{(l)} (1 - \text{out}_u^{(l)}) \vec{\text{in}}_u^{(l)}.$$

Error Backpropagation: Cookbook Recipe

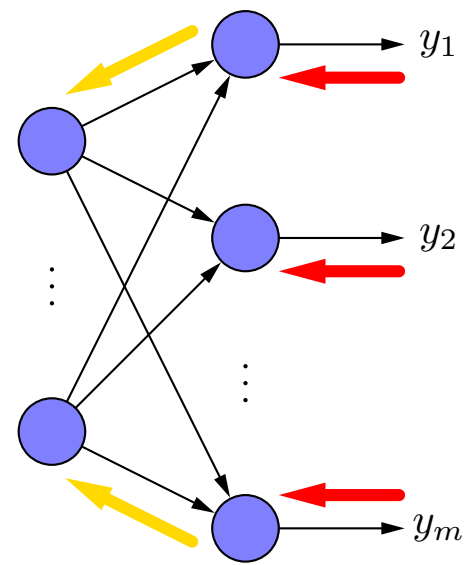
$$\forall u \in U_{\text{in}} : \text{out}_u^{(l)} = \text{ext}_u^{(l)}$$

forward propagation:

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



...



- logistic activation function
- implicit bias value

error factor:

backward propagation:

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

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

activation derivative:

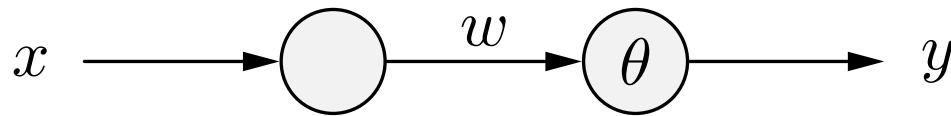
$$\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)} \right)$$

weight change:

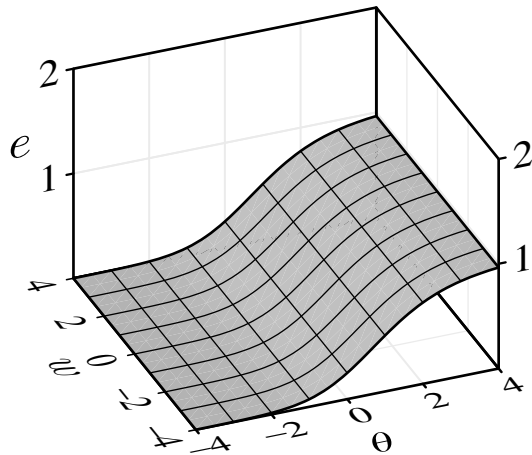
$$\Delta w_{up}^{(l)} = \eta \delta_u^{(l)} \text{out}_p^{(l)}$$

Gradient Descent: Examples

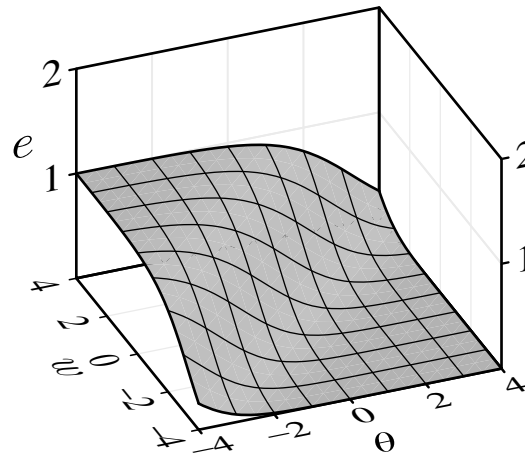
Gradient descent training for the negation $\neg x$



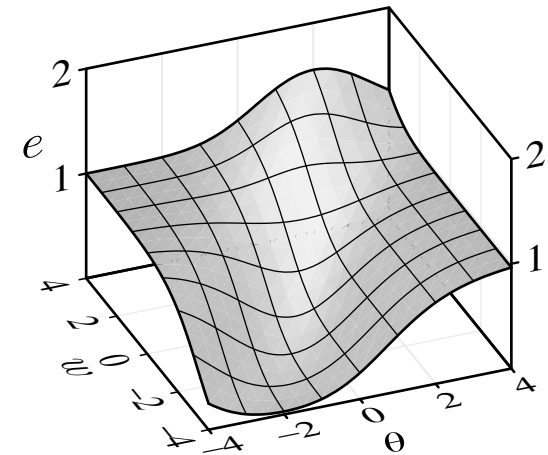
x	y
0	1
1	0



error for $x = 0$



error for $x = 1$



sum of errors

Note: error for $x = 0$ and $x = 1$ is effectively the squared logistic activation function!

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

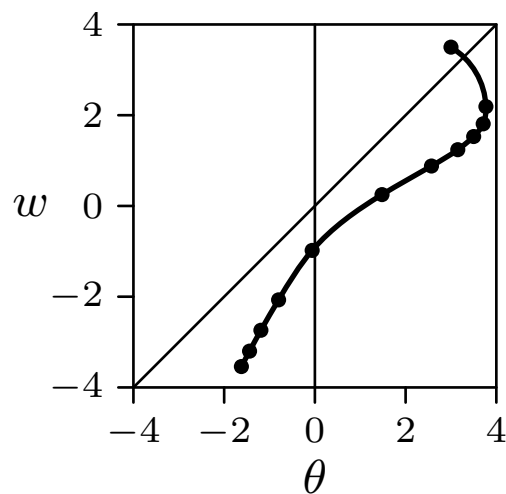
Online Training

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

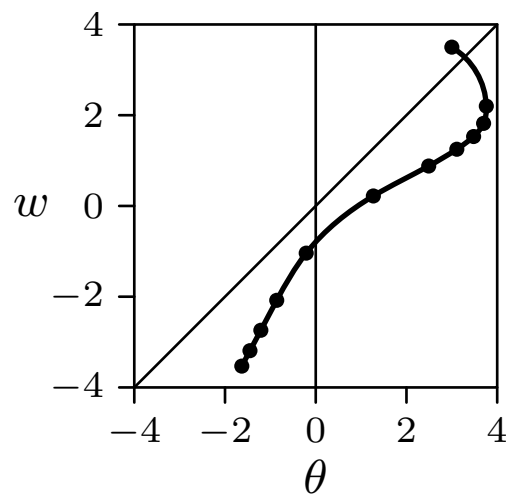
Batch Training

Gradient Descent: Examples

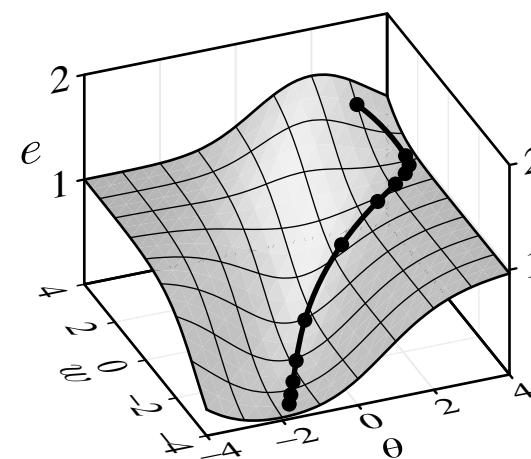
Visualization of gradient descent for the negation $\neg x$



Online Training



Batch Training



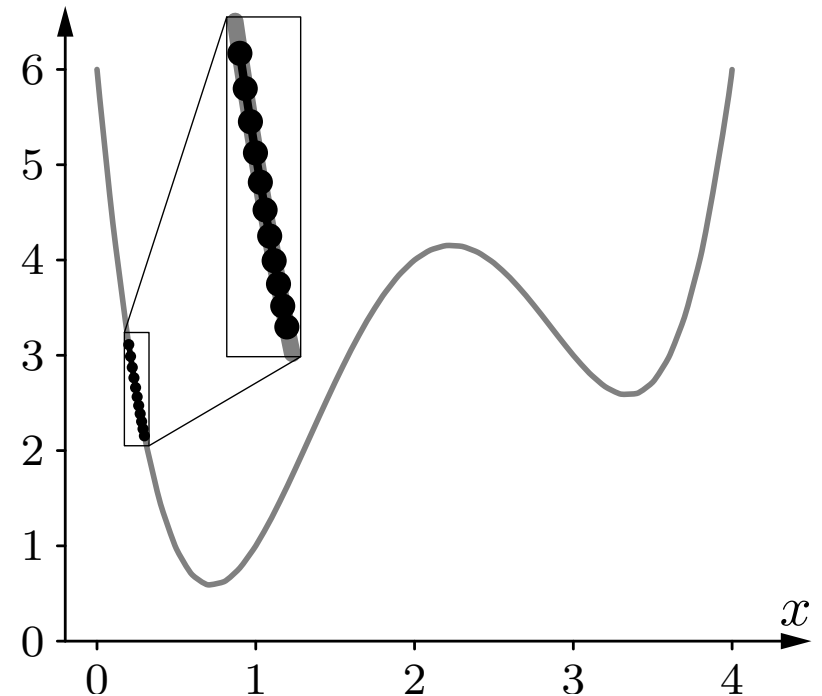
Batch Training

- 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		



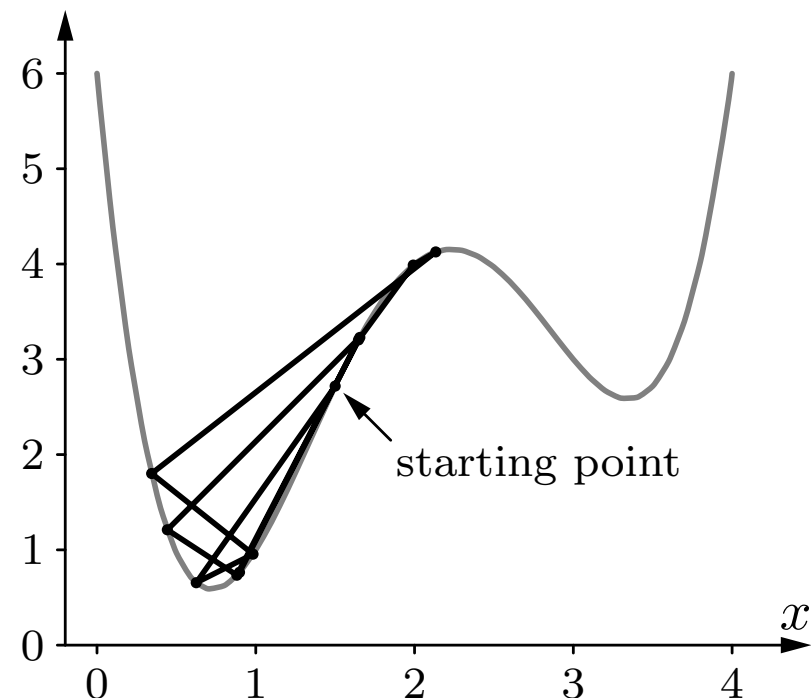
Gradient descent with initial value 0.2 and learning rate 0.001.

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		



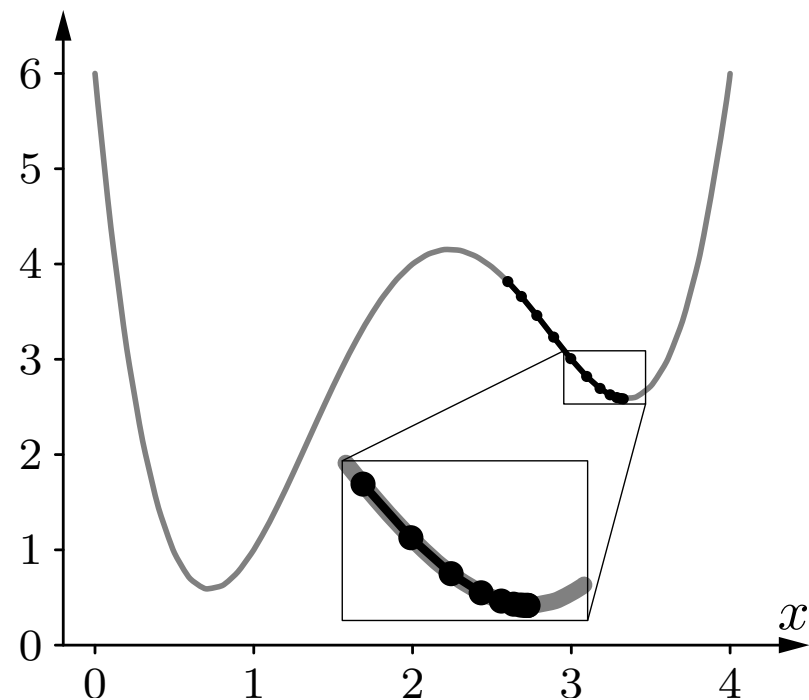
Gradient descent with initial value 1.5 and learning rate 0.25.

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:

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

Fixed step width (grid), only sign of gradient (direction) is evaluated.

Momentum term:

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

Part of previous change is added, may lead to accelerated training ($\beta \in [0.5, 0.95]$).

Gradient Descent: Variants

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:

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

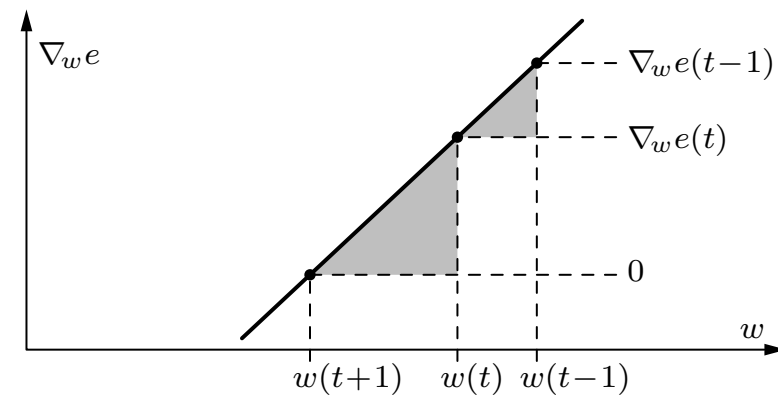
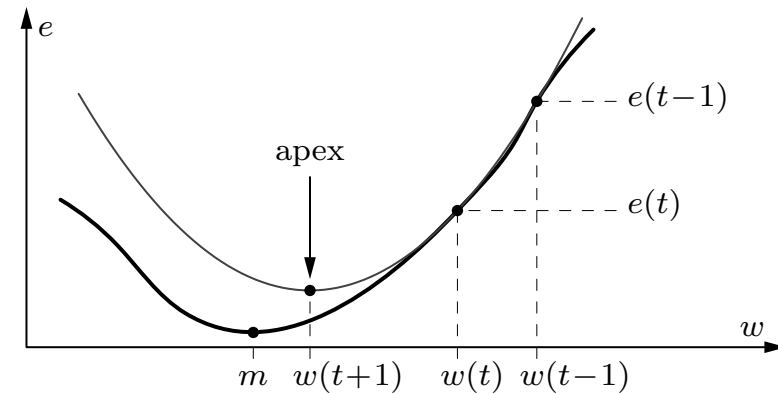
Typical values: $c^- \in [0.5, 0.7]$ and $c^+ \in [1.05, 1.2]$.

Gradient Descent: Variants

Quickpropagation

The weight update rule can be derived from the triangles:

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



Gradient Descent: Examples

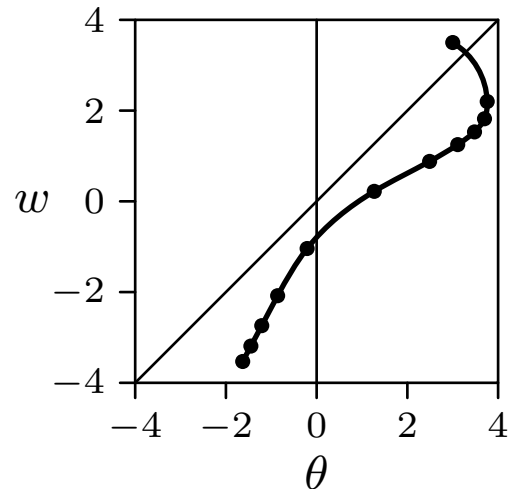
epoch	θ	w	error
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	-0.21	-1.04	0.292
160	-0.86	-2.08	0.140
180	-1.21	-2.74	0.084
200	-1.45	-3.19	0.058
220	-1.63	-3.53	0.044

without momentum term

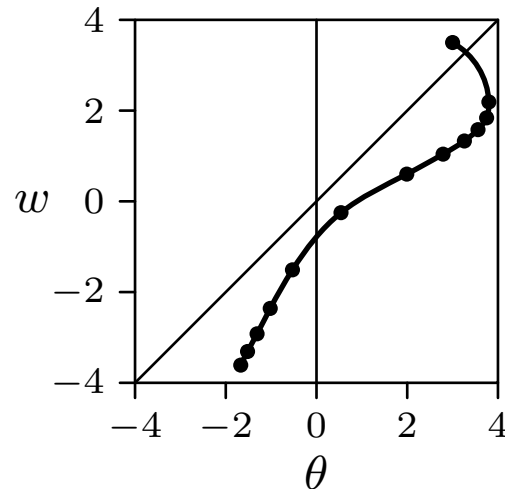
epoch	θ	w	error
0	3.00	3.50	1.295
10	3.80	2.19	0.984
20	3.75	1.84	0.971
30	3.56	1.58	0.960
40	3.26	1.33	0.943
50	2.79	1.04	0.910
60	1.99	0.60	0.814
70	0.54	-0.25	0.497
80	-0.53	-1.51	0.211
90	-1.02	-2.36	0.113
100	-1.31	-2.92	0.073
110	-1.52	-3.31	0.053
120	-1.67	-3.61	0.041

with momentum term ($\beta = 0.9$)

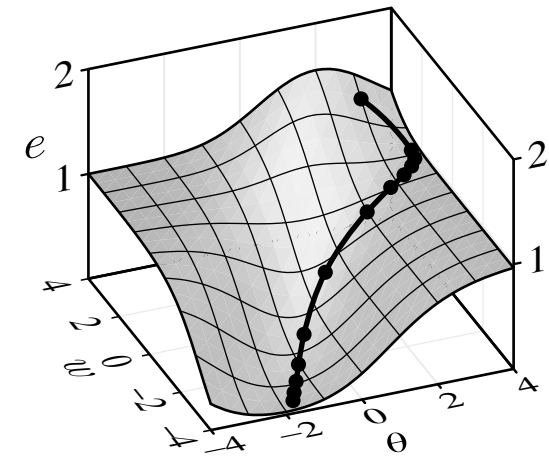
Gradient Descent: Examples



without momentum term



with momentum term



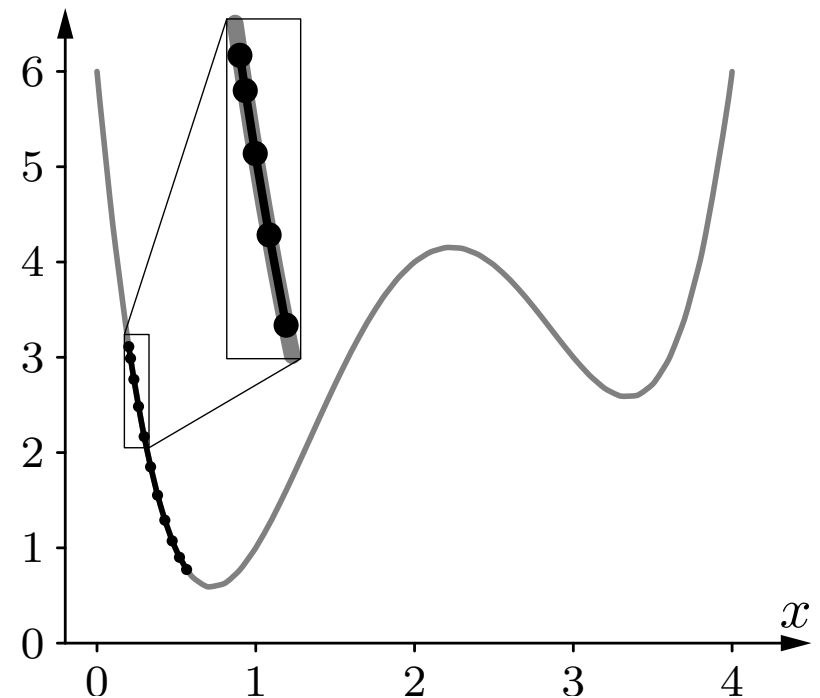
with momentum term

- Dots show position every 20 (without momentum term) or every 10 epochs (with momentum term).
- Learning with a momentum term ($\beta = 0.9$) 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		



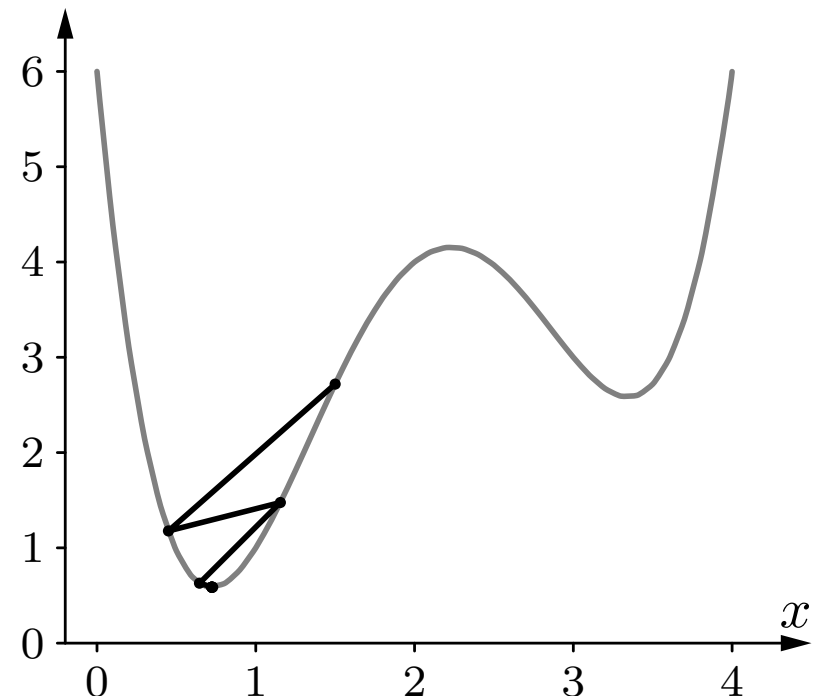
Gradient descent with initial value 0.2, learning rate 0.001
and momentum term $\beta = 0.9$.

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 initial value 1.5, initial learning rate 0.25, and self-adapting learning rate ($c^+ = 1.2$, $c^- = 0.5$).

Other Extensions of Error Backpropagation

Flat Spot Elimination:

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

- Eliminates slow learning in saturation region of logistic function ($\zeta \approx 0.1$).
- 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 ($\xi \leq 10^{-3}$).
- Can be derived from an extended error function penalizing large weights:

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

Number of Hidden Neurons

- Note that the approximation theorem only states that there *exists* a number of hidden neurons and weight vectors \vec{v} and \vec{w}_i and thresholds θ_i , but not how they are to be chosen for a given ε of approximation accuracy.
- For a single hidden layer the following **rule of thumb** is popular:
number of hidden neurons = (number of inputs + number of outputs) / 2
- Better, though computationally expensive approach:
 - Randomly split the given data into two subsets of (about) equal size, the **training data** and the **validation data**.
 - Train multi-layer perceptrons with different numbers of hidden neurons on the training data and evaluate them on the validation data.
 - Repeat the random split of the data and training/evaluation many times and average the results over the same number of hidden neurons. Choose the number of hidden neurons with the best average error.
 - Train a final multi-layer perceptron on the whole data set.

Number of Hidden Neurons

Principle of training data/validation data approach:

- **Underfitting:** If the number of neurons in the hidden layer is too small, the multi-layer perceptron may not be able to capture the structure of the relationship between inputs and outputs precisely enough due to a lack of parameters.
- **Overfitting:** With a larger number of hidden neurons a multi-layer perceptron may adapt not only to the regular dependence between inputs and outputs, but also to the accidental specifics (errors and deviations) of the training data set.
- Overfitting will usually lead to the effect that the error a multi-layer perceptron yields on the validation data will be (possibly considerably) greater than the error it yields on the training data.

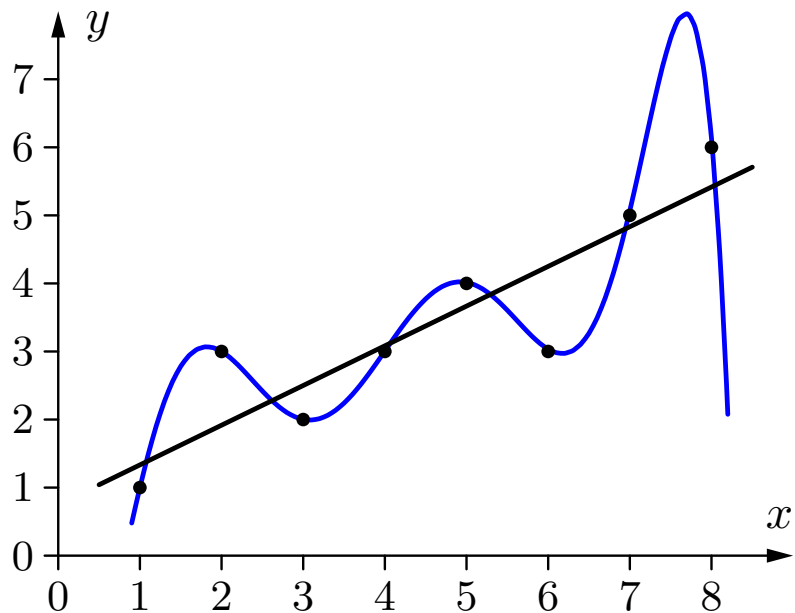
The reason is that the validation data set is likely distorted in a different fashion than the training data, since the errors and deviations are random.

- Minimizing the error on the validation data by properly choosing the number of hidden neurons prevents both under- and overfitting.

Number of Hidden Neurons: Avoid Overfitting

- Objective: select the model that best fits the data, **taking the model complexity into account.**

The more complex the model, the better it usually fits the data.



black line:
regression line
(2 free parameters)

blue curve:
7th order regression polynomial
(8 free parameters)

- The blue curve fits the data points perfectly, **but it is not a good model.**

Number of Hidden Neurons: Cross Validation

- The described method of iteratively splitting the data into training and validation data may be referred to as **cross validation**.
- However, this term is more often used for the following specific procedure:
 - The given data set is split into n parts or subsets (also called folds) of about equal size (so-called **n-fold cross validation**).
 - If the output is nominal (also sometimes called symbolic or categorical), this split is done in such a way that the relative frequency of the output values in the subsets/folds represent as well as possible the relative frequencies of these values in the data set as a whole. This is also called **stratification** (derived from the Latin *stratum*: layer, level, tier).
 - Out of these n data subsets (or folds) n pairs of training and validation data set are formed by using one fold as a validation data set while the remaining $n - 1$ folds are combined into a training data set.

Number of Hidden Neurons: Cross Validation

- The advantage of the cross validation method is that one random split of the data yields n different pairs of training and validation data set.
- An obvious disadvantage is that (except for $n = 2$) the size of the training and the test data set are considerably different, which makes the results on the validation data statistically less reliable.
- It is therefore only recommended for sufficiently large data sets or sufficiently small n , so that the validation data sets are of sufficient size.
- Repeating the split (either with $n = 2$ or greater n) has the advantage that one obtains many more training and validation data sets, leading to more reliable statistics (here: for the number of hidden neurons).
- The described approaches fall into the category of **resampling methods**.
- Other well-known statistical resampling methods are **bootstrap**, **jackknife**, **subsampling** and **permutation test**.

Avoiding Overfitting: Alternatives

- An alternative way to prevent overfitting is the following approach:
 - During training the performance of the multi-layer perceptron is evaluated after each epoch (or every few epochs) on a **validation data set**.
 - While the error on the training data set should always decrease with each epoch, the error on the validation data set should, after decreasing initially as well, increase again as soon as overfitting sets in.
 - At this moment training is terminated and either the current state or (if available) the state of the multi-layer perceptron, for which the error on the validation data reached a minimum, is reported as the training result.
- Furthermore a stopping criterion may be derived from the shape of the error curve on the training data over the training epochs, or the network is trained only for a fixed, relatively small number of epochs (also known as **early stopping**).
- Disadvantage: these methods stop the training of a complex network early enough, rather than adjust the complexity of the network to the “correct” level.