V ALGORITHMIK

S. BARTELS, 18.6.2018

1. Zielsetzungen

Unter *Algorithmik* versteht man die Entwicklung und Analyse von Algorithmen, also von Computer-realisierbaren Verfahren zur systematischen Lösung einer Aufgabe. Typische Aufgaben sind das Sortieren von Listen, die Bestimmung der Lösung eines mathematischen Problems oder die Berechnung der Wahrscheinlichkeit eines Ereignisses auf Basis gegebener Daten. Bei der Analyse eines Algorithmus sind die folgenden Kriterien relevant.

(i) *Durchführbarkeit:* Es muss sicher gestellt werden, dass alle Schritte wohldefiniert sind, so dass zum Beispiel keine Divisionen durch Null auftreten.
(ii) *Terminierung:* Es muss gewährleistet sein, dass das Verfahren regulär stoppt und nicht beispielsweise in eine Endlosschleife gerät.
(iii) *Korrektheit:* Es muss nachgewiesen werden, dass für alle zulässigen Eingaben das richtige Ergebnis berechnet wird.
(iv) *Effizienz:* Es muss untersucht werden, ob der Algorithmus für realistische Problemgrößen in akzeptabler Zeit terminiert.

Einige dieser Aspekte erscheinen auf den ersten Blick trivial, ihre Relevanz zeigt sich jedoch, wenn man unterschiedliche Algorithmen betrachtet.

2. Algorithmusarten

Die am häufigsten autretenden Algorithmen sind dynamische, iterative und rekursive Algorithmen.

2.1. **Dynamische Algorithmen.** Bei *dynamischen Algorithmen* wird eine häufig numerierte Folge von Anweisungen sukzessive abgearbeitet, was meist in Form einer `for`-Schleife realisiert wird. Beispiele sind die Berechnung einer Summe und die Suche eines Elements in einer Liste.

2.2. **Iterative Algorithmen.** Bei *iterativen Algorithmen* wird eine Rechenvorschrift, die auch Fallunterscheidungen enthalten darf, wiederholt auf einen Datensatz angewendet, bis ein geeignetes Abbruchkriterium erfüllt ist. Abstrakt lässt sich dies mit einer Abbildung $T : X \to X$ beschreiben, die für einen Startwert $x^0 \in X$ die *Iterierten* $(x^k)_{k=0,1,\dots}$ durch die *Iterationsvorschrift*

$$x^{k+1} = T(x^k)$$

definiert, bis sich die Iterierten nicht mehr stark ändern, also beispielsweise bis $\|x^{k+1} - x^k\| \leq \delta$ für eine kleine Zahl $\delta > 0$ gilt. Die Termininerung und Korrektheit lässt sich in einigen Situationen mit dem Banachschen Fixpunktsatz nachweisen. Ein Beispiel eines iterativen Verfahrens ist die Berechnung der Quadratwurzel nach Heron. Man beachte, dass hier eine Aufgabe nur approximativ gelöst wird.

2.3. **Rekursive Algorithmen.** *Rekursive Algorithmen* basieren meist auf der Beobachtung, dass sich ein Problem einer bestimmten Größe auf ähnliche Probleme kleinerer Größe zurückführen und für eine gewisse minimale Problemgröße direkt lösen lässt. Ein Beispiel ist die Fakultätsfunktion $f(n) = n!$, die die *Rekursionsformel*

$$f(j) = jf(j-1)$$

für $j \geq 1$ und die *Rekursionsverankerung*

$$f(0) = 1$$

erfüllt. In diesem Fall ist die Problemgröße das Argument n und wir führen die Berechnung von $n!$ auf die Berechnung von $(n-1)!$ zurück. Rekursionen zeichnen sich dadurch aus, dass Funktionen oder Routinen sich selbst aufrufen und entstehen auf natürliche Weises bei induktiv definierten Objekten. Es gibt aber Fälle, die in keiner Form dem intuitiven menschlichen Handeln entsprechen.

Beispiel 2.1 (Türme von Hanoi)**.** *Beim Problem der Türme von Hanoi soll ein der Größe nach sortierter Stapel mit n Scheiben unterschiedlicher Größe von einer Standposition A auf eine Zielposition B unter Verwendung einer Hilfsposition C versetzt werden. Dabei dürfen nur einzelne Scheiben versetzt werden und sie müssen stets der Größe nach geordnet sein. Im Fall von drei Scheiben ist das Problem noch intuitiv lösbar, aber schon bei vier Scheiben müssen sehr viele Bewegungen durchgeführt werden. Angenommen, wir wissen, wie man einen Stapel mit $(n-1)$ Scheiben von einer Position auf eine andere unter Verwendung der dritten als Hilfsposition versetzen kann. Dann können wir den Stapel mit n Scheiben als Zusammensetzung eines Stapels mit $(n-1)$ Scheiben und der untersten, größten Scheibe betrachten und das Problem in drei Schritten lösen:*

(1) *Versetze den oberen $(n-1)$-Teilstapel von A nach C.*
(2) *Versetze den unteren 1-Teilstapel von A nach B.*
(3) *Versetze den $(n-1)$-Stapel von C nach B.*

Das Vorgehen ist in Abbildung 1 illustriert. In diesen Schritten ist nur die Versetzung kleinerer Stapel erforderlich. Während die Versetzung eines Stapels mit einer Scheibe trivial ist, lässt sich die Versetzung von Stapeln mit $(n-1)$ Scheiben wieder jeweils auf zwei Versetzungen von Stapeln mit $(n-2)$ Scheiben und eine eines Stapels mit einer Scheibe zurückführen.

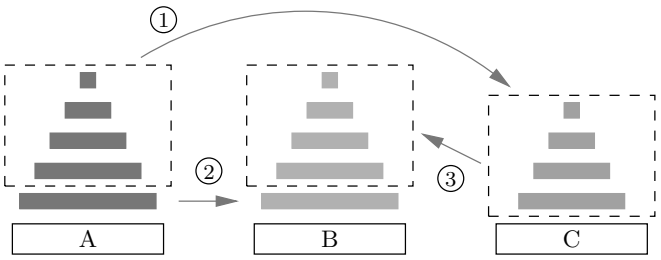

ABBILDUNG 1. Lösung des Problems der Türme von Hanoi durch Reduktion auf kleinere Probleme derselben Art.

Im Beispiel wird eine sehr komplexe Aufgabenstellung mit einer einfachen Rekursionsvorschrift gelöst. Etwas problematisch ist dabei, dass der Aufwand in jedem Reduktionsschritt um einen Faktor 2 vergrößert wird, was zu mehr als 2^n Operationen führt. Vor diesem Hintergrund ist es häufig sinnvoll, eine Rekursion zu vermeiden und durch einen dynamischen, vorwärtsgerichteten Algorithmus zu ersetzen, was im Fall der Fakultät und der Fibonacci-Folge nicht jedoch im Fall der Türme von Hanoi einfach möglich ist.

3. KOMPLEXITÄT

Wir wollen die Zeit, die ein Algorithmus zur Lösung eines Problems benötigt, unabhängig von der Art und Weise seiner praktischen Umsetzung quantifizieren. Dazu untersuchen wir, wie sich der Aufwand, das heißt die Anzahl der benötigten Operationen wie arithmetischen Operationen oder Vergleichen, verändert, wenn wir die Problemgröße verändern.

Definition 3.1. *(i) Die* (Problem-) Größe *einer Aufgabe ist eine charakteristische Größe des Problems wie die Anzahl der zu bearbeitenden Daten, die Anzahl von Summanden oder der Index eines zu berechnenden Folgeglieds.*
(ii) Der Aufwand *eines Algorithmus zur Lösung einer Aufgabe der Größe n ist die Anzahl $\mathcal{A}(n)$ der erforderlichen arithmetischen Operationen und Vergleiche.*

Wir sind meist an einer oberen Schranke für den maximalen Aufwand interessiert, das heißt dem Aufwand für ein *Worst-Case-Scenario*.

Beispiele 3.2. *(i) Das Suchen eines Elements in einer Liste mit n Elementen erfordert maximal n Vergleiche, das heißt $\mathcal{A}(n) \leq n$.*
(ii) Sind k arithmetische Operationen zur Bestimmung der Summanden s_j in der Summe $S_n = s_1 + s_2 + ... + s_n$ erforderlich, so erhalten wir den Gesamtaufwand $\mathcal{A}(n) \leq k(n+1)$. Im Fall der Gauß-Summe mit $s_j = j$ lässt sich der Aufwand mit der Formel $S_n = n(n+1)/2$ auf drei Operationen reduzieren, unabhängig von n.
(iii) Die rekursive Berechnung des Folgeglieds $f_n = f_{n-1} + f_{n-2}$ mit Initialisierung $f_0 = 0$ und $f_1 = 1$ der Fibonacci-Folge erfordert den Aufwand $\mathcal{A}(n) = 2^n$. Eine dynamische Realisierung hingegen nur $\mathcal{A}(n) \leq n$.

Um die Praktikabilität eines Algorithmus einordnen zu können, führen wir gewisse Aufwandsklassen ein, die prägnant angeben, wie sich der Aufwand vergrößert, wenn die Problemgröße beispielsweise verdoppelt wird.

Definition 3.3. *(i) Der Aufwand $\mathcal{A}(n)$ ist polynomiell, wenn Zahlen $p, c \geq 0$ existieren, sodass für alle $n \geq 1$ gilt*

$$\mathcal{A}(n) \leq cn^p.$$

Im Fall $p = 0, 1, 2, 3$ nennen wir den Aufwand konstant*,* linear*,* quadratisch *beziehungsweise* kubisch*.*
(ii) Der Aufwand $\mathcal{A}(n)$ ist exponentiell*, wenn Zahlen $s, c > 0$ existieren, sodass für all $n \geq 1$ gilt*

$$\mathcal{A}(n) \geq cs^n.$$

Mit diesen Begriffen lassen sich die erforderlichen Berechnungen von Aufgaben übersichtlich einordnen.

Beispiele 3.4. *(i) Das dynamische Suchen eines Elements in einer Liste erfordert linearen Aufwand.*
(ii) Die Bestimmung der Gauß-Summe erfordert linearen Aufwand bei dynamischer Realisierung und konstanten Aufwand bei Verwendung der Summenformel.
(iii) Die Berechnung eines Glieds der Fibonacci-Folge besitzt exponentiellen Aufwand bei rekursiver Realisierung und linearen Aufwand bei dynamischer Realisierung.

In der Regel erfordert die Lösung einer Aufgabe der Größe n mindestens n Operationen, da meist jede der n Informationen in die Berechnungen eingeht. Ein Algorithmus mit linearem Aufwand ist daher meist optimal und die Aufgabe kann *effizient* für eine große Spanne von Problemgrößen sinnvoll vom Computer gelöst werden. Bei exponentiellem Aufwand beispielsweise mit $\mathcal{A}(n) = 2^n$ stößt man sehr schnell an Grenzen des Realisierbaren, denn schon für ein Problemgrößen $n \geq 70$ ergeben sich

$$\mathcal{A}(n) \geq 10^{20}$$

Operationen, was selbst auf Giga-Hertz-Rechnern, die 10^{10} Operationen pro Sekunde durchführen können, Schwierigkeiten verursacht, denn selbst beim Einsatz von mehreren hundert Rechnern würde sich eine Rechenzeit von mehr als einem Jahr ergeben. Auch kubischer Aufwand führt schnell zu Laufzeitproblemen. In vielen Anwendungen ist ist quadratischer Aufwand noch vertretbar, besser ist jedoch superlinearer Aufwand, der zwischen linearem und quadratischem Aufwand liegt.

Bemerkungen 3.5. *(i) Neben der Wahl eines Algorithmus beeinflusst auch die konkrete Implementierung die Laufzeit eines Programms. Beispielsweise ist der Zugriff auf die Einträge einer Liste abhängig von deren technischer Realisierung.*

(ii) Probleme, die auf Algorithmen mit exponentiellem Aufwand führen, lassen sich meist der Klasse der NP-vollständigen Probleme *zuordnen. Dazu gehört beispielsweise das Problem des Handelsreisenden, der eine kürzeste Rundreise durch n Städte sucht. Für diese Klasse von Problemen sind bisher keine Lösungsalgorithmen mit polynomiellem Aufwand bekannnt, es ist jedoch auch noch nicht gelungen, deren Nichtexistenz formal nachzuweisen.*

4. Sortierverfahren

Das Sortieren von Listen von Datensätzen ist eine zentrale Aufgabe der Datenverarbeitung. Um die wichtigsten Ansätze darzustellen, betrachten wir eine Liste $I = [a_1, a_2, \ldots, a_n]$ mit n ganzen Zahlen, die aufsteigend sortiert werden soll.

4.1. **Bubblesort.** Das *Bubblesort*-Verfahren ähnelt dem Sortieren von Büchern in einem Regal, die der Reihe nach geprüft und gegebenenfalls nach vorne versetzt werden.

Algorithmus 4.1 (Bubblesort)**.** *Sei die Liste $I = [a_1, \ldots, a_n]$ gegeben und setze $i = 2$.*

(1) *Gilt $a_i \geq a_{i-1}$, so gehe zu Schritt (3).*
(2) *Gilt $a_i < a_{i-1}$, so vertausche wiederholt die Elemente a_{i-k} und a_{i-k-1} für $k = 0, 1, \ldots,$ bis $a_{i-k} \geq a_{i-k-1}$ oder $k = i - 1$ gilt.*
(3) *Sofern $i < n$ gilt, erhöhe $i \to i + 1$ und fahre fort mit Schritt (1); stoppe andernfalls.*

Im i-ten Schritt des dynamischen Algorithmus fallen ein Vergleich sowie bis zu $i - 2$ weitere an, sofern Einträge vertauscht werden müssen. Im schlechtesten Fall erhalten wir also

$$\mathcal{A}(n) \leq \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \leq \frac{1}{2}n^2,$$

das heißt, dass das Verfahren quadratischen Aufwand besitzt. Dies ist für Listen mit bis zu $n = 1000$ Einträgen akzeptabel, insbesondere da das Verfahren sehr leicht zu implementieren ist.

4.2. **Mergesort.** Das *Mergesort*-Verfahren basiert auf der rekursiven Verkleinerung der zu sortierenden Liste. Nach Auffüllen der Liste mit Nullen oder sehr großen Einträgen, können wir ohne Probleme annehmen, dass $n = 2^k$ mit einer ganzen Zahl k gilt, sodass wir die Liste wiederholt halbieren können, bis wir zu einelementigen Teillisten kommen. Zur Motivation des Verfahrens nehmen wir an, dass die linke und rechte Teilliste $I^{\ell}_{n/2}$ sowie $I^{r}_{n/2}$ bereits sortiert sind. Die sortierte Liste I_n erhalten wir dann durch ein einfaches Zusammenfügen, den sogenannten *merge*- Schritt, der beiden Teillisten. Dies führt auf den folgenden rekursiven Algorithmus, der für eine Liste I eine sortierte Liste $\widetilde{I}$ zurückgibt.

Algorithmus 4.2 (Mergesort). *Aufruf:* $\widetilde{I} = \mathrm{msort}(I,k)$.
Für die Länge n der Liste I gelte $n = 2^k$ für ein $k \geq 0$.
(1) *Gilt $k = 0$, so ist die Liste bereits sortiert, das heißt setze $\widetilde{I} = I$.*
(2) *Gilt $k > 0$, so setze*

$$\widetilde{I} = \mathrm{merge}\big(\mathrm{msort}(I^\ell_{n/2}, k-1), \mathrm{msort}(I^r_{n/2}, k-1)\big).$$

Die Realisierung der Zusammenführung zweier Listen der Länge m kann mit einem dynamischen Algorithmus mit linearem Aufwand $\mathcal{A}_{\mathrm{merge}}(m) = 2m$ realisiert werden. Der Aufruf von msort mit einer Liste der Länge n führt zu einem Aufruf von merge mit zwei Listen halber Länge und zwei Aufrufen von msort mit Listen halber Länge. Insgesamt erhalten wir den Aufwand

$$\mathcal{A}(n) = n + 2\mathcal{A}(n/2).$$

Diese Argumentation wenden wir wiederholt an und erhalten

$$\begin{aligned}\mathcal{A}(n) &= n + 2\big(n/2 + 2\mathcal{A}(n/4)\big) = 2n + 4\mathcal{A}(n/4)\\ &= 2n + 4\big(n/4 + 2\mathcal{A}(n/8)\big) = 3n + 8\mathcal{A}(n/8)\\ &= \cdots = kn + 2^k\mathcal{A}(n/2^k) = kn + 2^k\mathcal{A}(1) = kn,\end{aligned}$$

wobei wir verwendet haben, dass das Sortieren einelementiger Listen keinen Aufwand erfordert, also $\mathcal{A}(1) = 0$ gilt. Da $k = \log_2(n)$ ist, folgt

$$\mathcal{A}(n) = \log_2(n)n.$$

Dies ist ein Beispiel eines superlinearen Aufwands, der sich allerdings kaum von linearem Aufwand unterscheidet, da zum Beispiel $\log_2(10^6) \leq 20$. Problematisch ist bei rekursiven Algorithmen oft eine effiziente Speicherverwaltung.

4.3. **Andere Sortierverfahren.** Real auftretende Datensätze besitzen in der Regel zusätzliche Strukturen hinsichtlich ihrer Verteilung, beispielsweise sind die Geburtsdaten einer Personengruppe meist gleichmäßig über einen gewissen Zeitraum verteilt oder die Anfangsbuchstaben von Nachnamen treten mit bekannten Häufigkeiten auf. In diesem Fall kann man den Datenbereich einfach unterteilen, die Einträge den Teilbereichen zuordnen und sich auf das Sortieren der in der Regel erheblich kleineren Teillisten beschränken. Auf solchen Beobachtungen basieren die *Bucketsort*- und *Quicksort*-Verfahren. Unter geeigneten Bedingungen an die Einträge der Liste führen sie zu sehr effizienten Verfahren können aber im schlechtesten Fall auch zu quadratischem Aufwand führen.

5. Künstliche Intelligenz

In klassischen Algorithmen werden vom Nutzer erdachte Lösungsstrategien realisiert, die meist auf einem sehr guten Verständnis der zugrundeliegenden Aufgabe basieren. Algorithmen der *künstlichen Intelligenz*, kurz KI, versuchen, auf für Menschen schwer überschaubare oder nur mit sehr hohem Aufwand lösbare Probleme, Antworten mit vertretbarem Aufwand zu erzeugen.

Wir folgen in diesem Abschnitt der Darstellung des Buchs *Algorithmen kompakt und verständlich* von M. von Rimscha (Springer, 2008) und verweisen auf dieses Buch für weitere Details.

5.1. **Maschinelles Lernen.** Beim *maschinellen Lernen* wird ein menschliches Entscheidungsverhalten unter Berücksichtigung verschiedener Kriterien beobachtet und anschließend ausgewertet. Mit der statistischen Bewertung der Kriterien lässt sich dann ein Entscheidungsbaum konstruieren, der das Entscheidungsverhalten strukturiert widergibt. Ein populäres Beispiel ist die Freizeitgestaltung einer Person, die abhängig von Wetter und Wochenende sowie Verfügbarkeit interessanter Kinofilme und Freunden eine Entscheidung zwischen den Aktivitäten Radtour, Kino, Café, Schwimmen und zu Hause bleiben trifft. Die beobachteten Daten führen auf Tabelle 1, die jedoch nur einen Auszug eines größeren Datensatzes darstellt.

Freunde	Wetter	Kinofilm	Wochenende	Entscheidung
Ja	Sonne	Ja	Ja	Radtour
Ja	Regen	Nein	Nein	zu Hause
Nein	Sonne	Nein	Nein	Schwimmen
Nein	Bedeckt	Nein	Nein	Radtour
⋮	⋮	⋮	⋮	⋮

TABELLE 1. Entscheidungsverhalten bei Berücksichtigung verschiedener Kriterien.

Abhängig von der Häufigkeit des Auftretens eines Kriteriums wird seine Relevanz quantifiziert und damit die Priorität im Entscheidungsbaum, der in Abbildung 2 dargestellt ist, festgelegt.

5.2. **Schwarmintelligenz.** Das Konzept der *Schwarmintelligenz* imitiert den in der Natur beobachteten Effekt, dass eine Gruppe einfach entscheidender Individuen in ihrer Gesamtheit zu Lösungen für komplexe Aufgabenstellungen gelangen kann. Ameisen beispielsweise hinterlassen auf dem Weg von ihrem Nest zur Nahrungsstelle Duftmarken, an denen sie sich später orientieren. Der kürzeste Weg ist höher frequentiert, sodass eine größere Konzentration der Duftstoffe auftritt, und dieser Weg sich im Laufe der Zeit als der von den Ameisen bevorzugte durchsetzt. Dieses Populationsverhalten lässt sich algorithmisch beschreiben und führt zu effizienten Lösungsansätzen für schwer handhabbare Probleme, wie dem des Handelsreisenden. Dabei werden für einen gewissen Zeitraum virtuelle Ameisen auf Reisen zwischen den Städten geschickt, die Markierungen unterschiedlicher Intensität hinterlassen. Anschließend werden Wege mit geringen Markierungen verworfen und so kann häufig eine kurze Rundreise konstruiert werden, die in der Regel jedoch nicht optimal ist. Abbildung 3 zeigt einen so konstruierten Weg.

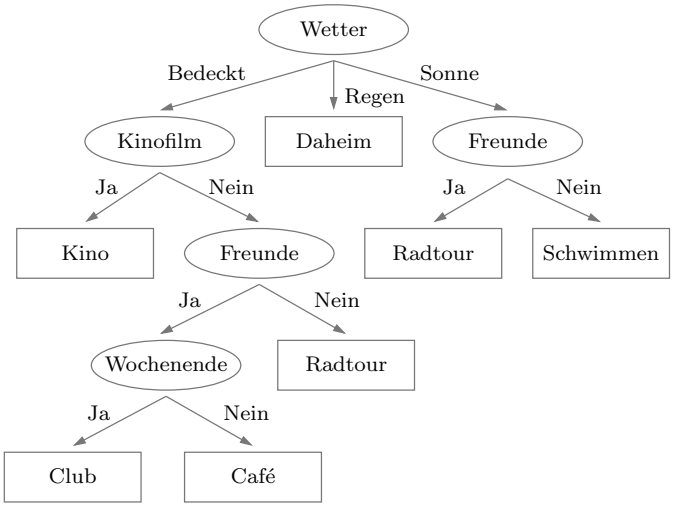

ABBILDUNG 2. Aus (einer Erweiterung von) Tabelle 1 gewonnener Entscheidungsbaum.

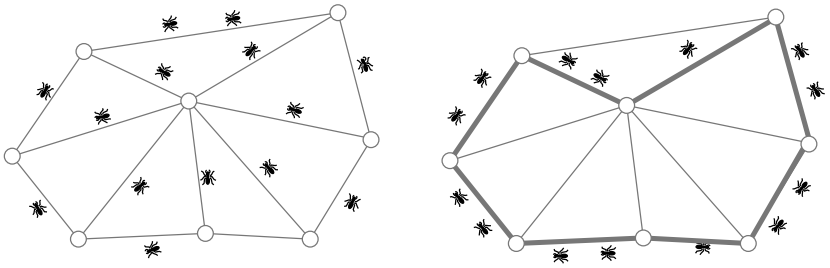

ABBILDUNG 3. Für einige komplexe Problemstellungen führen Ameisenalgorithmen zu qualitativ hochwertigen Ergebnissen.

5.3. **Neuronale Netze.** Bei *neuronalen Netzen* wird die Funktionsweise der Neuronen im menschlichen Gehirn imitiert, indem diese als Schaltzellen mit mehreren Eingangs- und einem Ausgangssignal beschrieben werden, die miteinander vernetzt sind. Das Ausgangssignal ergibt sich dabei als gewichtete Summe der Eingangssignale, wobei die Gewichte in einer Trainingsphase zu bestimmen sind. Besonders gut eignen sich neuronale Netze zur Mustererkennung, das heißt, wenn beliebige Muster mit Hilfe gewisser Referenzmuster klassifiziert werden sollen, wie in Abbildung 4 schematisch dargestellt ist. Die Bestimmung der erforderlichen Koeffizienten erfolgt in einer Trainingsphase, in der bekannte Muster den jeweiligen Referenzmustern zugeordnet werden.

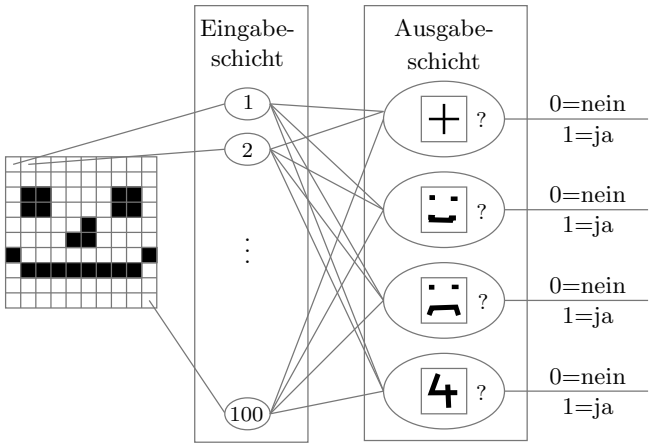

ABBILDUNG 4. Musterklassifizierung mit Hilfe eines neuronalen Netzes. Jedem Bildpixel und jedem Referenzmuster ist ein Neuron zugeordnet.