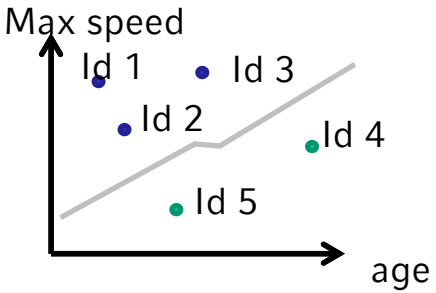
Chapter 6: Classification

1) Introduction
   - Classification problem, evaluation of classifiers, prediction
2) Bayesian Classifiers
   - Bayes classifier, naive Bayes classifier, applications
3) Linear discriminant functions & SVM
   1) Linear discriminant functions
   2) Support Vector Machines
   3) Non-linear spaces and kernel methods
4) Decision Tree Classifiers
   - Basic notions, split strategies, overfitting, pruning of decision trees
5) Nearest Neighbor Classifier
   - Basic notions, choice of parameters, applications
6) Ensemble Classification



Nearest Neighbor Classifiers

- Motivation:
  - Assume data in a non-vector representation: graphs, forms, XML-files, etc.
  - No simple way to use linear classifiers or decision trees

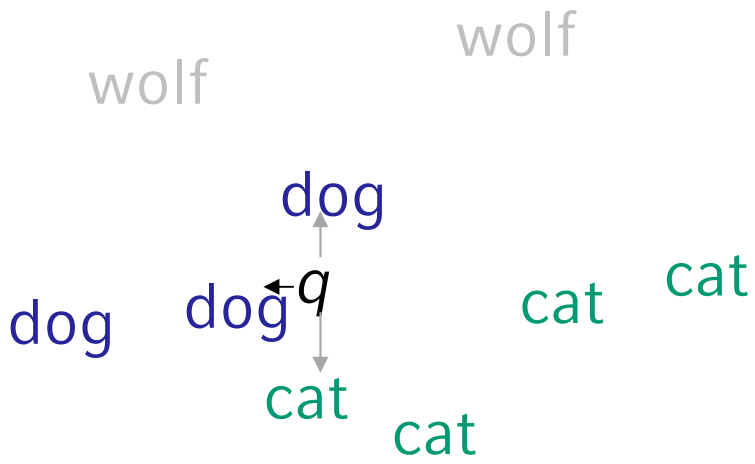
- Possible solutions
  - Definition of an appropriate kernel function for kernel machines (e.g. kernel SVM)
    - Not always clear how to define a kernel
  - Transformation of objects to some vector space (multidimensional scaling, histograms for color or shape distributions, etc.)
    - Difficult to choose an appropriate number of dimensions (intrinsic/fractal dimensionality)
  - Here: direct usage of the similarity of objects for classification
    → Nearest neighbor classifier
    - Requires a similarity function

Nearest Neighbor Classifiers: Example

- Procedure:

  Assign query object q to the class c_j of the closest training object $o \in O$

$$class(q) = class(NN(q))$$
$$NN(q) = \{o \in O \mid \forall o' \in O: d(o,q) \leq d(o',q)\}$$



- Classifier decides that query object q is a dog
- Instance-based learning

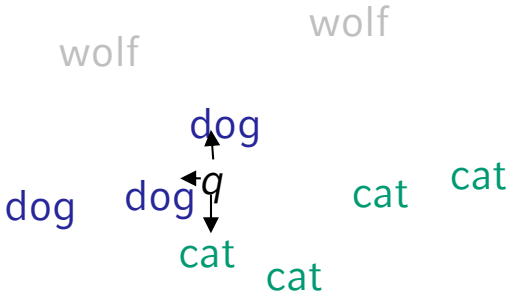
Instance-Based Methods

- Instance-based learning:
  - Store training examples and delay the processing ("lazy evaluation") until a new instance must be classified
  - Typical approaches : *k*-nearest neighbor approach
    - Instances represented as points in an Euclidean space or, more general, as points in a *metric* space
    - Index construction as training phase
      - The classification has to process a huge number of NN-queries → support by e.g. R-tree

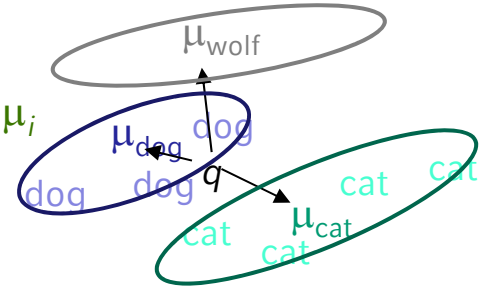
- Eager evaluation
  - Create models from data (training phase) and then use these models for classification (test phase)
  - Examples: Decision tree, Bayes classifier

Nearest Neighbor Classifiers: Variants

- NN Classifier
  - Assign query object to the class c_j of the closest training object
  - Parameter free approach
- k-NN Classifier
  - Consider the k>1 nearest neighbors for the class assignment decision
- Weighted k-NN Classifier
  - Use weights for the classes of the k nearest neighbors



- Mean-based NN Classifier:
  - Determine mean vector μ_i for each class c_j (in training phase)
  - Assign query object to the class c_j of the nearest mean vector μ_i
- Generalization: representative-based NN Classifier
  - Use more than one representative per class



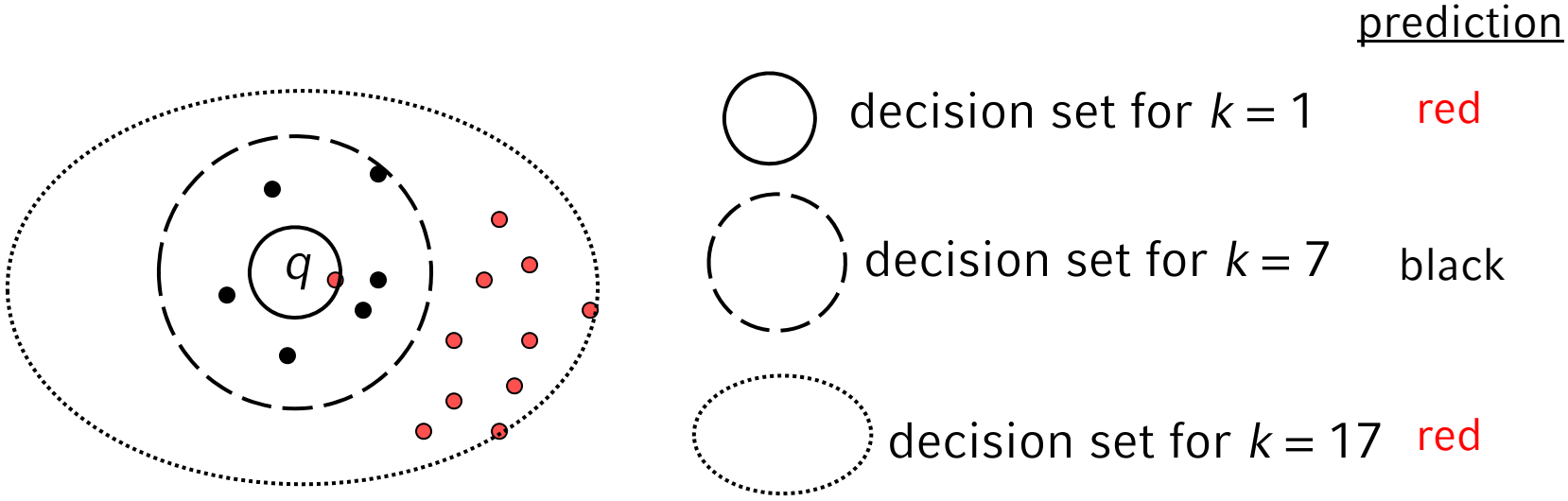
Nearest Neighbor Classifiers: Notions

- Distance function
  - Defines the (dis-)similarity for pairs of objects
- Number *k* of neighbors to be considered
- Decision set
  - Set of *k* nearest neighboring objects to be used in the decision rule
- Decision rule
  - Given the class labels of the objects from the decision set, how to determine the class label to be assigned to the query object?

Nearest Neighbor Classifiers: Parameter *k*

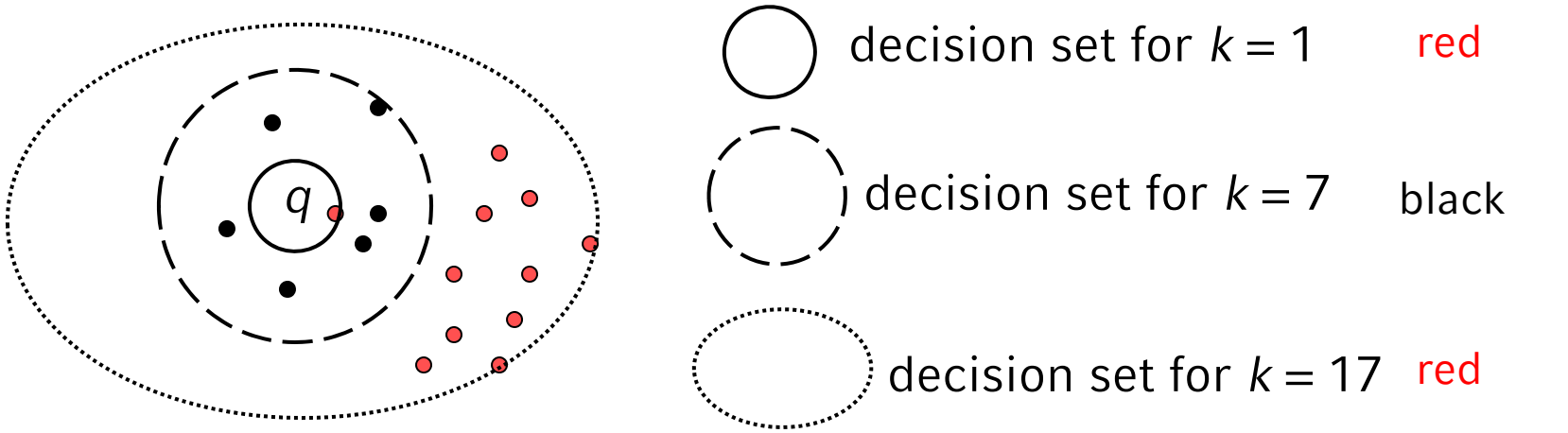
- Tradeoff between *overfitting* and *generalization*:

  Problem of choosing an appropriate value for parameter *k*
  - *k* too small: high sensitivity against outliers
  - *k* too large: decision set contains many objects from other classes
  - Empirically, $1 << k < 10$ yields a high classification accuracy in many cases



Nearest Neighbor Classifiers: Parameter *k*

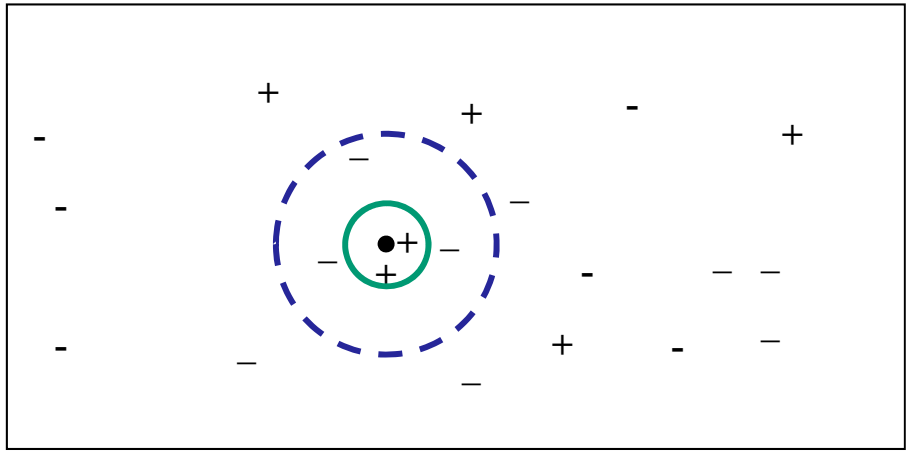
- Tradeoff between *overfitting* and *generalization*:
  Problem of choosing an appropriate value for parameter *k*
  - *k* too small: high sensitivity against outliers
  - *k* too large: decision set contains many objects from other classes
  - Different rules of thumb exist:
    - Based on theoretical considerations: Choose *k*, such that it grows slowly with *n*, e.g. $k \approx \sqrt{n}$ or $k \approx \log(n)$
    - Empirically, $1 << k < 10$ yields a high classification accuracy in many cases



Nearest Neighbor Classifiers: Decision Rules

- Standard rule
  - Choose majority class in the decision set, i.e. the class with the most representatives in the decision set
- Weighted decision rules
  - Use weights for the classes in the decision set
    - Use distance to the query object: $\frac{1}{d(o,q)^2}$
    - Use frequency of classes in the training set, i.e. the *a-priori probability* of the class
  - Example
    - Class a: 95%, class b: 5%
    - Decision set = {**a**, **a**, **a**, **a**, **b**, **b**, **b**}
    - Standard rule yields class "**a**"
    - Weighted rule yields class "**b**" (a-priori based)

Nearest Neighbor Classifiers: Example



Classes + and –

decision set for $k = 2$

decision set for $k = 5$

- Using unit weights (i.e., no weights) for the decision set
  - Simply called *"majority criterion"*
  - rule $k = 2$ yields class „+“, rule $k = 5$ yields class „–“
- Using the reciprocal square of the distances as weights
  - Both rules, $k = 2$ and $k = 5$, yield class „+“
- Using a-priori probability (=frequency) of classes as weights
  - Both rules, $k = 2$ and $k = 5$, yield class „+“

Nearest Neighbor Classification: Discussion

- Pro
  - applicability: training data and distance function required only
  - high classification accuracy in many applications
  - easy incremental adaptation to new training objects
  - useful also for prediction
  - robust to noisy data by averaging *k*-nearest neighbors
- Contra
  - naïve implementation is inefficient
    - requires k-nearest neighbor query processing
    - support by database techniques may help to reduce from $O(n)$ to $O(\log n)$ for n training objects → training phase: create index structure
  - does not produce explicit knowledge about classes
    - But provides some explanation information
  - Curse of dimensionality: distance between neighbors could be dominated by irrelevant attributes
    - To overcome it, stretch axes or eliminate least relevant attributes

Chapter 5: Classification

Neuronale Netze

- *Grundlagen* [Bigus 1996], [Bishop 1995]

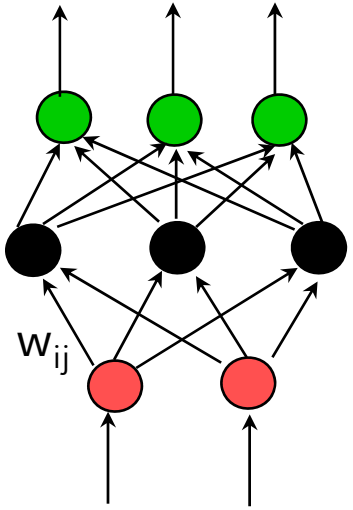
- Paradigma für ein Maschinen- und Berechnungsmodell
- Funktionsweise ähnlich der von biologischen Gehirnen

  Neuronales Netz: Menge von Neuronen, über Kanten miteinander verbunden

  Neuron: entspricht biologischem Neuron: Aktivierung durch Input-Signale an den Synapsen
- Beim menschlichen Gehirn:
  - 10^{11} Neuronen, jedes verbunden mit ~10^4 anderen
  - schnellste Aktionszeit 10^{-3} Sek. (3GHz Prozessor: ~10^{-10})
  - komplexe Entscheidungen sehr schnell (visuelle Erkennung der eigenen Mutter: 10^{-1} Sek. ≈ 100 Schritte)
- Neuronen erzeugen Output-Signal, das zu anderen Neuronen weitergeleitet wird
- Organisation eines neuronalen Netzes:
  - *Input-Schicht*
  - *verborgene Schichten*
  - *Output-Schicht*
  - Knoten einer Schicht mit allen Knoten der vorhergehenden Schicht verbunden

Grundlagen

- Netzwerk: Knoten (Neuronen) und (gerichtete) Kanten
- Kanten besitzen Gewichte
- Funktion eines neuronalen Netzes:



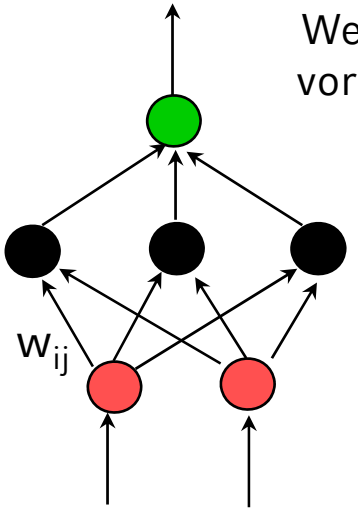
Output-Vektor y

Output-Schicht

verborgene Schicht

Input-Schicht

Input-Vektor x



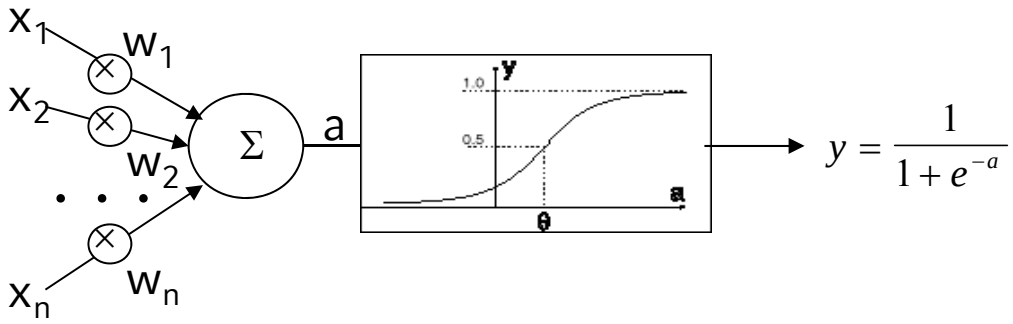
Output 1-dim:
Wert von y ist
vorhergesagte
Klasse

Input-Vektor x

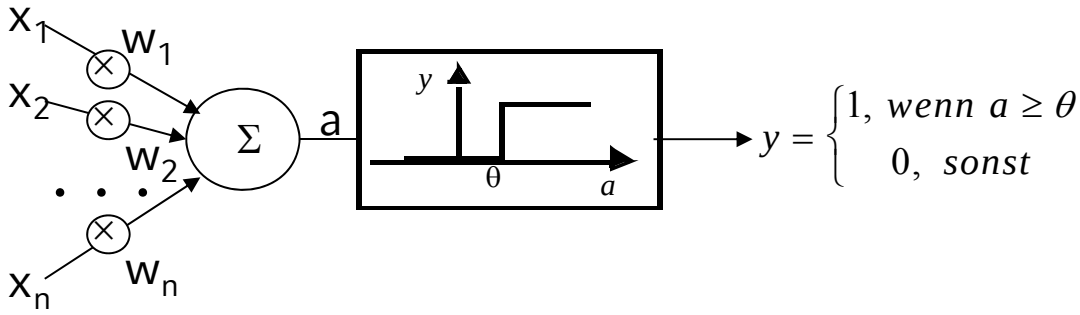
Arten von Neuronen

- allgemeines Neuron (auch: Perzeptron)

Aktivierungswert $a = \sum_{i=1}^{n} w_i \cdot x_i$



- *Threshold Logic Unit (TLU)*

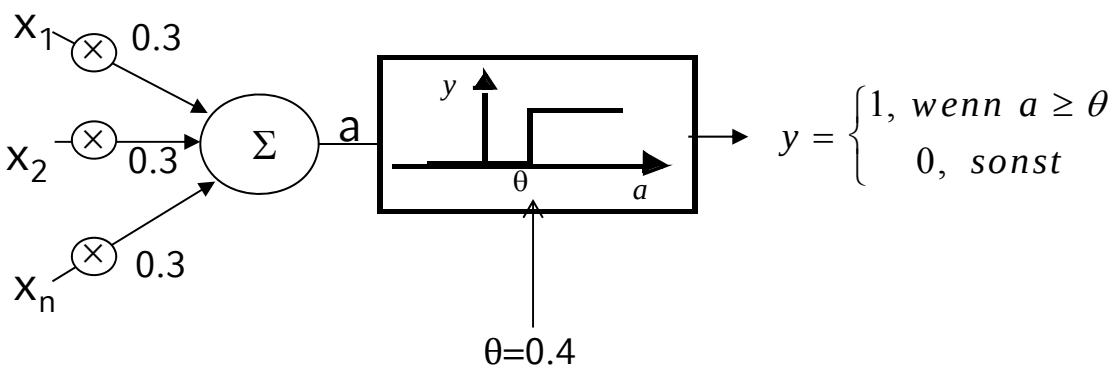


Neuronen

- Beispiel: Modellierung einer booleschen Funktion mit TLU

  (zwei oder mehr Variablen = 1 $\Rightarrow$ y=1, sonst 0)

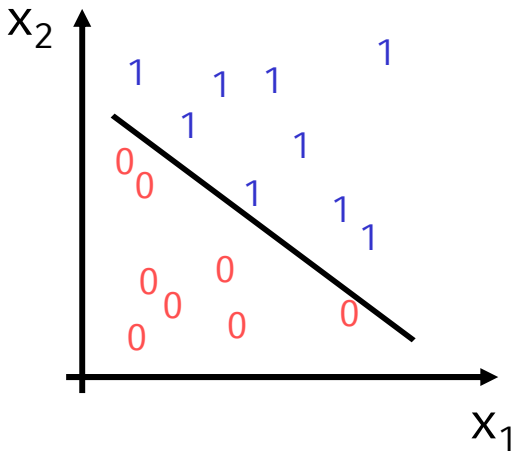
x_1	x_2	x_3	y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0



$\Rightarrow$ Lernen der Gewichte w_i

Neuronen

- Klassifikation mit Hilfe einer TLU
  - repräsentiert eine (Hyper-)Ebene $\sum_{i=1}^{n} w_i \cdot x_i = \theta$
  - links von der Ebene: Klasse 0
  - rechts von der Ebene: Klasse 1



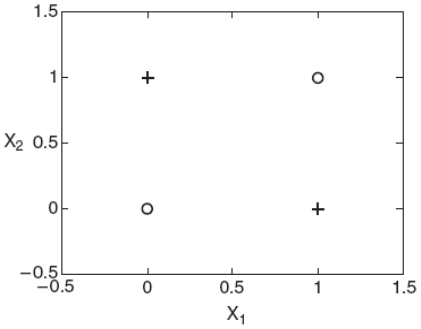
- Trainieren einer TLU
  - Lernen der „richtigen“ Gewichte zur Unterscheidung der zwei Klassen: iterative Anpassung der Gewichte w_{ij}
  - Rotation der durch w und θ gegebene Hyperebene um einen kleinen Betrag in Richtung v, wenn v noch nicht auf der richtigen Seite der Ebene liegt

XOR-Problem

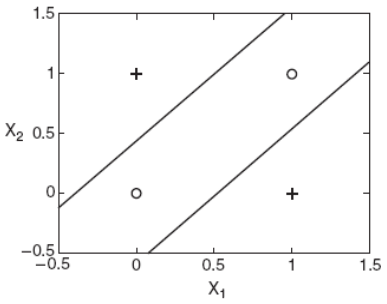
- Nicht linear separierbares Problem

⇒ mehrere innere Knoten (hidden layer) und ein Output-Knoten (zwei-Klassen-Problem, sonst mehr)

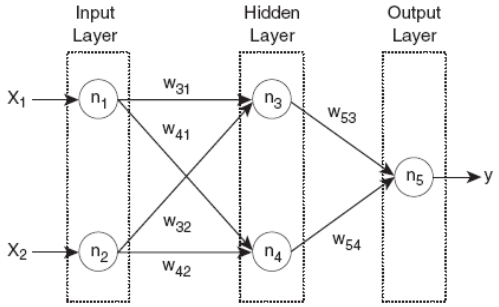
X_1	X_2	y
0	0	−1
1	0	1
0	1	1
1	1	−1



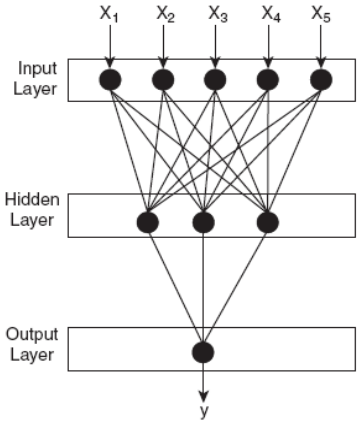
- Varianten: feed-forward (Bild rechts unten) vs. recurrent (auch Verbindungen auf derselben Ebene oder zu vorherigen Ebenen möglich)



(a) Decision boundary.

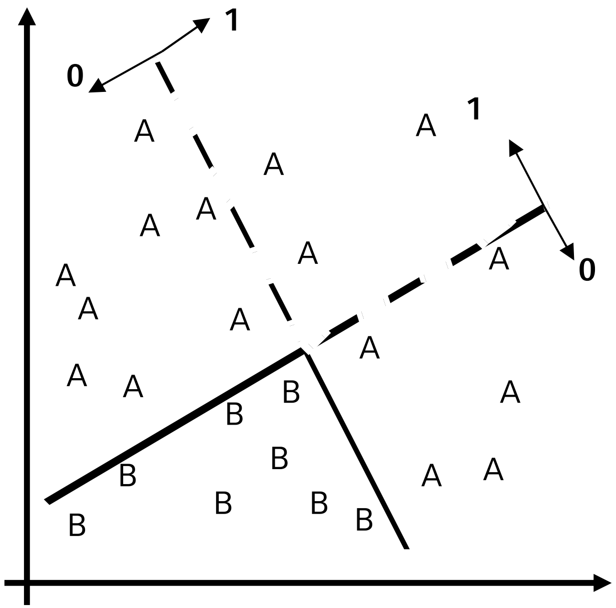


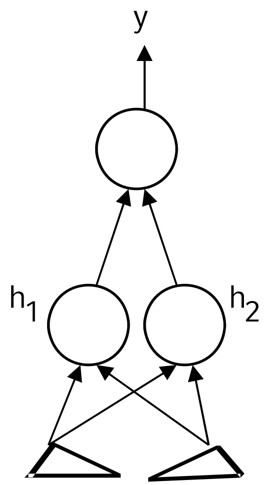
(b) Neural network topology.



Kombination mehrerer Neuronen

- Beispiel





$$h_1 = 0 \;\wedge\; h_2 = 0:\;\; y = 0 \;(Klasse \;\; B)$$

$$andernfalls:\; y = 1 \;(Klasse \;\; A)$$

Lernalgorithmus für komplexe Neuronale Netze

- **Bei Abweichung von vorhergesagter und tatsächlicher Klasse (Trainingsfehler):**

  Anpassung der Gewichte mehrerer Knoten

- **Frage:**

  In welchem Maße sind die verschiedenen Knoten an dem Fehler beteiligt?

- **Anpassung der Gewichte:**
  - durch Gradientenverfahren, das den Gesamtfehler minimiert
  - *Gesamtfehler*: Summe der (quadratischen) Abweichungen des tatsächlichen Outputs *y* vom gewünschten Output *t* für die Menge der Inputvektoren (Least Squares Optimierung)
  - *Voraussetzung:* Output *y stetige* Funktion der Aktivierung *a*

Algorithmus Backpropagation

- **für jedes** Paar(*v*,*t*) // *v* = Input,*t* = gewünschter Output

  „*forward pass*“:

  Bestimme den tatsächlichen Output *y* für Eingabe *v*;

  „*backpropagation*“:

  Bestimme den Fehler (*t* - *y*) der Output-Einheiten
  und passe die Gewichte der Output-Einheiten in die
  Richtung an, die den Fehler minimiert;
  Solange der Input-Layer nicht erreicht ist:
  Propagiere den Fehler auf die nächste Schicht und
  passe auch dort die Gewichte der Einheiten in
  fehlerminimierender Weise an;

Design der Netztopologie

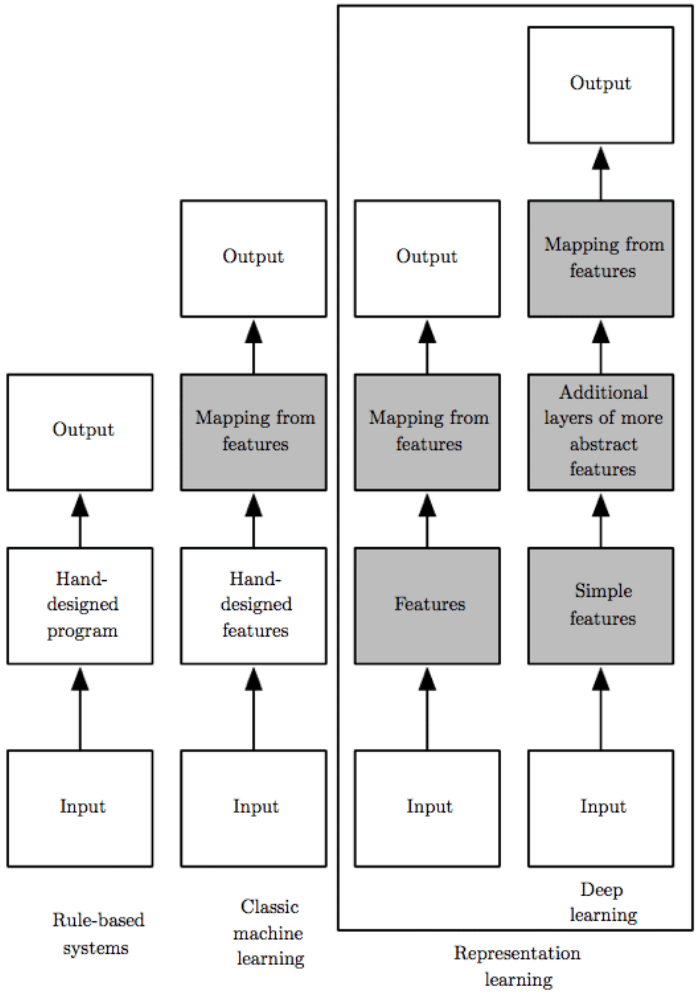
- Bestimmung von
  - Anzahl der Input-Knoten
  - Anzahl der inneren Schichten und jeweilige Anzahl der Knoten
  - Anzahl der Output-Knoten
- Starker Einfluss auf die Klassifikationsgüte:
  - zu wenige Knoten $\Rightarrow$ niedrige Klassifikationsgüte
  - zu viele Knoten $\Rightarrow$ Overfitting

Design der Netztopologie

- In diesem Zusammenhang ein bisschen BB:

Deep Learning:

  - neuronalen Netze, mit mehr als einem Hidden Layer
  - Nutzen mehrerer Neuronen-Schichten: zwischen den Schichten können "neue" Informationen gebildet werden, die eine „neue“ Repräsentation (Abwandlung bzw. Abstaktion) der ursprünglichen Informationen darstellen
  - Dieser Abstraktions-Mechanismus heißt auch ***Representation Learning***
  - Durch diese Abstraktion können Deep Learning Modelle in der Regel sehr gut auf neue Datenpunkte abstrahieren

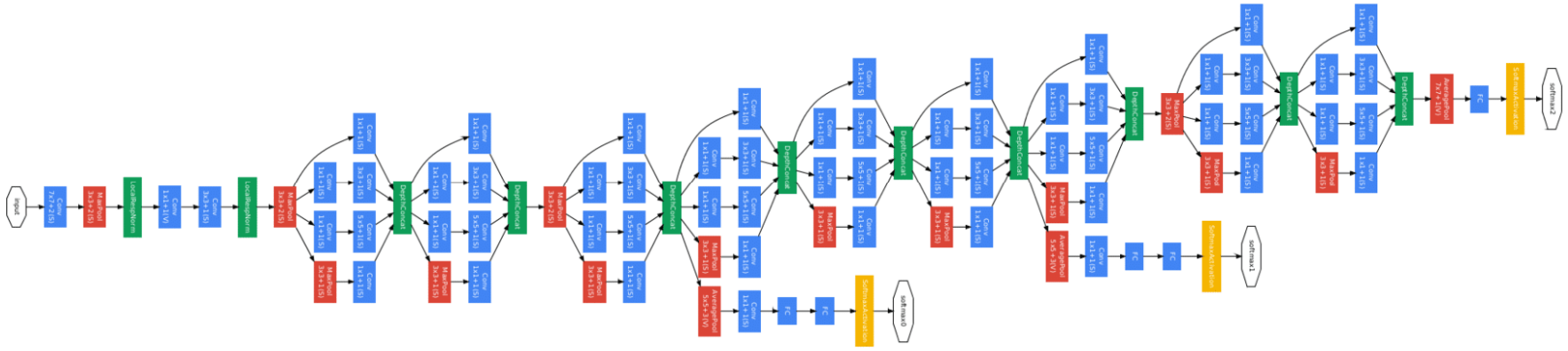


Graphik aus: Stephan Heinz: Deep Learning – Teil 1: Einführung
Siehe:
https://www.statworx.com/de/blog/deep-learning-teil-1-einfuehrung/

Design der Netztopologie

- Beiepiel: Topologie von GoogleNet zur Bildklassifikation

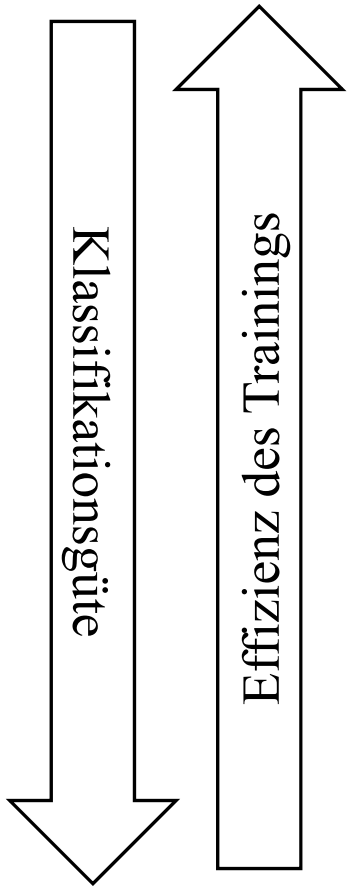
  (nach: Stephan Heinz: Deep Learning – Teil 1: Einführung https://www.statworx.com/de/blog/deep-learning-teil-1-einfuehrung/)



Bestimmung der Netztopologie

(nach [SPSS Clementine 2000])

- Statische Topologie
  - Topologie wird a priori festgelegt
  - eine verborgene Schicht reicht in vielen Anwendungen aus
- Dynamische Topologie
  - dynamisches Hinzufügen von Neuronen (und verborgenen Schichten) solange Klassifikationsgüte signifikant verbessert wird
- Multiple Topologien
  - Trainieren mehrerer dynamischer Netze parallel z.B. je ein Netz mit 1, 2 und 3 verborgenen Schichten
- Pruning
  - Trainieren eines Netzes mit statischer Topologie
  - nachträgliches Entfernen der unwichtigsten Neuronen solange Klassifikationsgüte verbessert wird



Diskussion: Neuronale Netze

Vorteile:

- im Allgemeinen sehr hohe Klassifikationsgüte
- beliebig komplexe Entscheidungsflächen (möglichst einfache Topologie auswählen, um Overfitting zu vermeiden)
- redundante Features relativ problemlos, Gewichte werden klein
- Effizienz der Anwendung

Nachteile:

- schlechte Verständlichkeit (lernt nur Gewichte, aber keine Klassenbeschreibung)
- Ineffizienz des Lernens (sehr lange Trainingszeiten)
- keine Integration von Hintergrundwissen
- empfindlich gegen Fehler in den Trainingsdaten
- mögliche Konvergenz in lokales Minimum