5. Robotik

5.1. Einführung

Der Begriff Roboter hat seinen Ursprung in Science-Fiction-Erzählungen. Er taucht erstmals 1921 in dem Theaterstück „Rossum's Universal Robots“ von Karel Capek auf. In dieser Geschichte entwickeln der Wissenschaftler Rossum und sein Sohn eine chemische Substanz, die sie zur Herstellung von Robotern verwenden. Der Plan war, dass die Roboter den Menschen gehorsam dienen und alle schwere Arbeit verrichten sollten. Im Laufe der Zeit entwickelte Rossum den „perfekten“ Roboter. Am Ende fügten sich die perfekten Roboter jedoch nicht mehr in ihre dienende Rolle, sondern rebellierten und töteten alles menschliche Leben. In den slawischen Sprachen (z.B. Tschechisch oder Polnisch) hat „robota“ die Bedeutung „arbeiten“. Auch Isaak Asimov schrieb mehrere Erzählungen über Roboter. Er formulierte sogar Gesetze für Roboter, deren oberstes besagt, dass Menschen kein Schaden zugefügt werden darf.

Im allgemeinen Sprachgebrauch wird unter „Roboter“ meist eine Maschine verstanden, die dem Aussehen des Menschen nachgebildet ist und/oder Funktionen übernehmen kann, die sonst von Menschen ausgeführt werden. Bei einem menschenähnlichen Aussehen des Roboters spricht man auch von Androiden.

5.1.1. Was versteht man unter Robotern?

Es gibt keine einheitliche Definition des Begriffes Roboter. Die offiziellen Definitionen gehen weit auseinander. Einige Beispiele:

- Die **Japanese Industrial Robot Association** (**JIRA**) unterteilt Roboter in die folgenden Kategorien:
  - Manueller Manipulator: Handhabungsgerät mit mehreren Freiheitsgraden, das kein Programm hat, sondern direkt vom Bediener bewegt wird.
  - Roboter mit festem Aktionsablauf: Handhabungsgerät, das wiederholt nach einem konstanten Bewegungsmuster arbeitet. Das ändern des Bewegungsmusters ist relativ aufwendig.
  - Roboter mit variabler Folge: Handhabungsgerät, wie vorher beschrieben, jedoch mit der Möglichkeit, den Bewegungsablauf problemlos und schnell zu ändern.
  - Playback Roboter: Der Bewegungsablauf wird diesem Gerät einmal durch den Bediener vorgeführt und dabei im Programmspeicher gespeichert. Mit der im Speicher enthaltenen Information kann der Bewegungsablauf beliebig oft wiederholt werden.
  - Numerisch gesteuerter Roboter: Dieses Handhabungsgerät arbeitet ähnlich wie eine NC-gesteuerte Werkzeugmaschine. Der Bediener entwirft für den Roboter ein Computerprogramm für seinen Bewegungsablauf, statt mit ihm die Aufgabe manuell durchzugehen.
  - Intelligenter Roboter: Diese höchste Roboterklasse ist für Geräte gedacht, die über verschiedene Sensoren verfügen und damit in der Lage sind, ihre Umgebung zu verstehen und eine Aufgabe trotz Veränderungen in den Umgebungsbedingungen erfolgreich zu lösen.
- **Robotics Institute of America** (**RIA**): Ein Roboter ist ein programmierbares Mehrzweck-Handhabungsgerät für das Bewegen von Material, Werkstücken, Werkzeugen oder Spezialgeräten. Der frei programmierbare Bewegungsablauf macht ihn für verschiedenste Aufgaben einsetzbar.
- Definition nach **VDI** (Richtlinie 2860): Industrieroboter sind universell einsetzbare Bewegungsautomaten mit mehreren Achsen, deren Bewegungen hinsichtlich Bewegungsfolge und Wegen

bzw. Winkeln frei (d.h. ohne mechanischen Eingriff) programmierbar und gegebenenfalls sensorgeführt sind. Sie sind mit Greifern, Werkzeugen oder anderen Fertigungsmitteln ausrüstbar und können Handhabungs- und/oder Fertigungsaufgaben ausführen.

5.1.2. Autonome mobile Roboter

Die meisten Roboter, die heute in der Industrie verwendet werden, sind Manipulatoren, die an einem festen Arbeitsplatz eingesetzt werden und nicht mobil sind. Diese Industrieroboter funktionieren nur in einer strukturierten kontrollierten Umgebung.

Mobile Roboter unterscheiden sich davon grundlegend. Sie können ihren Standort durch Lokomotion verändern.

Arten von mobilen Robotern:

1. Automatisiertes spurgeführtes Fahrzeug: Diese führen Transportaufgaben entlang vorgegebener Routen aus. Jede Änderung der Route ist kostspielig, und alle unvorhergesehenen Veränderungen (z.B. Hindernisse, die den Weg blockieren) können bewirken, dass die Aufgabe nicht erfolgreich ausgeführt wird.

2. Autonome mobile Roboter: Sie zeichnen sich durch folgende Merkmale aus:
   - Sensoren vorhanden
   - aufgabenorientierte und implizite Programmierung
   - Anpassung an Veränderungen in seiner Umgebung
   - Lernen aus Erfahrung und entsprechende Verhaltensänderung
   - Entwicklung eines internen Weltbildes
   - Selbstständige Planung und Durchführung komplexer Aufgaben in unbekannter Umgebung (Navigationsplanung)
   - Manipulation physikalischer Objekte in der realen Welt

In der Vorlesung werden vorwiegend autonome mobile Roboter behandelt.

5.1.3. Arbeitsgebiet Robotik

Robotik ist ein sehr vielseitiges Gebiet. Es hat Beziehungen zu folgenden Disziplinen:

1. Informatik
2. fast alle Gebiete der KI (Bilderkennung, Spracherkennung, Lernen, Multiagentensysteme, Planung, ...)
3. Elektrotechnik (z.B. Hardware, Sensorik)
4. Maschinenbau (z.B. Kinematik, mechanischer Aufbau)
5. Mathematik (Fehlerkalkulation, statistische Auswertung)
6. Kognitionswissenschaften
7. Psychologie
8. Biologie (Tierverhalten)

Roboter sind real vorhanden und es bewegt sich etwas. Man verzichtet auf eine Simulation, was bei der Vielfalt der Möglichkeiten in der realen Welt oft nicht möglich ist.

5.1.4. Anwendungen autonomer mobiler Roboter

1. Aufgaben, die Transport, Überwachung, Führung (z.B. von Personen), Inspektion usw. erfordern
2. Aufgaben, bei denen die Umgebung für Menschen unzugänglich oder gefährlich ist
   - Unterwasserroboter
   - Planetenrover
   - Arbeiten in verseuchter Umgebung
   - Bergung von Verletzten
3. Biomimetische Robotik: Überprüfung und Verbesserung von Hypothesen über intelligentes Verhalten, Wahrnehmung und Kognition (z.B. Navigation der Wüstenameise, Sechsbeinige Fortbewegung der Grille)

5.1.5. Geschichte der Robotik

Die Vorläufer heutiger Roboter können in den frühen mechanischen Geräten (Automaten) gesehen werden. Hinweise auf Automaten und bewegliche Statuen in Ägypten und Griechenland finden sich schon einige hundert Jahre vor Christus. Schwerpunkte des Automatenbaus waren dann später Frankreich, die Schweiz und Deutschland (Augsburg, Nürnberg). Erste Androiden sind im 18. Jahrhundert gebaut worden.

1. um 1738: J. d. Vaucansen (Frankreich) baut Musik spielende mechanische Puppen in menschlicher Größe (Flötenspieler, Tamburinspieler) und eine Ente. Der Mechanismus der Ente besteht aus über 1000 Teilen. Sie kann „fressen“ und „Exkremente ausscheiden“.
2. um 1774: P. Jaquet-Droz sein Sohn und J.-F. Leschot (Schweiz) bauen drei Androiden (Zeichner, Schriftsteller und Musikerin).
3. 1805: H. Maillardet konstruiert eine Bilder malende mechanische Puppe
4. um 1805: J.M. Jacquard erfindet den programmierbaren Webstuhl. Dieser wird durch Lochkarten gesteuert und kann als spezialisierter Roboter betrachtet werden.
5. 1810: J. Gottfried und F. Kaufmann (Dresden) bauen einen Trompeter

Die Entwicklung heutiger (Industrie)Roboter begann aber erst Mitte des 20. Jahrhunderts. Wichtige Ereignisse in der Entwicklung mobiler Roboter waren:

1. 1950: William Grey Walter entwickelt zwei mobile Roboter, die Aufgaben wie Hindernisausweichen und Phototaxis erlernen können (durch Veränderung der Ladung eines Kondensators, der das Verhalten des Roboters steuerte).
2. 1958: M. Minsky versucht einen Tischtennis spielenden Roboter zu bauen. Wegen der technischen Schwierigkeiten wird daraus allerdings ein Roboter, der einen Ball mit Hilfe eines Korbes fängt.
3. 1968: Entwicklung des mobilen experimentellen Roboters „SHAKEY“ am SRI (Stanford Research Institute); Integration von Sichtsystem (Kamera) und taktilen Sensoren Hindernisausweichen und Objektmanipulation in sehr strukturierter Umgebung (quader- und keilförmige Objekte). Es gibt allerdings noch technische Schwierigkeiten mit der Roboterhardware.
4. 1977: JPL ROVER für Planetenexploration (Jet Propulsion Laboratory in Pasadena) Kamera, Laserentfernungsmesser, Berührungssensoren, Kompass

5. 1978: mobiler Roboter CART (Stanford), die Aufgabe dieses Roboters ist, mit Hilfe eines Kamerasensors Hindernissen auszuweichen. Er benötigte 15 Minuten um 9 Bilder auszuwerten und sein zweidimensionales Weltmodell zu überarbeiten. Danach bewegt er sich einen Meter weiter.
6. 1978: HILARE bei LAAS in Toulouse digitale Bildverarbeitung, Laserentfernungsmesser und Ultraschallsensoren für Navigation (Rhode Island University)
7. 1984: Entwicklung des anthropomorphen (menschenähnlich) Roboters WABOT-2 mit der Fähigkeit des Keyboardspielens mit 2 Händen, 2 Beinen sowie Kamera (Waseda University, Tokio)
8. 1989: Erster Roboter-Wettbewerb am MIT AI Laboratory. Studenten erhalten einen Satz von Bauteilen und Computer und werden aufgefordert, eigene Problemstellungen zu entwickeln und zu lösen (Staubsauger, Laserstrahl-Spielzeugroboter, autonom manövrierende Kleinluftschiffe, kleine Geländefahrzeuge). Vgl. dazu http://www.mit.edu:8001/courses/6.270/home.html.
9. 1997: Erste Weltmeisterschaft im Roboterfußball (RoboCup) in Japan, wird in mehreren Klassen ausgetragen:
   - Simulation
   - Small-Size League (5 reale Roboter, max.18cm, Tischtennisplatte, Deckenkamera)
   - Middle-Size League (3 Spieler + Torwart, Feld ca. 9m x 5m)
   - Sony Legged Robot League

Vgl. dazu http://www.robocup.org/02.html.

5.1.6. Beispiele für mobile Kleinroboter

1. TuteBot

TuteBot kommt von Tutorial Robot (Lernroboter). Er ist wohl der einfachste Roboter der Welt, der aber nichtsdestotrotz ein funktionierendes System darstellt. Der TuteBot besitzt keinen Mikroprozessor, sondern besteht lediglich aus:

1. zwei Motoren
2. zwei Rädern
3. zwei Schaltern
4. Kollisionssensor
5. zwei Potentiometer
6. Relais
7. einige weitere elektronische Bauteile, die alle recht einfach zu integrieren sind

Abbildung 5.1 zeigt den TuteBot.

Die Reflexe des TuteBot lassen sich einfach durch Einstellen zweier Potentiometer justieren. Fähigkeiten des TuteBot:

1. Der Roboter bewegt sich vorwärts auf einer Geraden, bis er auf ein Hindernis stößt
2. Danach setzt er zurück, dreht sich eine bestimmte Zeit lang nach links und nimmt dann seine Geradeausfahrt wieder auf.
3. Der Roboter fährt auf einer Kreisbahn nach links vorwärts. Wenn er auf ein Hindernis trifft, fährt der Roboter zurück, dreht auf der Stelle nach rechts und setzt dann seine Fahrt auf der Kreisbahn nach links vorne fort. Dieses Verhalten nennt man auch Wandverfolgung.

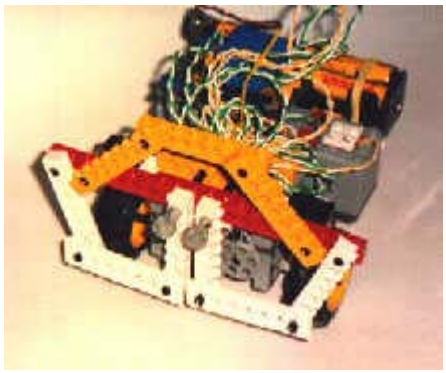


Abbildung 5.1

Die ersten beiden Verhalten lassen sich durch verschiedene Mechanismen erzielen.

1. Man könnte sich vorstellen, dass der TuteBot Stuhlbeine und Wände als solche erkennt und in jedem Fall neu entscheidet, ob er zurückweicht und um wie viel Grad er sich dreht. Er besitzt also eine differenzierte Wahrnehmung und ein entsprechendes Modell seiner Umwelt.
2. Der TuteBot besitzt aber nur einen einfachen Analogschaltkreis als Steuerungssystem. Dieser dirigiert die beiden Räder so lange geradeaus, bis ein Kollisionssensor (Bumper) auf der Vorderseite einen Zusammenstoß meldet. Das Bumpersignal bewirkt einen Richtungswechsel an beiden Motoren, woraufhin der Roboter zurücksetzt. Die Drehung wird durch eine Zustandsänderung oder eine Zeitschleife im System hervorgerufen. Diese wird durch eine (bzw. zwei, d.h. für jedes Rad eine) Kondensator-Widerstandsschaltung (RC-Schaltung) realisiert. Sind die RC-Glieder auf beiden Rädern unterschiedlich gesetzt, dreht eines der Räder länger rückwärts als das andere. Dies bewirkt, dass der TuteBot auf der Stelle dreht. Wenn der Roboter seine Vorwärtsbewegung dann wieder aufnimmt, „blickt“ er nicht mehr in dieselbe Richtung. So wird vermieden, dass er erneut in das Hindernis fährt, mit dem er zuvor zusammengestoßen ist. Mit einer ähnlichen Strategie kann die dritte Fähigkeit realisiert werden. Man erreicht dies, indem sich ein Rad während der Fahrt schneller drehen darf als das andere (z.B. indem ein Widerstand mit einem Motor in Reihe geschaltet wird).

2. Rug Warrior

Dieser Roboter besitzt einen Mikroprozessor. Durch den Einsatz von Software kann eine größere Anzahl von Sensoren und Aktuatoren eingesetzt werden. Trotz einer großen Anzahl an verschiedenen Sensoren bleibt dieser Roboter überschaubar und preiswert.

Einige typische Daten der Rug Warrior Roboter:

- Spannungsversorgung für die Schaltlogik: 4,6 - 7 Volt
- Spannungsversorgung für die Motoren: 4 - 9 Volt
- Hinderniserfassungsreichweite: 15 - 20 cm
- Robotergeschwindigkeit: ca 45 cm/s
- Gewicht (mit Batterien): ca 900g
- Durchmesser der Räder: ca 6 cm
- Durchmesser des Roboters: ca 16 cm
- Höhe des Roboters: ca 10 cm

Abbildung 5.2 zeigt den Rug Warrior.

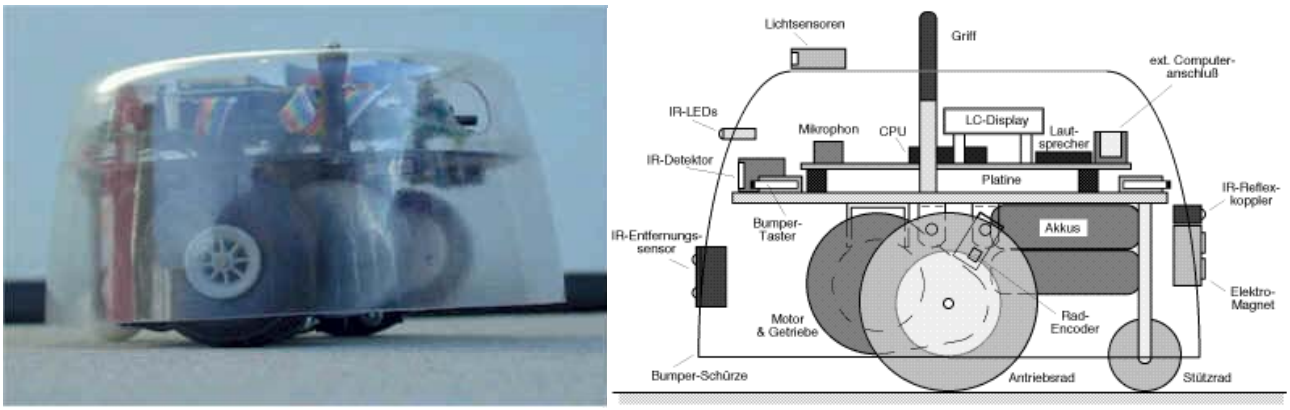


Abbildung 5.2

Fähigkeiten des Rug Warrior:

- Vorwärtsbewegen auf einer Geraden, bis er auf ein Hindernis stößt oder ein Hindernis erkennt.
- Ausweichen in einer neuen Richtung
- Wandverfolgung
- Reagieren auf Geräusche (aktiviert Mikrophonschaltung und spielt ein Lied)
- Verfolgen von intensiven Lichtpunkten
- Personenverfolgung (hält aber auch Heizkörper für Personen)
- Verstecken in dunklen Ecken
- Abstandsmessung zu Hindernissen (IR-Entfernungsmesser)
- Erkennen von Markierungen auf dem Boden (Reflexkoppler)

Auch dieser Roboter besitzt keine explizite geometrische Repräsentation der Welt, nach der er seine Aktionen plant. Statt dessen verfügt er über eine Reihe von Regelkreisen, die für eine enge Kopplung von Wahrnehmung und Handlung sorgen. Sein Steuerungssystem arbeitet mit einer sogenannten Subsumptionsarchitektur. Dieser Begriff bezeichnet eine bestimmte Weise der Strukturierung von Steuerungssystemen, die sich dadurch auszeichnet, dass einzelne bedingte Verhalten hierarchisch gegliedert werden. Dieser Ansatz greift weder auf Weltwissensmodelle noch auf die Erfassung und Auswertung von Sensordaten zurück. Subsumption beschreibt die Art und Weise, wie zwischen mehreren Ebenen bedingter Verhalten entschieden wird. Bei einer Subsumptionsarchitektur werden die Verhalten bei der Entwicklung des Steuerungssystems so definiert, dass niedrigerwertige Verhalten bei Auslösen von höherwertigen Verhalten deaktiviert werden.

Beispiel

Verhalten: Umherwandern | niedrige Priorität
Verhalten: Lichtverfolgung | hohe Priorität

Der Roboter wird solange in einer beliebigen Richtung herumfahren, bis jemand eine Lichtquelle auf ihn richtet. Dann wird das Umherwandern so lange unterdrückt, wie die Lichtquelle auf den Roboter gerichtet ist. Der Roboter beginnt, sich auf die Lichtquelle zu zu bewegen. Beim Abschalten des Lichts wird das Verhalten Lichtverfolgung wieder deaktiviert. Die Unterdrückung des Herumwanderns wird beendet und der Roboter beginnt wieder ziellos umherzuwandern.

An der TU Chemnitz wird jährlich ein Praktikum mit abschließenden Wettbewerb RoboKing durchgeführt. Weitere Informationen: http://www-user.tu-chemnitz.de/stj/lehre/praktiku.htm.

3. Khepera

Khepera ist ein mobiler Kleinst-Roboter, der an der TH Lausanne entwickelt wurde. Er hat einen Durchmesser von 55mm, eine Höhe von 30mm und ein Gewicht von 70g. Die geringe Größe ermöglicht Versuche auf engem Raum (Büro, Schreibtisch). Khepera besitzt 8 Infrarot Entfernungs-/Lichtsensoren und ein Prozessorsystem mit hoher Rechenleistung. Abbildung 5.3 zeigt zwei Kheperas mit verschiedenen Aufbauten.

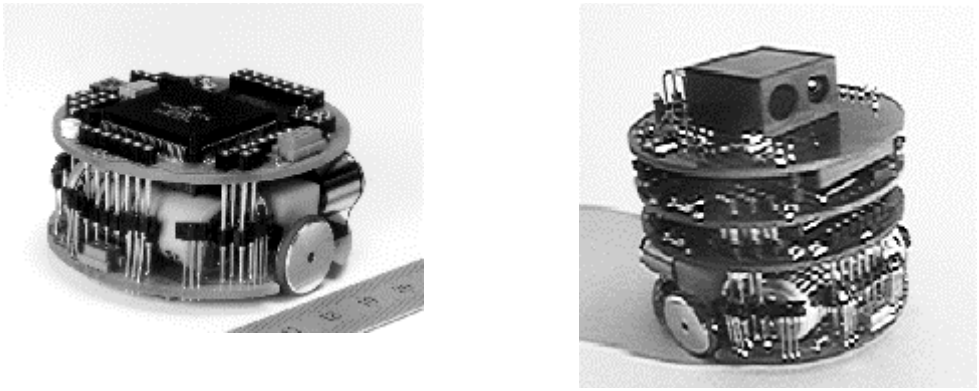


Abbildung 5.3

Weitere Informationen: http://www.k-team.com/robots/khepera/khepera.html

4. Pioneer2 - AT

Dieser an der Fakultät für Elektrotechnik und Informationstechnik der TU Chemnitz vorhandene Roboter besitzt wesentlich mehr Fähigkeiten:

- Geländegängigkeit
- Kompass / Neigungssensor
- Wegmessung
- Ultraschallsensoren
- hochgenaue Lasermessung bis 40m Entfernung
- Farbkamera, Schwenk-Neigekopf, veränderbare Brennweite
- kraftgeregelter Greifer mit 2 Freiheitsgraden
- Sprachein- und ausgabe

Abbildung 5.4 zeigt einen Pioneer2 AT.

Abbildung 5.4

Technische Daten: http://www.infotech.tu-chemnitz.de/~robotik-ag/hardware/techdat.htm
Weitere Informationen: http://www.activrobots.com/

5. Joker Robotics

Hier werden eine Reihe von mobilen Kleinrobotern angeboten:

- Rug Warrior Pro (siehe oben)
- EyeBot Controller (Ein leistungsfähiger 32 Bit Mikrocontroller speziell für mobile Robotikanwendungen)
- EyeWalker robot (Laufmaschine mit 12 Servoantrieben - 6 Beine)
- Soccer EyeBot (Eine mobile Plattform für Forschung und Lehre - nicht nur zum Fussballspielen)

Abbildung 5.5 zeigt den EyeWalker robot.

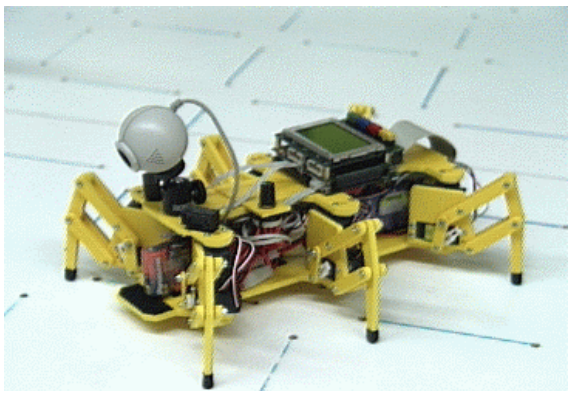


Abbildung 5.5

Weitere Informationen: http://www.joker-robotics.com/
http://www.eyebot.de/

6. Lego-Roboter

Wettbewerb (MIT): http://www.mit.edu:8001/courses/6.270/home.html
Schüler-AG in Chemnitz: http://www.infotech.tu-chemnitz.de/ robotik-ag/index.html

Abbildung 5.6 zeigt Schüler der Schüler-AG.

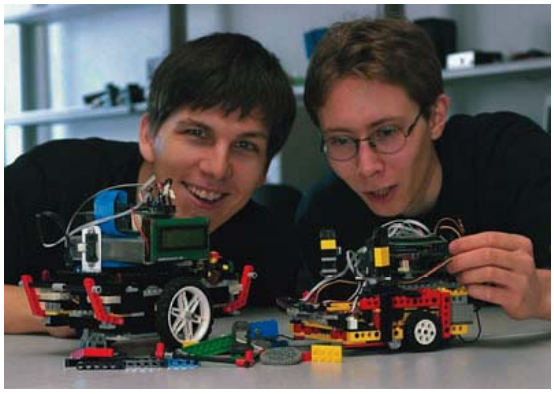


Abbildung 5.6

7. Fischertechnik

Hier werden folgende Baukästen angeboten, um eigene Roboter zu bauen:

- Mobile Robots (erkennt Hindernisse, Abgründe (Schreibtischkanten), folgt einer Lichtquelle oder einer Spur), http://www.knobloch-gmbh.de/¯scher/¯-30400.htm
- Industry Robots, http://www.knobloch-gmbh.de/¯scher/¯-30408.htm
- Pneumatics Robots, http://www.knobloch-gmbh.de/¯scher/¯-34948.htm

Abbildung 5.7 zeigt einen Fischertechnik-Roboter des Alfried-Krupp-Schülerlabors.

Abbildung 5.7

5.2. Aufbau und Teilsysteme eines Roboters

Wir unterscheiden folgende Teilsysteme:

- Mechanik
- Kinematik
- Achsregelung und Antrieb (Motoren, Stromversorgung)
- Sensoren
- Steuerung
- Programmierung

5.2.1. Mechanik

Durch die *mechanische Struktur* des Roboters ist seine Fähigkeit bestimmt, in seiner Umgebung mit dem (*End*)*Effektor* eine definierte *Position* und *Orientierung* einzunehmen bzw. eine *Bewegungsbahn* (Trajektorie) im dreidimensionalen euklidischen Raum abzufahren.

3 Teilsysteme:

- Fahrzeug (Lokomotion)
- Roboterarm (Führung des Effektors)
- Effektor, Endeffektor, Hand (Manipulation von Werkstücken und Werkzeugen)

Jedes dieser Teilsysteme weist eine spezifische Beweglichkeit auf, die sich aus seinen *Freiheitsgraden* ergibt. Im 3D-Raum besitzt ein frei beweglicher starrer Körper 6 Freiheitsgrade:

- *Translation* entlang der x-, y- und z-Achse
- *Rotation* um die x-, y- und z-Achse

Die kombinierte Ausnutzung dieser Freiheitsgrade erlauben die beliebige *Positionierung* und *Orientierung* eines Objektes im 3D-Raum.

Fahrzeug

Ein auf dem Boden bewegliches Fahrzeug besitzt i.a. 3 Freiheitsgrade:

- Translation auf der Bodenfläche (x- und y-Achse)
- Drehung um die senkrecht zur Bodenfläche stehende z-Achse

Bei Robotersystemen für Anwendungen im Weltraum oder Unterwasser kann das Fahrzeug bis zu 6 Freiheitsgraden besitzen. Manchmal übernehmen Zusatzachsen (Bodenschienen) oder Portale (Laufkräne unter der Decke) die Funktion des Fahrzeugs (1 Freiheitsgrad)

Die *Fortbewegung* kann mit Hilfe von *Rädern*, *Ketten* oder *Beinen* erfolgen.

Roboter mit Rädern haben eine einfache Mechanik und lassen sich leicht zusammenbauen. Der prinzipielle Nachteil von Rädern ist, dass sie auf unebenem Grund nicht so leistungsfähig sind wie auf ebenem Boden.

Systeme mit Beinen oder Ketten benötigen in der Regel eine komplexere und schwerere Hardware als Systeme mit Rädern, die für dieselbe Nutzlast ausgelegt sind. Bei Robotern, die in einer natürlichen Umgebung operieren sollen, sind Ketten allerdings geeigneter, da die größere Lauffläche den Roboter in die Lage versetzt, relativ große Hindernisse zu überwinden. Außerdem sind Kettenfahrzeuge weniger anfällig für Unebenheiten, z.B. lose Erde oder Steine. Der größte Nachteil von Ketten ist jedoch ihr geringer Wirkungsgrad. Gehende Roboter können in der Regel unebenes Gelände besser überwinden als Fahrzeuge mit Rädern oder Ketten. Allerdings gibt es viele Schwierigkeiten. Jedes Bein muss mit mindestens zwei Motoren ausgestattet sein. Außerdem ist der Fortbewegungsmechanismus komplexer und folglich fehleranfälliger. Darüber hinaus sind mehr Steueralgorithmen erforderlich, weil mehr Bewegungen zu koordinieren sind. Eine optimale Steuerung von gehenden oder laufenden Maschinen gibt es derzeit noch nicht.

Roboterarm:

Der Roboterarm führt den Effektor. Er besteht aus Armelementen (Gliedern), die über Bewegungsachsen (Gelenken) miteinander verbunden sind

Effektor:

Er bewirkt die Interaktion des Roboters mit der Umwelt. „Effektor“ ist ein Oberbegriff für:

- *Greifer*(Systeme) zur Handhabung und Manipulation von Objekten
- *Werkzeuge* zur Werkstückbearbeitung
- *Messmittel* zur Ausführung von Prüfaufträgen
- *Kamera* bei einem nur beobachtenden Roboter

5.2.2. Kinematik

Die *Kinematik* beschäftigt sich mit der Geometrie und den zeitabhängigen Aspekten der Bewegung, ohne die Kräfte, die die Bewegung verursachen, in die Überlegungen mit einzubeziehen. Parameter, die für die Kinematik von Interesse sind:

- Position (Verschiebung, Drehung)
- Geschwindigkeit
- Beschleunigung
- Zeit

Die Kinematik legt die Beziehung zwischen der Lage des Effektors bezüglich der Roboterbasis und der Einstellung der *Gelenkparameter* fest. Parameter zur Beschreibung der Gelenke sind *Drehwinkel* bzw. *Translationswege*.

5.2.3. Achsregelung und Antriebe

Achsregelung und Antriebe bewirken die Fortbewegung und die Bewegung der Glieder des Roboterarms bzw. des Effektors. Durch den Antrieb wird die erforderliche Energie auf die Bewegungsachsen übertragen. Der Antrieb muss auch die Kräfte und Momente durch das Gewicht der Glieder des Roboters und der Objekte im Effektor kompensieren. Energie wird also auch dann benötigt, wenn der Roboter sich nicht bewegt. Abbildung 5.8 zeigt den prinzipiellen Aufbau eines Antriebssystems.

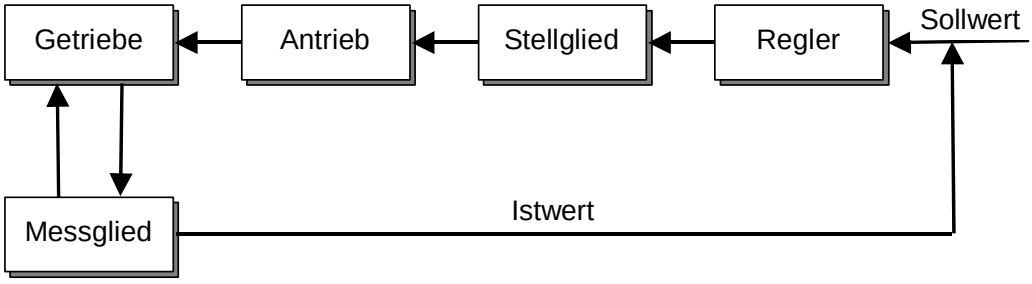


Abbildung 5.8

Es gibt drei Antriebsarten, die im Folgenden mit ihren Vor- und Nachteilen beschrieben werden:

- **pneumatisch**
  - komprimierte Luft bewegt Kolben, kein Getriebe
  - billig, einfacher Aufbau, schnelle Reaktionszeit, auch in ungünstigen Umgebungen brauchbar
  - laut, keine Steuerung der Geschwindigkeit bei der Bewegung, nur Punkt-zu-Punkt-Betrieb, schlechte Positioniergenauigkeit
  - Einsatz für kleinere Roboter mit schnellen Arbeitszyklen und wenig Kraft, beispielsweise zur Palettierung kleinerer Werkstücke

- **hydraulisch**
  - Öldruckpumpe und steuerbare Ventile
  - sehr große Kräfte, mittlere Geschwindigkeit
  - laut, zusätzlicher Platz für Hydraulik, Ölverlust führt zu Verunreinigung, Ölviskosität erlaubt keine guten Reaktionszeiten und keine hohen Positionier- oder Wiederholgenauigkeiten
  - Einsatz für große Roboter, beispielsweise zum Schweißen

- **elektrisch**
  - Motoren
  - wenig Platzbedarf, kompakt, ruhig, gute Regelbarkeit der Drehzahl und des Drehmoments, hohe Positionier- und Wiederholgenauigkeit, daher auch Abfahren von Flächen oder gekrümmten Bahnen präzise möglich
  - wenig Kraft, keine hohen Geschwindigkeiten
  - Einsatz für kleinere Roboter mit Präzisionsarbeiten, beispielsweise zur Leiterplattenbestückung

5.2.4. Sensoren

Funktionen:

- Erfassung der inneren Zustände des Roboters (z.B. Lage, Geschwindigkeit, Kräfte, Momente)
- Erfassen der Zustände der Handhabungsobjekte und der Umgebung
- Messen physikalischer Größen
- Identifikation und Zustandsbestimmung von Werkstücken und Wechselwirkungen
- Analyse von Situationen und Szenen der Umwelt

Man unterscheidet 3 Hauptkategorien:

- **interne Sensoren**: Sie messen Zustandsgrößen des Roboters
  - Position und Orientierung des Roboters selbst
  - Messung des Batteriestands
  - Temperatur im Innern des Roboters
  - Motorstrom
  - Stellung der Gelenke
  - Geschwindigkeit, mit der sich Gelenke bewegen
  - Kräfte und Momente, die auf die Gelenke einwirken

  Beispiele für interne Sensoren sind Positionssensoren, Kraftsensoren, Rad-Encoder, Kreisel, Kompass.

- **externe Sensoren**: Sie erfassen Eigenschaften der Umwelt des Roboters
  - Licht
  - Wärme
  - Schall
  - Kollision mit Hindernissen
  - physikalische Größen im technischen Prozess
  - Entfernungen
  - Lage von Positioniermarken und Objekten
  - Kontur von Objekten
  - Pixelbilder der Umwelt

  Beispiele für externe Sensoren sind Lichtsensoren, Wärmesensoren, Abstandsmesser, Schallsensoren, Kameras, Mikrophone

- **Oberflächensensoren**: z.B. Tastsensoren

Sensordaten sind oft verrauscht (d.h. unpräzise), widersprüchlich oder mehrdeutig. Die Steuerung und Auswertung der Sensordaten kann große Rechner erfordern (z.B. bei der Bildverarbeitung).

5.2.5. Robotersteuerung

Als *Robotersteuerung* bezeichnet man die Hard- und Software eines einzelnen Roboters.

Funktionen:

- Entgegennahme und Abarbeitung von Roboterbefehlen oder -programmen
- Steuerung, Überwachung von Bewegungs- bzw. Handhabungssequenzen und Fahraufträgen
- Synchronisation und Anpassung des Manipulators an den Handhabungsprozess

- Vermeidung bzw. Auflösung von Konfliktsituationen

Abbildung 5.9 zeigt die Struktur und die Schnittstellen einer Robotersteuerung.

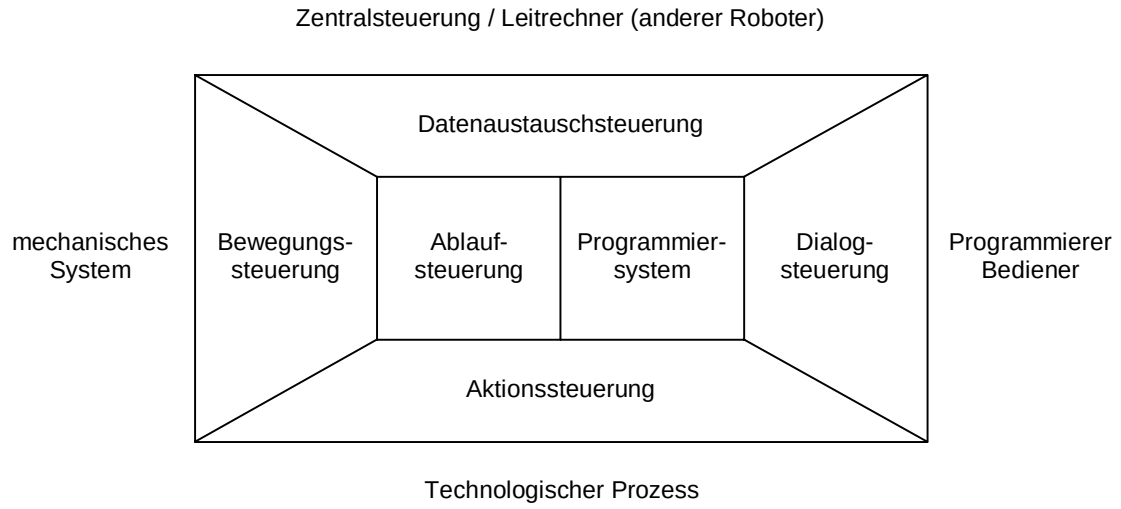


Abbildung 5.9

- **Ablaufsteuerung**: Sie übernimmt die Realisierung des Gesamtablaufs der gestellten Aufgabe entsprechend dem eingegebenen Programm.
- **Bewegungssteuerung**: Steuerung, Koordinierung und Überwachung der Bewegungen des mechanischen Grundgerätes (z.B. Punkt-zu-Punkt-Steuerung, Bahnsteuerung)
- **Aktionssteuerung**: Kommunikation mit der technologischen Umwelt, z.B.:
  - Koordinierung von Aktions- und Bewegungsabläufen
  - Synchronisation mit dem technologischen Prozess
  - Auswertung von Sensordaten
- **Programmiersystem**:
  - Online-Eingabe der Programme
  - Einlesen von Bahnpunkten
  - Interpretation Offline generierten Programmcodes

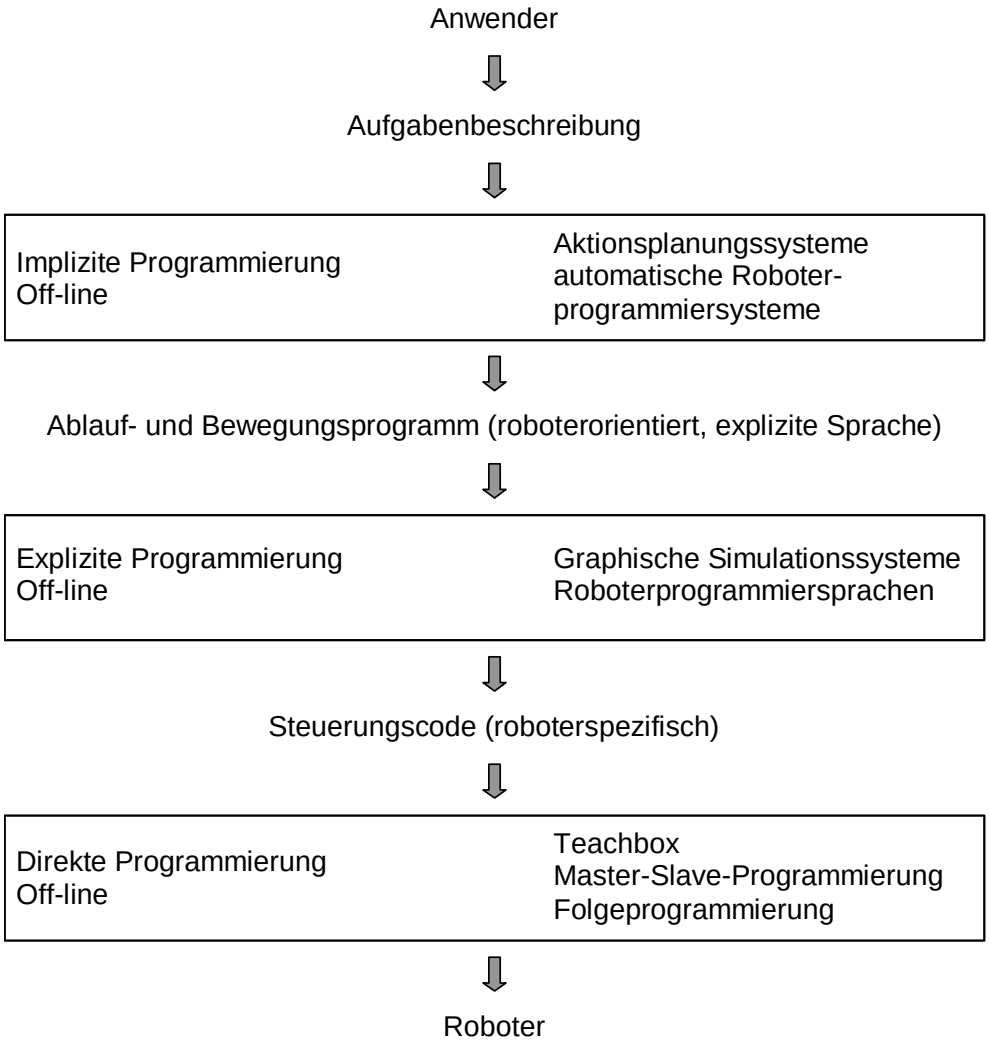
5.2.6. Roboterprogrammierung

Ein Roboter muss frei programmierbar sein. Für die Programmierung eines Roboters (z.B. für die Vorgabe von Bewegungsbahnen), kann man *on-line* und *off-line* Methoden unterscheiden. Bei den on-line Methoden wird der Roboter zur Programmierung benötigt. Bei den off-line Methoden wird das Roboterprogramm ohne Benutzung des Roboters erstellt. Der Roboter wird erst zum Test benötigt.

Funktionen:

- Erstellung der Steuerprogramme mit z.B. Compiler, Interpreter oder Simulator
- Interaktive / automatische Planung der Roboteraufgabe

Stufen der Roboterprogrammierung



- **Teachbox**: Über ein spezielles Eingabegerät (Teachbox) können die Gelenke angesprochen werden oder die Hand (Effektor) relativ zum Roboterbasissystem bewegt werden.
- **Folgeprogrammierung (manuelle Programmierung)**: Die Gelenkmotoren und die Bremsen des Roboters sind so eingestellt, dass der Roboter durch einen Menschen bewegt werden kann. Der Roboterarm wird positioniert und die zugehörigen Gelenkvariablen werden abgespeichert. Der Nachteil dieses Verfahrens ist, dass schwere Roboter nicht so leicht von Hand bewegbar sind. Die manuelle Programmierung kommt heute in der Praxis kaum noch vor.
- **Master-Slave-Programmierung**: Die Master-Slave-Programmierung ist eine Möglichkeit, auch schwerste Roboter manuell zu programmieren. Der menschliche Bediener führt hier einen *Master-Roboter*, der klein ist und leicht bewegt werden kann. Gleichzeitig wird die Bewegung auf den (großen, schweren) *Slave-Roboter* übertragen. Die Bereitstellung von jeweils zwei Robotern mit geeigneter Kopplung kann das Verfahren allerdings sehr teuer machen.

  Es wird daher fast ausschließlich in der Ausprägung *Teleoperation* verwendet. Teleoperation wird immer dort benötigt, wo der Aufenthalt für Menschen schwierig zu bewerkstelligen oder gefährlich ist, wie in verstrahlten Räumen, unter Wasser oder im Weltraum. Das Verfahren arbeitet wie die Master-Slave-Programmierung, jedoch werden normalerweise keine Zwischenpunkte abgespeichert. Da der Master oft sehr weit vom Slave entfernt ist, wird die aktuelle Szene beim Slave zum Bediener am Master per Kamera übertragen. Bei bestimmten Manipu-

lationen ist eine skalierte Rückführung der Kräfte, die auf den Slave einwirken, zum Master notwendig. Dies ist beispielsweise nötig, wenn Gegenstände mit Gefühl gegriffen werden sollen.

Diese Online-Programmierarten erlauben lediglich Bewegungsabläufe zu spezifizieren. Die Integration von Sensoranweisungen und die bedingte Programmablaufkontrolle ist im allgemeinen nicht möglich.

Implizite Programmierung

Aus einer *zielorientierten* Aufgabenbeschreibung sollen die Roboteraktionen und das zugehörige Roboterprogramm *automatisch* erzeugt werden. Die Eingabe beschreibt dabei zu erreichende Teilziele, z.B.: „Füge Stift X in Bohrung 1 von Platte“. Automatische Planungssysteme nutzen Methoden der KI, um das eingegebene Ziel zu erreichen.

Für die Programmierung *autonomer mobiler Roboter* gibt es zwei Ansätze:

Traditioneller Ansatz: Weltmodellierung, Aktionsplanung, Sensordatenfusion. Erfordert sehr leistungsstarke Rechner

Subsumptionsansatz

6. Lernen aus Beobachtungen

6.1. Induktives Lernen

Beim überwachten Lernen bekommt der Lerner eine Menge von Beispielen vorgelegt und soll daraus eine Funktion erlernen. Die **Beispiele** haben die Form von Paaren (x, $f(x)$). x ist die Eingabe und $f(x)$ die Ausgabe der zu lernenden Funktion. Die **reine induktive Inferenz** (kurz **Induktion**) besteht aus der folgenden Aufgabe: Bestimme auf Grund einer Menge von Beispielen von f eine Funktion h, genannt **Hypothese**, die f approximiert.

Im Allgemeinen können viele Hypothesen aus Beispielen generiert werden. Wird einer Hypothese gegenüber einer anderen der Vorzug gegeben, dann heißt dies ein **Bias**. Weil es fast immer eine große Zahl möglicher konsistenter Hypothesen gibt, haben alle Lernalgorithmen einen irgendwie gearteten Bias.

Die einfachste Form eines Lernverfahrens ist das reflexive Lernen. Damit können (*Wahrnehmung*, *Aktion*)-Paare gelernt werden. Die Grundstruktur eines solchen Lernverfahrens besteht aus einem REFLEX-PERFORMANCE-ELEMENT und einem REFLEX-LEARNING-ELEMENT.

global Beispiele ← {}

function REFLEX-PERFORMANCE-ELEMENT(*Wahrnehmung*) **returns** eine *Aktion*

  if (*Wahrnehmung*, *Aktion*) **in** *Beispiele* **then return** *Aktion*
  else $h \leftarrow$ INDUCE(*Beispiele*)
    return h(*Wahrnehmung*)

procedure REFLEX-LEARNING-ELEMENT(*Wahrnehmung*, *Aktion*)
  inputs: *Wahrnehmung* Feedback Wahrnehmug
    Aktion Feedback Aktion

  Beispiele ← *Beispiele* ∪ {(*Wahrnehmung*, *Aktion*)}

6.2. Lernen von Entscheidungsbäumen

6.2.1. Entscheidungsbäume

Ein **Entscheidungsbaum** nimmt als Eingabe ein Objekt oder eine Situation, beschrieben durch eine Menge von Eigenschaften, und erzeugt als Ausgabe eine Ja/Nein-Entscheidung. Entscheidungsbäume repräsentieren also Boolesche Funktionen, es können aber auch höherwertige Funktionen repräsentiert werden. Ein innerer Knoten eines Entscheidungsbaums entspricht einem Test auf den Wert einer der Eigenschaften und die nach unten von dem Knoten wegführenden Kanten sind mit den möglichen Werten der Eigenschaft markiert. Ein Blattknoten entspricht einem Booleschen Wert, der als Ergebnis ausgegeben wird, wenn das Blatt erreicht wird.

Als ein Beispiel wird das Entscheidungsproblem betrachtet, ob man in einem Restaurant auf einen freien Tisch warten soll, wenn alle Tische besetzt sind. Man möchte also eine Definition für das **Zielprädikat** *Warten* in Form eines Entscheidungsbaums lernen. Zunächst müssen die Eigenschaften oder *Attribute* festgelegt werden, mit denen die Beispiele beschrieben werden. Die folgenden Attribute werden gewählt:

1. *Alternative*: Gibt es in der Nähe ein geeignetes anderes Restaurant?
2. *Bar*: Gibt es in dem Restaurant eine komfortable Bar zum Warten?
3. *Fr/Sa*: Ist Freitag oder Samstag?
4. *Hungrig*: Bin ich hungrig?
5. *Gäste*: Wie viele Leute sind im Restaurant (*keine*, *einige* oder *voll*)?
6. *Preis*: In welchem Preissegment liegt das Restaurant (€, €€, €€€)?
7. *Regen*: Regnet es draußen?
8. *Reservierung*: Habe ich reserviert?
9. *Typ*: Um welche Art von Restaurant handelt es sich (*Französisch, Italienisch, Chinesisch* oder *Burger*)?
10. *Wartezeit*: Welche voraussichtliche Wartezeit wird vom Restaurant geschätzt?

Ein Entscheidungsbaum für das Problem könnte z.B. der in Abbildung 6.1 dargestellte sein. Die Attribute *Preis* und *Typ* sind darin nicht benutzt, weil sie angesichts der Datenlage irrelevant sind. Der Baum kann in eine Menge von Implikationen (8 insgesamt) umgesetzt werden, die den Pfaden von der Wurzel zu den mit *Ja* markierten Blättern entsprechen. Zum Beispiel ergibt der Pfad über die Knoten *Wartezeit* und *Hungrig* die Implikation

$$\forall r\ \mathit{Gäste}(r, \mathit{Voll}) \wedge \mathit{Wartezeit}(r, 10\text{-}30) \wedge \mathit{Hungrig}(r, \mathit{Nein}) \Rightarrow \mathit{Warten}(r)$$

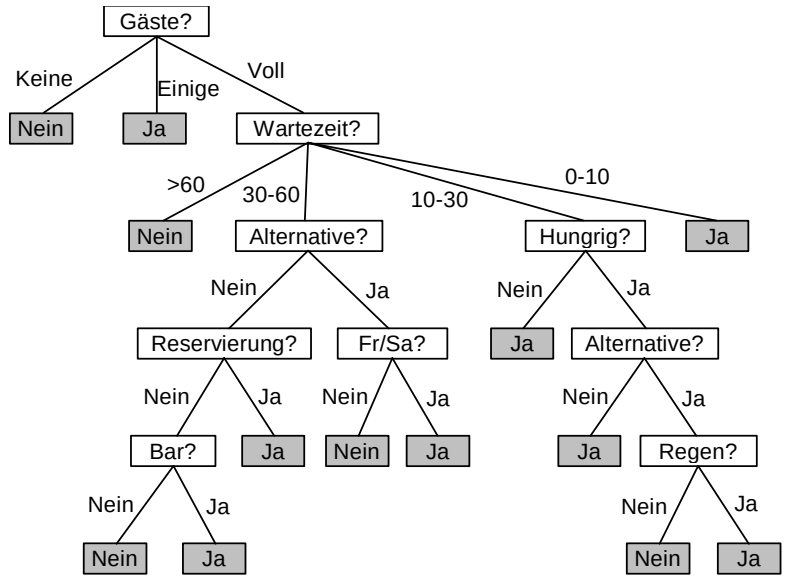


Abbildung 6.1

6.2.2. Ausdruckskraft von Entscheidungsbäumen

Entscheidungsbäume repräsentieren Mengen von Implikationen, sie können aber nicht beliebige Mengen von Sätzen der Logik erster Stufe repräsentieren, da sie prinzipiell nur Aussagen über einzelne Objekte machen können, niemals gleichzeitig über mehrere Objekte. Die Entscheidungsbaum-Sprache ist im Wesentlichen aussagenlogisch.

Die Ausdruckskraft der Entscheidungsbäume entspricht genau der Aussagenlogik, d.h. jede Boolesche Funktion oder jeder aussagenlogische Satz kann als Entscheidungsbaum dargestellt werden. Man braucht nur von der Wahrheitstabelle der Booleschen Funktion auszugehen und jede Zeile als Pfad in einem Entscheidungsbaum darzustellen.

Bei manchen Booleschen Funktionen ist aber die Wiedergabe als Entscheidungsbaum sehr aufwändig. Zur Darstellung der Paritätsfunktion zum Beispiel, die eine 1 ausgibt, wenn geradzahlig viele

Eingaben 1 sind, wäre ein exponentiell großer Entscheidungsbaum notwendig. Auch die Repräsentation der Majoritätsfunktion, die eine 1 ausgibt, wenn mehr als die Hälfte ihrer Eingaben 1 ist, ist schwierig.

6.2.3. Induktion von Entscheidungsbäumen aus Beispielen

Ein **Beispiel** wird durch die Werte der Attribute und den Wert des Zielprädikats beschrieben. Der Wert des Zielprädikats heißt **Klassifikation** des Beispiels. Ist der Wert wahr für ein Beispiel, so heißt dieses ein **positives** Beispiel, andernfalls ein **negatives** Beispiel. Die gesamte Menge der Beispiele heißt **Trainingsmenge**. Tabelle 6.1 enthält die Beispiele für das Restaurant-Beispiel.

Beispiele	Attribute										Ziel
	Alternative	Bar	Fr/Sa	Hungrig	Gäste	Preis	Regen	Reservierung	Typ	Wartezeit	Warten
X_1	Ja	Nein	Nein	Ja	Einige	€€€	Nein	Ja	Franz	0-10	Ja
X_2	Ja	Nein	Nein	Ja	Voll	€	Nein	Nein	Chin	30-60	Nein
X_3	Nein	Ja	Nein	Nein	Einige	€	Nein	Nein	Burger	0-10	Ja
X_4	Ja	Nein	Ja	Ja	Voll	€	Nein	Nein	Chin	10-30	Ja
X_5	Ja	Nein	Ja	Nein	Voll	€€€	Nein	Ja	Franz	>60	Nein
X_6	Nein	Ja	Nein	Ja	Einige	€€	Ja	Ja	Ital	0-10	Ja
X_7	Nein	Ja	Nein	Nein	Keine	€	Ja	Nein	Burger	0-10	Nein
X_8	Nein	Nein	Nein	Ja	Einige	€€	Ja	Ja	Chin	0-10	Ja
X_9	Nein	Ja	Ja	Nein	Voll	€	Ja	Nein	Burger	>60	Nein
X_{10}	Ja	Ja	Ja	Ja	Voll	€€€	Nein	Ja	Ital	10-30	Nein
X_{11}	Nein	Nein	Nein	Nein	Keine	€	Nein	Nein	Chin	0-10	Nein
X_{12}	Ja	Ja	Ja	Ja	Voll	€	Nein	Nein	Burger	30-60	Ja

Tabelle 6.1

Bei der Erstellung eines Entscheidungsbaums aus Beispielen kommt es darauf an, ein Muster zu extrahieren, das eine große Zahl von Fällen in knapper Form beschreibt. Der Entscheidungsbaum muss vor allem eine knappe, aber informationsreiche Repräsentation von Fällen sein und er sollte alle Trainingsbeispiele erfassen. Dies ist ein Beispiel für ein allgemeines Prinzip des induktiven Lernens, genannt **Ockhams Rasiermesser**: *Die wahrscheinlichste Hypothese ist die einfachste, die mit allen Beobachtungen konsistent ist*.

Ein Entscheidungsbaum repräsentiert eine einfache Hypothese, wenn er möglichst klein ist. Das Problem, den kleinsten Entscheidungsbaum zu finden, ist nicht lösbar. Aber man kann versuchen einen möglichst kleinen zu finden. Zu diesem Zweck geht der DECISION-TREE-LEARNING-Algorithmus so vor, dass er immer das wichtigste Attribut zuerst testet. Damit ist dasjenige Attribut gemeint, nach dem sich die Beispiele am stärksten unterscheiden. Auf diese Weise wird versucht eine korrekte Klassifikation mit einer kleinen Menge von Tests zu bekommen, wodurch die Pfade im Entscheidungsbaum kurz werden und der Baum insgesamt klein wird.

function DECISION-TREE-LEARNING(*Beispiele*, *Attribute*, *Default*) **returns** einen *Baum*
  inputs: *Beispiele* eine Menge von Beispielen
  Attribute eine Menge von Attributen
  Default ein Default-Wert für das Zielprädikat

  if *Beispiele* ist leer **then return** *Default*
  else if alle *Beispiele* haben die selbe Klassifikation **then return** die Klassifikation
  else if *Attribute* ist leer **then return** MAJORITY-VALUE(*Beispiele*)

else
best ← CHOOSE-ATTRIBUTE(*Attribute*, *Beispiele*)
Baum ← ein neuer Entscheidungsbaum mit Wurzeltest *best*
for each Wert v_i von *best* **do**
Beispiele$_i$ ← {Elemente von *Beispiele* mit *best* = v_i}
subtree ← DECISION-TREE-LEARNING(*Beispiele*$_i$, *Attribute - best*,
MAJORITY-VALUE(*Beispiele*))
füge einen Zweig zu *Baum* hinzu mit Marke v_i und Teilbaum *subtree*
end
return *Baum*

Nachdem die erste Aufteilung der Testmenge durch das erste Attribut erfolgt ist, entstehen mehrere neue Entscheidungsbaum-Lernprobleme mit weniger Beispielen und einem Attribut weniger. Die Konstruktion des Entscheidungsbaums erfolgt also rekursiv. Vier Fälle sind dabei zu unterscheiden:

1. Wenn die Teilmenge einige positive und einige negative Beispiele enthält, dann wähle das beste Attribut um sie weiter zu unterteilen.
2. Wenn alle Beispiele in der Teilmenge positiv sind (bzw. alle negativ), dann ist der Zweig des Baums fertig. Man kann mit *Ja* (bzw. *Nein*) antworten.
3. Wenn die Teilmenge leer ist bedeutet dies, dass kein Beispiel mit der durch die bisher benutzten Attribute beschriebenen Eigenschaften vorliegt. Es wird ein Defaultwert zurückgegeben, der auf Grund der mehrheitlichen Klassifikation am Vorgängerknoten berechnet wird.
4. Wenn keine Attribute übrig sind aber die Teilmenge nicht leer ist und sowohl positive als auch negative Beispiel enthält, bedeutet dies, dass diese Beispiele zwar verschiedene Klassifikation haben, aber die gleichen Eigenschaften, zumindest so weit sie auf Grund der definierten Attribute beschreibbar sind. Dieser Fall tritt ein, wenn einige Daten falsch sind, d.h. wenn sie **verrauscht** sind, oder wenn die Attribute nicht genügend Information enthalten um die vorliegende Situation zu beschreiben oder wenn die Domäne nicht deterministisch ist. In diesem Fall trift man eine Mehrheitsentscheidung.

Konstruiert man den Entscheidungsbaum für die Beispiele von Tabelle 6.1 nach dieser Vorschrift, dann erhält man den Entscheidungsbaum von Abbildung 6.2.

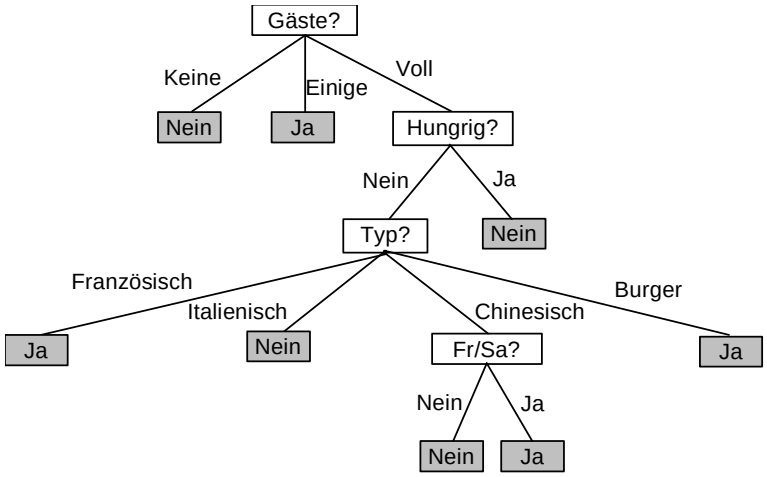


Abbildung 6.2

Die Diskrepanz zwischen dem Ausgangsbaum und dem Ergebnisbaum ist sicher auch der kleinen Anzahl von Beispielen zuzuschreiben. Mit mehr Beispielen dürfte der Unterschied geringer ausfal-

len. Der Baum von Abbildung 6.2 ist auch keine sehr gute Hypothese, denn z.B. kommt der Fall, dass ein Restaurant voll und die Wartezeit 0 – 10 Minuten beträgt, gar nicht vor. Im Fall von Hunger sagt der Baum, dass nicht gewartet werden soll, aber wenn die Wartezeit höchstens 10 Minuten beträgt, wird man in der Regel warten. Diese Beobachtung führt zu der Frage nach der Korrektheit der Hypothese.

6.2.4. Abschätzung der Performanz des Lernalgorithmus

Die Qualität eines Lernalgorithmus kann danach eingeschätzt werden, wie gut er neue, vorher nicht betrachtete Beispiele klassifiziert. Dazu muss die Klassifikation dieser Beispiele aber vorab bekannt sein. Die Menge der Beispiele, die für diesen Zweck verwendet werden, heißt **Testmenge**. Diese Menge reserviert man üblicherweise von vornherein für den Test, ansonsten müsste man nach neuen Beispielen suchen und deren Klassifikation bestimmen. Im Einzelnen geht man in folgenden Schritten vor:

1. Sammle eine große Menge von Beispielen.
2. Unterteile die Menge in zwei disjunkte Teilmengen, die Trainingsmenge und die Testmenge.
3. Erzeuge mit dem Lernalgorithmus auf der Basis der Trainingsmenge eine Hypothese *H*.
4. Bestimme den Prozentsatz der Beispiele in der Testmenge, die durch *H* korrekt klassifiziert werden.
5. Wiederhole die Schritte 1. bis 4. für verschieden große und zufällig ausgewählte Trainingsmengen.

Durch Anwendung dieser Schritte erhält man eine Menge von Daten, auf Grund derer die durchschnittliche Vorhersagequalität als eine Funktion der Größe der Trainingsmenge angegeben werden kann. Diese Funktion ergibt in grafischer Form dargestellt die so genannte **Lernkurve** des Algorithmus auf der betrachteten Domäne. Abbildung 6.3 zeigt den ungefähren Verlauf der Lernkurve für den Algorithmus DECISION-TREE-LEARNING auf der Restaurant-Domäne.

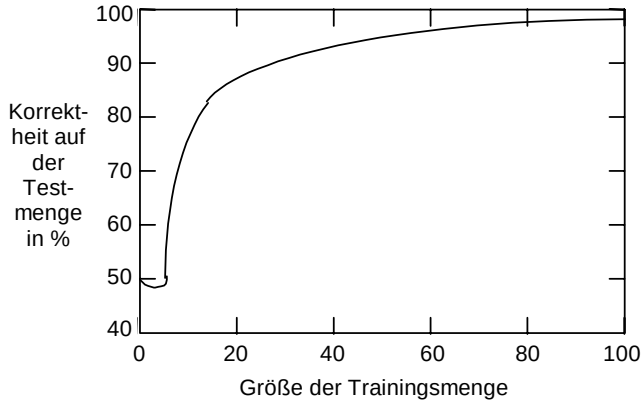


Abbildung 6.3

6.2.5. Praktischer Gebrauch des Entscheidungsbaum-Lernens

Gas-Öl-Trenner

BP brachte 1986 ein Expertensystem mit Namen GASOIL zum Einsatz. Sein Zweck war der Entwurf von Gas-Öl-Trennanlagen für Offshore-Ölplattformen. Die Trennung von Gas und Öl erfordert ein sehr großes, komplexes und teures Trennsystem, dessen Entwurf eine Anzahl von Attributen betrifft, u.a. die relativen Anteile von Gas, Öl und Wasser, die Flussrate, den Druck, die Dichte, die Viskosität und die Temperatur. GASOIL enthielt ungefähr 2500 Regeln. Der Bau eines solchen

Expertensystems von Hand erfordert ungefähr 10 Personen-Jahre. Statt dessen wurde ein Entscheidungsbaum-Lernverfahren auf eine Datenbasis von existierenden Entwürfen angewandt. Damit wurde das Expertensystem in 100 Personen-Tagen entwickelt. Es soll besser sein als menschliche Experten und viele Millionen Dollar eingespart haben.

Fliegen lernen im Flugsimulator

Sammut et al. entwickelten 1992 einen automatischen Piloten für eine Cessna. Die Beispieldaten wurden durch Beobachtung von drei erfahrenen Piloten im Flugsimulator gewonnen. Jeder der Piloten führte 30 Minuten lang einen vordefinierten Flugplan aus. Jedes Mal wenn ein Pilot eine Steueraktion durchführte indem er eine Steuervariable setzte, z.B. Schub oder Flügelklappen, wurde ein Beispiel erzeugt. Insgesamt wurden 90000 Beispiele erzeugt, jedes durch 20 Zustandsvariable beschrieben und mit der durchgeführten Aktion markiert. Aus den Beispielen wurde mit Hilfe des C4.5-Algorithmus ein Entscheidungsbaum erzeugt. Der Baum wurde in ein C-Programm umgesetzt und dieses in die Steuerschleife des Flugsimulators integriert, so dass das Programm das Flugzeug selbst fliegen konnte.

Die Ergebnisse waren überraschend. Das Programm konnte nicht nur fliegen, sondern sogar besser fliegen als die menschlichen Lehrer. Der Grund dafür ist, dass beim Lernen gelegentliche Fehler der einzelnen Menschen durch Verallgemeinerung ausgeglichen wurden. Das deutet an, dass für schwierige Aufgaben wie das Fliegen von Hubschraubern, die schwere Lasten bei starkem Wind tragen, mit den Methoden des Maschinellen Lernens Autopiloten entwickelt werden könnten. Bis jetzt gibt es solche nicht und es gibt auch nur wenige menschliche Piloten dafür.

6.3. Die Verwendung der Informationstheorie

6.3.1. Das Prinzip des Informationsgewinns

Zur Implementierung der CHOOSE-ATTRIBUTE-Funktion benötigt man ein Maß für gute und unnütze Attribute. Dieses Maß muss seinen größten Wert für perfekte Attribute und seinen kleinsten Wert für völlig wertlose Attribute haben. Ein solches Maß ist der erwartete Gehalt an **Information**, der von einem Attribut geliefert wird. Der Begriff *Information* ist dabei im Sinne der Shannonschen Informationstheorie zu verstehen. Eine Information in diesem Sinn ist eine Antwort auf eine Frage. Wenn man eine auf die Frage schon relativ gut zutreffende Vermutung hat, dann ist die Antwort weniger informativ als wenn man überhaupt nichts weiß.

In der Informationstheorie wird der Gehalt einer Information in **Bits** gemessen. Ein Bit Information genügt um eine Ja/Nein-Frage zu beantworten, über deren Antwort man keinerlei Vorstellung hat, wie dem Fall einer Münze. Wenn die möglichen Antworten v_i auf die Frage die Wahrscheinlichkeiten $P(v_i)$ haben, dann ist der Informationsgehalt I der tatsächlichen Antwort definiert durch

$$I(P(v_i),...,P(v_n)) = \sum_{i=1}^{n} - P(v_i) \log_2 P(v_i)$$

Dies ist der durchschnittliche Informationsgehalt der verschiedenen Antworten (die $-\log_2 P$-Ausdrücke), gewichtet mit den Wahrscheinlichkeiten der Antworten.

Beim Entscheidungsbaum-Lernen ist die zu beantwortende Frage: Was ist die korrekte Klassifikation für ein gegebenes Beispiel? Ein korrekter Entscheidungsbaum liefert die Antwort. Eine Schätzung der Wahrscheinlichkeiten der möglichen Antworten vor dem Test der Attribute ist durch das Verhältnis der positiven und negativen Beispiele in der Trainingsmenge gegeben. Angenommen,

die Trainingsmenge enthalte *p* positive und *n* negative Beispiele. Dann wird der Informationsgehalt einer korrekten Antwort wie folgt abgeschätzt:

$$I\left(\frac{p}{p+n},\frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

Der Test eines einzelnen Attributs gibt im Allgemeinen nicht so viel Information, aber immerhin einen Teil davon. Das Maß an Information, das ein Attribut liefert, wird daran gemessen, wie viel Information *nach* dem Attributtest für eine korrekte Antwort noch benötigt wird. Jedes Attribut *A* unterteilt die Trainingsmenge *E* in die Teilmengen E_1, ..., E_v, entsprechend ihren Werten für *A*, wobei angenommen ist, dass *A v* verschiedene Werte haben kann. Jede Teilmenge E_i enthält p_i positive und n_i negative Beispiele. Wenn man also auf dem *i*-ten Zweig des Baums fortfährt, benötigt man zusätzlich $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ Bits an Information um die Frage zu beantworten. Ein zufällig ausgewähltes Beispiel hat mit der Wahrscheinlichkeit $(p_i+n_i)/(p+n)$ den *i*-ten Wert des Attributs, deshalb benötigt man im Durchschnitt nach dem Test des Attributs *A*

$$Rest(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n}\cdot I\left(\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}\right)$$

Bits an Information um das Beispiel zu klassifizieren. Der **Informationsgewinn** aus dem Attributtest ist definiert durch die Differenz zwischen dem ursprünglichen Informationsbedarf und dem neuen Informationsbedarf:

$$Gain(A) = I\left(\frac{p}{p+n},\frac{n}{p+n}\right) - Rest(A)$$

Die Heuristik, die in der CHOOSE-ATTRIBUTE-Funktion benutzt wird, ist, das Attribut mit dem größten Informationsgewinn zu wählen.

6.3.2. Rauschen und Overfitting

Wenn es bei einem Lernproblem eine große Menge möglicher Hypothesen gibt, besteht die Gefahr, dass man beim Lernen bedeutungslose Regelmäßigkeiten in den Daten entdeckt. Dieses Problem heißt **Overfitting**. Es ist ein sehr allgemeines Phänomen und kann in allen möglichen Lernverfahren vorkommen, nicht nur bei Entscheidungsbäumen.

Eine einfache Technik zur Vermeidung des Overfitting ist das Pruning. Durch das Pruning wird verhindert, dass Attribute für die Aufteilung einer Menge verwendet werden, die nicht wirklich relevant sind, selbst wenn die Beispiele an diesem Knoten nicht uniform klassifiziert sind. Die Frage ist, wie irrelevante Attribute entdeckt werden. Der Informationsgewinn bietet dafür eine Möglichkeit. Angenommen man teilt eine Menge von Beispielen mit einem irrelevanten Attribut auf. Die entstehenden Teilmengen haben dann in der Regel etwa dieselbe Verteilung von positiven und negativen Beispielen wie die ursprüngliche Menge. Dann ist der Informationsgewinn annähernd Null.

6.3.3. Erweiterung der Anwendbarkeit von Entscheidungsbäumen

Um den Einsatzbereich des Entscheidungsbaum-Lernens zu erweitern muss man Maßnahmen ergreifen um fehlende Daten zu ergänzen und um Attribute mit besonderen Eigenschaften verwendbar zu machen.

- **Fehlende Daten** In vielen Anwendungsbereichen sind nicht alle Attributwerte für jedes Beispiel bekannt. Das kann daran liegen, dass sie nicht erfasst wurden oder dass es zu aufwändig ist sie zu ermitteln. Hier sind zwei Probleme zu lösen: 1. Wie modifiziert man die Konstruktion des Entscheidungsbaums, insbesondere die Definition des Informationsgewinns, wenn bei einigen Beispielen Attributwerte fehlen? 2. Wie klassifiziert man ein neues Beispiel mit einem bestehenden Entscheidungsbaum, wenn eines der Testattribute nicht anwendbar ist?
- **Attribute mit vielen Werten** Ist die Zahl der Werte eines Attributs sehr hoch, dann kann der Extremfall eintreten, dass das Attribut für jedes Beispiel einen eigenen Wert hat und damit lauter einelementige Mengen entstehen. Der Informationsgewinn hätte dann für dieses Attribut den höchsten Wert. Trotzdem wäre das Attribut irrelevant oder nutzlos. Um solche Attribute zu behandeln kann man den **Gain Ratio** verwenden.
- **Attribute mit kontinuierlichen Werten** Manche Attribute haben kontinuierliche Wertebereiche, z.B. Attribute wie *Größe* oder *Gewicht*. Um sie für das Entscheidungsbaum-Lernen verwendbar zu machen diskretisiert man üblicherweise die Wertebereiche. Das wird meistens von Hand gemacht. Eine bessere Methode ist, die Attribute im Rahmen des Lernprozesses vorab zu behandeln um herauszufinden, welche Unterteilung des Wertebereichs die nützlichste Information für die Zwecke der Klassifikation liefert.

7. Lernen in Neuronalen Netzen

7.1. Gehirn und Computer

7.1.1. Struktur und Arbeitsweise des Gehirns

Das **Neuron**, auch Nervenzelle genannt, ist die elementare funktionale Einheit von allem Nervengewebe, einschließlich des Gehirns. Es besteht aus einem Zellkörper, genannt **Soma**, der den Zellkern enthält. Aus dem Zellkörper verzweigen eine Anzahl kurzer Fasern, genannt **Dendriten**, und eine einzelne lange Faser, genannt **Axon**. Die Dendriten verzweigen in ein buschiges Netz von Endigungen um die Zelle herum, während das Axon sich über eine lange Distanz erstreckt, meist etwa 1 cm (damit das 100-fache des Durchmessers des Zellkörpers) in extremen Fällen bis zu 1 m. Das Axon verzweigt sich am Ende ebenfalls in einzelne Fäden, die mit Dendriten oder Zellkörpern anderer Neuronen verbunden sind. Die Verbindungsstellen zwischen Neuronen heißen **Synapsen**. Jedes Neuron bildet Synapsen mit irgendwelchen anderen Neuronen, die Anzahl ist sehr unterschiedlich, sie geht von einem Dutzend bis 100000. Abbildung 7.1 zeigt eine vereinfachte Darstellung eines Neurons.

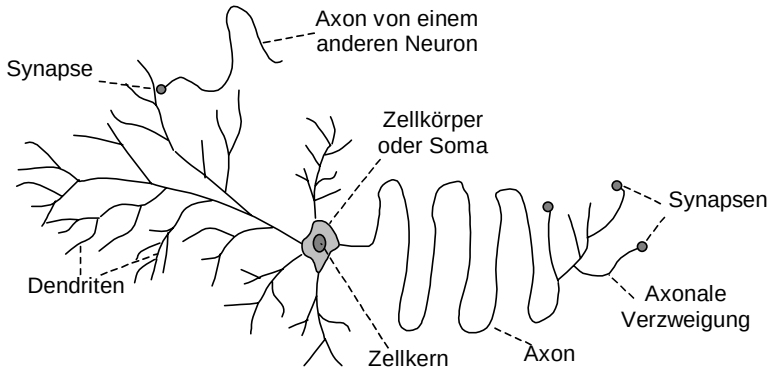


Abbildung 7.1

Die Übertragung von Signalen von einem Neuron auf ein anderes erfolgt durch einen komplizierten elektrochemischen Prozess. Wenn ein elektrischer Impuls am Axonende ankommt, erzeugt die Synapse eine Transmittersubstanz, die in den Dendrit einer anderen Zelle eindringt und dort das elektrische Potential erhöht oder erniedrigt (exzitatorische bzw. inhibitorische Synapsen). Wenn das Potential einen Schwellenwert erreicht, wird ein Impuls, genannt das **Aktionspotenzial**, durch das Axon geschickt. Dieser löst dann wieder die Erzeugung von Transmittersubstanz in den entsprechenden Synapsen aus. Synapsen können unterschiedlich stark sein, d.h. unterschiedliche Leitfähigkeit haben, und diese kann sich ändern. Ebenso können Neuronen neue Verbindungen zu anderen Neuronen aufbauen und ganze Neuronengruppen können ihren Platz verändern. Durch diese Möglichkeiten der Modifikation ist das Gehirn lernfähig.

7.1.2. Vergleich zwischen Gehirn und Computer

Vergleicht man Speicherkapazität und Verarbeitungskapazität von heutigen Personal Computern mit denen des Gehirns, so kommt man etwa zu den Zahlen von Tabelle 7.1.

	Computer	Menschliches Gehirn
Recheneinheiten	1 CPU, 10^6 Transistoren	10^{11} Neuronen
Speichereinheiten	10^{10} Bits RAM, 10^{11} Bits Platte	10^{11} Neuronen, 10^{14} Synapsen
Zykluszeit	10^{-8} sec	10^{-3} sec
Bandbreite	10^{10} Bits/sec	10^{14} Bits/sec
Geschaltete Elemente pro sec	10^5	10^{14}

Tabelle 7.1

7.2. Neuronale Netze

7.2.1. Grundbegriffe

Ein Neuronales Netz besteht aus einer Menge von Knoten oder **Einheiten**, die durch **Kanten** verbunden sind. Jeder Kante ist ein numerisches **Gewicht** zugeordnet. Die Gewichte sind veränderlich und durch ihre Veränderung kann das Netz lernen, während die Struktur im Allgemeinen als fest betrachtet wird. Durch das Lernen bringt das Netz seine Ausgaben, die es auf Grund bestimmter Eingaben erzeugt, mit den gewünschten Ausgaben in Übereinstimmung. Einige der Einheiten sind als Eingabe- bzw. Ausgabeeinheiten ausgezeichnet.

Jede Einheit hat eine Anzahl von Eingabekanten, die von anderen Einheiten kommen, und von Ausgabekanten, die zu anderen Einheiten führen. Sie hat weiterhin ein bestimmtes Aktivierungsniveau und ein Hilfsmittel zum Berechnen des Aktivierungsniveaus im nächsten Schritt aus den Eingaben und Gewichten. Im Prinzip führt jede Einheit ihre Berechnung lokal auf Grund der Eingaben von den Nachbareinheiten aus, eine globale Kontrolle ist nicht erforderlich. In der praktischen Realisierung, meist als Software, werden die Berechnungen aber synchronisiert, damit die Einheiten in einer festen Reihenfolge arbeiten.

7.2.2. Notationen

Für die Bezeichnung der Bestandteile eines Neuronalen Netzes sind einige formale Notationen erforderlich. Sie sind in Tabelle 7.2 zusammengestellt.

Notation	Bedeutung
a_i	Aktivierungswert der Einheit i (gleichzeitig die Ausgabe der Einheit)
$\mathbf{a}_i$	Vektor der Aktivierungswerte der Eingaben in Einheit i
g	Aktivierungsfunktion
g'	Ableitung der Aktivierungsfunktion
Err_i	Fehler der Einheit i (Differenz zwischen Ausgabe und Ziel)
Err^e	Fehler für Beispiel e
I_i	Aktivierung der Einheit i in der Eingabeschicht
$\mathbf{I}$	Vektor der Aktivierungen aller Eingabeeinheiten
$\mathbf{I}^e$	Vektor der Eingaben für Beispiel e
in_i	Gewichtete Summe der Eingaben in Einheit i
N	Gesamtzahl der Einheiten im Netz
O	Aktivierung der einzigen Ausgabeeinheit eines Perzeptrons
O_i	Aktivierung der Einheit i in der Ausgabeschicht
$\mathbf{O}$	Vektor der Aktivierungen aller Einheiten der Ausgabeschicht
t	Schwellenwert für eine Schrittfunktion
T	Zielausgabe für ein Perzeptron (gewünschte Ausgabe)
$\mathbf{T}$	Zielvektor bei mehreren Ausgabeeinheiten

$\mathbf{T}^e$	Zielvektor für Beispiel *e*
$W_{j,i}$	Gewicht der Kante von Einheit *j* zu Einheit *i*
W_i	Gewicht der Kante von Einheit *i* zur Ausgabeeinheit in einem Perzeptron
$\mathbf{W}_i$	Vektor der Gewichte an den zu Einheit *i* führenden Kanten
$\mathbf{W}$	Vektor aller Gewichte im Netz

7.2.3. Einfache Rechenelemente

In Abbildung 7.2 ist eine typische Einheit dargestellt. Ihre Grundfunktion ist es aus den über die Eingabekanten empfangenen Signalen ein neues Aktivierungsniveau zu berechnen und dieses über die Ausgabekanten an die nächsten Einheiten zu schicken. Die Berechnung erfolgt in zwei Schritten: Im ersten Schritt wird durch eine *lineare* Komponente, die **Eingabefunktion** in_i, die gewichtete Summe der Eingabewerte berechnet. Im zweiten Schritt wird durch eine nichtlineare Komponente, die **Aktivierungsfunktion** *g*, das Aktivierungsniveau a_i berechnet. Gewöhnlich haben alle Einheiten in einem Netz die selbe Aktivierungsfunktion.

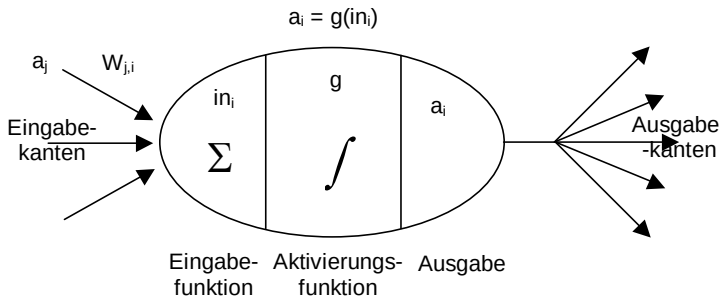


Abbildung 7.2

in_i wird berechnet durch

$$in_i = \sum_j W_{j,i} a_j = \mathbf{W}_i \cdot \mathbf{a}_i$$

Der neue Wert des Aktivierungsniveaus wird berechnet durch die Funktion *g*.

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$

Für die Wahl der Funktion *g* gibt es eine Fülle von Möglichkeiten. Drei besonders gebräuchliche Funktionen sind die Schrittfunktion, die Signumsfunktion und die Sigmoidfunktion. Sie sind in Abbildung 7.3 illustriert.

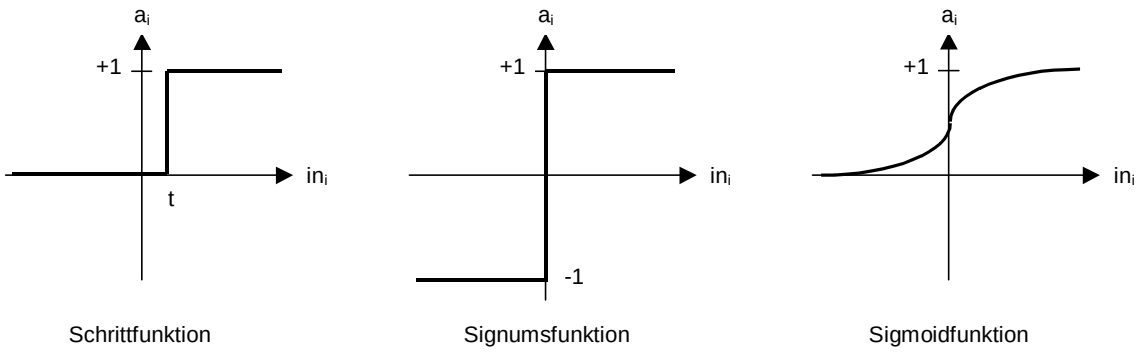


Abbildung 7.3

Die Definition dieser Funktionen ist wie folgt:

$$step_t(x) = \begin{cases} 1, & falls\ x \geq t \\ 0, & falls\ x < t \end{cases} \qquad sign(x) = \begin{cases} +1, & falls\ x \geq 0 \\ -1, & falls\ x < 0 \end{cases} \qquad sigmoid(x) = \frac{1}{1+e^{-x}}$$

Statt Aktivierungsfunktionen mit Schwellenwert zu verwenden, kann man solche ohne Schwellenwert verwenden und statt dessen noch eine zusätzliche Eingabe mit Gewicht vorsehen. Dadurch wird das Lernen im Netz einfacher, denn man muss nur noch die Gewichte verändern, nicht Gewichte *und* Schwellenwerte. Die zusätzliche Eingabe hat die konstante Aktivierung $a_0 = -1$. Das zusätzliche Gewicht $W_{0,i}$ dient zusammen mit der Aktivierung a_0 als Schwellenwert wenn $W_{0,i} \cdot a_0 = -t$. Dann kann für alle Einheiten der feste Schwellenwert 0 verwendet werden. Die zusätzliche Eingabe kann im Netz durch eine spezielle Einheit, genannt **Bias**, die eine Kante zu jeder anderen Einheit, aber keine Eingabekante besitzt, realisiert werden.

7.2.4. Netzstrukturen

Es gibt viele verschiedene Netzstrukturen mit unterschiedlichen Berechnungsleistungen. Sie lassen sich in zwei Hauptgruppen einteilen, die **zyklenfreien** (feed forward) und die **rekurrenten** Netze. In zyklenfreien Netzen sind die Kanten gerichtet und es gibt keine Schleifen, während in rekurrenten Netzen beliebige Strukturen erlaubt sind. Graphentheoretisch ist ein zyklenfreies Netz ein DAG. Üblicherweise wird die Anordnung der Einheiten im Netz in Schichten vorgenommen. In einem geschichteten zyklenfreien Netz führen von jeder Einheit aus nur Kanten zu Einheiten in der nächsten Schicht, es gibt keine Kanten zu Einheiten in der selben Schicht oder zu Einheiten in vorangehenden Schichten oder Kanten, die eine Schicht überspringen (Ausnahme: das Bias). In Abbildung 7.4 ist ein zweischichtiges zyklenfreies Netz dargestellt. Die Eingabeeinheiten werden nicht als eigene Schicht gezählt.

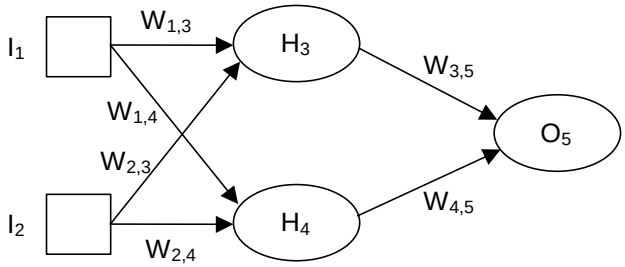


Abbildung 7.4

Zyklenfreie Netze realisieren Funktionen der Eingaben, die nur von den Gewichten abhängen. Sie haben keine inneren Zustände, da bei der Berechnung der Aktivierungswerte einer Schicht die Aktivierungswerte der vorhergehenden Schichten keine Rolle spielen. Die Netze können dazu verwendet werden, adaptive Versionen von Reflexagenten zu realisieren oder einzelne Komponenten komplexerer Agenten.

In rekurrenten Netzen wird die Aktivierung als Eingabe der selben oder vorhergehender Einheiten verwendet. Deshalb haben diese Netze einen inneren Zustand, der in den Aktivierungsniveaus der Einheiten gespeichert ist. Die Berechnung in rekurrenten Netzen verläuft weniger geordnet als in zyklenfreien. Die Netze können instabil werden, oszillieren oder chaotisches Verhalten zeigen. Es kann lange dauern, bis sie auf eine Eingabe eine stabile Ausgabe erzeugen und der Lernprozess ist schwieriger. Ihre Repräsentationsfähigkeit ist aber dafür größer als die zyklenfreier Netze. Sie können komplexe Agenten und Systeme mit inneren Zuständen darstellen.

In einem zyklenfreien Netz gibt es eine Schicht von Einheiten, die Eingaben aus der Umgebung aufnehmen, die **Eingabeeinheiten** (in Abbildung 7.4 I_1 und I_2). Ihr Aktivierungswert ist durch die Umgebung bestimmt. Ferner gibt es eine Schicht von Einheiten, die ihre Ausgaben in die Umgebung abgeben, die **Ausgabeeinheiten** (in Abbildung 7.4 O_5). Dazwischen gibt es Schichten von Einheiten, die keine Verbindung zur Außenwelt haben, sie heißen deshalb **verborgene Einheiten** (hidden units) (in Abbildung 7.4 gibt es eine Schicht verborgener Einheiten, H_3 und H_4). Ein bestimmter Typ zyklenfreier Netze, die **Perzeptrone**, haben keine verborgenen Einheiten. Bei ihnen ist der Lernprozess einfach, dafür ist ihre Repräsentationsfähigkeit stark beschränkt. Netze mit einer oder mehreren Schichten verborgener Einheiten heißen **mehrschichtige Netze**. Schon mit einer genügend großen Schicht verborgener Einheiten ist es möglich, jede kontinuierliche Funktion der Eingaben zu repräsentieren, mit zwei Schichten können sogar diskontinuierliche Funktionen repräsentiert werden.

Bei fester Struktur und festen Aktivierungsfunktionen g haben die Funktionen, die durch ein zyklenfreies Netz gelernt (und damit repräsentiert) werden können, eine spezifische parametrisierte Form. Die Gewichte bestimmen, welche Funktionen tatsächlich repräsentiert werden. Stellt man sich die Gewichte als Parameter oder Koeffizienten der repräsentierten Funktion vor, dann ist das Lernen der Prozess, bei dem die Parameter getunt werden, um die Daten der Trainingsmenge möglichst genau zu treffen. In der Statistik heißt dieser Prozess **nichtlineare Regression**.

7.2.5. Optimale Netzstrukturen

Ein generelles Problem bei der Konstruktion Neuronaler Netze ist die Festlegung der Struktur. Wird das Netz zu klein angelegt, dann kann es die gewünschte Funktion nicht repräsentieren. Wird es zu groß angelegt, dann kann es alle Beispiele der Trainingsmenge in der Art einer Tabelle speichern, aber es hat keine Verallgemeinerungsfähigkeit und kann neue Beispiele nicht richtig klassifizieren. Dieses Phänomen ist das Overfitting, es ist typisch für statistische Modelle.

Zyklenfreie Netze mit einer verborgenen Schicht können jede kontinuierliche Funktion approximieren, Netze mit zwei verborgenen Schichten können überhaupt jede Funktion approximieren. Aber die Zahl der Einheiten in jeder Schicht kann exponentiell mit der Zahl der Eingaben anwachsen. Es gibt noch keine gute Theorie um die mit einer kleinen Zahl von Einheiten approximierbaren Funktionen zu kennzeichnen.

Das Problem eine gute Netzstruktur zu finden kann als ein Suchproblem betrachtet werden. Um dieses zu lösen wurden verschiedene Verfahren versucht, z.B. genetische Algorithmen. Bei ihnen ist aber der Rechenaufwand sehr hoch, weil man Populationen von Netzen untersuchen und mit Fitnessfunktionen testen muss. Ein anderes mögliches Verfahren ist Bergsteigen. Dabei wird das Netz schrittweise modifiziert, und zwar entweder von einem kleinen Netz ausgehend vergrößernd, oder von einem großen Netz ausgehend verkleinernd.

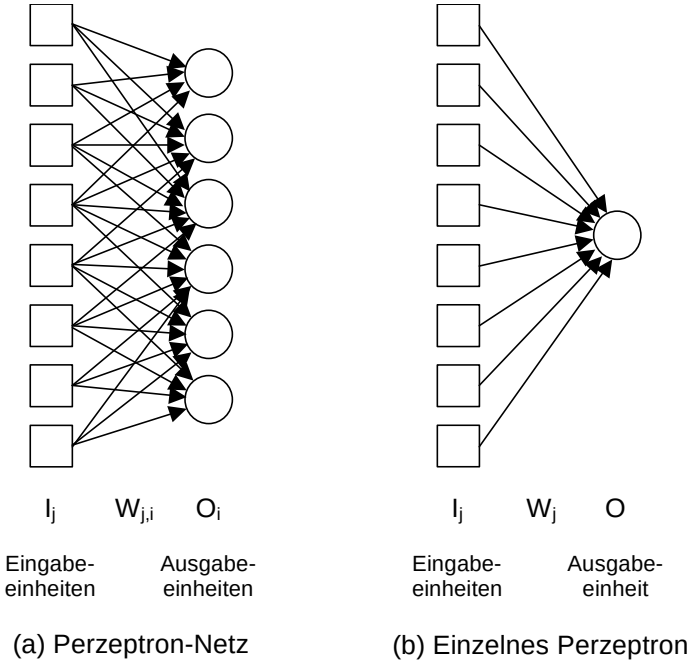
7.3. Perzeptrone

7.3.1. Struktur von Perzeptronen

Ein Perzeptron ist ein einschichtiges, zyklenfreies Netz. Abbildung 7.5(a) zeigt ein Perzeptron. Die Ausgabeeinheiten sind unabhängig voneinander, denn jedes Gewicht beeinflusst nur eine Ausgabe. Deshalb kann man ein Perzeptron als zusammengesetzt aus lauter einfachen Perzeptronen mit jeweils nur einer Ausgabeeinheit betrachten, vgl. Abbildung 7.5(b), und man kann die Untersuchung auf solche einfachen Perzeptrone beschränken. Die Indizes lassen sich hier vereinfachen, die

Ausgabeeinheit wird einfach mit *O* bezeichnet und das Gewicht von der Eingabe *j* zu *O* mit W_j. Die Aktivierung der Eingabeeinheit *j* ist durch I_j gegeben. Damit ist die Aktivierung der Ausgabe

$$O = step_0\left(\sum_j W_j I_j\right) = step_0(\mathbf{W}\cdot\mathbf{I}) \tag{7.1}$$



(a) Perzeptron-Netz (b) Einzelnes Perzeptron

Abbildung 7.5

7.3.2. Repräsentationsfähigkeit von Perzeptronen

Es können einige komplexe Boolesche Funktionen repräsentiert werden, z.B. die **Majoritätsfunktion**, die eine 1 ausgibt, wenn mehr als die Hälfte ihrer *n* Eingaben 1 ist. Dafür benötigt man ein Perzeptron, bei dem alle Gewichte $W_j = 1$ sind und der Schwellenwert $t = n/2$. Bei einem Entscheidungsbaum würde man für diese Funktion $O(2^n)$ Knoten benötigen.

Leider können viele Boolesche Funktionen nicht repräsentiert werden. Der Grund dafür ist, dass jede Eingabe I_j die Ausgabe nur in einer Richtung beeinflussen kann, unabhängig von den übrigen Eingabewerten. Man betrachte zur Illustration einen Eingabevektor **a**. Jede Komponente des Vektors sei $a_j = 0$ und die Ausgabe sei 0. Angenommen, beim Wechsel von a_j zum Wert $a_j = 1$ werde die Ausgabe 1. Dann muss W_j positiv sein. Ferner gibt es dann keinen Vektor **b**, für den die Ausgabe 1 ist wenn $b_j = 0$, aber andererseits die Ausgabe 0, wenn $b_j = 1$. Diese Eigenschaft schränkt die Repräsentationsfähigkeit der Perzeptrone erheblich ein.

7.3.3. Erlernen linear separierbarer Funktionen

Der Lernalgorithmus Neural-Network-Learning geht von zufällig aus dem Intervall [-0.5, 0.5] ausgewählten Gewichten in einem Perzeptron aus. Dann werden die Gewichte schrittweise verändert, und zwar so, dass die Ausgabe bei den vorgelegten Beispielen an die gewünschte Ausgabe angepasst wird. Dieser Anpassungsschritt muss für jedes Beispiel mehrere Male wiederholt werden um Konvergenz zu erzielen. Der Lernprozess wird dazu meist in **Epochen** unterteilt. In jeder Epoche wird für jedes Beispiel eine Anpassung durchgeführt. Der Algorithmus ist wie folgt definiert.

function NEURAL-NETWORK-LEARNING(*Beispiele*) **returns** ein *Netz*

Netz ← ein Neuronales Netz mit zufällig ausgewählten Gewichten
repeat
  for each *e* **in** *Beispiele* **do**
    O ← NEURAL-NETWORK-OUTPUT(*Netz*, *e*)
    T ← die erwarteten Ausgabewerte für *e*
    aktualisiere die Gewichte in Netz auf der Basis von *e*, **O** und **T**
  end
until alle Beispiele werden korrekt vorhergesagt oder das Stopp-Kriterium ist erreicht
return *Netz*

Dieser Algorithmus lässt sich auf beliebige zyklenfreie Netze anwenden. Es ist nur jeweils die Regel für das Aktualisieren der Gewichte geeignet zu definieren.

Die Regel für das Aktualisieren der Gewichte (*Lernregel*) für das Perzeptron ist einfach. Die vorhergesagte Ausgabe für ein Beispiel sei *O*, die korrekte Ausgabe sei *T*. Dann ist der Fehler

$$Err = T - O$$

Ist der Fehler positiv, dann muss der Wert von *O* vergrößert werden, ist er negativ, muss er verkleinert werden. Jede Eingabeeinheit hat den Anteil $W_j \cdot I_j$ an der Gesamteingabe. Ist I_j positiv, dann führt eine Vergrößerung von W_j tendenziell zu einer größeren Ausgabe, ist I_j negativ, dann führt eine Vergrößerung von W_j tendenziell zu einer kleineren Ausgabe. Fasst man beide Einflüsse zusammen, dann erhält man die folgende Regel:

$$W_j \leftarrow W_j + \alpha \cdot I_j \cdot Err$$

α ist eine Konstante, genannt die **Lernrate**. Die Regel ist eine leicht geänderte Version der Perzeptron-Lernregel von Rosenblatt (1960). Rosenblatt bewies, dass die Lernregel gegen eine Menge von Gewichten konvergiert, die die Beispielmenge korrekt repräsentiert, sofern die Beispiele eine linear separierbare Funktion repräsentieren.

7.4. Mehrschichtige zyklenfreie Netze

7.4.1. Back-Propagation-Lernen

Mit der Methode des Back Propagation können auch mehrschichtige zyklenfreie Netze lernen. Prinzipiell verläuft der Lernprozess wie bei den Perzeptronen: Dem Netz werden Beispiele vorgelegt. Stimmt der Ausgabevektor mit den erwarteten Werten überein, dann muss nichts gemacht werden. Liegt aber ein Fehler vor, d.h. eine Differenz zwischen Ausgabe und Ziel, dann müssen die Gewichte angepasst werden. Die Idee des Back-Propagation-Algorithmus ist es, alle für den Fehler verantwortlichen Gewichte in die Korrektur einzubeziehen. Für die Ausgabeeinheiten ist dies am einfachsten und die Regel ist der für Perzeptrone ähnlich. Es gibt zwei Unterschiede: Anstelle eines Eingabewerts wird die Aktivierung einer verborgenen Einheit benutzt und es wird die erste Ableitung g‘ der Aktivierungsfunktion g verwendet. Ist $Err_i = T_i - O_i$ der Fehler an der Ausgabeeinheit *i*, dann ist die Aktualisierungsregel für das Gewicht an der Kante von Einheit *j* zur Ausgabeeinheit *i*

$$W_{j,i} \leftarrow W_{j,i} + \alpha \cdot a_j \cdot Err_i \cdot g'(in_i)$$

Es ist üblich, einen speziellen Fehlerausdruck $\Delta_i = Err_i \cdot g'(in_i)$ zu definieren. Dieser Fehlerwert wird nämlich zurückpropagiert. Damit wird die Regel verkürzt zu

$$W_{j,i} \leftarrow W_{j,i} + \alpha \cdot a_j \cdot \Delta_i$$

Der Fehler Δ_i an der Ausgabeeinheit i wird nun auf die vorhergehenden verborgenen Einheiten aufgeteilt und zwar proportional zu den Gewichten der Kanten von den verborgenen Einheiten zu i. Die Fehler Δ_j der verborgenen Einheiten werden durch die Propagierungsregel wie folgt berechnet:

$$\Delta_j = g'(in_j)\sum_i W_{j,i}\Delta_i$$

Die Aktualisierungsregel für die Gewichte von den Eingabeeinheiten zu den verborgenen Einheiten ist damit

$$W_{k,j} \leftarrow W_{k,j} + \alpha \cdot I_k \cdot \Delta_j$$

Der vollständige Algorithmus wird durch die Funktion BACK-PROP-UPDATE implementiert.

function BACK-PROP-UPDATE(*Netz*, *Beispiele*, α) **returns** ein *Netz* mit modifizierten Gewichten

inputs: *Netz*, ein mehrschichtiges zyklenfreies Netz
Beispiele, eine Menge von Eingabe/Ausgabe-Paaren
α, die Lernrate

repeat
for each *e* **in** *Beispiele* **do**
/* Berechnung der Ausgabe für dieses Beispiel */
$\mathbf{O} \leftarrow$ RUN-NETWORK(*Netz*, $\mathbf{I}^e$)
/* Berechnung des Fehlers und von Δ für die Ausgabeeinheiten */
$\mathbf{Err}^e \leftarrow \mathbf{T}^e - \mathbf{O}$
/* Aktualisierung der Gewichte, die zu der Ausgabeschicht führen */
$W_{j,i} \leftarrow W_{j,i} + \alpha \cdot a_j \cdot Err_i^e \cdot g'(in_i)$
for each nachfolgende Schicht **in** *Netz* **do**
/* Berechnung des Fehlers an jeder Einheit */
$\Delta_j \leftarrow g'(in_j) \cdot \sum_i W_{j,i} \cdot \Delta_i$
/* Aktualisierung der Gewichte, die in die Schicht führen */
$W_{k,j} \leftarrow W_{k,j} + \alpha \cdot I_k \cdot \Delta_j$
end
end
until *Netz* ist konvergiert
return *Netz*

Der Algorithmus kann folgendermaßen zusammengefasst werden:

- Berechne die Δ-Werte für die Ausgabeeinheiten, ausgehend vom beobachteten Fehler.
- Wiederhole die folgenden Schritte für jede Schicht im Netz, beginnend mit der Ausgabeschicht, bis die erste verborgene Schicht erreicht ist:
  - Propagiere die Δ-Werte zurück zu der vorhergehenden Schicht.
  - Aktualisiere die Gewichte zwischen den beiden Schichten.

7.4.2. Back Propagation als Suche mit absteigendem Gradienten

Back Propagation kann als Suche im Raum der Gewichte mit absteigendem Gradienten betrachtet werden. Der Gradient liegt auf der Fehler-Oberfläche. Diese Oberfläche beschreibt den Fehler auf jedem Beispiel als Funktion aller Gewichte im Netz. Abbildung 7.6 zeigt ein Beispiel für eine Fehler-Oberfläche im zweidimensionalen Fall, d.h. für zwei Gewichte.

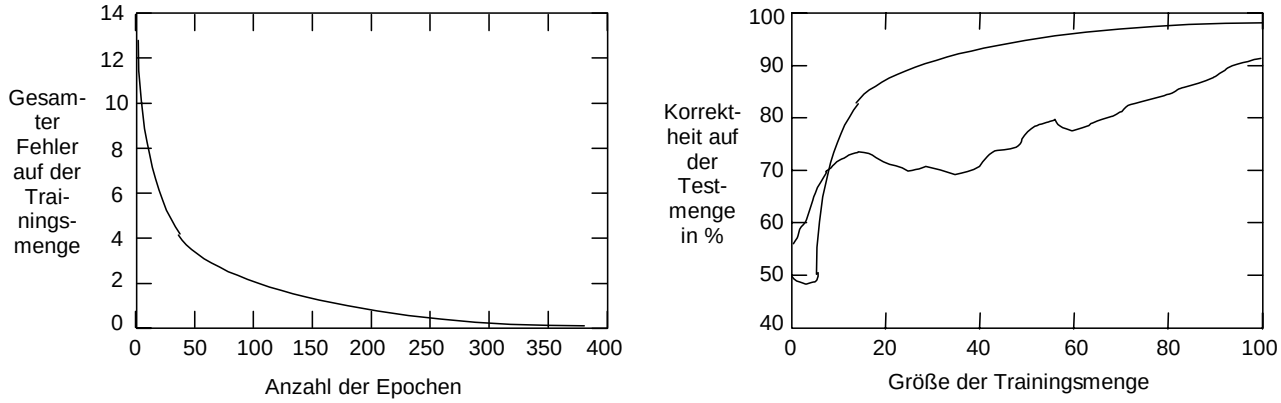


Abbildung 7.d

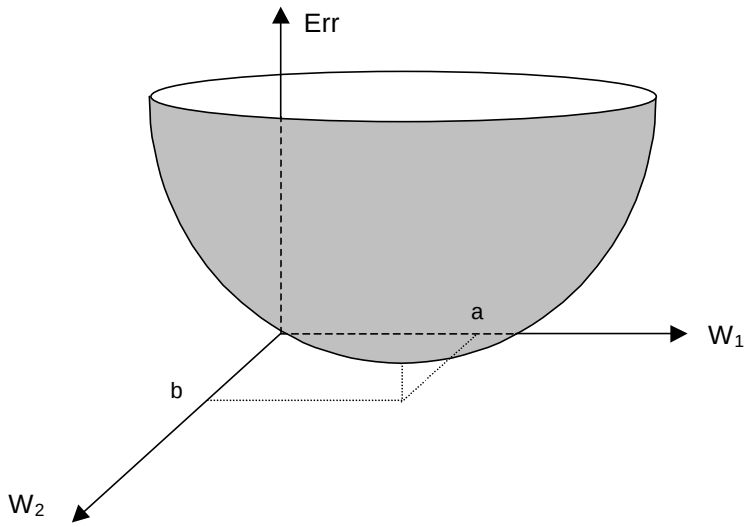


Abbildung 7.6

Die aktuelle Menge der Gewichte definiert einen Punkt auf der Oberfläche. Von einem solchen Punkt aus betrachtet man die Neigung der Oberfläche entlang den Achsen, die die Gewichte repräsentieren. Diese Neigung stellt die partielle Ableitung auf der Oberfläche bezüglich jedes Gewichts dar, d.h. sie beschreibt, um wie viel sich der Fehler ändert, wenn man eine kleine Änderung an dem jeweiligen Gewicht vornimmt. Macht man dies mit jedem Gewicht gleichzeitig, dann bewegt sich das Netz als Ganzes entlang dem steilsten Abfall auf der Fehler-Oberfläche.

Der Schlüssel zum Verständnis des Back Propagation ist, dass es eine Möglichkeit bietet die Berechnung des Gradienten auf die Einheiten im Netz aufzuteilen, so dass die Änderung in jedem Gewicht durch die Einheit berechnet werden kann, der das Gewicht zugeordnet ist, wobei sie nur lokale Information benutzt.

7.4.3. Eigenschaften des Back-Propagation-Lernens

Das Back-Propagation-Lernen kann nach den selben Eigenschaften beurteilt werden, die man auch an andere Lernverfahren als Kriterien anlegt.

Ausdruckskraft Neuronale Netze sind eine Attribut-basierte Repräsentation und haben deshalb nicht die selbe Ausdruckskraft wie allgemeine logische Beschreibungen. Sie sind für kontinuierlich-wertige Ein- und Ausgaben geeignet, im Unterschied zu Entscheidungsbäumen. Die Klasse aller mehrschichtiger Netze als Ganze kann zwar jede Funktion der Eingaben repräsentieren, aber ein einzelnes Netz kann zu wenige verborgene Einheiten haben.

Effizienz Die Effizienz ist wesentlich durch die Rechenzeit bestimmt, die für das Trainieren des Netzes, um eine Beispielmenge korrekt zu klassifizieren, benötigt wird. Bei m Beispielen und $|\mathbf{W}|$ Gewichten braucht jede Epoche $O(m|\mathbf{W}|)$ Zeit. Die worst-case-Anzahl der Lernepochen hängt aber exponentiell von der Zahl n der Eingaben ab. Ein Problem stellen die lokalen Minima in der Fehler-Oberfläche dar.

Generalisierung Neuronale Netze haben eine relative gute Generalisierungsfähigkeit. Dies ist insbesondere bei Funktionen der Fall, für die sie besonders geeignet sind. Von diesem Typ sind Funktionen, bei denen die Interaktionen zwischen den Eingaben nicht zu verwickelt sind und bei denen die Ausgaben sich „stetig“ mit den Eingaben verändern.

Empfindlichkeit gegen Rauschen Da die Neuronalen Netze nichtlineare Regression durchführen sind sie relativ unempfindlich gegenüber verrauschten Daten. Sie bestimmen diejenige Ausgabe, die unter den Gegebenheiten des Netzes am besten zu einer Eingabe passt. Dafür können sie auch keine Wahrscheinlichkeitsverteilung für die Ausgaben liefern.

Transparenz Neuronale Netze sind im Prinzip Black Boxes. Sie können nicht angeben, warum eine bestimmte Ausgabe gemacht worden ist, d.h. sie haben keine Erklärungsfähigkeit.

Vorwissen Liegt bereits Wissen über ein Anwendungsgebiet vor, dann ist es meist vorteilhaft, dieses in einem lernenden System zu verwenden. Der Lernerfolg hängt manchmal entscheidend davon ab, ob solches Vorwissen vorhanden ist. Bei Neuronalen Netzen ist die Verwendung von Vorwissen wegen ihrer mangelnden Transparenz schwierig.

7.5. Anwendungen Neuronaler Netze

7.5.1. Englische Aussprache

Die Aufgabe ist geschriebenen englischen Text flüssig auszusprechen. Es geht dabei darum den Strom der Textzeichen in Phoneme, die grundlegenden Lautelemente, umzusetzen und diese einem elektronischen Sprachgenerator zu übergeben. Das Problem ist also die Abbildung der Textzeichen in Phoneme. Die Schwierigkeit dabei ist vor allem im Englischen, dass die meisten Aussprache-regeln nur näherungsweise korrekt sind und es sehr viele Ausnahmen gibt. Deshalb eignen sich hier Neuronale Netze.

Sejnowski und Rosenberg entwickelten 1987 das Programm NETtalk, ein Neuronales Netz, das lernt geschriebenen Text auszusprechen. Eingabe ist eine Folge von Zeichen, die in einem Fenster präsentiert werden, das über den Text gleitet. Das Fenster zeigt immer das gerade auszusprechende Zeichen zusammen mit den drei vorangehenden und den drei nachfolgenden. Das Netz hat 29 Eingabeeinheiten, je eine für die 26 Buchstaben, eine für Blank, eine für den Punkt und eine für die restlichen Satzzeichen. Es hatte ferner 80 verborgene Einheiten und Ausgabeeinheiten, die die

Merkmale der zu produzierenden Laute repräsentierten: laut oder leise, betont oder unbetont usw.. An Stellen, wo zwei oder mehr Buchstaben einen einzigen Laut darstellen, ist die Ausgabe für die folgenden Zeichen leer.

Das Netz wurde mit einem Text aus 1024 Wörtern trainiert. Für diesen wurde phonetische Schreibweise vorgegeben. Das Netz konnte nach 50 Durchläufen durch die Trainingsbeispiele diese mit 95% Genauigkeit wiedergeben. Eine hundertprozentige Genauigkeit kann grundsätzlich nicht erreicht werden, weil die Aussprache vielfach kontextabhängig ist. Ein Programm, das nur einen begrenzten Textausschnitt in einem Fenster zu sehen bekommt, kann verschiedene Aussprachen des selben Worts nicht auseinander halten.

Die Erfolgsquote bei den Testdaten betrug allerdings nur 78%. Das erscheint zwar verständlich, ist aber ein wesentlich schlechterer Wert als der, den kommerziell verfügbare Programme zeigen. Allerdings ist der Aufwand für die Erstellung dieser Programme erheblich höher, er beträgt mehrere Personenjahre, während NETtalk in wenigen Monaten entwickelt und in einigen Dutzend Stunden trainiert wurde.

7.5.2. Erkennen von Handschrift

Le Cun und Kollegen entwickelten 1989 ein Neuronales Netz zum Erkennen von Postleitzahlen auf handgeschriebenen Briefumschlägen. Die Positionsbestimmung und die Segmentierung in einzelne Ziffern besorgt ein Preprozessor, das Netz muss nur die Ziffern identifizieren. Es benutzt einen 16×16-Pixelarray als Eingabe, hat drei verborgene Schichten mit 768, 192 bzw. 30 Einheiten und 10 Ausgabeeinheiten für die Ziffern 0 bis 9. Bei vollständiger Verknüpfung zwischen den Einheiten würde man 200000 Gewichte bekommen, so dass das Netz nicht trainierbar wäre. Statt dessen wurde nur teilweise verknüpft, und die Verknüpfungen dienten als Merkmalsdetektoren. Die erste verborgene Schicht wurde außerdem in Gruppen von Einheiten unterteilt, die alle die gleiche Menge von 25 verschiedenen Gewichten benutzten. Dadurch konnte die verborgene Schicht 12 verschiedene Merkmale identifizieren. Das gesamte Netz benutzte nur 9760 Gewichte.

Das Netz wurde mit 7300 Beispielen trainiert und auf 2000 Beispielen getestet. Es kann in schwierigen Fällen bei zwei oder mehreren Ausgabeeinheiten einen hohen Wert ausgeben und damit andeuten, welche Lesart hohe Wahrscheinlichkeit hat. In den restlichen Fällen erreichte es eine Genauigkeit von 99%.

7.5.3. Autofahren

Pomerleau entwickelte 1993 das System ALVINN (Autonomous Land Vehicle In a Neural Network). Es lernte durch Beobachten eines menschlichen Fahrers ein Auto zu steuern. Die Aktionen waren Lenken, Beschleunigen und Bremsen. Als Sensoren wurden eine Farb-Stereo-Videokamera, Laser und Radar verwendet. Das Bild der Kamera wurde in einen Pixelarray umgesetzt und an ein 30×32-Gitter von Eingabeeinheiten umgesetzt. Die Ausgabeschicht hat 30 Einheiten, jede entspricht einer Steueraktion. Es wird die Aktion ausgeführt, deren zugehörige Einheit den höchsten Aktivierungswert hat. Es gibt außerdem eine verborgene Schicht mit 5 Einheiten, die voll mit den Eingabe- und Ausgabeeinheiten verknüpft sind.

Die zu lernende Funktion ist eine Abbildung von den Kamerabildern auf Steueraktionen. Die Trainingsdaten wurden durch Aufzeichnen der Aktionen eines menschlichen Fahrers zusammen mit den Kamerabildern gewonnen. Nach Aufzeichnen von 5 Minuten Fahrt und einem 10-minütigen Training des Netzes konnte ALVINN bereits allein fahren.

Die Lernergebnisse sind beeindruckend. ALVINN fuhr mit 70 mph über eine Entfernung von 90 Meilen auf öffentlichen Landstraßen in der Nähe von Pittsburgh. Es kann auch auf Straßen ohne Fahrspuren, auf geteerten Fahrradwegen und auf zweispurigen Vorortstraßen fahren. Es kann aber nur auf Straßen fahren, für die es trainiert worden ist und es hat sich als nicht sehr robust gegenüber wechselnden Lichtverhältnissen und anderen Fahrzeugen erwiesen.