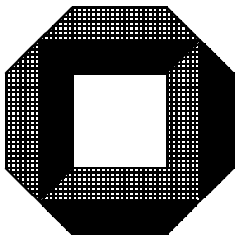
Proseminar

Künstliche Intelligenz



Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Algorithmen und Kognitive Systeme

Prof. Dr. J. Calmet
Dipl.-Inform. A. Daemi

Wintersemester 2003/2004

Institut für Algorithmen und Kognitive Systeme
Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
76 128 Karlsruhe

Lernen

Markus Przybylski
Michael Mai

Inhaltsverzeichnis

5 Lernen **2**

5.1 Einleitung 2

5.2 Lernen durch Beobachten 2

5.2.1 Entscheidungsbäume 3

5.2.2 Ensemble Learning 5

5.3 Statistische Lernmethoden 6

5.3.1 Naive Bayes Modelle 6

5.3.2 Neuronale Netzwerke 7

5.4 Reinforcement Learning 14

5.4.1 Passiv 15

5.4.2 Aktiv 16

5.5 Zusammenfassung 19

Kapitel 5

Lernen

5.1 Einleitung

Im Blickpunkt des Interesses der Künstlichen Intelligenz steht das Problem der Wissensrepräsentation: Jedes „intelligente“ System besitzt ein Modell der Wirklichkeit, anhand dessen es seine Entscheidungen trifft. Lernen bedeutet in diesem Kontext die sukzessive Anpassung der Parameter des Modells, um eine immer bessere Wiedergabe der Wirklichkeit zu erreichen und damit dem System „bessere“ Entscheidungen zu ermöglichen.

Zunächst soll ein grober Überblick über die verschiedenen hier behandelten Ansätze des Lernens gegeben werden.

Lernen durch Beobachten beschäftigt sich mit Entscheidungsfindung, die aufgrund von Beobachtungsdaten antrainiert wurden. Wir wenden uns zunächst den intuitiven **Entscheidungsbäumen** zu, um anschließend das demokratische **Ensemble Learning** näher zu betrachten.

Statistische Lernmethoden stützen sich auf Methoden der Wahrscheinlichkeitstheorie. **Naive Bayes Modelle** stellen einen ebenso einfachen wie leistungsfähigen Ansatz dar und sollen nur kurz gestreift werden. **Neuronale Netze** und ihre Möglichkeiten sollen ausführlich besprochen werden.

Reinforcement Learning nutzt Rückmeldungen aus der Umgebung, die erlernt werden soll. Bei **Passivem Reinforcement** wird nur das Modell erlernt, beim **Aktivem Reinforcement** zusätzlich die Aktionen, mit denen die Umgebung manipuliert wird.

5.2 Lernen durch Beobachten

Lernen durch Beobachten wird auch „Supervised Learning“ genannt. Als einleitendes Beispiel kann man sich die folgende Situation vorstellen:

> **Fahrschulunterricht**: Der Lehrer sitzt neben dem Schüler. Angenommen der Fahrschüler kann die Straßenschilder nicht lesen und brettert deswegen durch die Gegend. Der Lehrer wird dann irgenwann das Steuer übernehmen und seinem Fahrschüler erst einmal zeigen, wie man richtig fährt.

Bei den Methoden des induktiven Lernen wird eine approximative Anpassung versucht. Die beobachteten Daten stellen die Grundlage dafür da. Die Idee des induktiven Lernen geht auf die Grundannahme zurück, dass es überhaupt etwas zu lernen gibt. Ist diese Annahme nicht erfühlt so scheitert dieses Verfahren – wie auch alle anderen Methoden, die Lernfähigkeit simulieren sollen. Fairerweise muss erwähnt werden, dass auch der Mensch in einem solchen Fall nicht in der Lage ist, etwas zu lernen.

Hier wird im speziellen eine Lösung bzw. eine Annäherung innerhalb einer bestimmten Menge gesucht, des so genannten Hypothesen-Raums. In ihm findet man zum Beispiel die Annahme, dass das zu Lernende sich mit einer linearen Funktion oder einem Polynom beschreiben läßt. Wählt man die Grundmenge ungeeignet, so kann man niemals einen guten Algorithmus finden.

Würde man zum Beispiel als Hypothesen-Raum den Raum der linearen Funktionen annehmen, so wäre es nicht möglich einen Algorithmus zu finden, der ein Polynom 12ten Grades gut vorhersagt.

Man könnte natürlich auch auf die Idee kommen, als Grundmenge gleich die Menge aller berechenbaren Algorithmen, also die Menge der Turingmaschinen ($\mathcal{TM}$), anzunehmen. Dadurch hätte man sämtliche möglichen Algorithmen, die einen Lernprozess simulieren können, eingeschlossen. Aber durch die Komplexität auf dieser Grundmenge zu operieren ist es nicht ratsam die $\mathcal{TM}$ als Grundmenge zu wählen.

In der Wissenschaft gibt es eine „goldene“ Regel, *Ockam's Gesetz*, dass die einfachste Hypothese, die im Einklang mit den Daten steht, zu bevorzugen ist.

Lernen soll kein Selbstzweck sein, sondern es soll natürlich auch etwas Nützliches daraus entstehen. Alleine schon aus diesem Grund beschränkt man sich auf einen handhabbaren Hypothesen-Raum.

5.2.1 Entscheidungsbäume

Entscheidungsbäume stellen eine einfache Art des Induktiven Lernens da. Je nach Art der Ausgabe des Entscheidungsbaumes muss man weitere Unterscheidungen treffen.

Ausgabe	Bezeichnung
diskret	Klassifizierung
kontinuierlich	Regression

Pfade im Entscheidungsbaum kann man prädikatenlogisch ausdrücken. Hier ein Beispiel:

$$\forall s: \qquad WillWait(s) \Leftrightarrow (P_1(s) \vee P_2(s) \vee \cdots \vee P_n(s))$$

Wobei jedes $P_i(s)$ ein konjunktiv verknüpfter Ausdruck sein kann, der einen Test in einem in einem Teilbaum darstellt.

Das Erstellen der Bäume kann entweder gezielt durch manuelle Konstruktion geschehen oder mittels Algorithmen, die mit Trainingsdatensätzen den Baum konstruieren. Solche Algorithmen werden Decision-Tree-Learning-Algorithmen genannt.

Die Decision-Tree-Learning-Algorithmen verwenden Erkenntnisse und Techniken aus der Informationstheorie, um zu entscheiden, welcher Weg bei jedem Schritt als nächstes zu wählen ist. Hierzu wird der Informationsgehalt (→ information-gain) eines Pfades berechnet, der in [Bit] gemessen wird, und mit dessen Hilfe entschieden, ob es sich lohnt, diesen Weg oder einen anderen einzuschlagen.

Natürlich muss ein Entscheidungsbaum getestet werden. Ein Fehler, der hierbei häufig gemacht wird, ist, dass man man mit dem Trainingsdaten testet. Dies ist übrigens ein genereller Fehler beim Testen von Programmen und Algorithmen. Hierbei erhält man dann natürlich eine hunderprozentige Übereinstimmung. Diese exakte Übereinstimmung ist allerdings trügerisch, denn man kann sich ohne weiteres vorstellen, dass Zusammenhänge, die es scheinbar in den Trainingsdaten gab, in Wirklichkeit gar nicht gibt. Ein Teil dieses Phänomens werden wir beim Overfitting näher besprechen.

Ein guter Lernalgorithmus zeichnet sich dadurch aus, dass er gute Vorhersagen zu Daten machen kann, die ihm zuvor nicht antrainiert wurden. Aus dieser einleuchtenden Definition heraus kann man leicht erkennen, dass mehr Training – in der Regel – einen besseren Algorithmus fördert. Konkret heißt dies, dass man den Algorithmus mit umfangreichen Trainingsdatensätzen trainieren sollte, damit seine „Güte“ steigt bzw. verbessert wird.

Overfitting

Bei einem solchen Vorgehen können allerdings auch negative Effekte auftreten. „Overfitting“ nennt man das Anpassen an Rauschen („Noise“) in den Daten. Als Rauschen bezeichnet man ein Datum, dass nichts bedeutet – eine unbedeutende Abweichung. Mit statistischen Techniken kann man dieses Rauschen mildern bzw. sogar entfernen.

- Die χ^2-Methode ist hierfür gut geeignet.
  Mit der χ^2-Technik wird der Umfang der Trainingsdaten reduziert, nämlich um das Rauschen; der daraus entstehende Baum ist in der Regel einfacher und leichter zu verstehen.

- Cross-Validation nennt sich eine andere Technik.
  Cross-Validation gleicht die Vorhersagen mit den Werten aus einem Testdatensatz, welche nicht beim Training verwendet wurden, ab. Eine große Stärke von Cross-Validation ist es, dass man diese Technik auf alle Lernkonzepte anwenden kann, nicht nur auf Entscheidungsbäume.

Einige wichtige Anmerkung zur realen Welt sollte man hier noch machen. Wenn wir uns zunächst der Eigenschaft eines einzelnen Datum zuwenden fallen folgende Situationen ins Auge:

Fehlende Daten	Nicht erfassbar oder zu teuer, diese Daten zu erheben.
Superposition	Daten, die mehrere Bedeutungen haben.

Im Hinblick auf die Parameter und die gewünschte Ausgabe können wir folgendes feststellen:

kontinuierliches Input-Attribut	Splitten ab einem bestimmten Punkt wird notwendig.
kontinuierliches Output-Attribut	*Regression-Tree*, der eine Funktion ausgibt.

Ein sehr großer Vorteil und auch im rechtlichen Sinne wichtiger Aspekt von Entscheidungsbäumen ist die Nachvollziehbarkeit. Bei anderen Lernalgorithmen ist die Entscheidungsfindung nur schwer oder gar nicht nachvollziehbar.

5.2.2 Ensemble Learning

Beim Ensemble Learning verwendet man mehrere Algorithmen, welche nicht notwendigerweise die gleichen Konzepte verfolgen müssen.

Ensemble Learning realisiert die Demokratie im Reiche der künstlichen Intelligenz. Nach dem Motto „Die Mehrheit kann nicht irren“ arbeitet Ensemble Learning. Ensemble Learning ist an für sich kein eigenes Lernkonzept sondern eine Art Meta-Konzept, das die Irren eines Algorithmus auszugleichen versucht.

In der Praxis sieht das dann so aus: Man trainiert mehrere Algorithmen an der selben Sache, so dass jeder einzelne die gleiche Sache gut beschreiben kann. Es liegt in der Natur der Algorithmen, die Lernen simulieren, dass immer noch Abweichungen von der realen Welt vorkommen.

Läßt man nun eine Prognose von allen beteiligten Algorithmen berechnen und alle prognostizieren den selben Wert, so ist dieser Wert mit sehr hoher Wahrscheinlichkeit ein guter Prognosewert.

Kommt hierbei allerdings heraus, dass etwa die eine Hälfte den Wert x und die andere Hälfte den Wert y vorhersagen, so weiß man, dass weder x noch y als besonders sicher gelten können. Hätte man nur einen Algorithmus verwendet, so hätte man nie erfahren, dass dieser Prognosewert heikel oder unsicher ist.

Für ein möglichst gutes Ergebnis kann man verschiedene Algorithmen verwenden, die nach unterschiedlichen Konzepten konstruiert wurden. Dadurch kann man die unterschiedlichen Vor- und Nachteile der einzelnen Algorithmen ausnutzen bzw. vermeiden. Man kann hierbei auch von unterschiedlichen Hypothesen-Räumen Gebrauch machen. Ein positiver Effekt ist hierbei die Möglichkeit, eine große Streuung zu nutzen - und dies mit einem günstigen Aufwand.

Boosting

Ein häufig verwendetes Ensemble Learning Muster nennt sich Boosting. Beim Boosting zieht man die Trainingsdaten heran, um das gemeinsame Ergebnis zu verbessern. Hierzu fügt man zu jedem Datum d_j im Trainingssatz ein Gewicht w_j hinzu. Der erste Durchlauf ergibt die erste Hypothese h_1, hierbei werden einige Daten korrekt und andere falsch klassifiziert.

Die Gewichte der korrekt klassifizierten Daten werden erniedrigt und die der falsch klassifizierten erhöht. Ein weiterer Durchlauf würde dann die Gewichte berücksichtigen und eine verbesserte Hypothese h_{n+1} erstellen. Durch mehrfaches Anwenden dieses Verfahrens wird Schritt für Schritt der Algorithmus verbessert. Ein berühmt gewordener Boosting-Algorithmus ist AdaBoost.

5.3 Statistische Lernmethoden

5.3.1 Naive Bayes Modelle

Die Bayes-Formel

Naiven Bayes Modellen liegt die Bayes-Formel zugrunde:

$$\mathrm{P}(U \mid W) = \frac{\mathrm{P}(W \mid U)\mathrm{P}(U)}{\mathrm{P}(W)} \tag{5.1}$$

U steht für Ursache, W für Wirkung.

Der Nutzen der Bayes-Formel besteht darin, dass sich mit ihrer Hilfe die Wahrscheinlichkeit von Zusammenhängen nicht nur in *kausaler*, sondern auch in *Diagnoserichtung* untersuchen lässt.

Zur Veranschaulichung soll folgendes Beispiel betrachtet werden. Eine Person hat Zahnschmerzen. Die *a-priori*-Wahrscheinlichkeiten $\mathrm{P}(Loch)$ und $\mathrm{P}(Schmerzen)$ sind bekannt, ebenso die bedingte Wahrscheinlichkeit $\mathrm{P}(Schmerzen \mid Loch)$, dass ein Loch im Zahn zu Zahnschmerzen führt. Mithilfe der Bayes-Formel kann nun berechnet werden, mit welcher Wahrscheinlichkeit $\mathrm{P}(Loch \mid Schmerzen)$ nun wirklich ein Loch im Zahn vorliegt, wenn das Vorhandensein von Schmerzen bekannt ist. Schließlich könnten die Zahnschmerzen ja auch eine andere Ursache haben, z.B. eine Erkältung.

Warum „Naiv“?

Naive Bayes-Modelle gehen von der Annahme aus, dass alle Folgeereignisse untereinander *bedingt unabhängig* sind, falls die Ursache gegeben ist. Somit kann die folgende Formel angewandt werden:

$$\mathrm{P}(Y \mid X_1, \ldots, X_i) = \frac{\mathrm{P}(Y)}{\mathrm{P}(X_1, \ldots, X_i)} \prod_{i=1}^{n} \mathrm{P}(X_i \mid Y) \tag{5.2}$$

Bleiben wir zur Erläuterung beim Beispiel mit dem Loch im Zahn, jedoch mit einem etwas abgewandelten Modell. Am Anfang steht wieder das Ereignis „Loch im Zahn“. Folge davon können nun zwei verschiedene Ereignisse sein: „Zahnschmerzen“ und „Zahnarztbesuch“. Dem geneigten Leser wird unmittelbar bewusst, dass in unserem Modell ein Fehler gemacht wurde: In der Realität führt normalerweise noch nicht das Loch im Zahn zu einem Zahnarztbesuch, sondern erst die daraus resultierenden Schmerzen. In einem Naiven Bayes Modell wird genau dieser Zusammenhang bewusst ignoriert, d.h. die Ereignisse „Zahnschmerzen“ und „Zahnarztbesuch“ werden unter der Voraussetzung, das ein Loch im Zahn vorliegt, als bedingt unabhängig betrachtet.

Auch wenn die bedingte Unabhängigkeit in der Praxis oft nicht gegeben ist, leisten Modelle, die auf dieser vereinfachenden Annahme aufbauen, in realen Anwendungen oft sehr gute Dienste.

Insbesondere muss die exzellente **Skalierbarkeit** hervorgehoben werden. Der Ansatz ermöglicht das vollständige Faktorisieren der kompletten Wahrscheinlichkeitsverteilung, wodurch eine Wissensrepräsentation mit Speicheraufwand $\mathrm{O}(n)$ (anstelle von $\mathrm{O}(2^n)$ für die vollständige Tabelle aller atomaren Ereignisse) erreicht wird, wenn n die Anzahl der Booleschen Variablen zur Beschreibung der Welt ist. Die Wahrscheinlichkeit jedes atomaren Ereignisses kann dabei durch Multiplikation gemäß Formel 5.2 gewonnen werden.

5.3.2 Neuronale Netzwerke

Was ist ein Neuron?

Ein Neuron ist eine Gehirnzelle, die zur Verarbeitung elektrischer Signale dient. Jede einzelne solche Zelle ist über die Synapsen mit bis zu 100.000 weiteren Neuronen verbunden. Erst die Vernetzung ermöglicht die Verarbeitung komplexer Informationen.

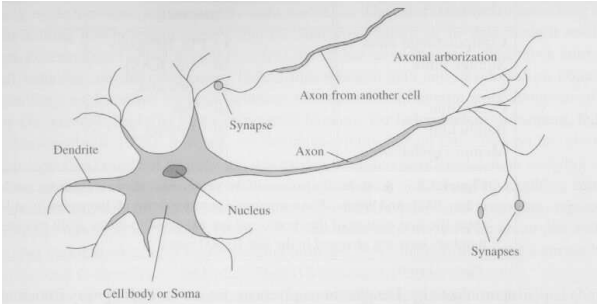


Abbildung 5.1: Bild eines Neurons

Künstliche Neuronale Netze - Aufbau

Ein neuronales Netz lässt sich veranschaulichen als gerichteter Graph mit Kantengewichten. Die einzelnen Neuronen werden durch Knoten repräsentiert. Die

Kanten zwischen den Knoten stehen für die Axonen und dienen der Weitergabe von elektrischen Signalen, wobei die Gewichte die Intensität der jeweiligen Verbindung und das Vorzeichen des übertragenen Signals darstellen.

Aufbau und Funktion eines Neurons

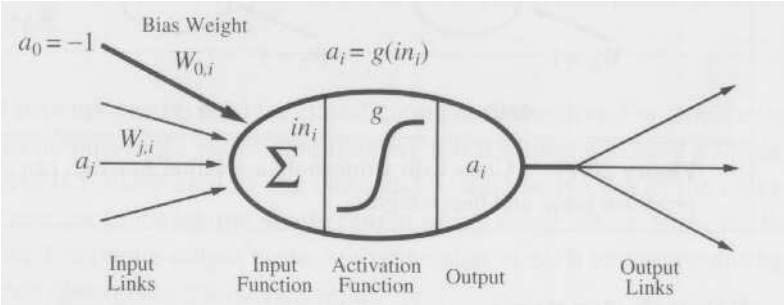


Abbildung 5.2: Mathematisches Modell eines Neurons

Funktion: Aus den Signalen aller Eingangskanten wird gemäß der Kantengewichte $W_{j,i}$ eine Linearkombination als Eingabewert des Neurons berechnet, wobei j das Start- und i das Endneuron der Kante ist:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j \tag{5.3}$$

Auf diesen Eingabewert wird eine sogenannte Aktivierungsfunktion g angewendet. Der Funktionswert ist die Ausgabe des Neurons:

$$a_i = g(in_i) = g(\sum_{j=0}^{n} W_{j,i} a_j) \tag{5.4}$$

Wahl einer Aktivierungsfunktion

Falls eine **diskrete Ausgabe** erwünscht ist, bietet sich beispielsweise die *Stufenfunktion* an. Sie eignet sich besonders gut, falls das Neuronale Netz eine Boolesche Funktion auswerten soll:

$$f(x) = \begin{cases} 0 & \text{falls } x < 0 \\ 1 & \text{sonst} \end{cases} \tag{5.5}$$

Für eine stetige Ausgabe eignet sich dagegen die Logistische Funktion gut:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.6}$$

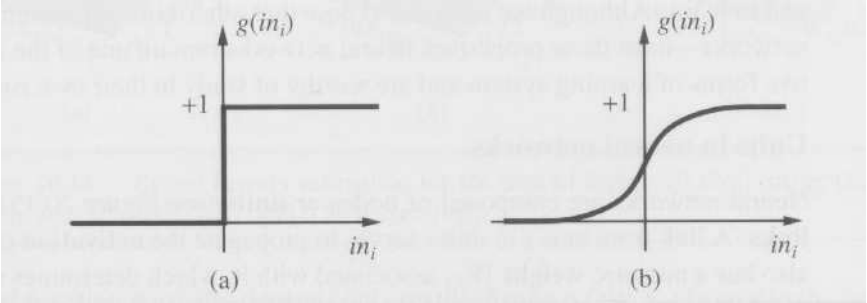


Abbildung 5.3: Die Stufenfunktion (a) und die Logistische Funktion (b)

Besondere Bedeutung kommt dem sogenannten **Bias Weight** $W_{0,i}$ zu: es bestimmt die **Empfindlichkeit** des Neurons. Das Neuron feuert genau dann, wenn die Linearkombination der Eingaben größer als dieser Schwellenwert ist, wenn also gilt:

$$\sum_{j=0}^{n} W_{j,i} a_j > W_{0,i} \tag{5.7}$$

Beispiel: Mithilfe eines einzelnen Neurons kann z.B. auf einfache Art und Weise die boolesche Funktion AND realisiert werden:

1. Wähle Neuron mit zwei Eingangskanten I_1 und I_2, beide mit Gewicht 1.
2. Lasse nur Werte aus {0,1} als Eingabe zu.
3. Wähle Treppenfunktion als Aktivierungsfunktion.
4. Setze die Aktivierungsschwelle (Bias Weight) auf den Wert $1,5$.

Wählt man dagegen für die Aktivierungsschwelle den Wert $0,5$, so erhält man die OR-Funktion.

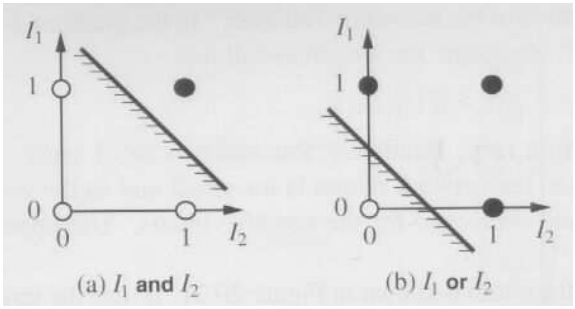


Abbildung 5.4: Darstellung der Booleschen Funktionen *AND* und *OR* durch ein Perzeptron mit einer Ausgabeeinheit

Verschiedene Netzwerkstrukturen

Neuronale Netze lassen sich ihrer *Struktur* nach weiter unterteilen:

Feed-Forward-Netze sind azyklisch. Sie berechnen lediglich eine *Funktion* ihrer Eingabewerte. Abgesehen von den Kantengewichten besitzen sie keinerlei internen Zustand.

Rekurrente Netzwerke dagegen enthalten Zyklen, d.h. Ausgabesignale werden wieder als Eingabe verwendet. Durch die Rückkopplung sind Systeme mit internem *Speicher* („Kurzzeitgedächtnis“) möglich. Rekurrente Netzwerke kommen der Struktur eines realen Gehirns näher, sind jedoch beträchtlich komplexer als Feed-Forward-Netzwerke.

Im Folgenden werden nur Feed-Forward-Netzwerke näher behandelt.

Feed-Forward-Netze ihrerseits lassen sich weiter unterscheiden anhand der Anzahl der *Neuronenschichten*:

Einstufige Netze bestehen lediglich aus Eingabe- und Ausgabeeinheiten, die direkt miteinander verbunden sind. Man bezeichnet sie auch als **Perzeptronen**. Beispiel hierfür sind die bereits weiter oben behandelten Netze zur Realisierung der *AND*- bzw. *OR*-Funktion.

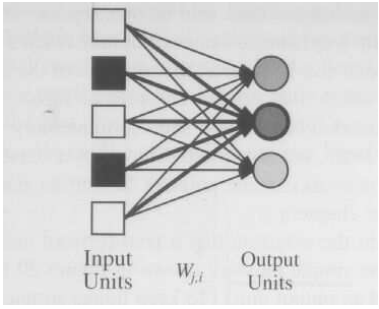


Abbildung 5.5: Ein Perzeptron mit drei Ausgabeeinheiten

Mehrstufige Netze enthalten zwischen den Ein- und Ausgabeeinheiten außerdem eine oder mehrere Schichten sogenannter „versteckter“ Neuronen.

Lernen bei einschichtigen Netzen

Welche Funktion ein Neuronales Netz repräsentiert, hängt im Wesentlichen von seinen einzelnen Kantengewichten ab. Ein wirkungsvoller Ansatz, einem Perzeptron eine Funktion „beizubringen“, besteht daher darin, die Gewichte solange anzupassen, bis die Funktion hinreichend gut approximiert wird.

Zu diesem Zweck wird zunächst eine Menge von **Trainingsdatensätzen** erzeugt, d.h. Eingabewerte der Funktion zusammen mit den zugehörigen Ergebnissen. Anschließend wird nach folgendem Algorithmus vorgegangen:

1. Die Eingabeeinheiten des Netzes erhalten die Trainingseingabe.
2. Das vom Netz errechnete Ergebnis wird mit dem in den Trainingsdaten hinterlegten Ergebnis verglichen; die Differenz bildet den Fehler.
3. Die Gewichte des Netzes werden dem Fehler entsprechend angepasst.
4. Der Vorgang wird mit den restlichen Eingaben aus der Menge der Trainingsdaten wiederholt.

Anschließend beginnt ein neuer Durchlauf durch die Trainingsdaten. Dies wird in der Regel solange fortgesetzt, bis die Änderung der Gewichte eine vorher festgelegte Schranke unterschreitet.

Mathematisch wird das Verfahren - zunächst für ein stetiges Perzeptron - wie folgt begründet:

Als Fehlermaß wird die *Summe der Fehlerquadrate* E herangezogen. Der quadratische Fehler wird beschrieben durch die Gleichung

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_w(x))^2 \tag{5.8}$$

wobei

x	die Eingabe,
y	die korrekte Ausgabe laut Trainingsbeispiel,
$h_w(x)$	die tatsächliche Ausgabe des Perzeptrons und
Err	den Fehler

bezeichnet.

Um die neuen Gewichte zu berechnen, wird für jedes einzelne Gewicht W_j jeweils die partielle Ableitung des quadratischen Fehlers nach diesem Gewicht berechnet:

$$\frac{\partial E}{\partial W_j} = -Err \times g'(in) \times x_j \tag{5.9}$$

g' ist die Ableitung der Aktivierungsfunktion.

Die Gewichte werden anschließend nach folgender Formel verändert:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j \tag{5.10}$$

α ist eine Konstante, die Lernrate. Anschaulich bedeutet die Formel, dass bei einem zu großen Ergebnis des Netzwerks die Gewichte verkleinert, bei einem zu kleinen Ergebnis vergrößert werden.

Im Falle eines Stufenperzeptrons funktioniert der Lernvorgang analog, jedoch wird in der Formel für die Gewichtsänderung g' weggelassen.

Beschränkung einschichtiger Netze - Lineare Klassifizierbarkeit

Die Grenzen der Leistungsfähigkeit eines Perzeptrons werden deutlich, wenn man sich das Funktionsprinzip nochmals vor Augen führt: Eine Aktivierungsfunktion lässt das Neuron „feuern“, falls die Linearkombination der Eingangssignale eine gewissen Schwellenwert überschreitet.

Besonders gut zu sehen ist das an den weiter oben erwähnten Booleschen Funktionen *AND* und *OR*, deren Realisierung durch ein Perzeptron sich nur durch die jeweilige Wahl des Schwellenwertes unterscheidet. Gemeinsam ist den beiden Funktionen, dass die beiden möglichen Klassen von Funktionswerten 1 und 0 durch eine Gerade getrennt werden.

Betrachtet man im Vergleich dazu nun die Boolesche Funktion *XOR*, so wird deutlich, dass in diesem Fall keine solche Gerade angegeben werden kann.

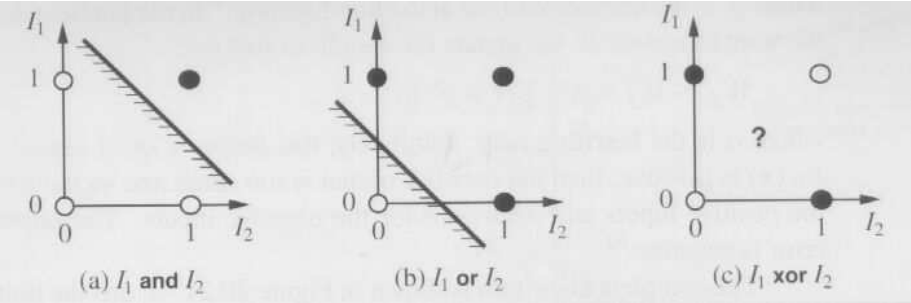


Abbildung 5.6: Die Boolesche Funktion *XOR* ist im Gegensatz zu *AND* und *OR* nicht linear klassifizierbar

Die *XOR*-Funktion ist nur eine Instanz des allgemeinen Problems der **Linearen Klassifizierbarkeit**. Der Schwellenwert definiert im Raum der möglichen Eingabewerte eine *Hyperebene* - in unserem zweidimensonalen Fall also eine Gerade - die den Raum in die beiden Klassen von Funktionswerten unterteilt. Kann für eine Funktion keine solche Hyperebene gefunden werden, so kann sie nicht von einem Perzeptron realisiert werden.

Mehrschichtige Netze

Das Problem der Linearen Klassifizierbarkeit kann durch Einfügen zusätzlicher sogenannter „versteckter“ Neuronenschichten zwischen den Ein- und Ausgabeeinheiten gelöst werden.

Verwendet man eine *stetige* Aktivierungsfunktion, ergeben sich vielfältige neue Möglichkeiten: Im Ergebnis einer Funktionsauswertung durch das Netz überlagern sich die Aktivierungsfunktionen der verschiedenen Neuronen in den versteckten Schichten.

Vorausgesetzt die versteckten Schichten sind groß genug, ist zur Darstellung aller stetigen Funktionen ein Netz mit einer versteckten Schicht ausreichend. Mit zwei oder mehr solchen Schichten können auch unstetige Funktionen dargestellt werden.

Lernen bei mehrschichtigen Netzen - Back Propagation

Der BACK-PROPAGATION-ALGORITHMUS ist eine Verallgemeinerung des Lernalgorithmus für Perzeptronen. Wir betrachten ein Netz mit mehreren versteckten Schichten und mehreren Ausgabeeinheiten.

Da wir jetzt anstelle eines Ergebniswertes einen Ergebnisvektor $h_w(x)$ haben, ist nun auch der Fehler ein Vektor: $\text{Err}(x) = y(x) - h_w(x)$

Die Gewichtsaktualisierungsregel der Ergebnisschicht des Netzes ist dieselbe wie beim Perzeptron, mit dem kleinen Unterschied, dass eben ein Ergebnisvektor vorliegt anstelle eines einzelnen Wertes.

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \tag{5.11}$$

wobei

$$\Delta_i = Err_i \times g'(in_i) \tag{5.12}$$

den Fehler in der i-ten Ausgabeeinheit bezeichnet.

Das eigentliche Problem besteht nun darin, ein brauchbares Modell dafür zu finden, wie sich der Fehler auf die Gewichte in den versteckten Schichten auswirkt.

Ein intuitiver Lösungsansatz findet sich, wenn man sich jeweils zwei benachbarte Schichten des Netzes als ein einzelnes Perzeptron vorstellt. Dann ist der Fehler in der Ausgabeschicht eine Folge der Fehler bei den Gewichten der Verbindungskanten zur Eingabeschicht. Im Hinblick auf ein mehrschichtiges Netz wird nun klar, dass ein Maß für die Beziehung der Fehlerwerte in der jeweils aktuellen Schicht zu denen der Vorgängerschicht gefunden werden muss. So können die Fehler in Rückwärtsrichtung bis zu den Eingabeeinheiten "durchgeschoben"werden.

Betrachten wir die letzte versteckte Schicht vor der Ausgabeschicht. Jeder der Knoten k_j trägt einen Teil zum Fehler in den Knoten der Ausgabeschicht bei, mit denen er verbunden ist.

Die Lösung ist nun leicht ersichtlich: Die Fehlerwerte Δ_i in den Ausgabeknoten werden entsprechend den Gewichten der Eingangskanten aufgeteilt und in die Knoten der davorliegenden Schicht verschoben, wodurch wir die einzelnen Fehlerwerte Δ_j in dieser Schicht erhalten. Dies geschieht nach folgender Regel:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \tag{5.13}$$

Daraus ergibt sich dann die folgende Formel für die Aktualisierung der Gewichte in den versteckten Schichten:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \tag{5.14}$$

wobei die k-te Schicht im Netz vor der j-ten Schicht liegt.

Lernen der Struktur

Der Vollständigkeit halber muss erwähnt werden, dass das Lernen durch Anpassen der Gewichte - ausgehend von einer fest vorgegebenen Netzwerkstruktur - nur eine von mehreren Möglichkeiten darstellt.

Ein anderer Ansatz sieht den Lernprozess als schrittweises Anpassen der Netzwerkstruktur an die Anforderungen der zu erlernenden Funktion.

Von Bedeutung sind in diesem Zusammenhang unter anderem folgende Verfahren:

Optimal Brain Damage geht von einem vollständig verbundenen Netz aus und eliminiert während des Lernens nach und nach Verbindungen und Neuronen, die für das Ergebnis unwesentlich sind.

Tiling lässt aus einem Netz mit nur einem Neuron ein Netz „wachsen“, das gerade groß genug ist, um die gewünschte Funktion zu realisieren.

5.4 Reinforcement Learning

Reinforcement Learning bezeichnet ein Verfahren, bei dem ein Feedback, also eine Rückmeldung über das Entschiedene, in die weiter Verbesserung des Lernalgorithmus miteinbezogen wird. Reinforcement Learning ist ein beliebtes Agenten-System-Design. Das Feedback, dass Reinforcement ausmacht, kann man in zwei unterschiedliche Klassen aufteilen:

Ständige Rückmeldungen sind Rückmeldungen nach jedem Spielzug in einem Spiel. Zum Beispiel durch die Bewertung des aktuellen Spielfelds.

Rückmeldungen am Ende eines Spiels. Der Agent würde sinnlos irgendwelche Züge machen und erst am Ende des Spiels erfahren, ob er gewonnen oder verloren hat.

Zur Vereinfachung setzten wir eine vollständig beobachtbare Umgebung voraus. Das bedeutet, dass jeder Zug – wenn wir bei unserer Spielumgebung bleiben – unserem Agenten ein beobachtbares und auswertbares Ergebnis auf dem Spielfeld hinterlässt. Diese Annahme ist nicht selbstverständlich.

Es gibt drei verschieden Designs, die das Konzept des Reinforcement Learning implementieren. Alle Designs haben für sich Vor- und Nachteile und unterscheiden sich in wichtigen Details:

Utility-based-Agent Erlernt eine Erwarungsfunktion (utility function) in Abhängigkeit eines Zustands und verwendet diese, um den nächsten Zug zu bewerten. Die Güte des nächsten Zuges soll, bemessen an dieser Funktion, maximal sein.

Q-learning Der Agent lernt in diesem Design ein $Aktion \times Wert$-Paar, auch Q-Funktion genannt, welches ihm in einem gegebenen Zustand den Erwartungswert des nächsten Zustandes bei einer gegebenen Aktion liefert.

Reflex Agent Ein Agent, der dieses Muster verwirklicht, erlernt „nur“ ein direktes Regelwerk, welches ihm sagt, was er im aktuellen Zustand zu tun hat.

Außerdem kann man noch Aktiv und Passiv unterscheiden. Beim passivem Design hat der Agent ein Modell seiner Umgebung zu erstellen, auch *Transistions-Modell* genannt. Dieses Merkmal wollen wir als erstes beleuchten.

Ein aktiver Agent muss zusätzlich noch seine Aktionen und seine Erfahrungen damit in ein Modell einbringen.

5.4.1 Passiv

Wie bereits erwähnt steht beim Passiven Reinforcement die Modell-Erstellung im Vordergrund. Ein Agenten-System muss beispielsweise in einer Spielumgebung die Spielregeln des Spiels erkennen und in ein Modell des Spiels umsetzen.

Direct utility estimation

Eine einfache Implementierung der *direct utility estimation*-Technik wurde in den 1950ern von Widrow und Hoff entwickelt. Die Idee war, dass der Erwartungswert eines Zustands gleich allen zu erwartenden Erwartungswerten der als nächstes besuchten Zustände sei. Die *direct utility estimation*-Technik läßt sich auch auf **Supervised Learning** (5.2) zurückführen, welches auf Induktivem Lernen basiert.

Es mag nicht ganz so einleuchtend sein, warum *direct utility estimation* auf **Lernen durch Beobachten** zurückführbar ist, stellt man sich aber die erfahrene Bewertung als Zuruf eines Lehrers vor, so wird die Ableitbarkeit doch recht schnell klar.

Die *direct utility estimation*-Methode konvergiert oft leider nur langsam gegen ein stabile Vorhersage, das heißt sie benötigt recht viele Evolutionsschritte, bis ein guter stabiler Algorithmus gefunden ist. Ein Grund für die langsame Konvergenz liegt in der Vorgehensweise des Verfahrens. Es werden, wie beim „Lernen durch Beobachten“, aus dem Raum der Hypothesen alle diejenigen herausgefiltert, die nicht passen.

Adaptive dynamic programming

Informationen über die Verbindungen zwischen den einzelnen Zuständen werden bei direct utility estimation nicht genutzt. Diese Informationen werden beim adaptive dynamic programming genutzt. Dieses Vorgehen hat große Ähnlichkeit mit dem Thema *Markov Decision*, das wir hier aber nicht behandeln möchten. Hierzu ist die Quelle [1] zu empfehlen.

Vereinfacht kann man adaptive dynamic programming so erklären: Man beobachtet, welche Aktion aus welchem Zustand wie häufig in einen anderen übergeht. Zur Erinnerung: die Übergänge sind mit Wahrscheinlichkeiten behaftet.

Mit dieser Methode kann man, wie bereits erwähnt, nur das Übergangsmodell erlernen, weswegen auch die Zuordnung zu dem passiven Zweig korrekt ist.

Die Erwartungsfunktion wird sozusagen an die Nachfolgerzustände angepasst. Auf diese Weise wird die Erwartungsfunktion besser in ihre Umgebung eingepasst, was einer besseren Modellbildung zugute kommt.

Temporal difference learning

Hier versucht man das Beste der beiden obigen Methoden zu nutzen. Das induktive Verhalten von direct utility estimation, und das Wissen, welches im adaptive dynamic programming eingesetzt wird. Leider erbt dieses Verfahren auch die langsame Konvergenz von direct utility estimation; dafür ist es allerdings einfacher zu berechnen als adaptive dynamic programming.

Temporal difference learning gehört wegen der Kombination dieser beiden Methoden zu den modellfreien Lernalgorithmen. Diese werden wir in 5.4.2 auf Seite 18 näher besprechen.

Ähnlich wie bei adaptive dynamic programming werden die Utility-Funktionen an die lokale Gegebenheit angepasst. Allerdings wird im Unterschied zu adaptive dynamic programming nicht die Anpassung aller Nachfolger gesucht, sondern nur die direkten Nachfolger, was sich in einem deutlich reduzierten Rechenaufwand widerspiegelt.

5.4.2 Aktiv

Im Passiv-Zweig der Reinforcement Technik zielten alle Anstrengungen auf eine Modellbildung ab. Im Gegensatz dazu beschäftigt sich der Aktiv-Zweig verstärkt mit den Aktionen, welche zum Ziel führen, wie auch immer dieses aussehen mag. Die Modellbildung steht eher im Hintergrund.

Durch geeignete Modifikationen lassen sich einige passive Reinforcement Algorithmen in aktive umbauen. Einen einfachen Start bietet uns das Konzept von adaptive dynamic programming. Zunächst muss unser Agent seine Umgebung kennen lernen und ein Modell dazu erstellen. Anschließend muss seine Wahlmöglichkeit zwischen unterschiedlichen Aktivitäten beachtet werden.

Die Integration von *Ein-Schritt-Voraussicht* (im englischen: „one-step lookahead“) ist hier sehr praktikabel. Eine Formel, die diesen Ansatz realisiert, lautet:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')U(s') \tag{5.15}$$

$U(s)$	Utitity-Funktion zum Zustand s
s'	Folgezustand von s
$R(s)$	Kurzfirstiger Reward in s
γ	Lernkurven-Konstante
$\max_a$	Das Maximum über alle Aktionen a
$T(s, a, s')$	Übergangsmodell

Diese Formel (auch BELLMAN-Gleichung gennant) beschreibt sozusagen die Anweisung: Finde den maximalen bzw. optimalen Weg mit Aktionen aus allen möglichen Kombinationen von Zustand-Folgezustand-Wegen heraus.

Man kann sich leicht vorstellen, dass diese Anweisung selbst bei einem recht kleinen Spielfeld mit nur wenigen möglichen Aktionen einen größeren Rechenaufwand bedeutet. Aus diesem Grund versucht man gewisse Abkürzungen zu nehmen. Zum Beispiel an einem Punkt aufzuhören, wenn es über längere Zeit keine Verbesserung mehr gegeben hat. So ein Vorgehen nennt man dann *greedy* oder gierig.

Exploration $\neq$ exploitation

Greedy-Algorithmen konvergieren nur selten zum optimalen Weg. Sie bieten aber in den meisten Fällen eine recht gute Lösung.

Selbst wenn man in dieser Umgebung mit der Formel 5.15 keinen greedy-Algorithmus verwenden würde, käme kein optimales Ergebnis heraus, sondern in den meisten Fällen ein unbefriedigendes. Dieses Modell kann die Wirklichkeit eben nicht hundertprozentig wiedergeben.

„The agent *does not* learn the utilities or the true optimal policy!“[1]

Hier fällt dann eine Unterscheidung zwischen exploration und exploitation auf. Exploration meint ein Maximum auf lange Sicht und exploitation meint ein Maximum der Rewards.

Mit einem Beispiel aus dem realen Leben kann man den Unterschied vielleicht besser erklären. Ein reiner Exploiter-Typ würde immer den maximalen Gewinn anstreben aber niemals Forschung betreiben. Ein reiner Explorer-Typ würde die ganze Zeit forschen aber niemals das erworbene Wissen umsetzen.

Um einen optimalen Mittelweg zu finden könnte ein Agent das **GLIE**-Schema implementieren. Das GLIE-Schema bedeutet, dass man in jedem Zustand jede nur denkbare Aktion in unbegrenzter Häufigkeit ausführt. Mittels dieser Technik wäre es für einen Agenten sogar möglich, ein optimales Modell seiner Umgebung zu erstellen.

$$U^+(s) \leftarrow R(s) + \gamma f\left(\sum_{s'} T(s,a,s')U^+(s'), N(a,s)\right) \tag{5.16}$$

mit

$$f(u,n) = \begin{cases} R^+ & \text{falls } n < N_e \\ u & \text{anderenfalls} \end{cases} \tag{5.17}$$

5.16 stellt die Anpassung von 5.15 an das GLIE-Schema dar; es berücksichtigt nun auch eine „optimistisches Erwartung“. 5.17 nennt man die Explorationsfunktion. Mit einer geeigneten Explorationsfunktion kann man sogar die Nachteile von greedy-Algorithmen mildern.

Modellfreies lernen

Einige Verfahren benötigen kein Modell zum Lernen; dies besitzt einige Eleganz. Eine direkte Gegenüberstellung von Lernen mit einem Modell und einer Utility-Funktion einerseits und Lernen eines Aktion-Wert-Paares ohne Modell andererseits wirft eine interessante Frage auf, die immer wieder in der Wissenschaftsgemeinde aufkommt: „Was ist der beste Weg, einen Agenten zu beschreiben?“

Ein häufig auf dem Gebiet der künstlichen Intelligenz gemachter Ansatz ist der des wissensbasierten Systems. Dieser Ansatz würde implizieren, einige Aspekte der Umgebung in das System abzubilden und von da aus dann weiter Entscheidungen zu treffen. Eine zutreffende Implementierung dieses Ansatzes wäre der eines modellbehafteten Lernens.

Ein anderer Ansatz, der auch von einigen Wissenschaftlern außerhalb des Forschungsbereiches KI häufiger geäußert wird, ist der, dass der wissensbasierte Ansatz überflüssig ist, da das modellfreie Konzept die Intuition nachbildet, welche den Menschen sowohl in Wissenschaft als auch in der Kunst voranbringt.

Eine Methode auf dem Gebiet des modellfreien Lernens nennt sich Q-learning. Die Besonderheit an Q-learning ist, dass es direkt die $Aktion \times Wert$-Relation lernt und keine Erwartungsfunktion. Dies befreit das Q-learning von einem Modell und läßt es zu den modellfreien Lernmethoden zählen.

Es besteht eine direkte Beziehung zwischen Utility-Funktionswert und Q-Wert:

$$U(s) = \max_a Q(a, s)$$

- a Aktion im
- s Zustand

Auch wenn diese beiden Funktionen so eng miteinander verknüpft erscheinen, so trennt sie doch modellbehaftetes Lernen vom modellfreiem Lernen.

5.5 Zusammenfassung

Abschließend sollen die einzelnen Konzepte mit ihren Vor- und Nachteilen noch einmal einander gegenübergestellt werden.

Entscheidungsbäume zeichnen sich durch verhältnismäßig schnelle Konvergenz der Lernverfahren sowie die einfache Nachvollziehbarkeit der Entscheidungen aus, sind aber in ihrer Mächtigkeit Neuronalen Netzen unterlegen.

Neuronale Netze hingegen können komplexe Sachverhalte darstellen und sind dabei sehr flexibel, einzelne Entscheidungsprozesse können aber nur schwer nachvollzogen werden.

Die große Stärke von Reinforcement Learning ist die Miteinbeziehung von Rückmeldungen aus dem manipulierten System. Dies lässt sich hier einfacher realisieren als in Neuronalen Netzen, die einzelnen Entscheidungen sind jedoch ähnlich schwer nachzuvollziehen.

Zum Schluss muss betont werden, dass keines der Konzepte universell einsetzbar ist. Je nach Problemstellung kann ein Ansatz seine Stärken besser ausspielen als ein anderer und für jede Aufgabe muss individuell abgewogen werden, welcher am besten geeignet ist.

Quellenverweis

Alle Bilder und Formeln wurden aus dem Buch[1] entnommen. Die Formel 5.2 wurde der besseren Lesbarkeit halber angegepasst.

Literaturverzeichnis

[1] Russel S., Norvig P.: *Artificial Intelligence – A Modern Approach*, Second Edition, Prentice Hall, 2003.

[2] Matt Ginsberg: *Essentials of articial intelligence*, Morgan Kaufmann Publishers 1993.

[3] G. Görz, C.-R. Rollinger, J. Schneeberger: *Handbuch der künstlichen Intelligenz*, 3. Auflage, Oldenburg 2000.