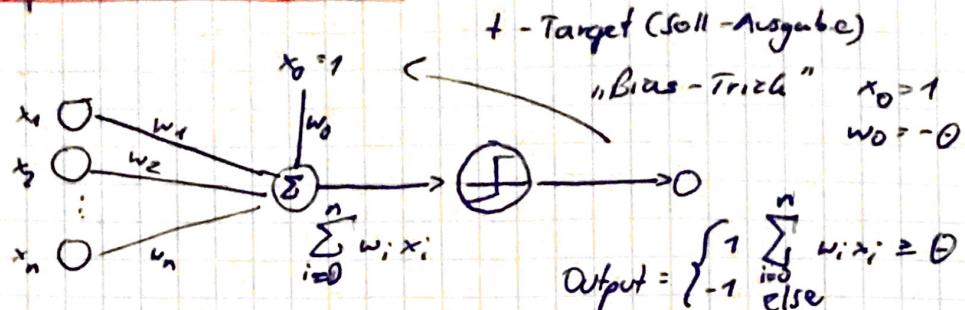


# 9. Maschinelles Lernen 1 Neuronale Netze

$\Rightarrow$  biologisch motiviert

## Perzeptron (Rosenblatt 1960)



$w$  = Gewichtsvektor

$x$  = Eingangsvektor

$O$  = Output (Ist-Ausgabe)

Das Perzeptron realisiert eine Trennhyperebene im  $\mathbb{R}^n$  und die Gewichte  $w_1, \dots, w_n$  beeinflussen die Orientierung dieser Trennebene.

„Bias-Trick“: „bewegen“ der decision boundary weg vom Ursprung bzw. implementiert eine Voraktivierung des Neurons  $\Rightarrow$  Bsp. 2-Input NN

- OR bspw. Gewichte:  $w_0 = -0.3, w_1 = w_2 = 0.5$
- AND bspw. Gewichte:  $w_0 = -0.8, w_1 = w_2 = 0.5$
- XOR nicht linear separierbar

## Perzeptron Lernalgorithmus

Start: Gegeben Lerndatenmenge  $P \cup N$   
 wo wird zufällig gewählt  
 setze  $t := 0$

positive Beispiele  
 negative Beispiele

Testen: Ein Punkt  $x$  in  $P \cup N$  wird zufällig gewählt  
 Falls  $x \in P$  und  $w(t) \cdot x > 0$  gehe zu Testen  
 Falls  $x \in P$  und  $w(t) \cdot x \leq 0$  gehe zu Addieren  
 Falls  $x \in N$  und  $w(t) \cdot x < 0$  gehe zu Testen  
 Falls  $x \in N$  und  $w(t) \cdot x \geq 0$  gehe zu Subtrahieren

Addieren: Setze  $w(t+1) = w(t) + x$   
 Setze  $t := t+1$ . Gehe zu Testen.

Subtrahieren: Setze  $w(t+1) = w(t) - x$   
 Setze  $t := t+1$ . Gehe zu Testen.

Kompat:

1. Initialisiere
2. Wähle  $x$ , berechne Output
3. Aktualisiere Gewichte:

$$w(t+1) = w(t) + (y - \hat{y})x; \quad (y: \text{wahres Label})$$

$t := t+1$   
 wiederhole

Problem beim Lernen: wenn  $w \gg x$ , dann sehr langsame Anpassung  
 Abhilfe: normiere  $x$  nach  $w$

Außerdem: Wenn Updates oszillieren (z.B. weil Updates zu groß/Lernrate nicht gut gewählt), dann werden „fast anti parallel vectors“ erzeugt

Abhilfe: Gradientenabstiegsverfahren mit Delta-Regel + Optimierung

## Gradientenabstiegsverfahren

②  $\Rightarrow$  Aktivierungsfunktion differenzierbar machen

### Fehlerfunktion

$$E(\vec{w}) = \frac{1}{2|D|} \sum_{d \in D} (t_d^{\text{real}} - o_d)^2$$

$D = \text{Lerndaten}$

Lernen: Minimieren von  $E$

Gradient:  $\nabla E(w) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$

### Anpassung des Gewichtsvektors:

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

$\eta = \text{Lernrate}$

$$\frac{\partial E}{\partial w_i} \approx \sum_{d \in D} (t_d - o_d)(-x_{id})$$

### Kapazität des Perzeptions

- ein Perzepton kann nur eine lineare Klassengrenze implementieren  
 $\Rightarrow$  komplexere Funktionen durch Kombination mehrerer Perzeptoren
- durch den Einsatz von Kerneln ist auch ein Perzepton in der Lage nichtlineare Klassengrenzen zu implementieren  $\Rightarrow$

### Kernel Perzepton

$\phi$ : Transformation

Idee: adaptiere Perzepton-Algorithmus für den Einsatz von Kerneln  
 $K(x, \vec{y}) = \langle \phi(x), \phi(y) \rangle$  (Transformation in höherd. Raum)

$\Rightarrow$  lineare Trennung erlaubt nichtlineare Trennung im Ursprungsräum

Anpassung (falsch klassifizierte Beispiele  $(\vec{x}_i, y_i)$ ):

$$\vec{w}_{+1} = \vec{w}_t + y_i \phi(\vec{x}_i)$$

Regel im Algorithmus:

$x_i$ : Anzahl Fehkklassifikationen für  $x_i$

$\Rightarrow$  Gewicht beim Kernel-Perzepton

if  $\text{sign} \left( \sum_{j=1}^l x_j y_j K(\vec{x}_j, \vec{x}_i) \right) \neq y_i$

$$f(\vec{x}) = \sum_{j=1}^n \alpha_j y_j K(\vec{x}_j, \vec{x})$$

### Multilayer Perceptrons

Vor Aktivierungsfunktion: net =  $\sum_{i=1}^n w_i x_i$

Output:  $o = f(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$

typische Aktivierungsfunktionen: Sigmoid  $f(x) = \frac{1}{1 + e^{-x}}$

Tan Hyp.  $f(x) = \tanh(x)$

# ③ Maschinelles Lernen 1 Neuronale Netze

## Backpropagation Algorithmus

Vorgaben: Menge  $T$  Trainingsbeispiele  
Lernrate  $\eta$   
Netz  
Eingabeknoten

Lernziel: Finden einer Gewichtsbeladung  $W$ , die  $T$  korrekt wiedergibt

Vorgehen: Gradientenabstiegsverfahren (allgemeine Deltaregel)

### Algorithmus:

- initialisiere Gewichte mit kleinen zufälligen Werten

Wiederhole:

- Auswahl eines Trainingabeispiels  $d$
- Bestimme Netzaustrgabe
- Bestimme Ausgabefehler (bzw. Sollausgabe)
- Suizessives Rückpropagieren des Fehlers auf die einzelnen Neuronen (Kettenregel)

$$\delta_j = \begin{cases} o_j(1-o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{jk} \\ o_j'(1-o_j)(t_j - o_j) \end{cases}$$

$j \notin \text{output}$   
 $j \in \text{output}$

Downstream( $j$ ):  
direkte  
Nachfolger  
des Neurons  $j$   
 $t_j$ : Zielausgabe

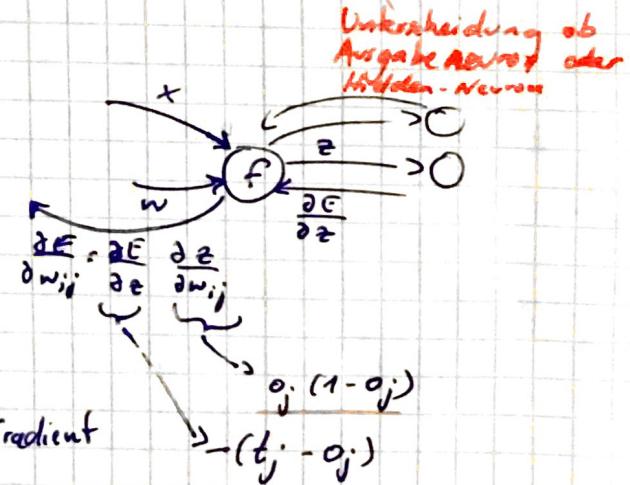
- Anpassung der Gewichte:

$$\Delta w_{ij} = \eta \delta_j x_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}}$$

Bis Abbruchkriterium erfüllt

Kettenregel:  $\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$

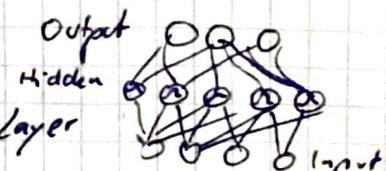
Gradient des  
daraufliegenden  
Layers



## Radial Basis Function - Netz

Topologie/Aufbau: - Feedforward-Netz

- dreischichtig mit einem hidden-layer



- Neuronen des Hidden Layer:

lokale reziproke Reize

$\hookrightarrow$  Gauß-Funktion mit:

$\mu$  - Zentrum (Mittelwert)

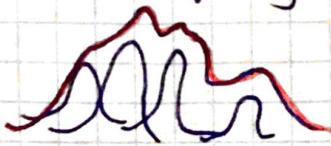
$\sigma$  - Reichweite (Std. Abweichung)

$\hookrightarrow$  nutzt RBF als Aktivierungsfunktion

## (4) Neuronen des Output-Layer:

$$o_j = \sum_{i \in \text{hidden}} o_i w_{ij} \quad j \in \text{output}$$

$\Rightarrow$  gewichtete Überlagerung: 9 Neuronen:



## Neuronen des Hidden Layer:

$$o_i(x) = \frac{-\sum_k S_k(x_k - \mu_{ik})^2}{2\sigma_i^2}$$

Backpropagation für  $\mu$ - und  $\sigma$ -Berechnung der Hidden-Neuronen:

$$\frac{\partial E}{\partial \mu_{ij}} = \sum_n \sum_k (o_k(x^n) - t_k^n) w_{kj} \exp\left(\frac{-\|x^n - \mu_j\|^2}{2\sigma_j^2}\right) \frac{\|x^n - \mu_j\|}{\sigma_j^2}$$

$$\frac{\partial E}{\partial \sigma_j} = \dots$$

## Backprop Nachteile:

- hoher Rechenaufwand (nichtlineare Fkt.)
- lokale Minima
- $\sigma$ /Reichweite kann sehr groß werden  $\rightarrow$  keine lokalen Felder

## Probleme / Optimierung von NN

Leistung ist abhängig von:

- Vorkommen lokaler Minima
- Steigung der Fehlerfläche
- Lernrate

## Abhilfe: Optimierung des Gradientenabstiegs

Idee: Täler/Schluchten besser „umgehen“ (Oberflächenkurven sind in einer Dimension häufig steiler als in anderen)  
 $\rightarrow$  „normaler Gradientenabstieg“ oszilliert



- Momentum Update: Momentum Term wird für Dimensionen erhöht deren Gradienten in dieselbe Richtung zeigen und wird für andere gerebelt  
 $\rightarrow$  Updates von Dimensionen deren Gradienten ihre Richtung häufig ändern werden reduziert

- Manhattan-Training: Normierung der Schrittweite

- Resilient Propagation (RPROP): individuelle Lernrate  $\eta$ , Modifikation mittels Gradientenvergleich:  
 $\begin{cases} \eta(t+1) > \eta(t) & \text{wenn } \text{sign}(\frac{\partial E}{\partial w}(t+1)) = \text{sign}(\frac{\partial E}{\partial w}(t)) \\ \eta(t+1) < \eta(t) & \text{sonst} \end{cases}$

$\Rightarrow$  Beschleunigung auf flachem Plateau

$\Rightarrow$  langsames Anpassen im Minimum

$\Rightarrow$  schnelle Konvergenz

## 5 Maschinelles Lernen 1

### Topologiedeckung

Prinzipiell: je höher Kapazität (VD-Dimension) eines NN, desto eher ist Overfitting möglich

### Verbesserung der Generalisierung von MLNN

- Anpassung der Kapazität

- a) großes Netz wird verkleinert  $\rightarrow$  verhindert Auswendiglernen
- b) kleines Netz wird solange vergrößert, bis Daten gelernt werden können

- Methoden der Regularisierung

a) Weight Decay / Strafterm: 12/11/1 Loss-Term in der Lossfunktion  $\rightarrow$  Bestrafen großer / "punktvoller" Gewichte

b) Dropout: großer Anteil Neuronen + Verbindungen zufällig deaktiviert

- zu Kapazität: b): Cascade Correlation

- Initialisiere 2-schichtiges Netz
- Abbruchkriterien festlegen (Fehlerschranke  $E(w)$ , # Neuronen ...)
- Solange  $E(w) > \text{Fehlerschranke}$ 
  - $\rightarrow$  füge neues Neuron in middle-Schicht ein (Kandidat-Neuron)
  - $\rightarrow$  trainiere neues Neuron
  - $\rightarrow$  trainiere Netz
  - $\Rightarrow$  „Correlation“: Korrelation zw. Kandidat-Neuronen und Output-Neuronen

- zu Kapazität: b): Dynamic Decay Adjustment (DDA)

- wie CC: einzigen neuen Neuronen
- Ausbilden einer RBF-Topologie

### Gewichtsinitialisierung

Wie?:  $\rightarrow$  zufällig, gleichverteilt (oder andere), klein  
 $\hookrightarrow$  Initialisierungsstrategie (MSRA, Xavier)

Wann?: a) Nach jedem Lernbeispiel  $\Rightarrow$  kein „echter“ Gradientenabstieg, kann mehr oscillieren

b) über mehrere Lernbeispiele (Batch)  $\Rightarrow$  Mittelung der Gewichtsänderung über Batch  $\Rightarrow$  „echter“ GD weniger anfällig für Ausreißer

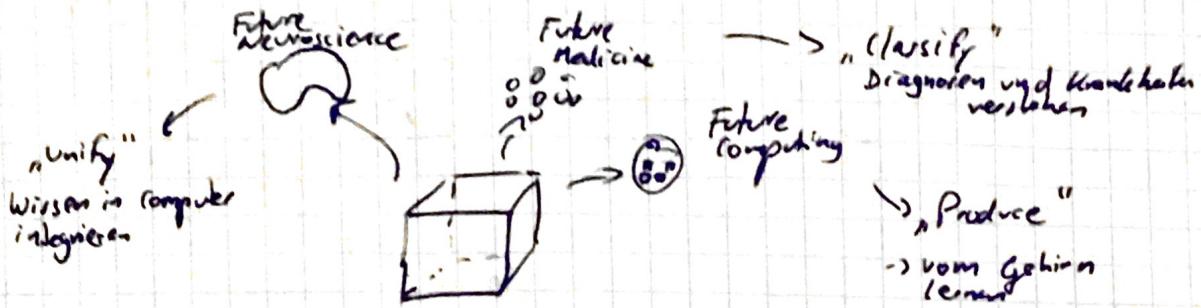
### Anwendungen von MCNN

- Gesichtserkennung / Gewichtsdetektion (Problem: Blickrichtung)
- Programmieren durch Vormachen (PDR)
  - sensorabhängiges Wissen (z.B. Klassifikation von Action Sequence)
  - Klassifikation von statischen Größen (Daten: Gelenkwinkel, Kräfte usw.)
- Autonomes Fahren (z.B. Lenkungssteuerung)
  - Problem: Daten bei Ausnahmesituationen  $\rightarrow$  simulieren

# Human Brain Project (EU-Projekt)

⑥

Ziel: Technologien entwickeln um Verständnis des menschlichen Gehirns zu erhöhen und das Wissen in neue Produkte zu transferieren



## Spiking Neural Networks

- Hintergrund: möglichst reale Abbildung natürlicher NN (pulscodiert)

- Idee: Neuronen feuern nicht wie bei MLP in jedem Propagationzyklus, sondern abhängig von ihrem Potential (ab Folge der Frequenz eingehender Spikes oder der Zeitspanne zw. Spikes)

=> Anzahl und Zeitpunkte der Spikes spielen eine Rolle  
z.B. 20-50 Spikes in kurzer Zeit aktivieren Folgenervon

- Signal besteht aus kurzen elektr. Pulsen
- ~100mV, 1-2ms
- Puls ändert sich bei Propagation entlang Axons nicht
- Spikes selbst tragen keine Information