Block 11 (Freitag 1.3.2024)

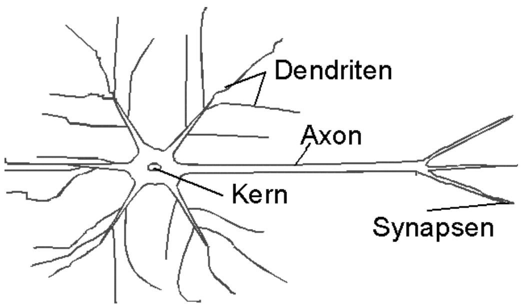
VIDEO: `video11a_nn_intro`

12 Neural Networks

Fundamental research: how does brain function?
Application: efficient self-learning algorithms (hand writing recognition, optimization, generalisation, ...)
Brain has about 10^{11} neurons. Communication by electrical impulses.

Structure neuron:

Dendritens: input signals, up to 2×10^5 per cell
Axon: output signal
Synapsens: couple axon to dendrites of other cells, up to 10^4 per cell, in total about 10^{15}, strengths can be changed.



von `www.lunaticpride.de`

Modelling by McCulloch/Pitts neurons (W.S. McCulloch and W. Pitts, Bull. Math. Biophys. 1943) [9]

- L inputs $x_i = 0, 1$ (silent/active)
- Strengths $w_i \in \mathbb{R}$ of synapses
- Threshold value s
- Output signal

$$y = \theta\left(\sum_{i=1}^{L} w_i x_i - s\right) \tag{65}$$

$\theta(x) = 1$ for $x \geq 0$ $\theta(x) = 0$ else.

Logical functions AND/NOT can be realised
$\Rightarrow$ arbitrary logical functions.

Supervised leerning: given function $y^{(\text{target})}(\underline{x})$, and input $\underline{x}$.

Contribution to weights: Hebb's learning rule (D. Hebb, Wiley, 1949) [10] ($\epsilon > 0$: "learning parameter")

$$\Delta w_i = \epsilon y^{(\text{target})}(\underline{x})x_i \tag{66}$$

Can be performed for many input vectors $\{\underline{x}^{(1)}, \underline{x}^{(2)}, \ldots, \underline{x}^{(n)}\}$.

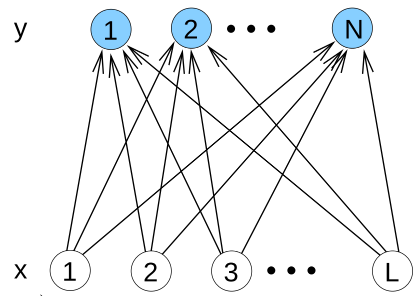
VIDEO: `video11b_perceptron`

12.1 Perzeptron

(Also) for classification of input petterns $\underline{x}$ and generalisation
$\to N$ output $y_r \in \{0, 1\}$ $(r = 1, \ldots, N)$ according to

$$y_r = \theta\left(\sum_{i=0}^{L} w_{ri}x_i\right) \tag{67}$$

$(s \leftrightarrow -w_{r0}$ via $x_0 = 1)$
Each output is independent of the others, pure layered structure



Perceptron learning algorithm ("training phase")

- start with random weights
- use different training vectors x.

  For each incorrect output $y_r(\underline{x})$ changed weights:

$$\Delta w_{ri} = \epsilon \cdot (y_r^{(\text{target})}(\underline{x}) - y_r(\underline{x})) \cdot x_i \tag{68}$$

As C function (see `perceptron.c`)

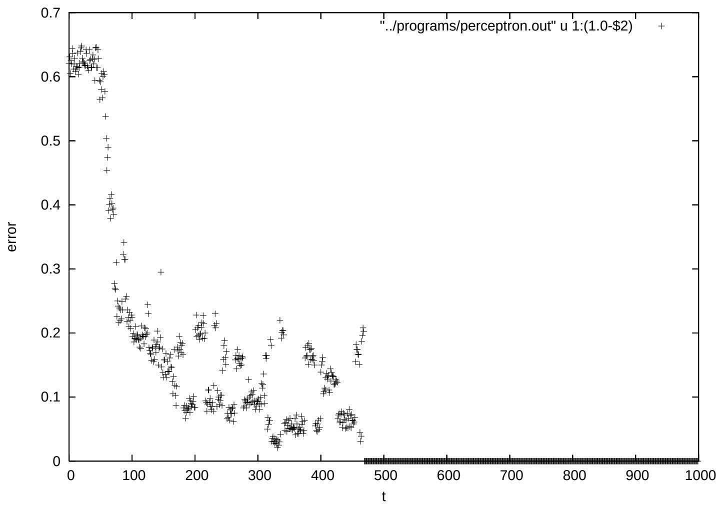
```
/***************** perceptron_learning() ****************/
/** Peforms 'K' steps of learning algorithm:           **/
/** generate random vector and adjust weights using    **/
/** parameter 'epsilon' to learn function 'f'          **/
/** PARAMETERS: (*)= return-paramter                   **/
/**           L: number of (real) values               **/
/**       (*) w: weight vector                         **/
/**     epsilon: learning rate                         **/
/**           f: target function                       **/
/**           K: number of iterations                  **/
/** RETURNS:                                           **/
/**     (nothing)                                      **/
/********************************************************/
void perceptron_learning(int L, double *w, double epsilon,
 int (*f)(int, int *), int K )
{
  int step, t;                                /* loop counters */
  int *x;                                      /* input vector */
  int y, y_wanted;                            /* output values */


  x = (int *) malloc( (L+1)*sizeof(int));
  x[0] = 1;                             /* bit 0 <-> threshold */
  for(step=0; step<K; step++)          /* main learning loop */
  {
    random_vector(L, x);
    y = output_neuron(L, x, w);
    y_wanted = f(L, x);
    if(y != y_wanted)
      for(t=0; t<=L; t++)                    /* adjust weights */
        w[t] += epsilon*(y_wanted- y)*x[t];
  }
  free(x);
}
```

Test: function Majority function

$$f(x) = \begin{cases} 1 & \text{more than half of the bits is 1} \\ 0 & \text{else} \end{cases} \tag{69}$$

For $L = 10$, failure rate as function of number of learning iterations:



resulting weights

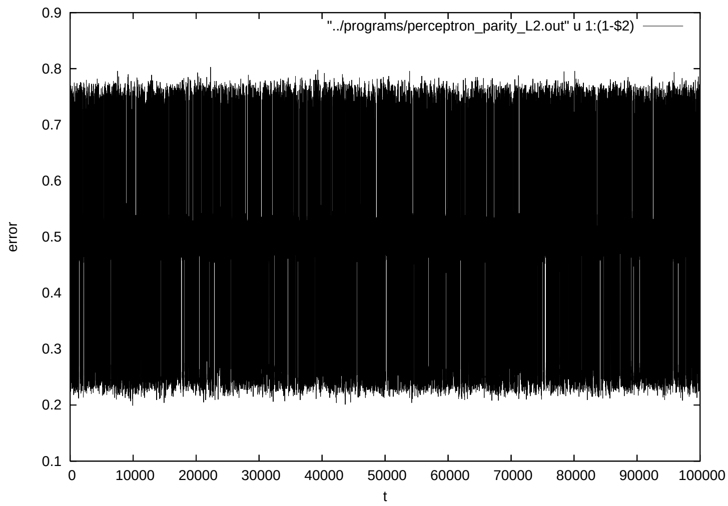
```
# w[0] = -1.150000
# w[1] = 0.250000
# w[2] = 0.200000
# w[3] = 0.200000
# w[4] = 0.200000
# w[5] = 0.200000
# w[6] = 0.200000
# w[7] = 0.200000
# w[8] = 0.250000
# w[9] = 0.200000
# w[10] = 0.200000
```

corresponds to exact solution, e.g. $w_0 = 1.1$, $w_i = 0.2$ $(i > 0)$.
Test: parity function

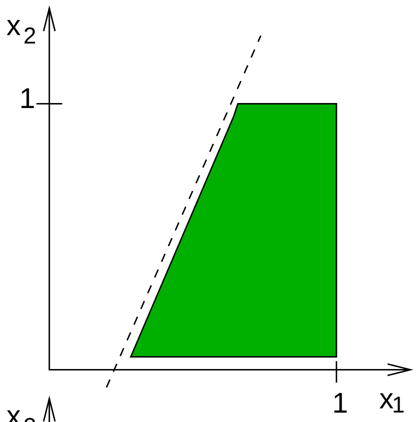
parity function

$$f(x) = \begin{cases} 1 & \text{number of 1 bits is odd} \\ 0 & \text{else} \end{cases} \tag{70}$$

For $L = 2$ (only !), failure rate as function of learning iterations:

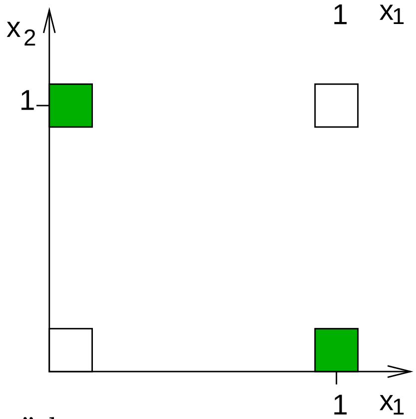


Result is "random", all obtained weights close to 0.



Analytical theory shows: pattern classification possible if there is a hyperplane in L-dim space which separates the 1/0 patterns (linear separable)

Analytical Theory: If pattern classification is possible: algorithm converges (if $\epsilon < 1/||\underline{x}||$)



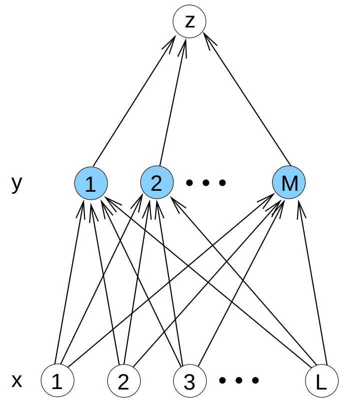
$\Rightarrow$ parity function for $L = 2$ (XOR function) not implementable.

Solution: multi-level structures, e.g. a "hidden" layer.

VIDEO: `video11c_backpropagation`

12.2 Back propagation

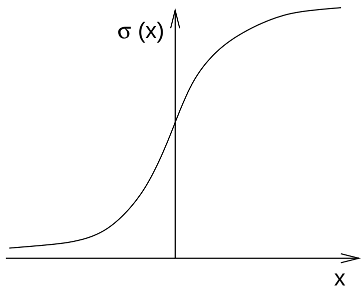
<u>Feed-forward network</u>:
Several layers, here one layer of M <u>hidden</u> Neurons.
w.l.o.g.: 1 output neuron
Transfer function ($x_0 = 1$):



$$y_j = \sigma\left(\sum_{k'=0}^{L} w_{jk'} x_{k'}\right) \quad (j = 1 \ldots M) \tag{71}$$

$$z = \sigma\left(\sum_{j'=0}^{M} \tilde{w}_{j'} y_{j'}\right) \tag{72}$$

sigmoid function



$$\sigma(x) = \frac{1}{1+\exp(-x)} \quad \in [0,1]$$

$$\sigma'(x) = (-1)(-1)\frac{\exp(-x)}{(1+\exp(-x))^2}$$

$$= \frac{1}{1+\exp(-x)} \frac{1+\exp(-x)-1}{1+\exp(-x)}$$

$$= \sigma(x)(1-\sigma(x)) \tag{73}$$

Aim: network shall learn p patterns+classifications $(\underline{x}^\nu, \hat{z}^\nu)$ $(\nu = 1, \ldots, p)$ patterns
(e.g. again a function $\hat{z} = f(\underline{x})$, then $\hat{z}^\nu = f(\underline{x}^\nu)$), $\underline{\hat{x}} \in \{0,1\}^L$);

Use as <u>energy function</u>: mean squared error

for each pattern nu, we compare the value of network x, to desired output z and square the difference

$$E = \frac{1}{2}\sum_\nu (\hat{z}^\nu - z(\underline{x}^\nu))^2 \tag{74}$$

$$= \frac{1}{2}\sum_\nu \left(\hat{z}^\nu - \sigma\left(\sum_{j=0}^{M} \tilde{w}_j y_j(\underline{x}^\nu)\right)\right)^2 \tag{75}$$

Look for optimale weights: most simple: gradient descent. Start with some weights, then (ϵ: Parameter):

gradient descent: whenever we can decrease the value of the E, we go a small step (epsilon) into this direction by changing the weights accordingly.

$$\Delta w_{jk} = -\epsilon \frac{\partial E}{\partial w_{jk}} \tag{76}$$

$$\Delta \tilde{w}_j = -\epsilon \frac{\partial E}{\partial \tilde{w}_j} \tag{77}$$

approximately

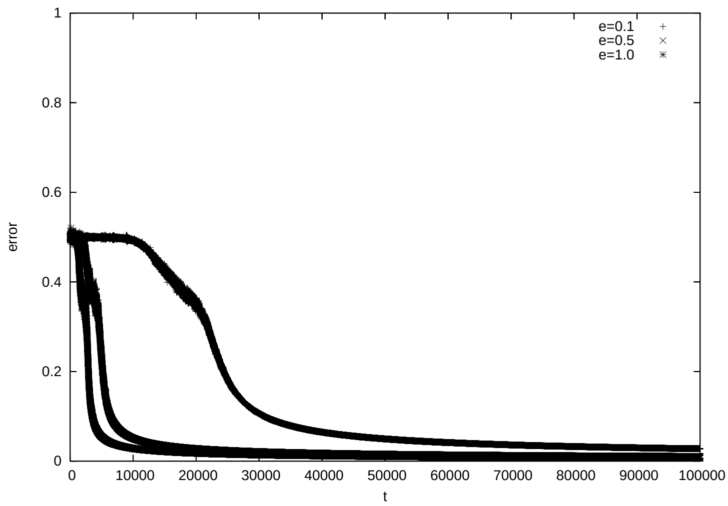
$$\Delta E \approx \sum_{\{w\}} \frac{\partial E}{\partial w} \Delta w = -\epsilon \sum_{\{w\}} \left(\frac{\partial E}{\partial w} \right)^2 \leq 0$$

$\rightarrow$ konverges to (local) minimum.

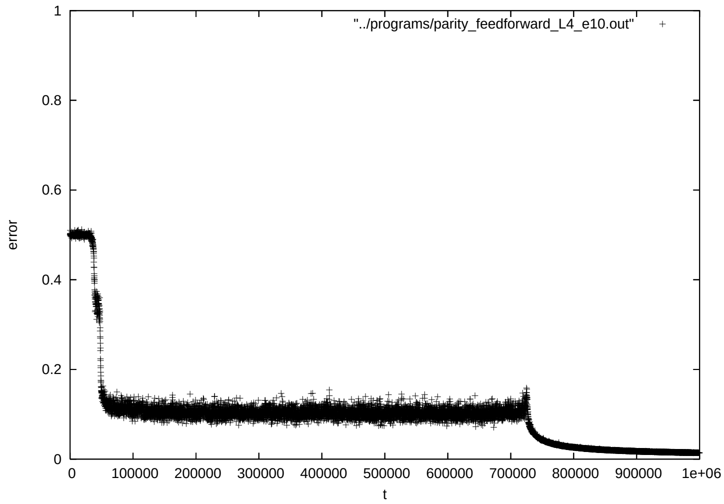
Here, contribution for a single pattern $(\underline{x}^\nu, \hat{z}^\nu) \rightarrow (\underline{x}, \hat{z})$

$$\begin{aligned}
\frac{\partial E}{\partial \tilde{w}_j} &\overset{(74)}{=} -(\hat{z} - z)\frac{\partial z}{\partial \tilde{w}_j} \overset{(72)}{=} -(\hat{z} - z)\sigma'(\sum_{j'} \tilde{w}_{j'} y_{j'}) y_j \\
&\overset{(73)}{=} -(\hat{z} - z) z(1 - z) y_j && (78) \\
\frac{\partial E}{\partial w_{jk}} &\overset{(75)}{=} -(\hat{z} - z) z(1 - z) \tilde{w}_j \frac{\partial y_j}{\partial w_{jk}} \\
&\overset{(71)}{=} -(\hat{z} - z) z(1 - z) \tilde{w}_j \sigma'(\sum_{k'} w_{jk'} x_{k'}) x_k \\
&\overset{(73)}{=} -(\hat{z} - z) z(1 - z) \tilde{w}_j y_j (1 - y_j) x_k && (79)
\end{aligned}$$

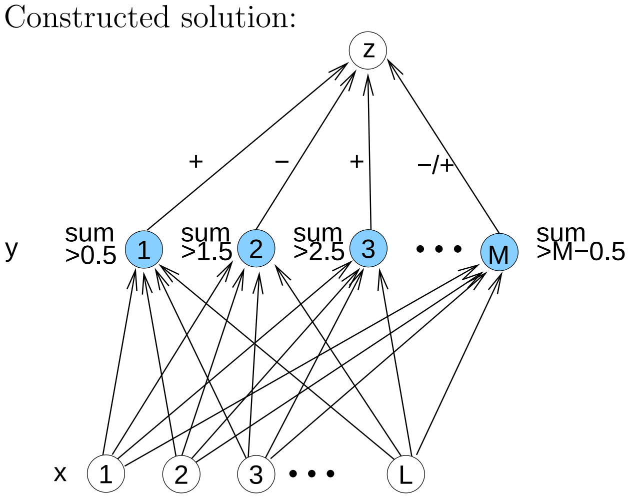
Error rate for parity function $L = 2$



konverges quickly.
Error rate for parity function $L = 4$



converges slowly.
Constructed solution:



Accelerated convergence: better minimization approach: conjugate gradient, Monte Carlo optimisation with Parallel Tempering, ...

So far supervised learning. Unsupervised learning: generalization, not $E = \frac{1}{2}\sum_\nu(\hat{z}^\nu - z(\underline{x}))^2$ but arbitrary function $f(\hat{z}^\nu, z(\underline{x}))$ is minimized.
Aim: System learns structures. Weights are adapted, such that set of given patterns is generated. Application: statistical analyses such as clustering of data points, “Deep Learning” (many layers, GO algorithm).

References

[1] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from ”good” random number generators. *Phys.*

Rev. Lett., 69:3382, 1992.

[2] B.J.T. Morgan. *Elements of Simulation.* Cambridge University Press, Cambridge, 1984.

[3] Alexander K. Hartmann. *Practical Guide to Computer Simulations.* World Scientific, Singapore, 2009.

[4] D. Stauffer and A. Aharony. *Perkolationstheorie.* Wiley-VCH, Weinheim, 1995.

[5] A. Dhar. Heat conduction in a one-dimensional gas of elastically colliding particles of unequal masses. *Phys. Rev. Lett.*, 86:3554, 2001.

[6] P. Grassberger, W. Nadler, and Lei Yang. Heat conduction and entropy production in a one-dimensional hard-particle gas. *Phys. Rev. Lett.*, 89:180601, 2002.

[7] L.E. Reichl. *A Modern Course in Statistical Physics.* John Wiley & Sons, New York, 1998.

[8] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087, 1953.

[9] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.*, 5:115–133, 1943.

[10] D. Hebb. *Organisation of Behavior.* Wiley, New York, 1949.