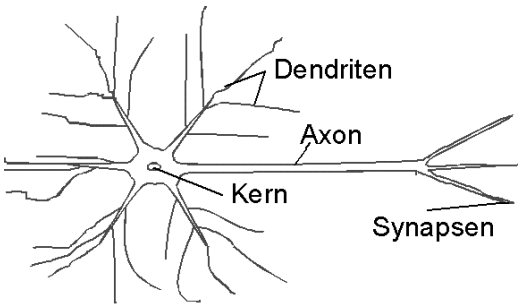
Vorlesung 8 (Dienstag 27.2.2018)

7 Neuronale Netze

Grundlagenforschung: Verständnis wie Gehirn funktioniert
Anwendung: effiziente selbstlernende Algorithmen (Handschriftenerkennung, Optimierung, Generalisierung, ...)
Gehirn hat ca. 10^{11} Neuronen. Kommunikation durch elektrische Impulse.

Aufbau Neuron:

Dendriten: Eingangssignale, bis zu $2 * 10^5$ pro Zelle
Axon: Ausgangsignal
Synapsen: koppeln Axon an Dendriten anderer Zellen, bis zu 10^4 pro Zelle, insgesamt ca 10^{15}, Kopplungsstärke veränderbar.



von www.lunaticpride.de

Modellierung McCulloch/Pitts Neuron [5]

- L Eingänge $x_i = 0, 1$ (ruhig/aktiv)
- Synapsenstärken $w_i \in \mathbb{R}$
- s: Schwellwert
- Ausgangsignal

$$y = \theta\left(\sum_{i=1}^{L} w_i x_i - s\right) \tag{49}$$

$\theta(x) = 1$ für $x \geq 0$ $\theta(x) = 0$ sonst.

Logische UND/NOT Funktionen realisierbar → beliebige logische Funktionen.

Einstellung der Gewichte: Hebbsche Lernregel [6]

$$\Delta w_i = \epsilon y^{(\text{soll})}(\underline{x}) x_i \tag{50}$$

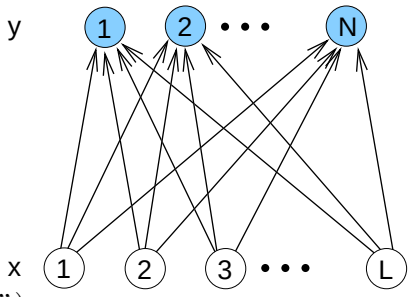
- $y^{(\text{soll})}(\underline{x})$ gewünschte Ausgabe
- $\epsilon > 0$: "Lernparameter"

7.1 Perzeptron

Zur Klassifikation von Eingabemustern $\underline{x}$ und Generalisierung
$\rightarrow N$ Ausgaben $y_r \in \{0, 1\}$ $(r = 1, \ldots, N)$ gemäß

$$y_r = \theta \left(\sum_{i=0}^{L} w_{ri} x_i \right) \tag{51}$$

$(s \leftrightarrow -w_{r0}$ via $x_0 = 1)$
Jede Ausgabe ist unabhängig von den anderen, reine Schichtstruktur



Perzeptron Lernalgorithmus ("Trainingsphase")

- Starte mit zufälligen Kopplungen
- Führe verschiedene Trainingsvektoren x zu.
  Für jede falsche Ausgabe $y_r(\underline{x})$ passe Gewichte an:

$$\Delta w_{ri} = \epsilon \cdot (y_r^{(\text{soll})}(\underline{x}) - y_r(\underline{x})) \cdot x_i \tag{52}$$

Als C Funktion (siehe `perceptron.c`)

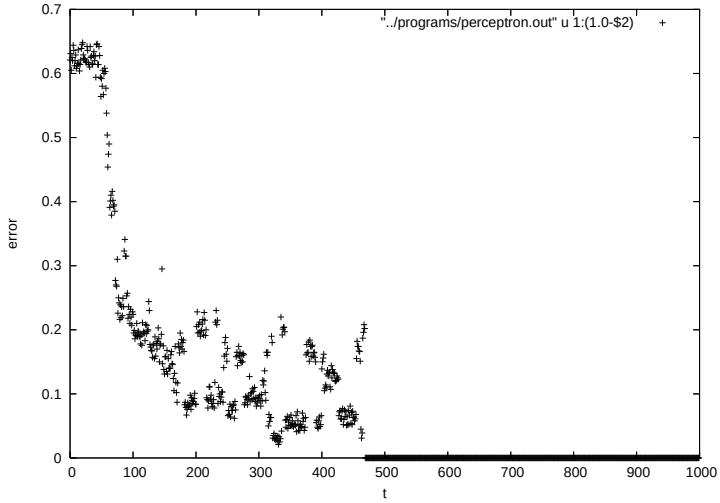
```
/***************** perceptron_learning() ***************/
/** Peforms 'K' steps of learning algorithm:          **/
/** generate random vector and adjust weights using   **/
/** parameter 'epsilon' to learn function 'f'         **/
/** PARAMETERS: (*)= return-paramter                  **/
/**            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: Funktion

$$f(x) = \begin{cases} 1 & \text{mehr als Hälfte der Bits ist 1} \\ 0 & \text{sonst} \end{cases} \tag{53}$$

Für $L = 10$, Fehlerquote als Funktion der Lernzyklen:



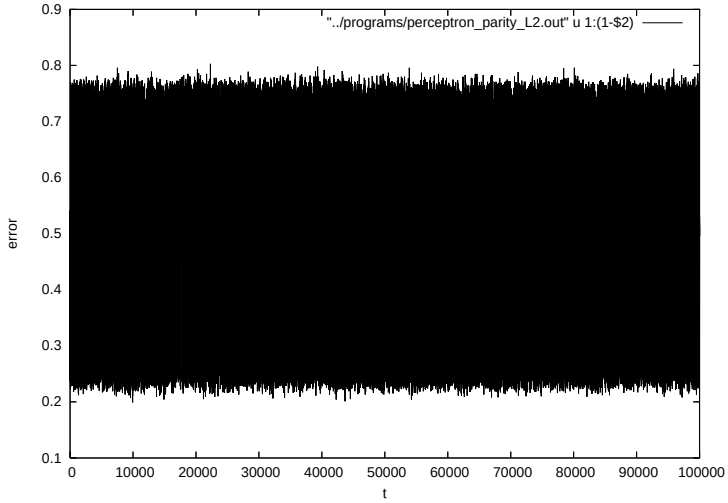
Resultierende Gewichte

```
# 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
```

entspricht exaktem Ergebnis, z.B. $w_0 = 1.1$, $w_i = 0.2$ $(i > 0)$.
Test: Funktion

$$f(x) = \begin{cases} 1 & \text{Zahl der 1 Bits ist ungerade} \\ 0 & \text{sonst} \end{cases} \tag{54}$$

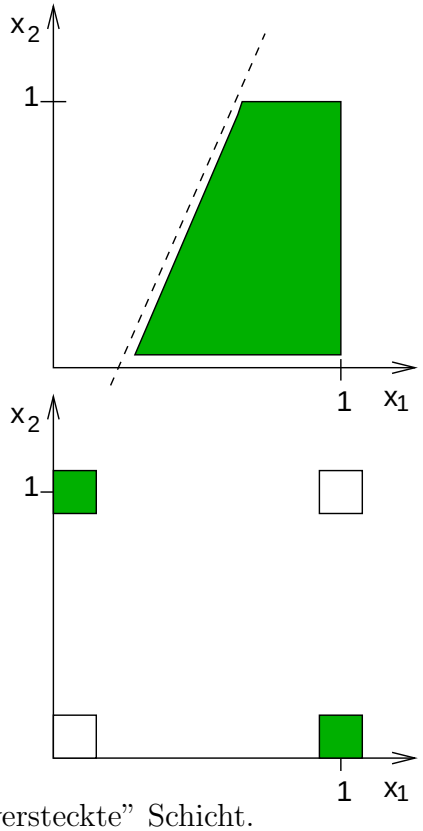
Für $L = 2$ (nur !), Fehlerquote als Funktion der Lernzyklen:



Ergebnis "zufällig", resultierende Gewichte alle nahe bei 0.

Analytische Theorie zeigen: Musterklassifikation möglich wenn es Hyperebene im L-dim Raum gibt, der 1/0 Muster trennt (linear separierbar)

Analytische Theory: Falls Musterklassifikation möglich: Algorithmus konvergiert (falls $\epsilon < 1/||\underline{x}||$)

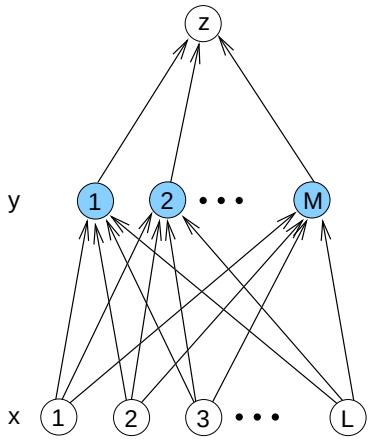


→ auch XOR-Funktion nicht realisierbar.

Ausweg: Mehrschichtstrukturen, z.B. eine "versteckte" Schicht.

7.2 Backpropagation

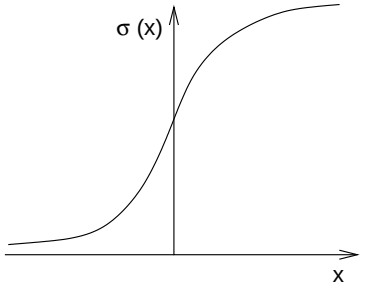
Feed-forward Netzwerke:
Mehrere Schichten, hier eine Schicht von M versteckter Neuronen
o.B.d.A: 1 Ausgabeneuron
Übergangsfunktionen ($x_0 = 1$):



$$y_j = \sigma\left(\sum_{k'=0}^{L} w_{jk'} x_{k'}\right) \quad (j = 1 \ldots M) \tag{55}$$

$$z = \sigma\left(\sum_{j'=0}^{M} \tilde{w}_{j'} y_{j'}\right) \tag{56}$$

$$\begin{aligned} \sigma(x) &= \frac{1}{1+\exp(-x)} \quad \in [0,1] \\ \sigma'(x) &= (-1)(-1)\frac{\exp(-x)}{(1+\exp(-x))^2} \\ &= \frac{1}{1+\exp(-x)}\frac{1+\exp(-x)-1}{1+\exp(-x)} \\ &= \sigma(x)(1-\sigma(x)) \end{aligned} \tag{57}$$



Ziel: Netz soll p Musterpaare $(\underline{x}^\nu, \hat{z}^\nu)$ $(\nu = 1, \ldots, p)$ lernen (z.B. wieder eine Funktion $\hat{z} = f(\underline{x})$, $\underline{\hat{x}} \in \{0,1\}^L$)

Einführung Energiefunktion: mittlerer quadratischer Fehler

$$E = \frac{1}{2}\sum_\nu (\hat{z}^\nu - z(\underline{x}^\nu))^2 \tag{58}$$

$$= \frac{1}{2}\sum_\nu \left(\hat{z}^\nu - \sigma\left(\sum_{j=0}^{M} \tilde{w}_j y_j(\underline{x}^\nu)\right)\right)^2 \tag{59}$$

Suche nach optimalen Gewichten: Am einfachsten: Gradientenabstiegsverfahren. Starte mit irgendwelchen Gewichten, dann (ϵ: Parameter):

$$\Delta w_{jk} = -\epsilon \frac{\partial E}{\partial w_{jk}} \tag{60}$$

$$\Delta \tilde{w}_j = -\epsilon \frac{\partial E}{\partial \tilde{w}_j} \tag{61}$$

näherungweise

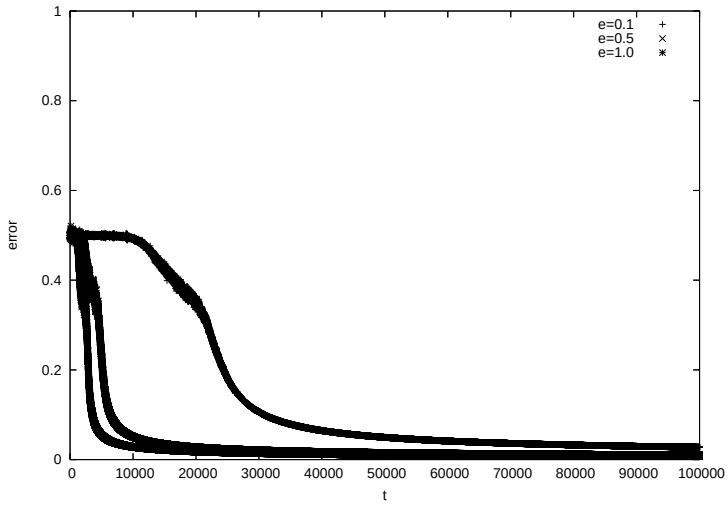
$$\Delta E \approx \sum_{\{w\}} \frac{\partial E}{\partial w} \Delta w = -\epsilon \sum_{\{w\}} \left(\frac{\partial E}{\partial w} \right)^2 \leq 0$$

$\rightarrow$ konvergiert in (lokales) Minimum.

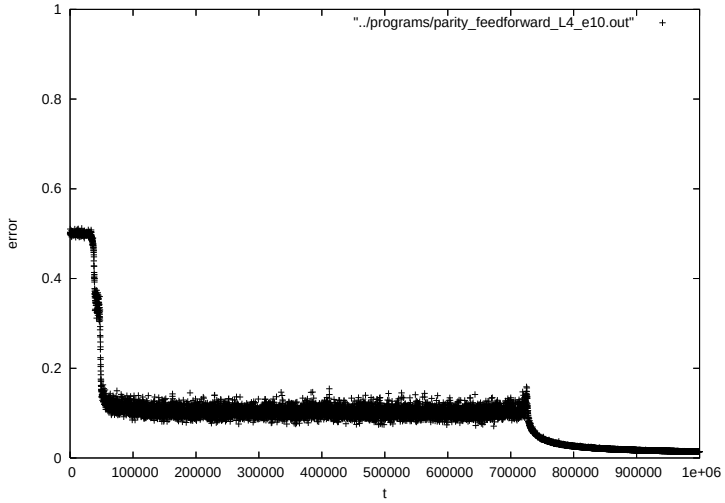
Hier, Beitrag für eine einzelnes Muster $(\underline{x}^\nu, \hat{z}^\nu) \rightarrow (\underline{x}, \hat{z})$

$$\begin{aligned}
\frac{\partial E}{\partial \tilde{w}_j} &\overset{(58)}{=} -(\hat{z} - z) \frac{\partial z}{\partial w_j} \overset{(56)}{=} -(\hat{z} - z) \sigma'(\sum_{j'} \tilde{w}_{j'} y_{j'}) y_j \\
&\overset{(57)}{=} -(\hat{z} - z) z (1 - z) y_j \qquad (62) \\
\frac{\partial E}{\partial w_{jk}} &\overset{(59)}{=} -(\hat{z} - z) z (1 - z) \tilde{w}_j \frac{\partial y_j}{\partial w_{jk}} \\
&\overset{(55)}{=} -(\hat{z} - z) z (1 - z) \tilde{w}_j \sigma'(\sum_{k'} w_{jk'} x_{k'}) x_k \\
&\overset{(57)}{=} -(\hat{z} - z) z (1 - z) \tilde{w}_j y_j (1 - y_j) x_k \qquad (63)
\end{aligned}$$

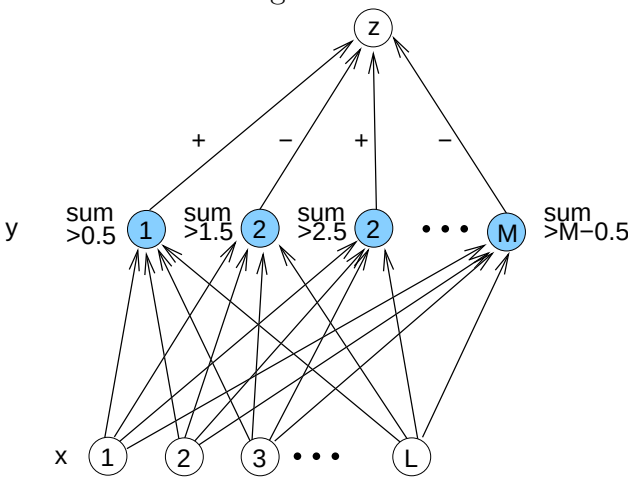
Fehler Rate für Parity-Funktion $L = 2$



konvergiert schnell.
Fehler Rate für Parity-Funktion $L = 4$



konvergiert langsam.
Konstruierte Lösung:



Beschleunigung der Konvergenz: bessere Minimierungsmethoden: Conjugate Gradient, Monte-Carlo Optimierung mit Parallel Tempering, ...
Bisher Überwachtes Lernen. Unüberwachtes Lernen: Verallgemeinerung, nicht $E = \frac{1}{2}\sum_\nu (\hat{z}^\nu - z(\underline{x}))^2$ sonder beliebige Funktion $f(\hat{z}^\nu, z(\underline{x}))$ wird minimiert. Anwednung: statistische Analysen wie Clusterung von Datenpunkten.