7. Künstliche Intelligenz

7.1. Turing-Test

Bevor wir uns über *künstliche* Intelligenz unterhalten können, müssen wir zunächst klären, was das Wort *Intelligenz* überhaupt bedeutet. Es gibt viele verschiedene Ansätze Intelligenz zu definieren. Zum Beispiel

- Die Fähigkeit Wissen und Fertigkeiten zu erwerben und anzuwenden.
- Die Fähigkeit Probleme zu lösen.
- Die Fähigkeit zur Abstraktion.

Diese Definitionen sind aber alle nicht sehr präzise und ihre Auslegung ändert sich mit der Zeit. Intuitiv würden wir heute sagen, dass Computer nicht intelligent sind, weil die Aufgaben, die sie lösen können alle mechanisch lösbar sind (per Definition!). Doch bevor es Computer gab, galt zum Beispiel das Lösen von Rechenaufgaben als intelligente Tätigkeit. Seitdem Computer das um viele Größenordnungen besser können als Menschen. Ebenso galt das Brettspiel Schach als etwas, das nur intelligente, strategisch denkende Spieler spielen können. Doch auch hier sind Computer mittlerweile stärker als Menschen: 1996 hat zum ersten Mal ein Schachweltmeister ein Spiel gegen einen Computer verloren (Deep Blue gegen Kasparov), zehn Jahre später gewinnen Computer überzeugend gegen Weltmeister (Deep Fritz gegen Kramnik). 2009 erreichte Pocket Fritz 4 Großmeisterniveau – obwohl es auf einem Mobiltelefon läuft.

In der Praxis ist es oft so, dass *Intelligenz* gerade das ist, was Menschen haben, aber Tiere oder Maschinen nicht.

Um die Definition von Intelligenz präzise zu machen, hat Alan Turing bereits 1950 eine Möglichkeit vorgeschlagen, um festzustellen, ob eine Maschine intelligent ist oder nicht. Beim sogenannten *Turing-Test* führt ein Prüfer ein Gespräch via Text (also wie in einem Chat) und muss feststellen, ob sein Gesprächspartner ein Mensch oder eine Maschine ist. Gelingt es dem Prüfer nicht Menschen von Maschinen zu unterscheiden, so müssen die Maschinen intelligent sein. Diese Definition von Intelligenz ist intuitiv sinnvoll. Schließlich stellen wir auch von unseren Mitmenschen fest, ob sie intelligent sind, oder nicht, indem wir mit ihnen interagieren. Wir können nicht in der Kopf anderer hineinschauen, darum muss ausschließlich das beobachtbare Verhalten ausschlaggebend sein.

In seiner Arbeit diskutiert Turing die ob es theoretisch möglich ist, dass eine Maschine den oben beschriebenen Test besteht. Er gibt ein paar mögliche Gegenargumente an.

1. Der theologische Einwand. Die Behauptung ist, dass Intelligenz nicht allein dem Gehirn, bzw. dem Körper entspringt, sondern es einer „metaphysischen Zutat“ bedarf.

2. Der Bewusstseinseinwand. Selbst wenn es möglich wäre, Maschinen zu konstruieren, die den Turing-Test bestehen, so wären sie nicht intelligent, weil sie kein Bewusstsein haben.

   Der Turing-Test kann im Sinne der Philosophie des Geistes als *funktionalistisch* interpretiert werden, weil wir uns das Verhalten eines Agenten ansehen, um etwas über seine Geisteszustände zu erfahren und behaupten, dass identisches Verhalten auf identische Geisteszustände schließen lässt. Der Bewusstseinseinwand geht von einem *dualistischen* Weltbild aus: Geisteszustände werden nicht mit funktionalen Zuständen gleichgesetzt, sondern sie sind Zustände einen immateriellen Substanz.

   Von Philosophen wurden viele Bände mit Diskussionen zu diesem Thema gefüllt, die wir hier nicht wiedergeben können. Zwei überzeugende Gegenargumente zum Dualismus sind wie folgt. Nimmt man an, dass es tatsächlich ein Ding gibt, das nicht durch die Gesetze der Physik gefasst wird und für die Aktionen eines Geistes verantwortlich ist, ist unklar wie die Interaktion zwischen physischer und metaphysischer Materie von statten gehen soll. Zieht man sich darauf zurück, dass es lediglich emergente Eigenschaften des Geistes sind, die sich nicht auf seine physikalischen Eigenschaften zurückführen lassen, so bleibt die Frage, warum Gehirne allein dazu prädestiniert sind solche Eigenschaften zu entwickeln.

3. Der „schwere Probleme“ Einwand. Wir haben schon gelernt, dass es Aufgaben gibt, die ein Computer nicht, oder nur sehr schwer, lösen kann. Zum Beispiel das Halteproblem, oder NP-vollständige Probleme. Die Behauptung ist, dass Menschen diese Probleme lösen können, und folglich Computer Menschen nie imitieren können werden.

   Das wesentliche Gegenargument zu diesem Einwand ist, zum einen, dass es völlig unklar ist, ob Menschen schwere Probleme lösen können oder nicht. Zwar können Menschen für viele Turingmaschinen durch scharfes hinsehen feststellen, ob sie halten oder nicht, aber das Halteproblem ist eine Aussage über *alle* Turingmaschinen. Bringt man Fähigkeiten wie zum Beispiel „Kreativität“ an, die Menschen offensichtlich besitzen, aber Computer nicht, bleibt die Frage, ob es sich hier um eine prinzipielle Unmöglichkeit handelt, oder wir lediglich noch nicht wissen, wie wir Computern diese Fähigkeiten beibringen können. Außerdem sind wir hier wieder im Bereich schwammiger Definitionen. Sind zum Beispiel Programme wie Randomusic, die Musik komponieren, kreativ? Die Kompositionen sind auf jeden Fall besser, als alles, was Adrian produzieren könnte.

4. Der Hardwareeinwand. Die Behauptung ist, dass Gehirne so fundamental anders funktionieren als Computer, dass es keine Regeln gibt, die ein Computer befolgen kann, die das gleiche Verhalten erzeugen wie ein Gehirn.

   Dieser Einwand ist ähnlich zum Bewusstseinseinwand. Wenn wir nicht glauben, dass es einer magischen Zutat bedarf um Intelligenz hervorzurufen, zu ist nicht klar, warum Maschinen nicht Gehirne simulieren können. Gehirne sind materielle Gegenstände, die den Regeln der Physik folgen. Diese Regeln kann man, mit genügend Mühe, von einem Computer simulieren lassen.

Turing gibt auch ein paar überzeugende Argumente an, warum es *im Prinzip* für eine Maschine möglich sein muss, den Turing-Test zu bestehen. Am stärksten ist meiner Meinung nach das folgende. Weil wir im Gespräch mit einem fremden sehr schnell feststellen, ob wir mit einem intelligenten Wesen reden oder nicht, reicht es eine (sehr große!) Tabelle mit allen sinnvollen Gesprächen, die nicht länger als z.B. 15 Minuten dauern, zu haben. Dann kann eine Maschine einfach in der Tabelle nachschlagen, was eine geeignete Antwort auf die Aussage des Gesprächspartners ist.

Die Versuche Computer so intelligent zu machen, dass sie den Turing-Test bestehen, haben bislang leider nur sehr unbefriedigende Ergebnisse geliefert. Schon das einfachere Problem der automatischen Übersetzung von Sprachen wehrt sich seit den 1950ern standhaft gegen überzeugende Lösungen. Es stellt sich als sehr schwierig heraus, die Bedeutung von Sätzen zu verstehen, wenn man nicht über eine große Menge an „gesundem Menschenverstand“ verfügt, um Mehrdeutigkeiten sinnvoll aufzulösen. Aktuelle Ansätze zu diesem Problem verbinden daher große Wissensdatenbanken mit ausgefeilten statistischen Methoden. Das Übersetzungssystem von Google, Google Translate, liefert mittlerweile Übersetzungen zwischen vielen Sprachen, die zumindest die groben Ideen der Texte erhalten.

Wegen der ausbleibenden Erfolge hat sich die Forschung in der künstlichen Intelligenz auf Spezialfälle zurückgezogen. Wichtige Aufgaben, die früher nur von Menschen gelöst werden konnten, werden heutzutage routinemäßig von Computern erledigt. Früher brauchte es einen erfahrenen Verkäufer, um zu wissen, welche Produkte man einem Kunden empfehlen muss, heute übernehmen Algorithmen diese Aufgabe in den großen Internetkaufhäusern. Autos können von Computern sicher im Straßenverkehr gesteuert werden, es ist nur noch eine Frage des Kostensenkung, bevor sie kommerziell verfügbar sind.

Im Rest dieses Kapitels werden wir uns mit zwei wichtigen Aspekten von intelligentem Verhalten in diesem auf Spezialfälle eingeschränkten Bild beschäftigen. Zum einen diskutieren wir, wie Computer aus Beispielen lernen und Objekte aus verschiedenen Kategorien voneinander unterscheiden können. Anschließend werden wir an Hand von Brettspielen wie Schach sehen, wie Computer sich planen und sich strategisch verhalten können.

7.2. Lernen

In diesem Abschnitt besprechen wir wie man an Hand von Beispielen lernt, Objekte zu erkennen und verschiedene Dinge voneinander zu unterscheiden. Diese sogenannte *Mustererkennung* hat zahllose Anwendungen. Am vertrautesten ist uns vielleicht das Filtern von unerwünschter Werbung aus unserem E-Mail Postfach. Das E-Mailprogramm lernt durch unsere Bewertungen, welche E-Mails gelöscht werden können und welche nicht. Von großer kommerzieller Bedeutung ist das Lernen von Nutzerpräferenzen. Konzerne wie Amazon lernen aus unseren Einkäufen und den Produkten, die wir uns ansehen wofür wir uns interessieren und kann so Produkte vorschlagen, die wir wahrscheinlich auch kaufen möchten.

Das Verstehen von gesprochener Sprache, das essentiell für künstlich intelligente Assistenten wie zum Beispiel Apples Siri ist, basiert auf ähnlichen Prinzipien. Man möchte gegeben eine Folge von Tönen entscheiden welchem Wort sie am ähnlichsten sind.

In diesem Kapitel werden wir sehen, dass bereits sehr einfache Methoden, die wir kaum als „Lernen“ erkennen würden, es Computern ermöglichen mit großer Genauigkeit verschiedene Aufgaben, bei denen Objekte erkannt werden müssen, zu lösen.

7.2.1. Spamerkennung

Wir werden uns zunächst auf den einfachen Fall beschränken, in dem wir Objekte einer von zwei Kategorien zuordnen möchten. Zum Beispiel möchten wir E-Mails nach Unerwünscht (*Spam*) und Erwünscht (*Ham*) unterscheiden. Dabei gibt uns der Nutzer der Mailsoftware die Beispiele aus denen wir lernen sollen, in dem er eingehende Mails als Spam oder Ham markiert.

Um Spam von Ham zu unterscheiden, müssen wir natürlich Daten sammeln. Welche Daten man genau misst kann einen großen Einfluss auf den Erfolg unserer Klassifikation haben. Man möchte Merkmale messen, die sich bei den beiden Kategorien möglichst unterscheiden. Wenn man mehrere Merkmale betrachtet sollten sie auch möglichst unabhängig voneinander sein. Wenn man zum Beispiel Äpfel von Bananen unterscheiden möchte, könnte man das Gewicht messen. Misst man gleichzeitig auch noch das Volumen erhält man aber nur wenig zusätzliche Informationen, weil Volumen und Gewicht voneinander abhängig sind. Außerdem müssen die Merkmale natürlich auch noch einfach zu messen sein, damit man ein praktikables Verfahren bekommt.

Am einfachsten zu messen ist die Anzahl an Spam-Mails. Hat der Nutzer zum Beispiel 1000 E-Mails empfangen und davon 600 Spam bzw. 400 als Ham markiert, können wir die Wahrscheinlichkeit für Spam empirisch abschätzen. Wir sagen die *a-priori Wahrscheinlichkeit* einer E-Mail Spam zu sein ist $p_S = 0.6$ und dementsprechend $p_H = 0.4$ für Ham.

Wenn wir nur diese Information haben, welche Antwort sollten wir geben, wenn uns jemand fragt, ob es sich bei einer E-Mail um Spam oder um Ham handelt? Mit Wahrscheinlichkeit 0.6 „Spam“ zu sagen und mit Wahrscheinlichkeit 0.4 „Ham“ klingt verlockend, ist aber nicht die beste Strategie. Stattdessen müssen wir *immer* sagen es sei Spam. Weil Spam-Mails häufiger sind als Ham, machen wir den kleinsten Fehler bei der Klassifikation, wenn wir immer „Spam“ antworten.[1] Wenn wir immer Spam sagen, liegen wir in $p_S = 60\%$ der Fälle richtig. Sagen wir mit p_S Spam und mit p_H Ham, liegen wir in

$$\begin{aligned} P(\text{Richtig}) &= P(\text{Spam}) \cdot P(\text{Wir sagen Spam}) + P(\text{Ham}) \cdot P(\text{Wir sagen Ham}) \\ &= p_S^2 + p_H^2 \\ &= 52\% \end{aligned}$$

der Fälle richtig.

Die korrekte Art zu Klassifizieren ist also immer Kategorie zu wählen, die wir für am wahrscheinlichsten halten. Wenn wir keine Daten außer der Häufigkeit haben, insbesondere also unsere Informationen von der konkreten E-Mail unabhängig sind, antworten wir immer gleich. Jetzt können wir uns aber natürlich die Mail genauer ankucken und ein paar Merkmale messen. Zum Beispiel können wir überprüfen, ob sich der Absender in unserem Adressbuch befindet oder nicht.

Nehmen wir an, der Absender ist im Adressbuch. Was haben wir gelernt? Wir schauen uns wieder alle 1000 E-Mails an und zählen in wievielen der Absender im Adressbuch ist. Wir stellen fest, dass 300 Mails von bekannten Absendern kommen, davon sind 100 Spam-Mails und 200 Ham-Mails. Bevor wir den Absendern angesehen haben, glaubten wir mit $p_S = 0.6$ daran, dass die E-Mail Spam ist. Jetzt müssen wir die neuen Informationen verwenden um unseren Glauben zu aktualisieren.

Durch Nachzählen der Mails wissen wir die Wahrscheinlichkeit, dass eine Spam-Mail einen bekannten Absender hat, wir schreiben dafür $P(\text{Bekannt}|\text{Spam}) = 1/6$. Wir möchten $P(\text{Spam}|\text{Bekannt})$ ausrechnen, also die Wahrscheinlichkeit, dass eine Mail Spam ist, wenn sie einen Bekannten Absender hat. Dazu kucken wir uns alle Mails mit bekanntem Absender an und zählen wieviele Spam sind. 100 von 300 Mails mit bekanntem Absender sind Spam, also ist $P(\text{Spam}|\text{Bekannt}) = 1/3$. Diese Art Rechnung nennt man die Formel von Bayes:

$$P(\text{Spam}|\text{Bekannt}) = \frac{P(\text{Bekannt}|\text{Spam}) \cdot P(\text{Spam})}{P(\text{Bekannt})}.$$

Wir haben also festgestellt, dass die Wahrscheinlichkeit, dass diese spezielle E-Mail Spam ist nur $1/3$ ist, weil sie einen bekannten Absender hat. Daher klassifizieren wir sie

[1] Das geht natürlich davon aus, dass beide Arten von Fehlern, sowohl eine Spam-Mail als Ham zu klassifizieren als auch eine Ham-Mail als Spam zu klassifizieren, gleich schlimm sind. In der Praxis ist es natürlich viel schlimmer, wenn wir fälschlicherweise eine Ham-Mail löschen und wir sollten unsere Antworten dahingehend anpassen, dass wir einen gewichteten Fehler minimieren.

Abbildung 7.1.: Thomas Bayes (1701-1761), Entdecker von Bayes Formel.

	Bekannt	Kredit	Gesamt
Spam	100	90	600
Ham	200	10	400
Gesamt	300	100	1000

als Ham. Man könnte den Wert $1/3$ auch durch einfaches nachzählen, ohne zu Rechnen, bestimmen. Wir wollen wissen wie groß die Wahrscheinlichkeit ist, unter den Mails mit bekanntem Absender eine Spam-Mail zu finden. Es gibt 300 solche Mails und nur 100 sind Spam, also ist die Wahrscheinlichkeit $1/3$. Man möchte aber eigentlich das Nachzählen vermeiden, weil das zu lange dauert. Für jede eingehende Mail das gesamte Postfach anzusehen ist in der Praxis nicht möglich.

Wenn wir noch weitere Merkmale messen, wird die Rechnung ein bisschen komplizierter. Testen wir zum Beispiel noch, ob die Mail ein bestimmtes Wort, wie zum Beispiel „Kredit“ enthält. Wieder müssen wir in den vom Nutzer klassifizierten Mails nachzählen. 100 Mails enthalten das Wort „Kredit“, davon sind 90 Spam und 10 Ham.

Wir nehmend vereinfachend an, dass *Kommt von einem bekannten Absender* und *Enthält das Wort Kredit* unabhängig voneinander sind, wir also nicht extra nachzählen müssen wieviele Mails das Wort „Kredit“ enthalten und von einem bekannten Absender kommen, sondern sofort sagen, dass ein Zehntel der Mails von bekannten Absenders das Wort „Kredit“ enthalten (weil ein Zehntel aller Mails das Wort enthalten). Wir rechnen

jetzt mit Bayes Formel:

$$\begin{aligned} P(\text{Spam}|\text{Bekannt \& Kredit}) &= P(\text{Spam}) \cdot \frac{P(\text{Bekannt \& Kredit}|\text{Spam})}{P(\text{Kredit \& Bekannt})} \\ &= P(\text{Spam}) \cdot \frac{\cdot P(\text{Bekannt}|\text{Spam}) \cdot P(\text{Kredit}|\text{Spam})}{P(\text{Bekannt}) \cdot P(\text{Kredit})} \\ &= \frac{600}{1000} \cdot \frac{\frac{100}{600} \cdot \frac{90}{600}}{\frac{300}{1000} \cdot \frac{100}{1000}} \\ &= \frac{1}{2} \end{aligned}$$

Der interessiere Leser kann verifizieren, dass genau das gleiche rauskommt, wenn wir zuerst $P(\text{Spam}|\text{Bekannt})$ ausrechnen und anschließend unsern neuen Glauben an Spam oder Ham mit der Kredit-Information aktualisieren.

Tatsächliche Spamfilter bestimmen eine große Zahl von solchen Merkmalen um die Wahrscheinlichkeit auszurechnen, dass eine Mail Spam ist. Der Lernschritt ist also das Bestimmen der Wahrscheinlichkeiten $P(M|\text{Spam})$ für jedes Merkmal M an Hand der bereits klassifizierten Mails. Der Nachdenkschritt ist dann eine Anwendung von Bayes Formel.

7.2.2. Nächste Nachbarn

Bei der Spamerkennung, haben wir nur Merkmale betrachtet, die zwei mögliche Werte haben. Entweder der Absender ist im Adressbuch, oder nicht; entweder die Mail enthält Wort X, oder nicht. In vielen Bereichen, in denen wir Werte messen, die beliebige Werte annehmen können. Zum Beispiel messen wir das Gewicht von verschiedenen Fischarten um zu entscheiden ob wir sie auf das Förderband der Fischstäbchenmaschine oder in den Abfall legen möchten. Um zu verstehen, wie man aus solchen Messwerten lernen kann, werden wir zunächst das Spamproblem von einem anderen Blickwinkel aus betrachten.

Wenn wir eine E-Mail klassifizieren wollen, testen wir eine Reihe von Eigenschaften, zum Beispiel: 1) Ist der Absender im Adressbuch? 2) Enthält die E-Mail das Wort „Kredit“?. Weil die Antworten auf diese Fragen entweder „Ja“ oder „Nein“ sind, können wir sie mit 1 und 0 kodieren. Für eine E-Mail M können wir etwa die Ergebnisse $(0, 1)$ erhalten, d.h. der Absender ist nicht im Adressbuch und das Wort „Kredit“ kommt in M vor.

So eine Folge von Zahlen nennt man einen *Vektor*. Man kann Vektoren als Punkte im Raum verstehen. Ein Vektor mit zwei Einträgen entspricht einem Punkt in der Ebene, mit drei Einträgen entspricht er einem Punkt im dreidimensionalen Raum und mit hundert Einträgen einem Punkt im 100D-Raum.[2]

[2] Dass man sich mehr als drei Dimensionen nicht vorstellen kann, hat Mathematiker natürlich nicht davon abgehalten über die Eigenschaften von Räumen mit vielen (manchmal sogar unendlich vielen!)

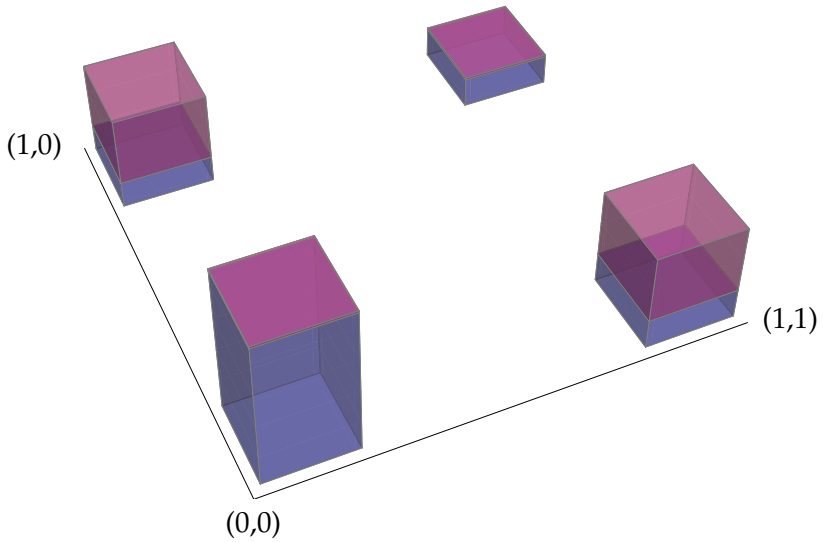


Abbildung 7.2.: Blaue Balken stehen für Spam, rote für Ham.

Die gleichen Tests haben wir auf allen bereits klassifizierten E-Mails auch ausgeführt. Wir haben also eine Menge von 1000 Punkten in der Ebene und für jeden Punkt wissen wir, ob es sich um eine Spam- oder eine Ham-Mail handelt. Im vorherigen Abschnitt haben wir ausgerechnet, wie wahrscheinlich es ist, dass eine Mail Spam ist, indem wir zunächst die Mittelwerte für jede Koordinate der Spam-Mails und Ham-Mail ausgerechnet und dann Bayes Formel angewendet haben. Bayes Formel funktioniert intuitiv so, dass eine Mail, die in einem Punkt liegt, in dem viele Spam-Mails liegen wahrscheinlicher Spam als Ham ist.

Im wesentlichen Rechnen wir mit Bayes Formel für einen Punkt (x, y) die Höhe der blauen und roten Balken in Abbildung 7.2 aus. Bayes Formel teilt noch durch die Höhe, weil eine mit Wahrscheinlichkeiten gerechnet wird und die Zahlen zwischen 0 und 1 sein müssen. Wenn der blaue Balken höher ist, sagen wir „Spam“, ist der rote höher, sagen wir „Ham“.

Wenn wir jetzt nicht nur 0 und 1 in unseren Messwertvektor schreiben können, sondern beliebige Zahlen, ändert sich das Bild. Wir haben weiterhin für jedes klassifizierte Objekt einen Punkt, aber es ist sehr unwahrscheinlich, dass wir zwei gleiche Punkte haben. Wir erhalten eine Punktwolke wie in Abbildung 7.3.

Wenn wir jetzt eine neues Objekt bekommen und somit einen neuen Punkt in unserem Bild, wie sollten wir es am besten klassifizieren? In Abbildung 7.2 haben wir nachgezählt wieviele Spam- bzw. Ham-Emails am gleichen Punkt liegen und nach der Mehrheit

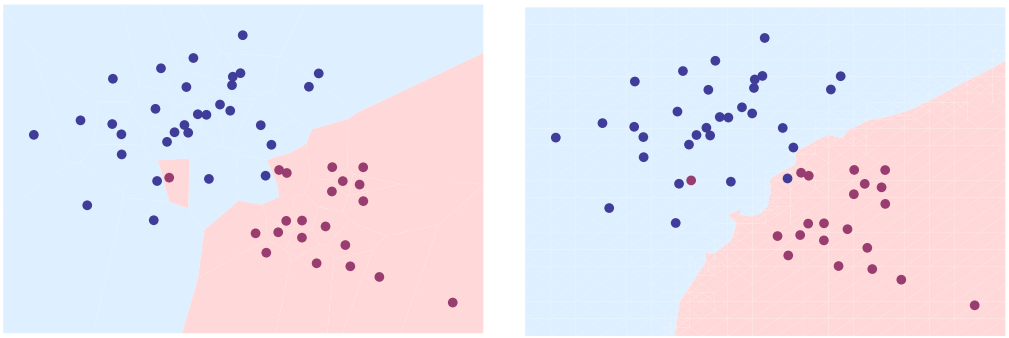
Dimensionen nachzudenken. Das hat sich als überaus praktisch nicht nur in der Informatik sondern auch in der Physik und den Ingenieurswissenschaften herausgestellt.

Abbildung 7.3.: Wenn Messwerte kontuinierliche Ergebnisse liefern, haben wir keine Balken wie in Abbildung 7.2, sondern Punktwolken.

klassifiziert. Es liegt nahe, dass wir uns jetzt die Nachbarschaft des neuen Punktes ansehen und an Hand seiner nächsten Nachbarn klassifizieren.

Wir rechnen also für einen neuen Punkt x die Abstände zu allen anderen Punkten aus. Jetzt können wir uns die k nächsten Nachbarn ansehen und einen einfachen Mehrheitsentscheid machen. Wieviele Nachbarn man sich am besten ansieht, also wie groß das k zu wählen ist, hängt vom genauen Problem ab. Zu klein und die Klasifikation ist nicht robust gegen Ausreißer–eine ungewöhnliche Spam-Mail sorgt dafür, dass ein paar Ham-Mails fälschlicherweise als Spam klassifiziert werden. Zu groß und die weiter entfernten Nachbarn gehören schon zur falschen Klasse. Abbildung 7.4 zeigt die Klassifikation aller Punkte der Ebene an Hand der Trainingsmenge aus Abbildung 7.3.

Dieses nächste Nachbarn Verfahren (man nennt es auch k-NN für k *nearest neighbors*), liefert recht gute Ergebnisse, hat aber eine Reihe von Nachteilen. Zunächst muss man für die Klassifikation eines neuen Punkten die Abstände zu allen anderen Punkten ausrechnen. Wenn man viele Punkte hat, und für jeden Punkt viele Messwerte, nimmt das viel Rechenzeit in Anspruch. Wichtiger ist jedoch, dass dieses Verfahren keine Abstraktion von den Trainingsdaten macht. Es lernt praktisch alle Daten auswendig und vergleicht dann nur. Wenn man etwas über die Struktur der Daten lernen möchte, zum Beispiel dass es unterschiedliche Gruppen innerhalb einer Klasse gibt, kann man dieses Verfahren dafür nicht benutzen. Im nächsten Abschnitt werden wir uns an Hand eines Beispiels ein paar weitere einfache Verfahren ansehen, die diese Nachteile teilweise beheben.



(a) 1-NN (b) 3-NN

Abbildung 7.4.: Klassifikation der ganzen Ebene an Hand der Punkte aus Abbildung 7.3. Es wird in 7.4(a) nur der nächste Nachbarn betrachtet, in 7.4(b) werden die drei nächsten Nachbarn betrachtet.

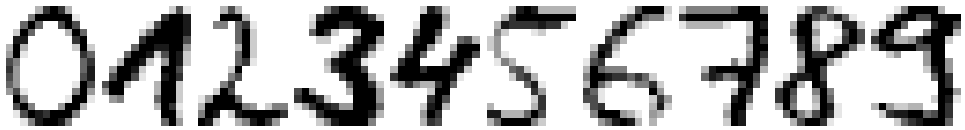


Abbildung 7.5.: Ein paar Beispiele für Ziffern aus der Datenbank handgeschriebener Ziffern.

7.3. Handgeschriebene Ziffern Erkennen

In diesem Abschnitt benutzen wir die k-NN Klassifizierung und verwandte Verfahren um handgeschriebene Ziffern zu erkennen. Wir haben eine Datenbank mit Ziffern, die auf 12×16 Pixel normiert wurden. Abbildung 7.5 zeigt ein paar der Ziffern. Die Datenbank ist in zwei Teile aufgeteilt. Eine *Trainingsmenge* von 199 Ziffern und eine *Testmenge* von 993 Ziffern. Wir trainieren unsere Erkennungsverfahren mit der Trainingsmenge und testen sie mit der Testmenge. Diese Trennung ist nötig, da sonst zum Beispiel das k-NN Verfahren perfekte Ergebnisse liefern würde–es lernt ja die gesamte Trainingsmenge auswendig.

Um die Ziffern in Punkte im Raum umzuwandeln, interpretieren wir jedes der $12 \cdot 16 = 192$ Pixel als eine Zahl zwischen 0 und 1, 0 bedeutet Weiß, 1 Schwarz, und Grauwerte werden durch Zahlen dazwischen ausgedrückt. Jede Ziffer entspricht also einem Punkt im 192-dimensionalen Raum. Wenn man nicht nur die statischen Bilder der Ziffern zur Verfügung hat, sondern auch noch Daten darüber hat, wie sich der Stift beim Zeichnen bewegt hat, kann man sich natürlich noch weitere Merkmale überlegen, die bei der Klassifizierung hilfreich sein könnten. Zum Beispiel kann man die Winkelsumme der Stiftbewegungen bestimmen, oder die Geschwindigkeit. Die korrekte Auswahl der Merkmale, die man zum Klassifizieren verwenden möchte hat einen großen Einfluss auf

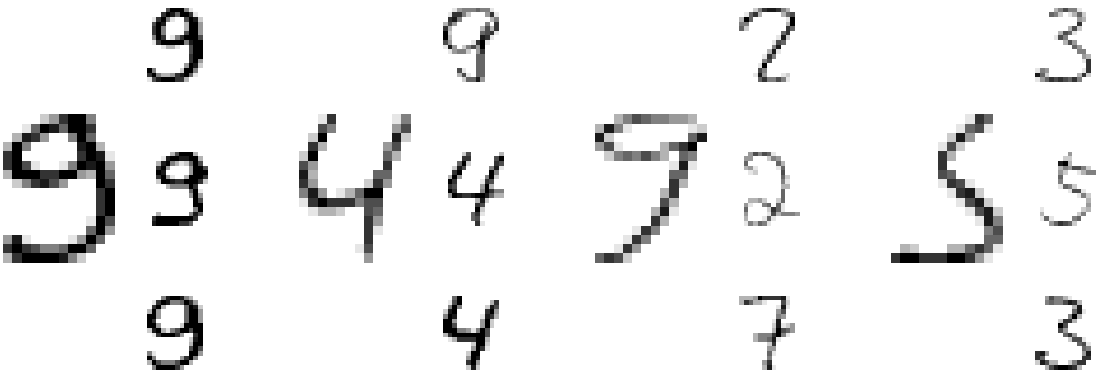


Abbildung 7.6.: Ziffern und ihre drei nächsten Nachbarn. Der nächste Nachbar ist oben. Manche der Ziffern sind auch für mich schwer zu erkennen. Die am weitesten rechts stehende Ziffer ist laut Datenbank eine 5. Meiner Meinung nach ist die 3-NN Klassifikation als 3 aber durchaus gerechtfertig.

den Erfolg und erfordert eine Menge Erfahrung.

Auch wie man die Distanz im Raum definiert spielt eine Rolle. Am einfachsten ist es, die euklidische Distanz zwischen den Punkten zu nehmen, also die Länge der Verbindungslinie. Eine Alternative ist die Winkeldistanz. Statt die Länge der Verbindungslinie zu messen, messen wir den Winkel zwischen den beiden Geraden, die die Punkte mit dem Ursprung verbinden. Wenn zwei Ziffern an den gleichen Pixeln dunkel sind, können sie eine große euklidische Distanz voneinander haben, wenn eine dunkler ist als die andere, der Schreiber also fester aufgedrückt hat. Die Winkeldistanz ist dann aber weiterhin gering. Für unsere Zwecke reicht es allerdings aus nur die einfache euklidische Distanz zu berechnen.

Für diese Klassifikationsaufgabe funktioniert bereits das einfache nächste Nachbarn Verfahren extrem gut. Wenn man nur den nächsten Nachbarn betrachtet, erreicht man schon mehr als 93% korrekte Ergebnisse. Betrachtet man die drei nächsten Nachbarn verbessert sich das auf fast 95%. Abbildung 7.6 zeigt ein paar Ziffern und ihre nächsten Nachbarn.

Wie aber schon im letzten Abschnitt gesagt, braucht die Klassifizierung mit diesen Verfahren recht viel Rechenzeit. Was kann man machen um das zu umgehen?

Eine einfache Idee ist es, die Anzahl der Punkte zu denen man Entfernungen ausrechnen muss zu verringern, indem man sich repräsentative Beispiele aussucht. Wenn man wie wir eine von Menschen mit den richtigen Klassen annotierte Trainingsmenge hat, kann man zum Beispiel für jede Zifferngruppe den Mittelpunkt ausrechnen. Dann benutzt man diese zehn „Durchschnittsziffern“ zur 1-NN Klassifikation. Statt wie vorher die Abstände zu fast zweihundert Trainingsziffern ausrechnen zu müssen, werden dann nur noch die Abstände zu zehn Ziffern ausgerechnet. Abbildung 7.7 zeigt die Durchschnittsziffern.

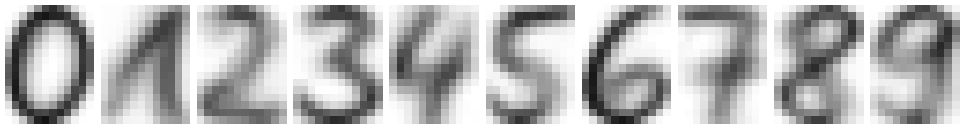


Abbildung 7.7.: Die Zentren der verschiedenen Ziffernklassen.

Weil man eine ganze Menge Informationen verliert, ist natürlich auch die Erkennungsrate viel geringer. Lediglich 85% der Testmenge werden an Hand der Zentren korrekt klassifiziert. Besonders schlecht funktioniert diese Methode, wenn die Punkte einer Klasse im Raum nicht eine ungefähr kugelförmige Wolke bilden. Wenn sie stattdessen lang gezogen sind, oder gar aus mehreren voneinander recht unterschiedlichen Untergruppen bestehen, ist das Zentrum wenig repräsentativ. Außerdem erfordert diese Methode eine bereits von Hand klassifizierte Trainingsmenge. Eine solche zu erzeugen ist natürlich recht aufwändig.

Ein weiteres Verfahren, das im Prinzip auf der gleichen Idee basiert, kommt ohne eine vorklassifizierte Trainingsmenge aus. Bei diesem *k-Means* genannten Verfahren sucht ein Algorithmus selbstständig Repräsentanten für k Gruppen. Es funktioniert sehr einfach: Wir beginnen damit uns k zufällige Repräsentanten im Raum zu wählen. Das können entweder Punkte der Trainingsmenge sein, oder komplett ausgewürfelte. Jetzt klassifizieren wird die gesamte Trainingsmenge an Hand dieser Repräsenten. Dadurch bekommt jeder Repräsentant eine Menge von Punkten, die zum ihm näher sind als zu den anderen Repräsentanten. Wir verschieben jeden Repräsentant in die Mitte der ihm zugewiesenen Gruppe. Wir wiederholen diesen Gruppieren-Verschieben Schritt solange bis sich die Repräsentanten nicht mehr bewegen.

Diese Repräsentanten sind natürlich noch nicht klassifiziert, weil die Trainingsdaten nicht vorklassifiziert sind. Aber weil es sich nur um k Stück handelt, können wir einen Menschen bitten ihnen Klassen zuzuweisen. Anschließend können wir Testdaten an Hand der so ausgewählten Repräsentanten klassifizieren.

Im Idealfall ist jeder Gruppe in den Trainingsdaten ein Repräsentant zugeordnet. Natürlich kann es aber passieren, dass zwei oder mehr Repräsentanten in die selbe Gruppe fallen und dafür manche Gruppen gar keinen Repräsentanten abbekommen. Daher ist es in der Regel eine gute Idee k größer zu wählen als die Zahl der Gruppen, die man in seinen Daten erwartet. Abbildung 7.8 zeigt die Repräsentanten die für 17-Means gefunden werden (weil der Algorithmus randomisiert ist, kommen natürlich immer andere Repräsentanten raus). Die Erkennungsrate liegt zwischen siebzig und achtzig Prozent, je nachdem wie viel Glück man bei der Berechnung der Repräsentanten hat.

Neben der Anwendung zur Klassifizierung, kann man den k-Means Algorithmus dazu benutzen um versteckte Strukturen in seinen Daten zu finden. Zum Beispiel kann man sehen, dass es verschiedene Arten gibt, die 1 zu schreiben, man gerader, mal dachförmiger. Ähnliches tritt bei der 9 auf. Manche Menschen machen einen geraden Strich, andere einen einen Haken.

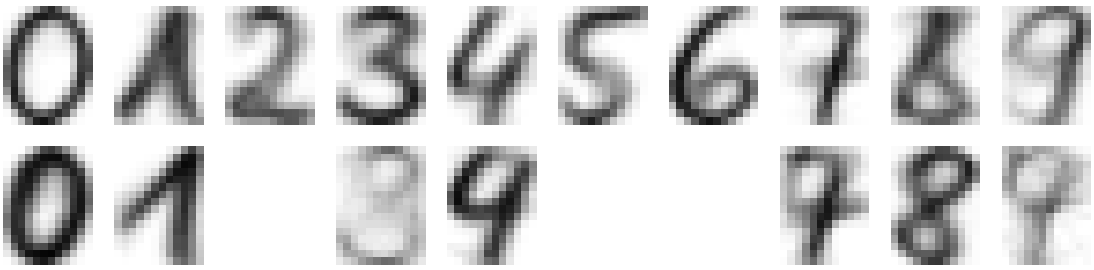


Abbildung 7.8.: Ein Beispiel für Zentren, die mit k-means automatisch gefunden werden. Man bemerke, dass es bei der 9 zwei Untergruppen gibt–die mit einem geraden Strich und die mit einem Hakenstrich. k-Means findet diese beiden Untergruppen. Ähnliches bei der 1.

Spannender ist das natürlich bei anderen Datensätzen als handgeschriebenen Ziffern. Man könnte zum Beispiel verschiedene chemische Marker an den Zellmembranen von Krebszellen messen und dann lernen, dass es mehrere Untergruppen dieser Krebsart gibt, die verschieden gut auf unterschiedliche Medikamente ansprechen.

7.4. Spiele

In diesem Abschnitt werden wir an Hand von Brettspielen Methoden kennen lernen, mit denen Computer Strategien entwickeln können. Wir schränken uns auf deterministische (also ohne Würfel funktionierende) Brettspiele mit zwei Spielern ein, bei denen beide Spieler einen Überblick über die gesamte Situation haben. Das ist ein sehr eingeschränktes Modell, beinhaltet aber bekannte Spiele wie Dame, Schach, und Go. Computer sind auch in etwas schwierigeren Disziplinen stark. In der Computerspieleindustrie gibt es zum Beispiel einen großen Bedarf an künstlich intelligenten Gegnern, die in nicht rundenbasierten Echtzeit-Spielen mit eingeschränkter Information Menschen ins Schwitzen bringen können. Diese Art von Spielen erfordert ganz andere Methoden, als die, die wir hier beschreiben.

7.5. Spielbäume

Wenn Menschen solche Brettspiele spielen, probieren sie mental ein paar Züge aus, und überlegen sich, welcher Zug zur besten Position führt. Computer machen das sehr ähnlich. Sie sind nur viel schlechter im Bewerten, dafür aber viel besser im Ausprobieren. Betrachten wir wie das für ein sehr einfaches Spiel wie Tic-Tac-Toe aussehen kann. Abbildung 7.9 zeigt einen Teil des *Spielbaums* für Tic-Tac-Toe. Wir beginnen mit dem leeren Spielfeld in der Wurzel. In der ersten Ebene zeichnen wir alle Züge des ersten

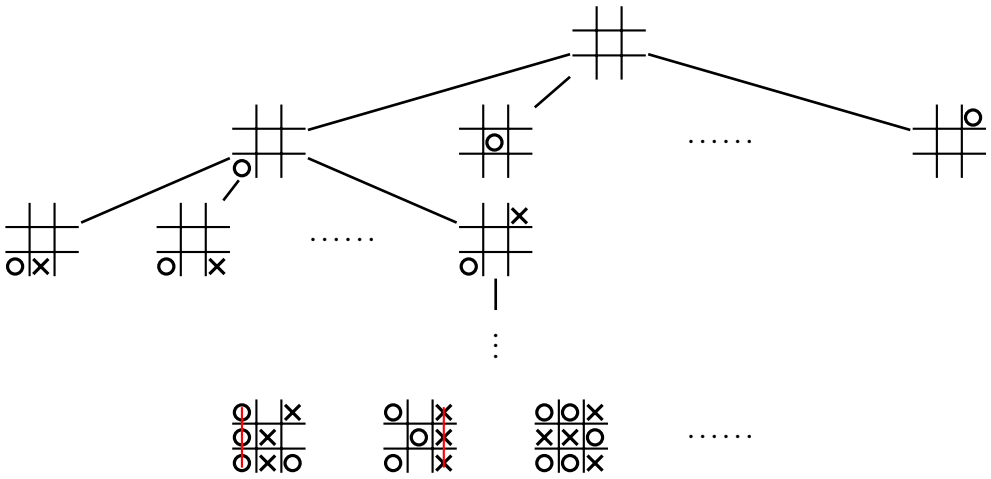


Abbildung 7.9.: Ein kleiner Ausschnitt aus dem Baum der möglichen Spielverläufe für Tic-Tac-Toe.

Spielers ein, in der zweiten Ebene zeichnen wir für jeden Zug des ersten Spielers alle möglichen Erwiderungen des zweiten Spielerns ein.

Einen solchen Spielbaum können wir leicht für ein beliebiges Brettspiel (mit den oben genannten Einschränkungen) aufstellen. Für sehr einfache Spiele wie Tic-Tac-Toe bleibt der Baum auch recht klein. Jedes Feld in Tic-Tac-Toe ist entweder leer, enthält ein ○ oder ein ×. Da das Spielfeld neun Felder hat, gibt es höchstens 3^9 Knoten im Baum, tatsächlich sind es viel weniger, weil viele Spielfelder nicht regelgerecht erzeugt werden können.

Wie helfen uns Spielbäume dabei zu entscheiden welchen Zug wir machen sollten? Woran sieht man zum Beispiel dem Zug von ○, der in der Mitte des Spielfeldes ist, dass er stärker ist als einer der anderen möglichen Züge?

Betrachten wir, wie uns ein Spielbaum am Ende des Spiels hilft. Nehmen wir an es geht um den letzten Zug und Spieler ○ ist an der Reihe. Der Spieler kann für jeden möglichen Zug bestimmen, ob er damit gewinnt, verliert, oder es zu einem Unentschieden kommt. Es ist klar, dass er den Zug wählen muss, der zu dem bestmöglichen Ergebnis führt. Gehen wir einen Zug zurück.

Um vorletzten Zug probiert der Spieler × jeden seiner möglichen Züge aus. Mental geht er dadurch eine Ebene nach unten um Spielbaum, zum letzten Zug. Für diesen Zug haben wir uns aber gerade überlegt wie ○ handeln muss. Der × Spieler kann also für jeden seiner Züge vorausbestimmen wie ○ handelt, wenn er annimmt, dass ○ keine Fehler macht. Das lässt sich so fortsetzen.

Schreiben wir für *Spieler ○ gewinnt* eine -1, für *Spieler × gewinnt* eine 1 und eine 0 für ein Unentschieden, so möchte ○ das Spielergebnis minimieren, × möchte es maximieren. In der untersten Ebene des Baumes stehen die fertigen Spiele, für die wir direkt sehen ob

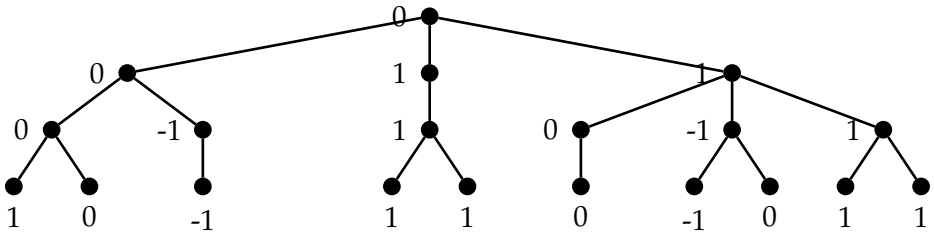


Abbildung 7.10.: Ein vollständig beschrifteter Spielbaum. Spieler 1 minimiert, Spieler 2 maximiert. Die Knoten der untersten Ebene sind beendete Spiele für die wir die Bewertung an Hand der Regeln des Spiels nachschlagen können. In der Ebene darüber ist jeder Knoten mit dem Minimum seiner Kinder beschriftet, die Ebene davor benutzt das Maximum usw. bis zur Wurzel, wo wieder das Minimum verwendet wird.

sie -1, 0, oder 1 sind. In der Ebene davor können wir die Knoten mit -1, 0, oder 1 beschriften, indem wir das Minimum, bzw. das Maximum der Ebene darunter ausrechnen. Abbildung 7.10 zeigt ein Beispiel, in dem der der minimierende Spieler zuerst am Zug ist. Algorithmus 10 zeigt den Pseudocode für diesen Algorithmus.

Algorithmus 10 : Minimax

Daten : Ein Spielbaum mit n Ebenen.
Ergebnis : Für jeden Knoten v im Spielbaum eine Bewertung w_v.
für jedes *Blatt* v *auf Ebene* n **tue**
  w_v = Bewertung des Endzustands
für jedes i *in* $n-1, n-2, \ldots, 1$ **tue**
  für alle *Knoten* v *auf Ebene* i **tue**
    wenn i *gerade ist* **dann**
      w_v = Minimum der w_u für alle Kinder u von v
    sonst
      w_v = Maximum der w_u für alle Kinder u von v

Wenn man sich für jeden Knoten noch die Kante merkt, von der aus er seinen Wert bekommen hat, findet man die *dominante Strategie* für den Spielbaum. Dieser Spielverlauf wird gewählt wenn beide Spieler optimal spielen.

Der Algorithmus, den wir verwendet haben, um den Baum zu beschriften heißt *Minimax* und stellt die Basis für viele Algorithmen dar, die bei Brettspielen verwendet werden können. Da wir aber zur Ausführung des gesamten Spielbaum durchsuchen müssen (beginnend bei den Blättern, ebenenweise), ist das nur praktikabel wenn das Spiel sehr einfach ist. Tic-Tac-Toe mit seinen paar tausend möglichen Spielfeldern passt leicht in den Speicher eines Computers und kann daher mit Minimax gelöst werden. Wie dem Leser sicher schon aus eigener Erfahrung bekannt ist, kommt dabei heraus, dass man immer unentschieden spielt, wenn beide Spieler optimal spielen.

Für die meisten Spiele passt der Spielbaum nicht mehr in den Speicher eines Computers. Schach zum Beispiel hat etwa 10^{50} mögliche Spielzustände, weit mehr als man realistisch ansehen könnte. An statt sich den ganzen Spielbaum anzusehen, und die Bewertungen des Spielzustände von den Endstellungen aus nach oben zu propagieren, sieht man nur eine Hand voll Züge in die Zukunft.

Jetzt muss man mit Hilfe der Spielzustände, die man so sieht, entscheiden, welchen Zug man macht. Dazu schaut man sich die Stellungen in der Zukunft scharf an und schätzt, ob es sich um günstige Stellungen für den ersten oder den zweiten Spieler handelt. Nach dem Gefühl, das man so entwickelt, gibt man den Stellungen dann Bewertungen. Diese Bewertungen kann man dann wieder mit dem Minimax-Algorithmus nach oben propagieren und so entschieden, wie man selbst zieht. Diese geschätzte Bewertung eines Spielzustands werden wir im Folgenden eine *Heuristik* nennen.

Es hängt natürlich von der Güte der Heuristik ab, wie stark man mit diesem Verfahren spielt. Je weiter man in die Zukunft schaut, desto eher sieht man schlechte Entwicklungen: im Schach zum Beispiel einen Verlust einer Dame. Wenn man also nur schwache Heuristiken hat, muss man sich viele Spielzustände ansehen, mit starken Heuristiken weniger. Menschen haben sehr gute Heuristiken, sie spielen zum Beispiel sehr gutes Schach, obwohl sie sich in einer Runde nur sehr wenige Züge mental vorstellen können. Dafür können sie einer Position aber ansehen, ob sie strategisch günstig oder ungünstig ist. Leider ist es schwierig herauszufinden, wie genau Menschen Positionen bewerten, darum ist es schwierig die gleichen Verfahren auch im Computer anzuwenden.

Menschen sehen sich beim Spielen aber nicht alle möglichen Züge an. Viele Züge werden als offensichtlich wertlos sofort ausgeschlossen und man konzentriert sein Denken auf die Züge, die interessant scheinen. Ein Verfahren, das Computern auf ähnliche Weise erlaubt einen Teil der möglichen Züge nicht zu betrachten (und stattdessen weiter in die Zukunft zu sehen um stärker zu spielen), nennt man α, β Suche.

Die Grundidee der α, β Suche ist identisch zu Minimax. Wir wollen wieder Bewertungen von weiter unten im Spielbaum liegenden Knoten nach oben propagieren, um uns zu entscheiden, welcher Zug für uns der beste ist. Um Züge wegzulassen, können wir natürlich nicht mehr Ebenenweise vorgehen. Wenn wir uns die unterste Ebene im Spielbaum angesehen haben, haben wir ja bereits einen großen Teil der Arbeit hinter uns.

Stattdessen werden wir die Bewertungen pfadweise nach oben propagieren. Wir rechnen, rekursiv, zuerst die Bewertung für das am weitesten links stehende Kind eines Knotens aus. Dann für das nächste Kind usw. bis alle Kinder bewertet sind. Dann nehmen wir das Minimum, bzw. das Maximum um die Bewertung für diesen Knoten zu finden. Der Trick der α, β Suche besteht nun darin, dass man manchmal mit dem Bewerten der Kinder aufhören kann, noch bevor man alle Kinder betrachtet hat.

Um zu sehen, wann das der Fall ist, muss man im Kopf behalten, dass man sich in einem hypothetischen Spielzustand befindet. Es hängt vom Verhalten des anderen Spielers ab, ob man in im tatsächlichen Spiel überhaupt erreichen kann. Wenn man feststellt, dass

Funktion AlphaBeta(v, α, β, spieler)

wenn *Maximale Tiefe oder ein Endzustand erreicht ist* **dann**
  gib Heuristischen Wert von v **zurück**
wenn *spieler=Min* **dann**
  $w_v = \beta$
  für jedes *Kind* u *von* v **tue**
    $w_u =$ AlphaBeta(u, α, w_v, Max)
    $w_v = \min(w_u, w_v)$
    wenn $w_v \leq \alpha$ **dann** Brich die Suche ab.
  gib w_v **zurück**
sonst
  $w_v = \alpha$
  für jedes *Kind* u *von* v **tue**
    $w_u =$ AlphaBeta(u, α, w_v, Min)
    $w_v = \max(w_u, w_v)$
    wenn $w_v \geq \beta$ **dann** Brich die Suche ab.
  gib w_v **zurück**

der andere Spieler einen nie bis hierhin gelangen lässt, braucht man sich auch nicht die Mühe zu machen, eine genaue Bewertung für diesen Zustand zu finden.

Wann lässt der andere Spieler einen gar nicht bis zu diesem Zustand kommen? Wenn er einen anderen Zug machen kann, bei dem er sicher besser abschneidet. Um das zu erkennen merken wir uns in jedem Knoten nicht nur seine Bewertung, sondern zwei Werte α und β. Der Wert α gibt den besten (kleinsten) Wert an, auf den der Min-Spieler noch hoffen hat, β den größtmöglichen Wert, den der Max-Spieler noch erreichen kann. Die Werte α und β sagen also aus, was der Gegner schon sicher hat. Wenn wir in einem Max-Knoten feststellen, dass die Bewertung größer ist als β, wissen wir, dass der Min-Spieler nicht diesen Pfad wählen wird und wir können die Berechnung hier abbrechnen. Stellen wir analog in einem Min-Knoten fest, dass die Bewertung kleiner als α ist, wird uns der Max-Spieler nicht bis hierhin kommen lassen.

Es bleibt zu klären wie sich die α und β Werte propagieren. Am Anfang sind $\alpha = -\infty$ und $\beta = +\infty$, weil wir noch nichts wissen. Wenn wir noch keine Bewertug in einem Knoten haben, übergeben wir das α und β, das wir von unserem Vater bekommen haben.

Wenn wir in einem Min-Knoten v mit vorläufiger Bewertung w und den Schranken α_v, β_v zu einem Kind u gehen, setzen wir $\alpha_u = \alpha_v$ und $\beta_u = w$. Denn stellt der Max-Spieler in seinem Knoten fest, dass etwas besseres als w möglich ist, dann nehmen wir diesen Pfad ohnehin nicht. Analog setzen wir von einem Max-Knoten ausgehend das α unserer Kinder auf unsere derzeitige Bewertung. In Min-Knoten machen wir also das β für unsere Kinder kleiner, in Max-Knoten machen wir das α für unsere Kinder größer.

Abbildung 7.11 zeigt ein Beispiel.

Wieviel die α β Suche vom Spielbaum abschneiden kann und nicht betrachten muss, hängt natürlich von der Reihenfolge ab, in der die Knoten betrachtet werden. Je früher man die starken Züge sieht, desto mehr schwache Züge kann man als irrelevant ignorieren. In der Praxis betreibt man deswegen beträchtlichen Aufwand, um die Züge heuristisch so zu sortieren, dass starke Züge vorne in der Reihenfolge stehen.

Diese Methoden funktionieren sehr gut für Spiele wie Schach. Man kennt gute Heuristiken und in jedem Zug ist die Anzahl der Möglichkeiten relativ klein. Für kompliziertere Spiele wie Go ist dieses Verfahren aber sehr schwach. Go wird auf großen Brettern gespielt und in jedem Zug kann man einen Stein an viele mögliche Positionen setzen. Dadurch ist der Spielbaum sehr, sehr groß und man kann ihn nur wenige Schritte weit durchsuchen. Außerdem ist es schwierig Heuristiken für Go zu finden. Die Go-Meister haben nur ein sehr intuitives Verständnis davon, was eine gute Stellung von einer schlechten unterscheidet, das macht es natürlich schwierig aus ihrem Wissen Algorithmen abzuleiten.

Außerdem ist es schwierig Verfahren zu finden, die für verschiedene Spiele gut funktionieren. Da die Heuristiken eine so große Rolle spielen und spielspezifisch sind, weiß man nur wenig darüber, wie man davon loskommen kann.In der Hoffnung, dass daraus Einsichten über generelle Intelligenz entstehen könnten, wird deswegen in der KI Forschung das sogenannte „General Game Playing“ erforscht. Hierbei bekommt das Programm eine Beschreibung der Regeln eines Spiels, damit es weiß, was legale Züge sind, und wer in einer Endstellung gewonnen hat, und muss dann möglichst gut spielen.

Hier kann man natürlich keine vom Programmierer vorgegebenen Heuristiken benutzen, weil ja die Beschreibung des Spiels bei der Erstellung des Programms noch gar nicht zur Verfügung stand. Derzeit verwendet man aber auch in diesem Gebiet Suchen im Spielbaum und braucht heuristische Bewertungen der Spielzustände. Natürlich müssen diese Heuristiken jetzt unabhängig vom Spiel funktionieren.

Eine Möglichkeit ist zufällige Suchen im Spielbaum zu machen. Wenn man einen Knoten v bewerten möchte, spielt man von v ausgehend eine große Zahl von Spielen mit zufälligen Zügen zuende. Die heuristische Bewertung von v ist dann das Verhältnis von gewonnenen zu verlorenen Spielen. Raffinierte Variationen von dieser Grundidee machen nicht vollkommen zufällige Züge. Zum Beispiel kann man aus dem Spielbaum mit der Zeit Züge lernen, die immer gut sind, wenn sie legal sind. Zum Beispiel ist es in Tic Tac Toe immer gut, wenn man das mittlere Feld für sich behaupten kann. Dann kann man seine zufällige Suche so anpassen, dass man bevorzugt solche Züge macht, die man für „immer gut“ hält.

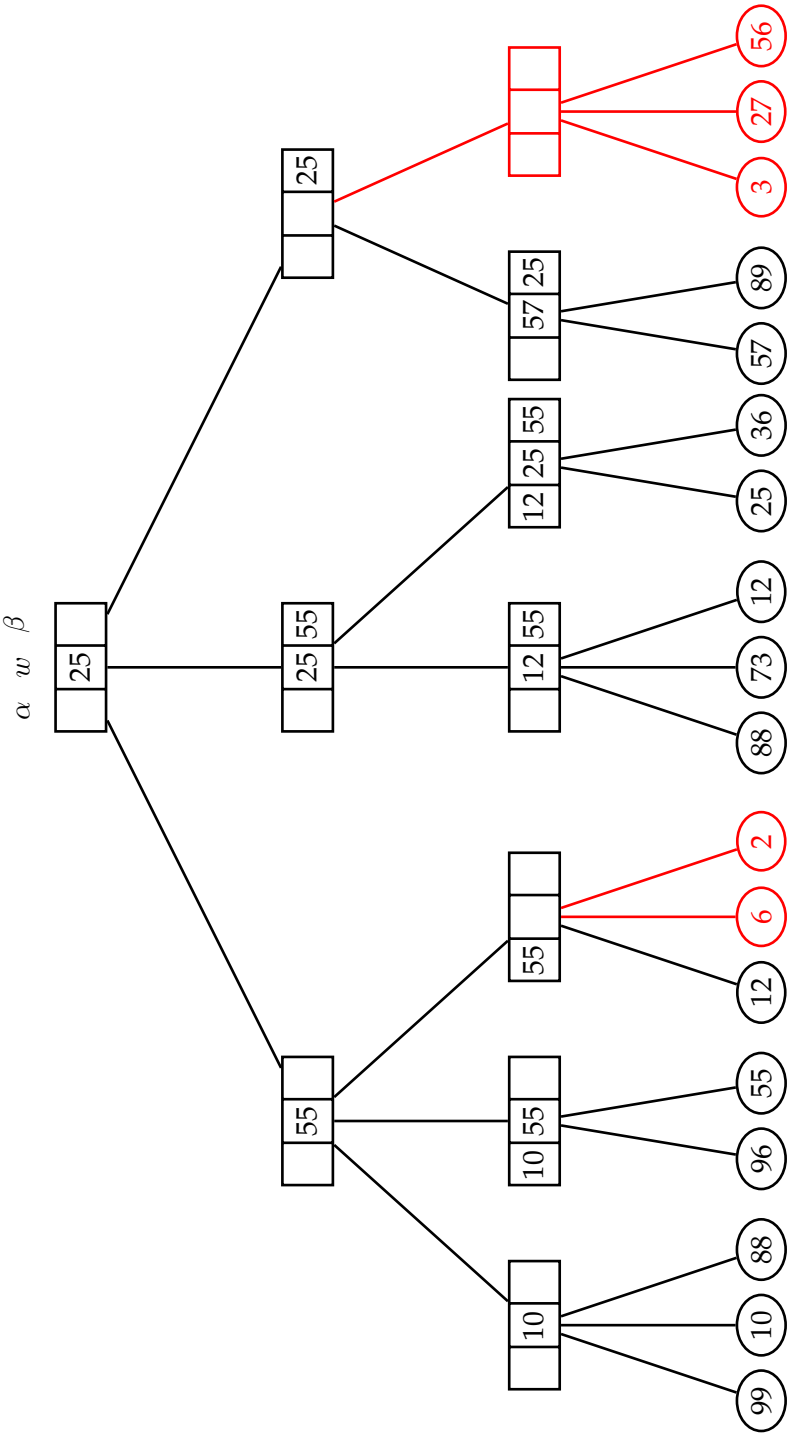


Abbildung 7.11.: Ein Beispiel für α, β Suche. Die runden Knoten sind Endzustände. In inneren Knoten notieren wir α, den Wert, und β. Wenn α oder β frei sind, sind sie $\pm\infty$. Rot gezeichnete Knoten haben wir nicht betrachtet.