www.javaspektrum.de

Ausgabe 3/2019
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 03

JavaSPEKTRUM

Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen

Doing by Learning – Praxiseinsatz maschinellen Lernens

Einführung in die Komplexität von Algorithmen

Maschinelles Lernen – Frameworks und Praxis

Interview

Glenn Gore, Worldwide Lead Architect bei Amazon Web Services über AI, über Sicherheit und die Kluft zwischen IT- und Business

Fachthemen

Wieso Datenschutz (jetzt) auch sichere Softwareerstellung betrifft

Testdaten mit den Patterns „Object Mother“ und „Builder“

Sonderbeilage: German Testing zum Inhalt „Testen in Zeiten der Digitalisierung“

Doing by Learning – Praxiseinsatz maschinellen Lernens

G30325

Warum künstliche Intelligenz so schwierig ist

Einführung in die Komplexität von Algorithmen

Christian Heitzmann

Künstliche Intelligenz ist in aller Munde. Was im Hype dieses Themas oft untergeht, ist eine nüchterne Betrachtung der theoretischen Machbarkeit: Was wird überhaupt jemals realisierbar sein, und wo setzt die Wissenschaft Obergrenzen? Anhand von einfachen, nachvollziehbaren Beispielen führt dieser Artikel den Grundbegriff der algorithmischen Komplexität ein und überträgt die daraus gewonnenen Konzepte auf die künstlichen neuronalen Netze.

Im Rahmen dieses Artikels ist eine Java-Applikation namens *Time Complexity Visualizer* entstanden, die dem Leser unter dem Link [TimeComVis] kostenlos für den privaten Eigengebrauch zur Verfügung steht. Dieses Programm stellt ein grafisches Framework dar, welches eine vom Benutzer zu implementierende Methode mit unterschiedlichen Eingabegrößen ausführt. Die Laufzeiten, die zur Ausführung des programmierten Algorithmus benötigt werden, werden automatisch gemessen und in einem Liniendiagramm dargestellt. So lässt sich sofort erkennen, wie sich die Laufzeit einer eigenen Methode, die Eingabedaten verarbeitet, in Bezug auf ihre Eingabegröße verhält (s. Abb. 1).

Das genaue Vorgehen, wie sich eigene Klassen implementieren und ausführen lassen, ist in der dem Programm beiliegenden Anleitung beschrieben. Alle Diagramme in diesem Artikel stammen aus dieser Applikation. Jeder Durchlauf wird drei Mal wiederholt, um Schwankungen, wie sie zum Beispiel durch die (Neben-)Auslastung des Computers, Optimierungen durch die Java-Virtual-Machine (JVM), Unterbrechungen durch den Garbage-Collector oder zufälligen Eingabedaten entstehen, auszugleichen.

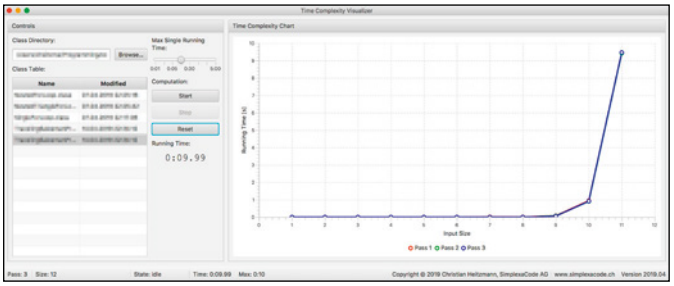
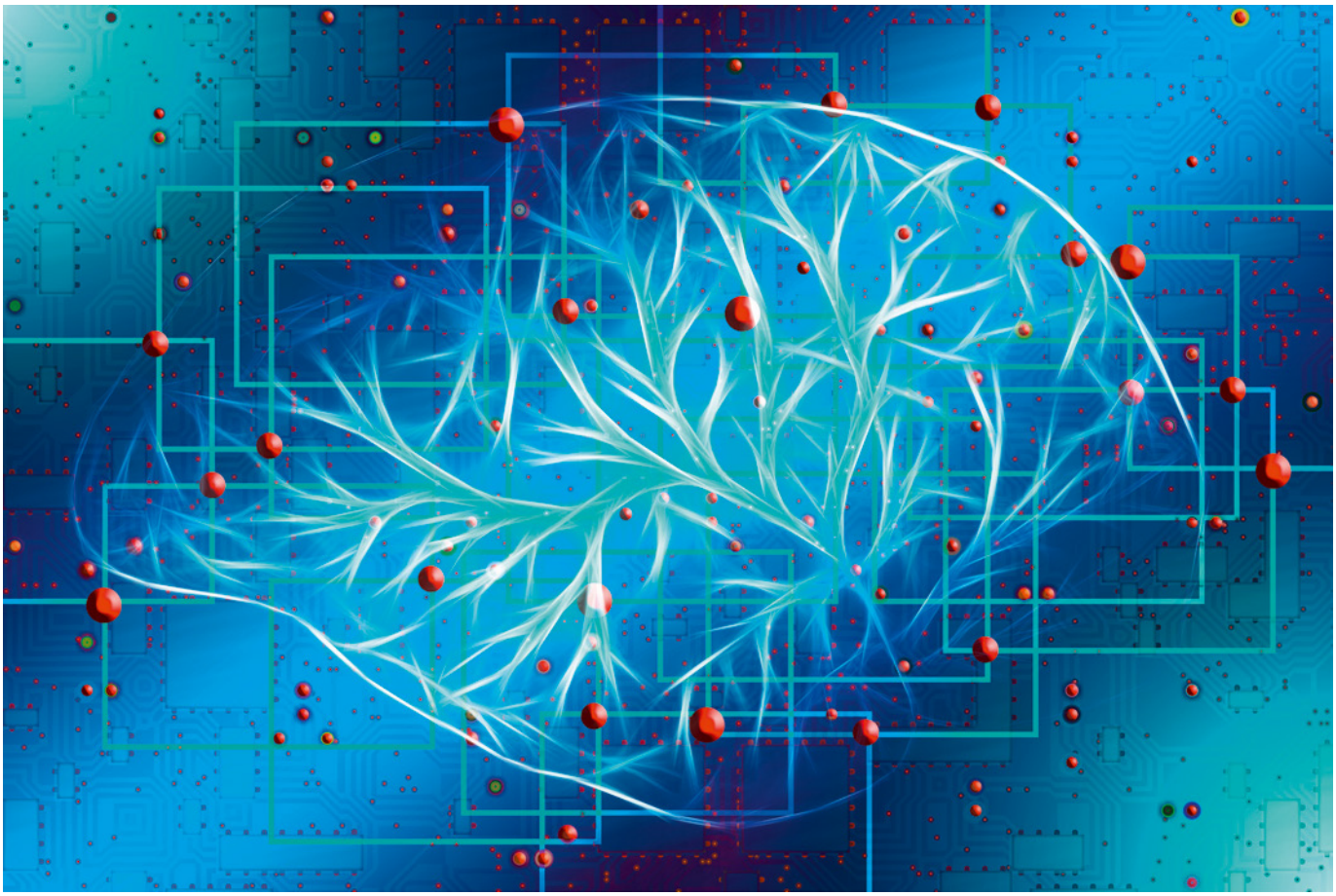


Abb. 1: Time Complexity Visualizer



Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung vor allem für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.
E-Mail: christian.heitzmann@simplexacode.ch

```
import ch.simplexacode.time_complexity_visualizer.*;

public final class SingleForLoop extends AbstractComputation {
  @Override
  public void compute() {
    final long inputSize = inputSize();
    for (long i = 1; i <= inputSize; i++) {
      if (Thread.currentThread().isInterrupted()) {
        return;
      }
      performLengthyComputation();
    }
  }
}
```

Listing 1: Einfache for-Schleife

Einfache for-Schleife

Der vermutlich einfachste „Algorithmus" ist eine einfache `for`-Schleife, die so oft ausgeführt wird, wie die sogenannte *Eingabegröße* (english: input size) es fordert. Listing 1 zeigt, wie eine vollständige, vom Framework ausführbare Implementierung einer einfachen `for`-Schleife aussieht.

Die `for`-Schleife läuft jeweils von `1` bis `inputSize`. Ganz wichtig – nicht nur für dieses Framework, sondern generell in allen Fällen, in denen mit Threads programmiert wird – ist eine regelmäßige Überprüfung des `interrupted`-Status des eigenen Threads sowie dessen zeitnahe Terminierung, sobald „von außen" ein Unterbrechungswunsch geäußert wird. Läuft ein Algorithmus zu schnell ab, sodass bei überschaubaren Eingabegrößen keine sinnvolle Zeitmessung möglich ist, so lassen sich die einzelnen Ausführungsschritte künstlich bremsen. Der Methodenaufruf `performLengthyComputation` bewirkt eine Verzögerung von – auf aktuellen Maschinen – ganz grob

```
public final class NestedForLoop extends AbstractComputation {
  @Override
  public void compute() {
    final long inputSize = inputSize();
    for (long i = 1; i <= inputSize; i++) {
      if (Thread.currentThread().isInterrupted()) {
        return;
      }
      for (long j = 1; j <= inputSize; j++) {
        performLengthyComputation();
      }
    }
  }
}
```

Listing 2: Verschachtelte for-Schleifen

1 µs, also etwa einer millionstel Sekunde. Erst durch diese künstliche Verzögerung werden vernünftig lange, messbare Laufzeiten erreicht; die Schleife verfügt dann quasi über genügend „Last".

Abbildung 2 zeigt die gemessenen Laufzeiten im Liniendiagramm. 278 Millionen Iterationen dauern hier etwa 276 Sekunden. Sehr schön zu erkennen ist die fast schon perfekte gerade Form des Graphen. Jeder Leser kann sich sofort ausmalen, wie sich die Laufzeiten verhalten würden, wenn die Eingabegröße weiter steigt. Wenn die Laufzeit bei 278 Millionen etwa 276 Sekunden dauert, dann wird sie bei 500 Millionen grob 500 Sekunden dauern, bei 800 Millionen grob 800 Sekunden usw. Die Laufzeit ist also offensichtlich direkt proportional zur Eingabegröße. Eine *doppelt* so große Eingabegröße resultiert in einer *doppelt* so langen Ausführungszeit. Man spricht hierbei auch von einer *linearen Komplexitätsklasse*, mathematisch notiert mit dem *Landau-Symbol* $O(n)$ (ausgesprochen: „O von n").

Verschachtelte for-Schleifen

Etwas interessanter wird die ganze Sache, wenn zwei `for`-Schleifen ineinander verschachtelt werden. Listing 2 zeigt einen solchen Fall. Die Eingabegröße dient nun als Obergrenze sowohl für die äußere als auch für die innere Schleife. Beträgt die Eingabegröße zum Beispiel 10, so wird die äußere Schleife (mit Index `i`) 10 Mal ausgeführt, und für jeden Schleifendurchlauf der äußeren Schleife wird die innere Schleife (mit Index `j`) ebenso 10 Mal ausgeführt. Der eigentliche Inhalt der inneren Schleife (namentlich `performLengthyComputation`) wird also $10 \cdot 10 = 100$ Mal aufgerufen.

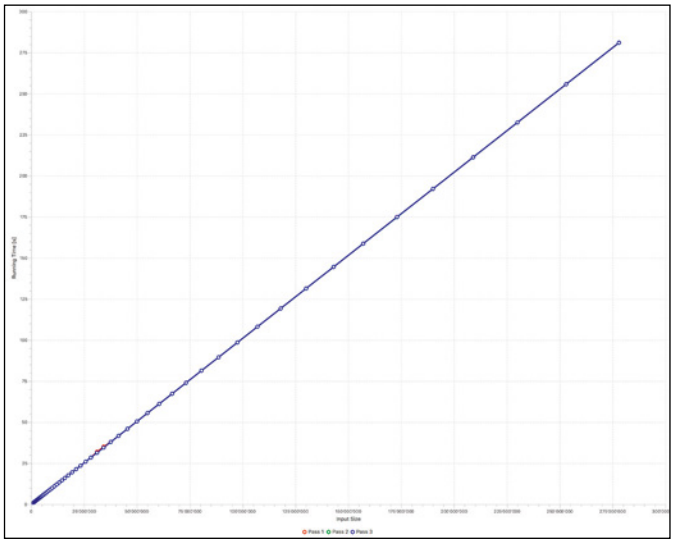


Abb. 2: Laufzeitdiagramm einer einfachen for-Schleife

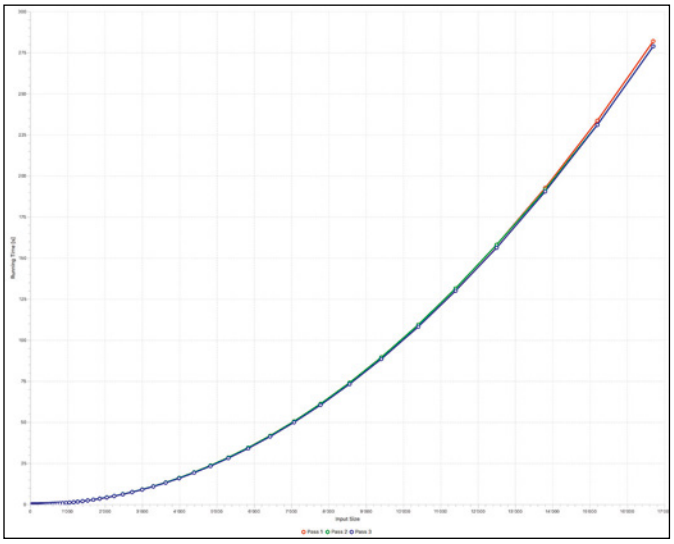


Abb. 3: Laufzeitdiagramm verschachtelter for-Schleifen

```
public final class NestedTriangleForLoop extends AbstractComputation {
  @Override
  public void compute() {
    final long inputSize = inputSize();
    for (long i = 1; i <= inputSize; i++) {
      if (Thread.currentThread().isInterrupted()) {
        return;
      }
      for (long j = 1; j <= i; j++) {
        performLengthyComputation();
      }
    }
  }
}
```

Listing 3: Verschachtelte for-Schleifen in „Dreiecksform“

Wird die Eingabegröße verdoppelt (hier: von 10 auf 20), so resultiert dies in insgesamt 20·20 = 400 Schleifendurchläufen. Eine *doppelte* Eingabegröße resultiert also in einer *vierfachen* Ausführungszeit, eine *dreifache* Eingabegröße in einer *neunfachen*. Man spricht hier von einer *quadratischen Laufzeit* respektive einer *quadratischen Komplexitätsklasse*, notiert als $O(n^2)$ und ausgesprochen als „O von n Quadrat“. Abbildung 3 zeigt das Liniendiagramm der verschachtelten `for`-Schleifen mit ihrer quadratischen Laufzeit.

Eine quadratische Laufzeit ist eigentlich schon an der Grenze des Akzeptablen für „alltägliche Algorithmen“. Der Leser kann dies im Time Complexity Visualizer gerne einmal nachvollziehen – man muss es einfach selbst erfahren, wie schnell unangenehm lange Laufzeiten erreicht werden, wenn man die Eingabegrößen zunehmend vergrößert. Ließ sich im ersten Beispiel mit der einfachen `for`-Schleife noch eine Eingabegröße von einer Viertelmilliarde innerhalb von 5 Minuten bearbeiten, so ist bei den verschachtelten `for`-Schleifen bereits bei circa 17 000 Schluss. Übertragen auf ein etwas realistischeres Beispiel: Dauert in einem Fotobearbeitungsprogramm die Anwendung eines grafischen Effekts auf einem (der Einfachheit halber quadratischem) Digitalfoto mit 4000 Pixeln Seitenlänge 20 Sekunden, so sind dies bei einem Foto mit 16 000 Pixeln Seitenlänge bereits über 5 Minuten.

Listing 3 zeigt ebenfalls zwei verschachtelte `for`-Schleifen, diesmal allerdings in „Dreiecksform“: Die innere Schleife zählt jeweils nur von `1` bis `i`, also bis zum aktuellen Wert der äußeren Schleife. So durchläuft der Index `j` (der inneren Schleife) folgende Werte:

```
1
1, 2
1, 2, 3
1, 2, 3, 4
1, 2, 3, 4, 5
...
```

Wie verhält sich die Laufzeit nun in diesem Fall? Der Time Complexity Visualizer in Abbildung 4 gibt die Antwort: Die „Form“ des Graphen (eine klassische Parabel) ist praktisch identisch zu derjenigen des vorherigen Beispiels, was klar für eine quadratische Laufzeit spricht. Allerdings haben sich die Wertebereiche geändert: Konnte im vorherigen Beispiel innerhalb von 5 Minuten nur eine Eingabegröße bis circa 17 000 bearbeitet werden, so beträgt diese neu etwa 23 000. Die vorherige Eingabegröße von 17 000 ist hingegen schon nach etwa 145 Sekunden bearbeitet.

Die mathematische Begründung des Quadrats leuchtet ziemlich schnell ein, wenn man sich an frühere Mathematikstunden in der Schule erinnert. Mit Sicherheit wurde dort einmal hergeleitet, dass sich die Summe aller aufsteigenden Zahlen von 1 bis n mit der

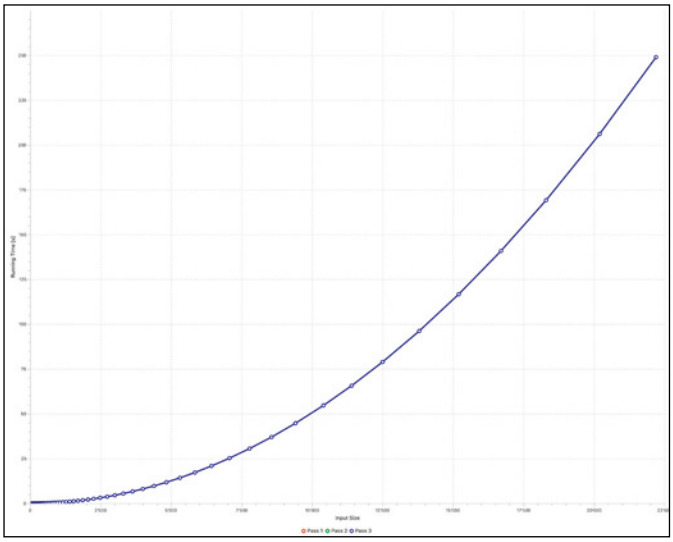


Abb. 4: Laufzeitdiagramm verschachtelter for-Schleifen in Dreiecksform

Formel $\frac{1}{2}\,n\,(n+1) = \frac{1}{2}\,(n^2+n)$ berechnen lässt (Stichwort: „kleiner Gauß“). Darin ist das Quadrat ersichtlich.

Zwei Punkte sind dabei außerordentlich wichtig:

- Relevant für die Angabe einer algorithmischen Komplexitätsklasse ist immer dasjenige Glied mit der höchsten Ordnung, das heißt derjenige Term, der vereinfacht gesagt am schnellsten wächst. Die „Dreiecks-`for`-Schleifen“ bestehen aus einem quadratischen Term (n^2) und einem linearen Term (n). Für große Zahlen wird das Gewicht des quadratischen Terms überwiegen, der lineare Term wird zunehmend an Gewicht verlieren.
- Völlig irrelevant für die Angabe einer algorithmischen Komplexitätsklasse sind konstante Faktoren. Wer genau hinschaut, sieht, dass die Dreiecks-Version „doppelt so schnell“ ausführt wie das Beispiel mit den „vollen“ `for`-Schleifen. In der oben gezeigten Formel ist dies am Faktor ½ zu erkennen. Dennoch haben beide Algorithmen die Laufzeitordnung $O(n^2)$.

Mit dem soeben genannten zweiten Punkt kann auch die eigentliche Kernbotschaft dieses Artikels eingeführt werden: *Konstante Faktoren in der Laufzeit sind unbedeutend. Entscheidend ist die Komplexitätsklasse, die angibt, wie sich das Wachstum der Laufzeit bezüglich der Eingabegrößen verhält.*

Konstante Faktoren lassen sich relativ einfach ändern: Vielleicht läuft ein Algorithmus auf einer GPU (Grafikprozessor) drei Mal schneller als auf einer normalen CPU (Hauptprozessor)? Vielleicht lässt sich mit dem Einsatz von Kotlin oder Python anstelle von Java ein Faktor 2 herausholen? Oder man wartet einfach 18 Monate ab und verlässt sich darauf, dass bis dahin – analog einer Art Mooreschem Gesetz – die Computer „doppelt so schnell“ sein werden wie heute.

All diese Optimierungsmaßnahmen sind aber nur ein Tropfen auf den heißen Stein, wenn man das generelle Laufzeitverhalten, also das Wachstum in Bezug auf die Eingabegrößen, außer Acht lässt. Hierzu ein Gedankenexperiment: Angenommen, ein Flaschengeist sei bereit, seinem Befreier einen Wunsch zu erfüllen. Er wird entweder a) dessen aktuelles Gehalt verdoppeln oder b) dessen aktuelles Gehalt quadrieren. Für welche Option a) oder b) sollte sich der Befreier wohl entscheiden? – Würde er sich auch für Option b) entscheiden, wenn es hieße, der Flaschengeist könne das Gehalt zwar quadrieren, dies koste dann aber einmalig 20 Euro Bearbeitungsgebühren? – Denkfehler dieser Art treten in der Praxis leider häufiger auf, als einem lieb ist. Nicht selten wird mit teilwei-

zum Inhalt

se großem Elan an Details herumgewerkelt, während die eigentlich ausschlaggebenden, großen Brocken übersehen werden.

In 80 Tagen um die Welt

Spulen wir ein paar Algorithmik-Kapitel vor. In der beiliegenden Software [TimeComVis] sind andere Beispiele enthalten, wie etwa verschiedene Sortieralgorithmen und deren Vergleiche zueinander. Dort sieht man auch, wie clevere Ideen dazu beitragen, dass sich die in der Regel quadratische Laufzeit von naiven Sortier-Implementierungen auf ein angenehmes Niveau von $O(n \log n)$ („O von n log n“) verbessert.

Am anderen Ende des Laufzeitspektrums befinden sich allerdings auch sehr viele Probleme, die in ihrer Beschreibung zwar einfach klingen, deren (exakte) Lösung aber nur sehr schwer zu ermitteln ist. Die Laufzeit dieser sogenannten *NP-Probleme* ist so dramatisch schlecht, dass selbst für überschaubare Eingabegrößen bereits Laufzeiten entstehen, die sich im Bereich von Jahren befinden, oder bei denen man sogar vorhersagt, dass ihre Lösung länger dauern würde, als das Universum alt ist (knapp 14 Milliarden Jahre).

Eines dieser einfach klingenden, aber schwer lösbaren Probleme ist das *Problem des Handlungsreisenden* (englisch: Traveling Salesman Problem). Die Aufgabe besteht darin, für einen Reisenden mit gegebenem Startort eine optimale Reihenfolge für alle anderen zu besuchenden Städte zu finden, sodass die Gesamtdistanz dieser Rundreise minimal wird – ein durchaus legitimes Anliegen.

Vorab: Es gibt diverse, recht effiziente Algorithmen, die dieses Problem „ziemlich“ gut lösen. Will man allerdings eine hunderprozent optimale Lösung, so funktionieren diese Algorithmen nicht mehr. Die einzige Möglichkeit, die richtige Lösung zu finden, besteht darin, alle erdenklichen Möglichkeiten auszuprobieren. Beginnend bei Stadt A kann der Handlungsreisende also die Städte B, dann C, dann D und dann zurück nach A besuchen. Er kann aber auch A, dann B, dann D, dann C, oder A, dann C, dann B, dann D, und so weiter, besuchen.

Etwas allgemeiner formuliert: Besteht das Problem aus 10 Städten, so hat der Handlungsreisende 9 weitere Städte zur Auswahl, die er als Erstes besucht. Ausgehend von dieser Stadt hat er 8 verbleibende Städte, die er als Zweites besucht, dann 7 Städte für den dritten Aufenthalt, 6 für den vierten usw. Letzten Ende resultiert das in $9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 362\,880$ verschiedenen Möglichkeiten. Bei 15 Städten mit gegebener Anfangsstadt sind es bereits 87 Milliarden Möglichkeiten, bei 20 Städten bereits 121 Billiarden (eine 18-stellige Zahl).

Abbildung 5 zeigt das Laufzeitverhalten eines allgemeinen Traveling Salesman Problem (TSP) im Time Complexity Visualizer. Innerhalb des Zeitlimits von 5 Minuten war lediglich die Problemstellung mit 12 Städten lösbar, was knapp 2 Minuten benötigte. Abbildung 6 zeigt ein Lösungsbild für 14 Städte, für das der Rechner des Autors knapp 8 Stunden beschäftigt war.

Das Aufmultiplizieren aufsteigender beziehungsweise absteigender Zahlen nennt man *Fakultät*. Sie ist bekannt für ihr explosives Wachstum. Probleme mit Laufzeiten, denen eine Fakultät zugrunde liegt, gehören zur *faktoriellen Komplexitätsklasse*, bezeichnet mit $O(n!)$ („O von n Fakultät“). Analog dazu – und sogar noch schlimmer – verhält sich die *exponentielle Komplexitätsklasse*, zum Beispiel $O(10^n)$ („O von 10 hoch n“). Die entsteht meist dann, wenn Eingabeparameter (hier zum Beispiel die Städte) jeweils mehrfach und nicht nur einmal verwendet werden dürfen.

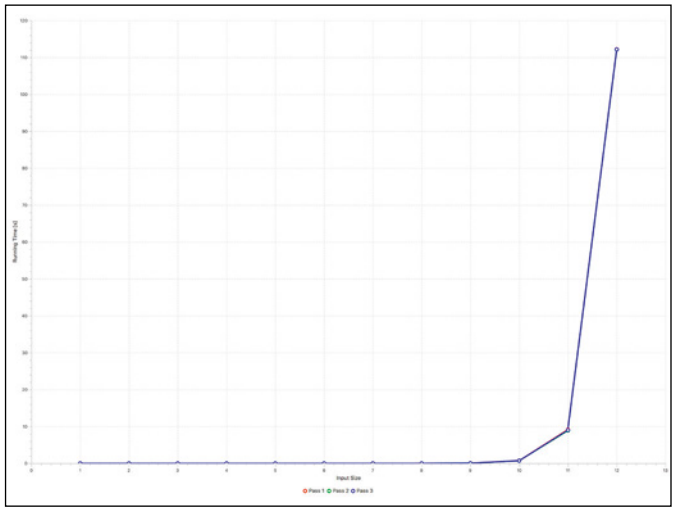


Abb. 5: Laufzeitdiagramm des Traveling Salesman Problem

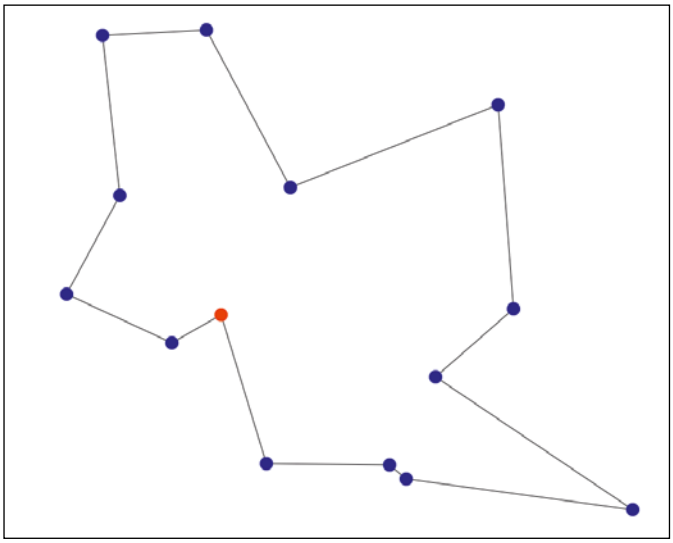


Abb. 6: Lösung des Traveling Salesman Problem für 14 Städte

Ein typisches Beispiel eines Problems mit exponentieller Laufzeit ist die Brute-Force-Attacke zum Erraten eines Passworts. Angenommen, es werden für ein Passwort die 26 Groß- und 26 Kleinbuchstaben, 10 Ziffern sowie die 8 gängigen Sonderzeichen verwendet. Dann gibt es pro Stelle 70 verschiedene Möglichkeiten für ein Zeichen. Jede zusätzliche Stelle vergrößert den Suchraum um den Faktor 70. Im Time Complexity Visualizer liegt eine Brute-Force-Suche als Beispiel bei, sodass sich der Leser gerne selbst einen Eindruck von der Laufzeitkomplexität $O(70^n)$ verschaffen kann.

Komplexität von künstlichen neuronalen Netzen

Übertragen wir die bis jetzt kennengelernten Konzepte auf den Klassiker der künstlichen Intelligenz, die *künstlichen neuronalen Netze* (KNN). Für eine Grundlagenbeschreibung ihrer Funktionalität fehlt an dieser Stelle leider der Platz. Soweit sollte bekannt sein, dass KNNs aus mindestens einer *Eingabeschicht*, einer *Ausgabeschicht* und beliebig vielen *verdeckten Schichten* aufgebaut sind, wobei jede Schicht eben aus künstlichen Neuronen besteht. Im Idealfall ist jedes Neuron einer Schicht mit jedem Neuron der benachbarten Schicht(en) verbunden.

Eingabe„reize“, zum Beispiel die einzelnen Pixel eines Bildes, breiten sich je nach aktueller, für jedes Neuron individueller Gewichtung auf die in der nachfolgenden Schicht befindlichen Neuronen in sogenannten *Feed-Forward-Schritten* aus. Am Ende lösen einige wenige Neuronen der Ausgabeschicht aus, denen meist die Bedeutung von *Kategorisierern* zukommt. Zeigt das Bild am Eingang zum Beispiel eine Katze, dann sollte das „Katzen-Ausgabe-Neuron“ auslösen, zeigt es einen Vogel, dann das „Vogel-Ausgabe-Neuron“.

Während der Trainings- oder Lernphase wird anschließend in der sogenannten *Backpropagation* der Fehler, der entstanden ist (zum Beispiel wenn das Katzen-Neuron ausgelöst hat, obwohl es sich um einen Vogel handelte), bis hin zur Eingabeschicht „zurückverteilt“. So weiß jedes Neuron, mit welchem Anteil es am Fehler beteiligt war, und kann sich (respektive seine Gewichtung) entsprechend für den nächsten Durchlauf korrigieren.

Erstes Problem

Das erste Problem an dieser Art des Lernens ist die unvorstellbare Menge an Trainingsdaten, die es benötigt, um das System so automatisch parametrisieren zu lassen. Ein Marktgigant wie Google mag über die erforderlichen Ressourcen – Hardware, Personal und Trainingsdaten – verfügen, um in Form Hunderttausender, bereits vorkategorisierter Bilder das System auf eine mehr oder weniger verlässliche Mustererkennung zu trimmen. Mit „Lernen“ hat das aber wenig zu tun. Keiner von uns allen (Leser wie Autor) benötigte in seiner Kindheit Tausende Bilder, bis er eine Katze von einem Vogel unterscheiden konnte. Oftmals reichte bereits ein einziges Bild, oder gar nur die abstrakte Beschreibung eines Tiers, um es ab dann benennen zu können (zum Beispiel „sehr langer Hals“ = Giraffe). Die Menge an Trainingsdaten muss man nicht nur besitzen, sondern auch die entsprechende Trainingszeit zur Verfügung haben.

Zweites Problem

Das zweite Problem ist die rein rechnerische Anzahl an Neuronen und Verbindungen, die es braucht, um ausreichend komplexe Problemstellungen lösen zu können – womit man dann direkt wieder auf die Laufzeitkomplexität stößt. Ein Rechenbeispiel: Für die Bilderkennung eines 640x640 Pixel großen Bildes (das war Instagram-Auflösung vor 2015 und ist nicht sonderlich groß) werden für die 409 600 Pixel entsprechend 409 600 Eingabeneuronen benötigt (sofern man sich keiner Tricks aus der Bildverarbeitung bedient, die man meist auch anwendet). Im Idealfall sollte jedes Neuron mit *allen* Neuronen der Nachbarschicht verbunden sein, was meist jedoch nicht realisiert werden kann. Hätte die nächste Zwischenschicht auch um die 400 Tausend Neuronen, so wären alleine für den ersten Schritt bereits 167 Milliarden Rechenschritte für die Signalweiterleitungen notwendig ($409\,600^2$). Simuliert man nur jeweils 1000 Verbindungen pro Neuron, so sind es immer noch über 409 Millionen ($409\,600 \cdot 1000$).

Das alles wäre für sich genommen ja noch handhabbar, gäbe es da nicht die vielen Zwischenschichten, die man benötigt, insbesondere im Bereich des Deep Learning, um zunehmend abstrakte Merkmale herausschälen zu können. Bleiben wir bei den 1000 Verbindungen pro Neuron, so kommen mit jeder zusätzlichen Schicht weitere 400 Millionen Rechenschritte dazu. Damit besteht die Komplexität unseres Systems schon aus fünf Faktoren: *Anzahl Trainingsdaten* x *Bildbreite* x *Bildhöhe* x *durchschnittliche Anzahl Verbindungen pro Neuron* x *Anzahl Schichten*. Wer jetzt nicht genau aufpasst und auch nur einen einzigen dieser Faktoren aus dem Ruder laufen lässt, riskiert, dass sich seine KI-Lösung nicht mehr mit vernünftigem Zeitaufwand realisieren lässt.

Drittes Problem

Das dritte Problem besteht darin, dass es keine Schwarz-Weiß-Antwort auf die Anzahl Neuronen, die Anzahl Verbindungen pro Neu-

zum Inhalt

ron, die Anzahl Schichten oder gar nur der Verteilung der Schichten gibt. Sollen die Zwischenschichten mehr oder weniger Neuronen als die Eingabeschicht besitzen? Soll die Anzahl konstant bleiben, oder zunehmen oder abnehmen? All das sind zeitaufwendige Parametrisierungsarbeiten, die wahrscheinlich auf ein reines Ausprobieren oder Heuristiken hinauslaufen werden.

Zum Vergleich: Das menschliche Gehirn verfügt geschätzt über circa 85 Milliarden Neuronen und etwa 100 Billionen Verbindungen, das heißt grob 1000 Verbindungen pro Nervenzelle – und die sind nicht einmal an streng hierarchische Schichten gebunden, sondern können „kreuz und quer" verlaufen. Hört sich das nicht sehr nach exponentieller Laufzeit an, wenn man das seriell mit einem Computer emulieren müsste?

So verkommt die erhoffte künstliche Intelligenz letzten Endes doch nur zu einem manuellen Prozess des „Schräubchendrehens", einer menschengemachten Parametrisierung und Optimierung, nicht selten, um überhaupt noch eine Berechenbarkeit in vertretbarer Zeit zu gewährleisten. Das gewünschte Verhalten wird der KI damit in einer mehr oder weniger direkten Form schon einprogrammiert. Mit wirklicher Intelligenz hat das dann nicht mehr viel zu tun.

Fazit

Computer sind eine geniale Erfindung und unschlagbar in der Bearbeitung großer Daten. Damit hat es sich aber auch schon. Die beschworene Intelligenz liegt noch in weiter Ferne, sind die heutigen KI-Algorithmen letzten Endes doch nichts anderes als (zugegeben, oft sehr gute) Mustererkennungsmaschinen. Muster genügen aber nicht für Intelligenz. Es fehlten ihnen bis heute an einem Verständnis abstrakter Konzepte wie zum Beispiel Gesellschaft oder Moral, Liebe oder Konkurrenz, Mimik oder Gestik – und es sieht nach aktuellem Kenntnisstand nicht danach aus, dass sich in dieser Richtung bereits etwas tun würde.

Im Rahmen dieses Artikels wurde versucht, die Herausforderungen und Limitierungen des klassischen KI-Ansatzes – den künstlichen neuronalen Netzen – rein aus der Perspektive der algorithmischen Komplexität zu beleuchten. Für Zahlen und Größenordnungen ist es immer sehr hilfreich, ein Gespür dafür zu entwickeln. Hoffentlich kann die diesem Artikel beiliegende Java-Software ein bisschen dazu beitragen. Alle Leser sind herzlich dazu eingeladen, darin mit ihren eigenen Algorithmen zu experimentieren.

Literatur und Links

[Cormen07] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Algorithmen – Eine Einführung, 2. Auflage, Oldenbourg Wissenschaftsverlag, 2007

[Pom08] G. Pomberger, H. Dobler, Algorithmen und Datenstrukturen, Pearson Studium, 2008

[Sedge14] R. Sedgewick, K. Wayne, Algorithmen, 4. Auflage, Pearson Deutschland, 2014

[TimeComVis] SimplexaCode, Time Complexity Visualizer, https://link.simplexacode.ch/7sns

[Trink16] A. Trinkwalter, Netzgespinste, in: c't 6/2016, Heise Medien, 2016

[Trink18] A. Trinkwalter, Statistik ist nicht Denken, in: c't 24/2018, Heise Medien, 2018

[WikiTSP] Wikipedia, Travelling salesman problem, https://en.wikipedia.org/wiki/Travelling_salesman_problem

Als innovatives, dynamisches Unternehmen mit rund 420 Mitarbeitern sind wir bereits seit mehr als 40 Jahren auf Erfolgskurs und expandieren weiter! Aktuell suchen wir in Mosbach, Stuttgart und Heidelberg

- Softwareentwickler und Softwarearchitekten (m/w/d)
- Fachliche und Technische Projektmanager (m/w/d)
- IT-Consultants Industrie 4.0 (m/w/d)
- Account Manager und Key Account Manager Sales (m/w/d)

Sind Sie bereit für spannende Aufgaben rund um **Industrie 4.0**, **Digitale Transformation** und **Internet of Things**? Dann werden Sie Teil der MPDV und begleiten mit uns unsere Kunden auf ihrem Weg zur **Smart Factory**!

Informieren!
jobs.mpdv.com

Fragen?
Tel.: 06261/92 09-0

Bewerben!
jobs@mpdv.com

MPDV Mikrolab GmbH ▪ Römering 1 ▪ 74821 Mosbach ▪ zum Inhalt / 92 09-0 ▪ jobs@mpdv.com