Einführung in die KI

Prof. Dr. sc. Hans-Dieter Burkhard

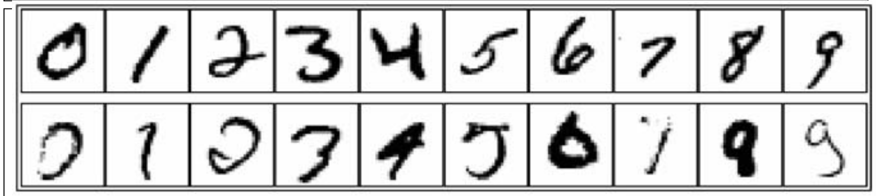
Vorlesung Winter-Semester 2006/07

Neuronale Netze

Neuronale Netze (Konnektionismus)

- Neuronale Netze sind biologisch motiviert
- Neuronale Netze können diskrete, reell-wertige und Vektor-wertige Funktionen berechnen
- Informationsspeicherung verteilt im gesamten Netz
- Lernverfahren auf der Basis von Fehleroptimierung (Gradientenabstieg), z.B. Back-Propagation
- Neuronale Netze sind robust bezüglich
  - Verrauschten Daten
  - Ausfall von Teilen des Netzes

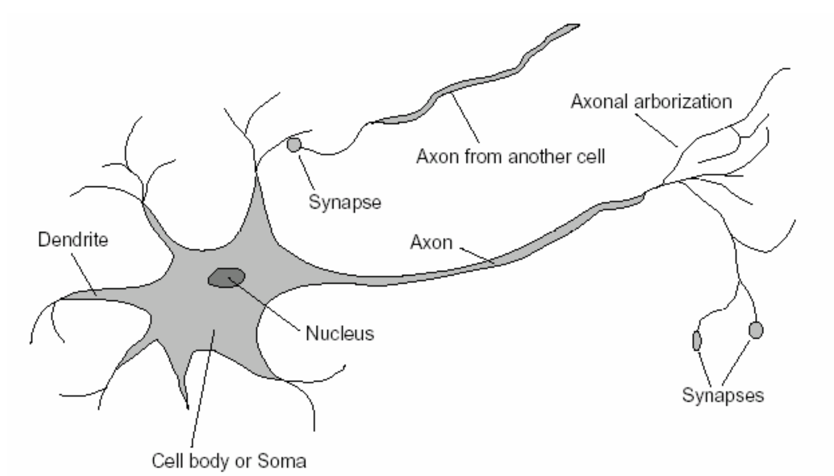
Beispiel-Aufgabe: Schrifterkennung



Es soll ein Verfahren entwickelt werden, das geschriebenen Zeichen die korrekte Bedeutung zuordnet.
Neuronale Netze können so trainiert werden, dass sie die Aufgabe mit geringer Fehlerrate lösen.
Trainieren bedeutet: Anpassung des Verhaltens aufgrund der Präsentation von Beispielen.
Hypothese: Das Netz kann so verallgemeinern, dass auch zuvor nicht gesehene Zeichen korrekt klassifiziert werden.

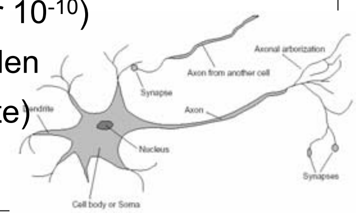
Biologisches Vorbild

Nervenzelle



Biologisches Vorbild

- Gehirn mit 10^{10}-10^{11} Neuronen, 20 Typen
- Axon bis zu 1m lang
- Gerichtete Reizleitung von Dendriten zu Synapsen
- Weiterleitung bei hinreichend starker Erregung
- Jeweils Verbindungen (Synapsen) zu 10-100000 Neuronen
- Insgesamt 10^{14} Synapsen
- Schaltzeit 10^{-3} Sekunden (Computer 10^{-10})
- Erfassung einer Szene: 10^{-1} Sekunden
  (d.h. ca. 100 Verarbeitungs-Schritte)
- Hohe Parallelität
- Lernfähigkeit

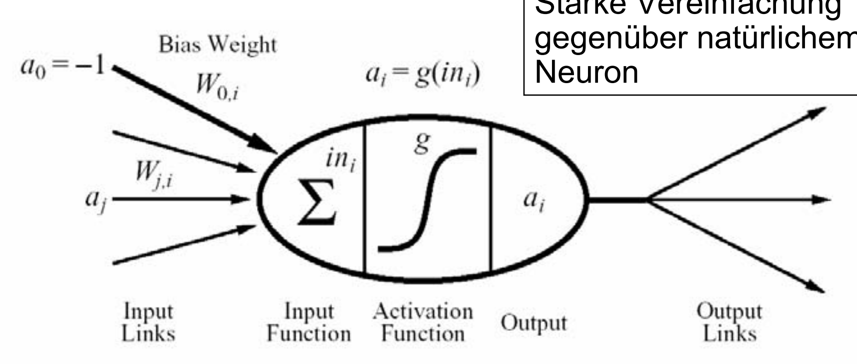


Modell: (Künstliches) Neuron, „Unit“

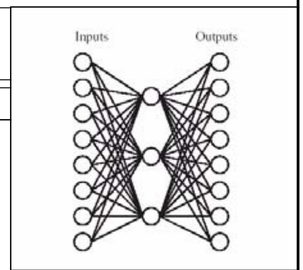
McCulloch-Pitts Unit

- Mehrere Eingänge, 1 (verzweigter) Ausgang

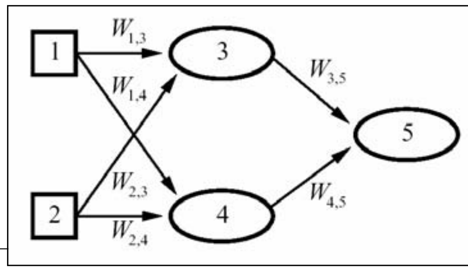
Starke Vereinfachung gegenüber natürlichem Neuron



Neuronales Netz

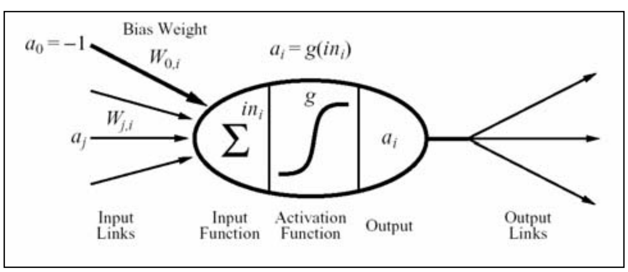


- Miteinander verknüpfte Neuronen $u_1,...,u_n$
- Die Verbindungen zwischen Neuronen u_j und u_i sind gewichtet mit Werten w_{ji}
- Darstellung der Verknüpfung als Matrix $((w_{ji}))$ (mit $w_{ji} = 0$ falls keine Verbindung besteht)



Verarbeitung an Einheit u_i

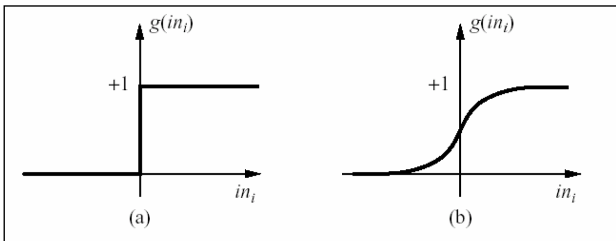
- Gewichtete Summe der Eingänge $in_i = \Sigma_{i=0,..,n}\ w_{ji}\ a_i$ mit
  a_j = ankommende Aktivierung von Einheit u_j für $i=1,..,n$
  $a_0 = -1$ „Bias"
- Aktivierungsfunktion g
- Ausgabe $a_i = g(in_i) = g(\Sigma_{i=0,..,n}\ w_{ji}\ a_i)$

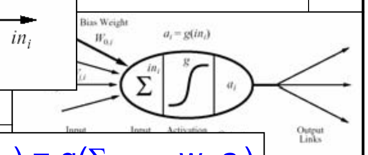


Aktivierungsfunktion g

„Schaltet Ausgabe ein“ falls Eingabe hinreichend groß bzw. falls Eingabe positiv (Justierung mittels Bias), z.B.

- Schwellwert oder
- Sigmoidfunktion $1/(1+e^{-x})$ (differenzierbar!) Flanken als Übertragungsfunktion (z.B. Verstärkung)



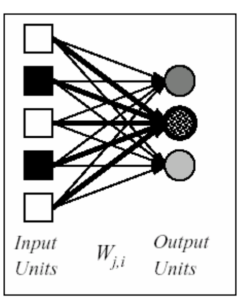


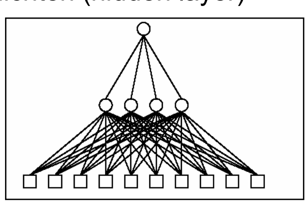
$a_i = g(in_i) = g(\Sigma_{i=0,..,n} w_{ji}\, a_i)$

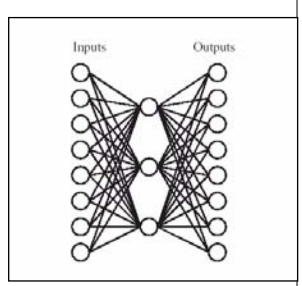
Netzstrukturen

Ohne Rückkopplung („feed forward“)

- Verarbeitung Eingabevektor → Ausgabevektor
- Typischerweise in Schichten (layer) angeordnet:
  - Eingabeschicht, Ausgabeschicht
  - Verdeckte Schichten (hidden layer)







Netzstrukturen

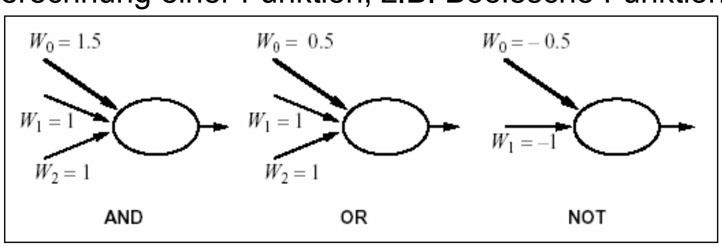
Mit Rückkopplung („recurrent“):

Dynamisches System mit Möglichkeiten für
- Stabilisierung (Zustand)
- Oszillation
- Chaotischem Verhalten

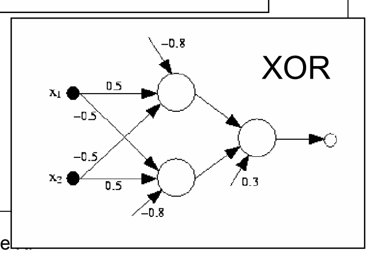
Antwort auf eine Eingabe vom aktuellen Zustand abhängig („Gedächtnis)

Netzstrukturen

Ohne Rückkopplung („feed forward“)
- Berechnung einer Funktion, z.B. Boolesche Funktionen

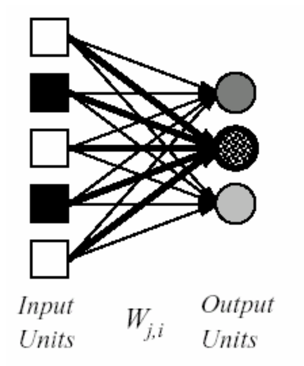


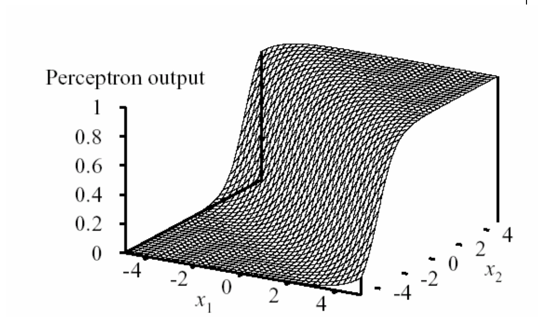
Mittels Verknüpfung beliebige Boolesche Funktionen darstellbar



Perzeptron

Einschichtig, Ohne Rückkopplung



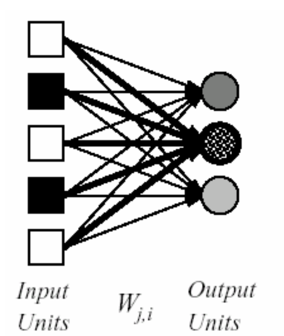


Ausgaben eines Neurons mit 2 Eingängen bei sigmoider Aktivierungsfunktion

Perzeptron

Ausgaben unabhängig voneinander.

Ausgabe einer Einheit: $a = g(in) = g(\Sigma_{i=0,..,n} w_j a_i)$



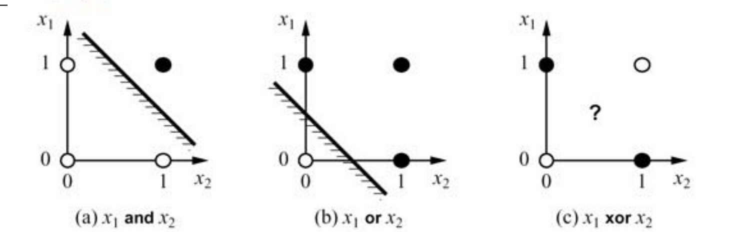
Gleichung einer Hyperebene falls g Schwellwertfunktion:

$\Sigma_{i=0,..,n} w_j a_i > 0$

oder $\mathbf{w} \cdot \mathbf{a} > 0$

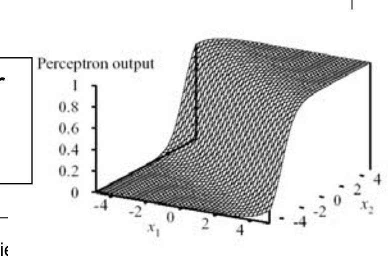
Nur Darstellung von linear trennbaren Funktionen

Nur linear trennbare Funktionen



$\Sigma_{i=0,..,n}\ w_j\ a_i > 0$ oder $\mathbf{w} \cdot \mathbf{a} > 0$
XOR ist nicht linear trennbar

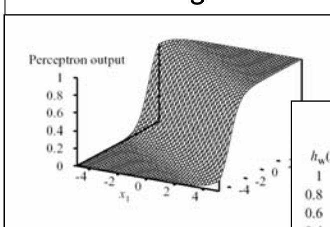
Analoges Resultat für sigmoide Aktivierungsfunktion

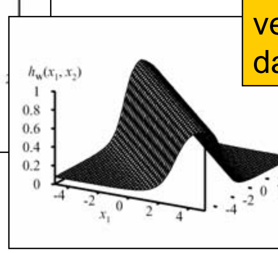


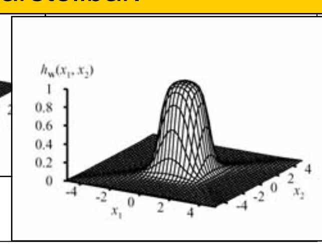
Mehrschichtige Netze

- Die Ausgaben von Netzen können als Eingaben nachgeschalteter Netze verwendet werden.
- Das ermöglicht komplexeres Verhalten.

Alle (stetigen) Funktionen in Netzen mit (zwei) drei verdeckten Schichten darstellbar.







Lernen

Insgesamt gegeben:

- Menge **F** von Funktionen
- Menge **N** von Neuronalen Netzen N
- Gütemaß g: **F** x **N** $\rightarrow$ W

Insgesamt gesucht:

Verfahren, das zu gegebener Funktion $f(x_1,...,x_n)$ ein Netz N liefert mit „hinreichender“ Güte g(f,N) .

i.a. keine genaue Übereinstimmung gefordert, sondern gute Approximation:

optimale Güte = minimaler Fehler

Lern-Verfahren

Gegeben: Funktion $f(x_1,...,x_n)$

Gesucht: Netz N mit „optimaler“ Güte g(f,N) .

Idee des Lernens:

Ein initiales Netz wird anhand von (endlich vielen) Trainingsdaten solange modifiziert, bis es auf diesen Daten hinreichend gut arbeitet.

Hypothese:

Wenn das Netz auf den Trainingsdaten gut arbeitet, wird es auch für alle anderen Daten gute Ergebnisse liefern.

Lern-Verfahren

Eingabe: Beispiele zu $f(x_1,\ldots,x_n)$, initiales Netz N_0

Kann als Suchproblem im Suchraum **N** betrachtet werden.

Start: $k:=0$

Schritt k:

Falls Güte $g(f,N_i) > \varepsilon$: Exit mit Lösung N_i .

Andernfalls:

N_{k+1} aus N_k berechnen durch Anpassen der Gewichte w_{ji}.

Heuristik für Anpassung:

Optimierung von $g(f,N_k)$ anhand der Lernbeispiele.

Weiter mit Schritt $k+1$.

Lernformen

- Überwachtes Lernen
  („supervised learning“, „mit Lehrer“):
  Es werden Beispiele korrekter Zuordnungen präsentiert
  $[x_1,\ldots,x_n ; f(x_1,\ldots,x_n)]$

- Unüberwachtes Lernen
  („unsupervised learning“, „ohne Lehrer“):
  Es werden Beispiele von Eingaben präsentiert
  $[x_1,\ldots,x_n]$

Unüberwachtes Lernen

Es werden Beispiele von Eingaben präsentiert

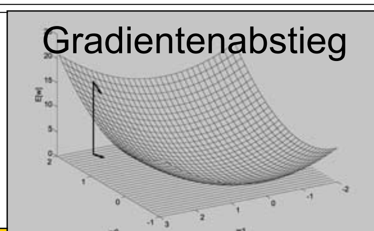
$[x_1,...,x_n]$

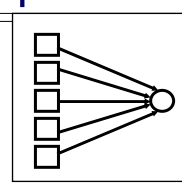
Das Netz soll z.B. Regularitäten (Clusterbildung) erkennen.

Mögliche Verfahren:

- Hebbsche Lernregel: Verbindungen zwischen gleichzeitig aktivierten Neuronen verstärken („festigen durch Wiederholung“)
- Competitive Learning (Winner takes it all): Eingänge hoch aktivierter Neuronen werden verstärkt.

Überwachtes Lernen für Perzeptron





<u>Idee:</u> Fehler bei Trainingsbeispielen minimieren

Summe der Fehlerquadrate:

$$E = \sum_{d \in D} (N (x^d_1,...,x^d_n) - f (x^d_1,...,x^d_n))^2$$

<u>Gradientenabstieg:</u>

Korrekturen der Gewichte: $w_i := w_i + \delta w_i$ („Delta-Regel“)

in Richtung der größten Abnahme des Fehlers E

Gradientenabstieg für Perzeptron

Start: $k:=0$ $w_i :=$ kleine zufällige Werte, Lernrate η festlegen.

Schritt k:

Iteration von 1-3 mit Netz N_k bis $E(N_k,f) < \varepsilon$:

1. Für jedes Trainingsbeispiel $d = [\,[x^d_1,\ldots,x^d_n]\,,\ f(x^d_1,\ldots,x^d_n)\,]$:
   Fehler $\delta^d := f(x^d_1,\ldots,x^d_n) - N_k(x^d_1,\ldots,x^d_n)$ berechnen.
2. Korrekturwerte der Gewichte w_i bestimmen:
   $$\delta w_i := \eta \cdot \sum_{d \in D} \delta^d \cdot x^d_i$$
3 Neues Netz N_{k+1} mit modifizierten Gewichten:
$$w_i := w_i + \delta w_i$$

Weiter mit Schritt $k+1$.

Gradientenabstieg für Perzeptron

Vorzugeben:

- Anzahl der Neuronen
  - an Problemgröße angepasst: Dimension des Raumes
- Initiale Gewichte
- Fehlerschranke ε
- Lernrate η bestimmt Stärke der Modifikation
  - Zu kleine Werte:
    geringe Variation, Gefahr lokaler Minima
  - Zu große Werte:
    schnelle Variation, am Ende schlechte Konvergenz

Simulated annealing: η mit der Zeit verringern

Inkrementeller Gradientenabstieg für P.

Bisher:

Modifikation bezüglich Gesamtfehler bei allen Beispielen

$$\delta w_i := \eta \cdot \sum_{d \in D} \delta^d \cdot x^d_i$$

Jetzt:

Einzelmodifikationen für jedes Beispiel

$$\delta w_i := \eta \cdot \delta^d \cdot x^d_i$$

Vorteile: Schnellere Konvergenz

Besseres Verlassen lokaler Minima

Inkrementeller Gradientenabstieg für P.

Start: $k:=0$ $w_i :=$ kleine zufällige Werte, Lernrate η festlegen.

Schritt k:

Für jedes Trainingsbeispiel $d = [\,[x^d_1,\dots,x^d_n]\,,\ f(x^d_1,\dots,x^d_n)\,]$:

a) Fehler $\delta^d := f(x^d_1,\dots,x^d_n) - N_k(x^d_1,\dots,x^d_n)$ berechnen.

b) Korrekturwerte der Gewichte w_i bestimmen:

$$\delta w_i := \eta \cdot \delta^d \cdot x^d_i$$

c) Neues Netz N_k mit modifizierten Gewichten:

$$w_i := w_i + \delta w_i$$

„Delta-Regel“

Falls $E(N_k,f) < \varepsilon$: Exit. Sonst $N_{k+1} := N_k$, weiter mit Schritt $k+1$.

Backpropagation

- Lernen in mehrschichtigen Netzen
- Fehler von Ausgabeschicht sukzessive in vorherige Schichten geeignet zurück propagieren
- Ausgangspunkt: Minimierung der Summe der Fehlerquadrate $E = \sum_{d \in D} (h (x^d_1,\ldots,x^d_n) - f (x^d_1,\ldots,x^d_n))^2$
- Verfahren: Gradientenabstieg (in Richtung Ableitung der Fehlerfunktion)
- Ausgabe (i.a. mehrere Neuronen)

  $f (x_1,\ldots,x_n) = [t_1,\ldots,t_k]$
- Beispiele in der Form

  $d = [[x_1,\ldots,x_n] , [t_1,\ldots,t_k]]$

Backpropagation

Start: $k:=0$ $w_i :=$ kleine zufällige Werte, Lernrate η festlegen.

Schritt k:

Für jedes Trainingsbeispiel $d = [[x_1,\ldots,x_n] , [t_1,\ldots,t_k]]$:

a) Vorwärtspropagierung
b) Rückwärtspropagierung des Fehlers für Ausgabe-Neuronen bzw. schichtweise für innere Neuronen
c) Korrekturwerte der Gewichte bestimmen:
d) Gewichte korrigieren:
e) Neues Netz N_k mit modifizierten Gewichten

Falls $E(N_k,f) < \varepsilon$: Exit. Sonst $N_{k+1} := N_k$, weiter mit Schritt $k+1$.

Backpropagation: innerer Zyklus

Für jedes Trainingsbeispiel $d = [[x_1,...,x_n] , [t_1,...,t_k]]$:

a) Vorwärtspropagierung:
Für jedes Ausgabe-Neuron u die Ausgabe o_u berechnen.

b) Berechnung des Fehlers für Ausgabe-Neuronen u

$$\delta_u := o_u \cdot (1- o_u) \cdot (t_u - o_u)$$

bzw. („nach rückwärts") schichtweise für innere Neuronen v

$$\delta_v := o_v \cdot (1 - o_v) \cdot \sum_u w_{vu} \cdot \delta_u$$

c) Korrekturwerte der Gewichte bestimmen:

$$\delta w_{uv} := \eta \cdot \delta_u \cdot x_{uv}$$

d) Gewichte korrigieren:

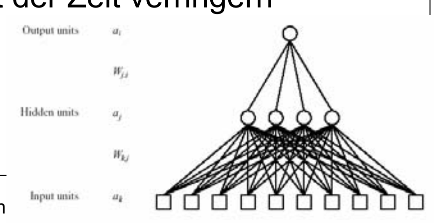
$$w_{uv} := w_{uv} + \delta w_{uv}$$

e) Neues Netz N_k mit modifizierten Gewichten

Lernen für Mehrschichtige Netze

Vorzugeben:

- Anzahl der Schichten und Neuronen pro Schicht
  „an Problemgröße angepasst"
- Initiale Gewichte
  häufig alle Neuronen zwischen Schichten verbunden
- Fehlerschranke ε
- Lernrate η bestimmt Stärke der Modifikation
  Simulated annealing: η mit der Zeit verringern



Korrekte Verallgemeinerung?

Wenn das Netz auf den Trainingsdaten gut arbeitet, wird es auch für alle anderen Daten gute Ergebnisse liefern.

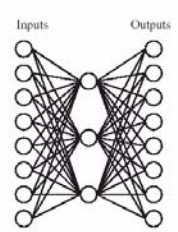
- Grundsätzlich könnte jedes Netz als Lernergebnis genommen werden, wenn es die Trainingsbeispiele reproduziert.
- Es gibt aber zusätzliche Annahmen, die bestimmte Netze bevorzugen, z.B.
  - Stetigkeit: Ähnliche Eingaben erzeugen ähnliche Ausgaben.
  - Gleichmäßige Interpolation zwischen Trainingsbeispielen
  - Gradientenabstieg für Modifikation der Netze.

Allgemeines Problem für Maschinelles Lernen:

Welche Lösungen werden bevorzugt?
Zusätzliche Annahmen: „Inductive bias".

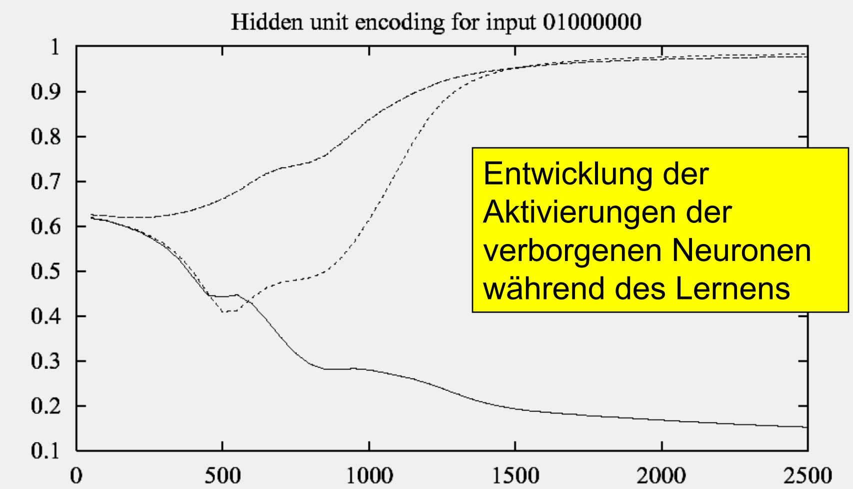
Welche Größe des Netzes ist sinnvoll?

- Viel hilft nicht viel:
- Anzahl und Größe der Schichten müssen dem Problem entsprechen (eine Spezialisierung der Neuronen auf eigene Aufgaben erlauben.
- Untersuchung: Lernen der Binärkodierung in der verborgenen Schicht (nach Mitchell)



Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Welche Größe des Netzes ist sinnvoll?



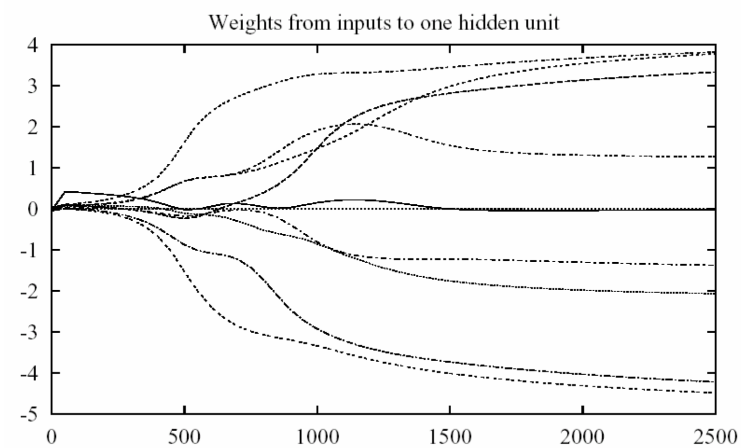
Entwicklung der Aktivierungen der verborgenen Neuronen während des Lernens

Welche Größe des Netzes ist sinnvoll?

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

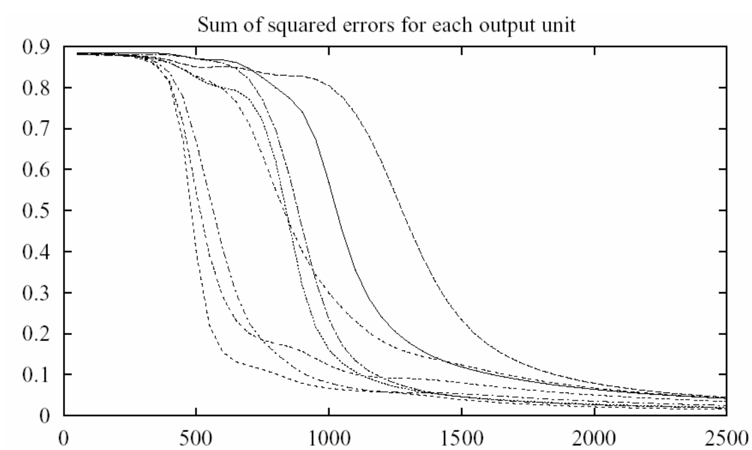
Gelernte Kodierung der verborgenen Neuronen

Welche Größe des Netzes ist sinnvoll?



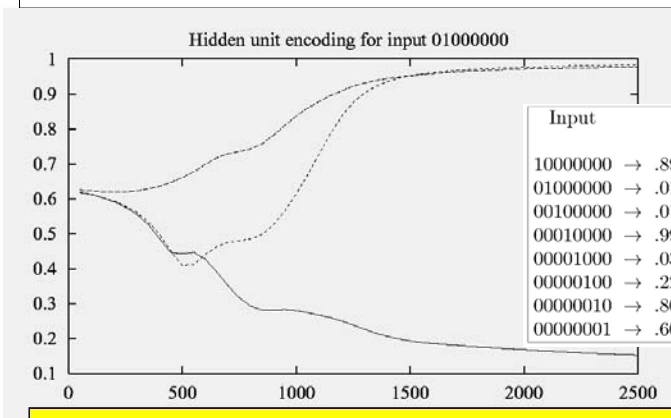
Anpassung der Gewichte der Eingabe-Neuronen (mit bias) zu einem der verdeckten Neuronen

Welche Größe des Netzes ist sinnvoll?



Fehler an den Ausgabe-Neuronen während des Lernens

Welche Größe des Netzes ist sinnvoll?



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Drei verdeckte Neuronen entsprechen genau der Dimension des Problems, weniger oder mehr Neuronen führen zu schlechteren Ergebnissen. Im besten Fall werden die überzähligen Neuronen mit der Zeit „ignoriert“.

Bestimmung der Größe von Netzen

- Probieren.
- Zunächst zu viele Neuronen, dann Neuronen mit wenig Einfluss (geringe Gewichte) eliminieren – LeCun et. al. „Optimal Brain Damage Approach“ 1990
- Zunächst zu wenig Neuronen, dann Neuronen hinzufügen (bei schlechter Konvergenz) – Fahlman and Lebiere „Cascade Correlation“ 1990.
- Evolutionäre Algorithmen.

Overfitting

Anpassung an zu spezielle Beispiele geht zulasten der Generalisierung (insbesondere bei verrauschten Daten)

Gegenstrategie:

Beispielmenge D aufteilen in

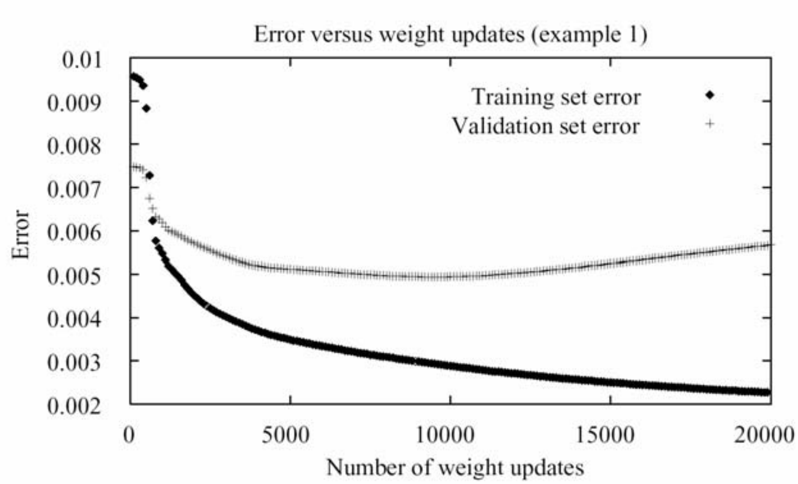
– Trainingsmenge D_T

– Validierungsmenge D_V

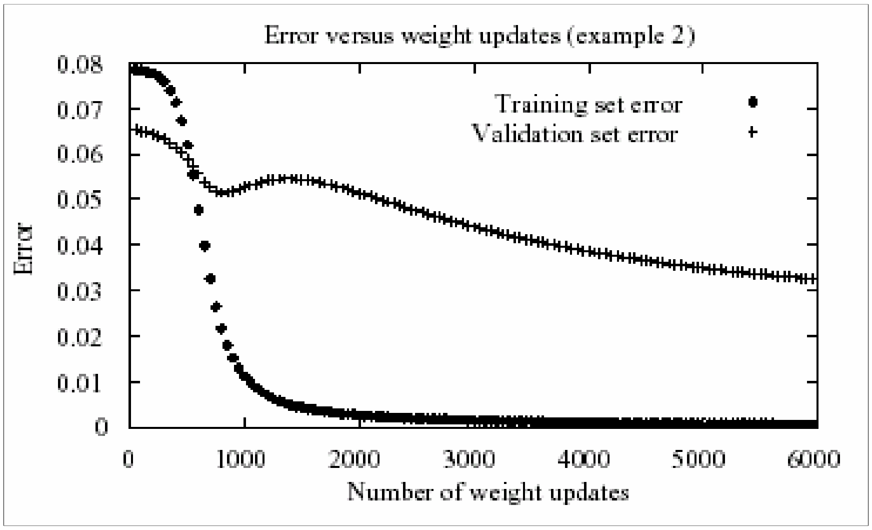
Abbruch-Kriterium nur beziehen auf D_V

Kreuzvalidierung: Mehrfache unterschiedliche Aufteilungen von D in D_T und D_V

Overfitting



Overfitting



Anwendungen

- Klassifizierung, z.B. Schriftzeichen
- Erkennung von Sprachmustern
- Reaktive Steuerungen

Vorteile:

- Schnelle Berechnung
- Robust bzgl. Rauschen in Trainingsdaten
- Robust bzgl. Ausfall von Neuronen

Charakteristika typischer Anwendungen

- Eingabe:
  - (reell-wertige) Merkmalsvektoren.
  - Merkmale können auch korreliert sein.
  - Daten können verrauscht sein.
- Ausgabe:
  - diskret/kontinuierlich
  - Vektor
- Trainingsverfahren:
  - evtl. langwierig, komplex, wiederholte Versuche
- Anwendung
  - Ausgabewert wird schnell berechnet
- Schlechte Kommunizierbarkeit
  - „Subsymbolischer Ansatz“

Modifikationsmöglichkeiten

- Fehlerfunktion
  - Summe der Fehlerquadrate
- Mögliche Modifikationen der Fehlerfunktionen:
  - $E = \sum_{d \in D} (h (x^d_1,\ldots,x^d_n) - f (x^d_1,\ldots,x^d_n))^2 + \gamma \sum_{i,j} w_{i,j}$

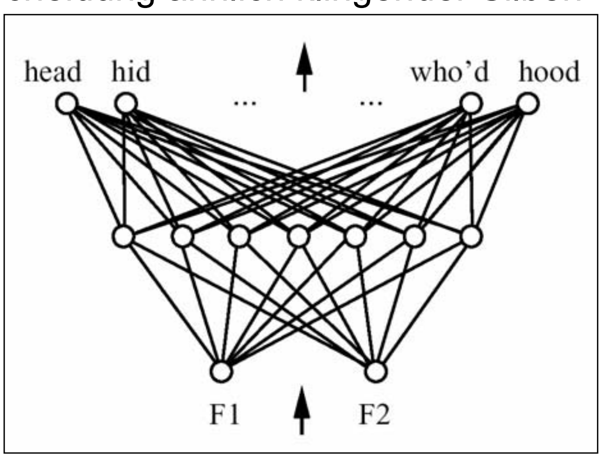
  bevorzugt Netze mit kleineren Gewichten
    - Modifikation der Fehlerfunktion bewirkt Modifikation der Updateregel durch Faktor

Modifikationsmöglichkeiten

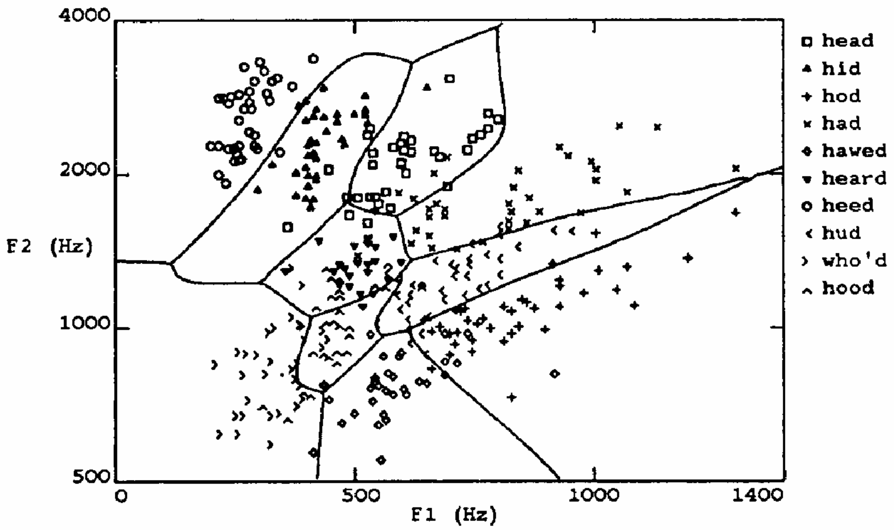
- Backpropagation benötigt häufig mehrere tausend Iterationsschritte für die Anpassung der Gewichte
- Alternative: Line Search: Bestimmung der Richtung des Gewichtsgradienten, die Entfernung bestimmt sich aus dem Minimum der Fehlerfunktion in dieser Richtung
- Vorteil: Schnellere Konvergenz der Gewichte
- Verschiedene Methoden tendieren zu Lernresultaten in verschiedenen lokalen Minima

Spracherkennung

Unterscheidung ähnlich klingender Silben

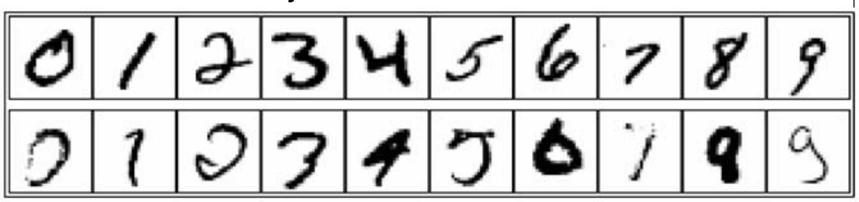


Spracherkennung



Schrifterkennung

Test mit mit
60 000 Zeichen aus je 20x20 Pixeln in 8 Graustufen.



Netz mit 400 Eingabe-Neuronen, 10 Ausgabe-Neuronen.
Beste Ergebnisse bei 300 verdeckten Neuronen (1 Schicht).
Trainingszeit: 7 Tage, Fehler-Rate: 1,6 %
Laufzeit 0,01 Sekunden, Speicherbedarf 0,5 MB

Bessere Klassifizierung mit Support-Vektor-Maschinen (SVM):
Fehler-Rate: 0,6 %