Numerik des Maschinellen Lernens

Jan Heiland

TU Ilmenau – Sommersemester 2024

Contents

Vorwort

Das ist ein Aufschrieb der parallel zur Vorlesung erweitert wird.

Korrekturen und Wünsche immer gerne als *issues* oder *pull requests* ans github-repo.

Chapter 1

Einführung

Was sind *Numerische Methoden für Maschinelles Lernen* (ML)?

Kurz gesagt, beim Training eines ML-Modells durchläuft ein Computer Millionen von Anweisungen, die in Form mathematischer Ausdrücke formuliert sind. Gleiches gilt für die Bewertung eines solchen Modells. Dann stellen sich Fragen wie *wird es einen Punkt geben, an dem das Training endet?* und *wird das Modell genau sein?*.

Um zu beschreiben, was passiert, und für die spätere Analyse führen wir die allgemeinen Konzepte von

- Algorithmus
- Konsistenz/Genauigkeit
- Stabilität
- Rechenaufwand

ein, von denen einige klassische *numerische Analysen* sind.

1.1 Was ist ein Algorithmus

Interessanterweise ist der Begriff *Algorithmus* ähnlich intuitiv und abstrakt. Es bedurfte großer Anstrengungen, um eine allgemeine und prägnante Definition zu finden, die den Anforderungen und Einschränkungen aller Bereiche gerecht wird (von *Kochrezepten* bis zur Analyse von *formalen Sprachen*).

Definition 1.1 (Algorithmus)**.** Ein Problemlösungsverfahren wird als *Algorithmus* bezeichnet, wenn und nur wenn es eine *Turing-Maschine* gibt, die dem Verfahren entspricht und die, für jede Eingabe, für die eine Lösung existiert, *anhalten* wird.

Diese Definition ist in ihrer Allgemeinheit nicht allzu hilfreich - wir haben nicht einmal definiert, was eine Turing-Maschine ist.

> Eine *Turing-Maschine* kann als eine Maschine beschrieben werden, die einen Streifen von Anweisungen liest und auf diesen Streifen schreiben kann. Abhängig davon, was sie liest, kann sie vorwärts bewegen, rückwärts bewegen oder anhalten (wenn der Streifen einen vordefinierten Zustand erreicht hat). Das Schöne daran ist, dass dieses Setup in einen vollständig mathematischen Rahmen gestellt werden kann.

Hilfreicher und gebräuchlicher ist es, die Implikationen dieser Definition zu betrachten, um zu überprüfen, ob ein Verfahren zumindest die notwendigen Bedingungen für einen Algorithmus erfüllt

- Der Algorithmus wird durch endlich viele Anweisungen beschrieben (Endlichkeit).
- Jeder Schritt ist *machbar*.
- Der Algorithmus erfordert eine endliche Menge an Speicher.
- Er wird nach endlich vielen Schritten beendet.
- In jedem Schritt ist der nächste Schritt eindeutig definiert (*deterministisch*).
- Für denselben Anfangszustand wird er im selben Endzustand anhalten (*bestimmt*).

Somit könnte eine informelle, gute Praxisdefinition eines Algorithmus sein

Definition 1.2 (Algorithmus – informell)**.** Ein Verfahren aus endlich vielen Anweisungen wird als *Algorithmus* bezeichnet, wenn es eine bestimmte Lösung – falls sie existiert – zu einem Problem in endlich vielen Schritten berechnet.

> Beachten Sie, wie einige Eigenschaften (wie endlich viele Anweisungen) a priori angenommen werden.

Als noch informelleren Verweis auf Algorithmen werden wir die Begriffe ***(numerische) Methode*** oder ***Schema*** verwenden, um ein Verfahren durch Auflistung seiner zugrundeliegenden Ideen und Unterprozeduren anzusprechen, wobei *Algorithmus* sich auf eine spezifische Realisierung einer *Methode* bezieht.

Weiterhin unterscheiden wir

- *direkte* Methoden – die die Lösung exakt berechnen (wie die Lösung eines linearen Systems durch *Gauß-Elimination*) und
- *iterative* Methoden – die iterativ eine Folge von Annäherungen an die Lösung berechnen (wie die Berechnung von Wurzeln mit einem *Newton-Schema*).

1.2 Konsistenz, Stabilität, Genauigkeit

Für die Analyse numerischer Methoden werden allgemein die folgenden Begriffe verwendet:

Definition 1.3 (Konsistenz)**.** Wenn ein Algorithmus in exakter Arithmetik die Lösung des Problems mit einer gegebenen Genauigkeit berechnet, wird er als *konsistent* bezeichnet.

Definition 1.4 (Stabilität (informell))**.** Wenn die Ausgabe eines Algorithmus kontinuierlich von Unterschieden in der Eingabe und kontinuierlich von Unterschieden in den Anweisungen abhängt, dann wird der Algorithmus als *stabil* bezeichnet.

Die *Unterschiede in den Anweisungen* sind typischerweise auf Rundungsfehler zurückzuführen, wie sie in *ungenauer Arithmetik* auftreten.

> Man könnte sagen, dass ein Algorithmus konsistent ist, wenn *er das Richtige tut* und dass er stabil ist, *wenn er trotz aller Arten von kleinen Ungenauigkeiten funktioniert.* Wenn ein Algorithmus konsistent und stabil ist, wird er oft als *konvergent* bezeichnet, um auszudrücken, dass er schließlich die Lösung auch in ungenauer Arithmetik berechnen wird.

Beachten Sie, dass Begriffe wie

- *Genauigkeit* – wie nahe die berechnete Ausgabe der tatsächlichen Lösung kommt oder
- *Konvergenz* – wie schnell (typischerweise in Bezug auf den Rechenaufwand) der Algorithmus sich der tatsächlichen Lösung nähert

keine intrinsischen Eigenschaften eines Algorithmus sind, da sie von dem zu lösenden Problem abhängen. Man kann jedoch von *Ordnungskonsistenz* eines Algorithmus sprechen, um die erwartete Genauigkeit für eine Klasse von Problemen zu spezifizieren, und einen Algorithmus als konvergent einer bestimmten Ordnung bezeichnen, wenn er auch stabil ist.

1.3 Rechenkomplexität

Die *Rechenkomplexität* eines Algorithmus ist sowohl theoretisch (um abzuschätzen, wie der Aufwand mit beispielsweise der Größe des Problems skaliert) als auch praktisch (um zu sagen, wie lange das Verfahren dauern wird und welche Kosten in Bezug auf CPU-Zeit oder Speichernutzung es generieren wird) wichtig.

Typischerweise wird die Komplexität durch Zählen der elementaren Operationen gemessen, oft als *FLOP*s bezeichnet, was für *floating point operations* steht. Um

die Algorithmen in Bezug auf Komplexität versus Problemgröße zu klassifizieren, sind die folgenden Funktionsklassen hilfreich

Definition 1.5 (Landau-Symbole oder große O-Notation)**.** Sei $g\colon \mathbb{R} \to \mathbb{R}$ und $a \in \mathbb{R} \cup \{-\infty, +\infty\}$. Dann sagen wir für eine Funktion $f\colon \mathbb{R} \to \mathbb{R}$, dass $f \in O(g)$, wenn

$$\limsup_{x \to a} \frac{|f(x)|}{|g(x)|} < \infty$$

und dass $f \in o(g)$, wenn

$$\limsup_{x \to a} \frac{|f(x)|}{|g(x)|} = 0.$$

Der Sinn und die Funktionalität dieser Konzepte wird vielleicht deutlich, wenn man sich die typischen Anwendungen ansieht:

- Wenn $h > 0$ ein Diskretisierungsparameter ist und, sagen wir, $e(h)$ der Diskretisierungsfehler ist, dann könnten wir sagen, dass $e(h) = O(h^2)$, wenn *asymptotisch*, d.h. für immer kleinere h – der Fehler mindestens so schnell wie h^2 gegen 0 geht.
- Wenn $C(n)$ die Komplexität eines Algorithmus für eine Problemgröße n ist, dann könnten wir sagen, dass $C(n) = O(n)$, um auszudrücken, dass die Komplexität *asymptotisch*, d.h. für immer größere n, mit derselben Geschwindigkeit wie die Problemgröße wächst.

Leider ist die übliche Verwendung der Landau-Symbole etwas nachlässig.

1. Das oft verwendete "="-Zeichen ist informell und keineswegs eine Gleichheit.
2. Was das Limit a ist, wird kaum jemals explizit erwähnt, aber glücklicherweise ist es in der Regel aus dem Kontext klar.

Als Beispiel betrachten wir zwei verschiedene Wege, ein Polynom p vom Grad n an der Abszisse x zu bewerten, basierend auf den zwei äquivalenten Darstellungen

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x) \cdots)) \end{aligned}$$

Für eine direkte Implementierung der ersten Darstellung erhalten wir

```
'''Berechnung von p(x) in Standarddarstellung
'''
n = 10                                          # Beispielwert für n
ais = [(-1)**k*1/k for k in range(1, n+2)]      # Liste der Beispielkoeffizienten
x = 5                                           # Ein Beispielwert für x
cpx = ais[0]                                    # der Fall k=0
for k in range(n):
    cpx = cpx + ais[k+1] * x**(k+1)             # der Beitrag des k-ten Schritts
print(f'x={x}: p(x)={cpx:.4f}')                 # Ausgabe des Ergebnisses
```

Im k-ten Schritt benötigt der Algorithmus eine Addition (wenn wir auch die Initialisierung als Addition zählen) und k Multiplikationen. Das ergibt eine Gesamtkomplexität von

$$C(n) = \sum_{k=0}^{n}(1+k) = n + 1 + \frac{n(n-1)}{2} = 1 + \frac{n}{2} + \frac{n^2}{2} = O(n^2)$$

Für die zweite Darstellung können wir das sogenannte *Horner-Schema* implementieren, das lauten würde

```
'''Berechnung von p(x) mit dem Horner-Schema
'''
n = 10                                          # Beispielwert für n
ais = [(-1)**k*1/k for k in range(1, n+2)]      # Liste der Beispielkoeffizienten
x = 5                                           # Ein Beispielwert für x
cpx = ais[n]                                    # der Fall k=n
for k in reversed(range(n)):
    cpx = ais[k] + x*cpx                        # der Beitrag des k-ten Schritts
print(f'x={x}: p(x)={cpx:.4f}')                 # Ausgabe des Ergebnisses
```

Insgesamt benötigt dieses Schema $n+1$ Additionen und n Multiplikationen, d.h. $2n+1$ FLOPs, so dass wir sagen können, dass *dieser Algorithmus $O(n)$ ist.*

1.4 Übungen

1. Vergleichen Sie die beiden Implementierungen zur Bewertung eines Polynoms, indem Sie die Komplexität als Funktion von n darstellen und die benötigte CPU-Zeit für eine Beispielbewertung im Vergleich zu n messen und darstellen.

Weiterführende Literatur:

- wikipedia:Algorithmus

Chapter 2

Fehler und Konditionierung

Berechnungen auf einem Computer verursachen unvermeidlich Fehler, und die Effizienz oder Leistung von Algorithmen ist immer das Verhältnis von Kosten zu Genauigkeit. Zum Beispiel:

- Allein durch Betrachtung von Rundungsfehlern kann die Genauigkeit einfach und signifikant verbessert werden, indem auf Hochpräzisionsarithmetik zurückgegriffen wird, was jedoch höhere Speicheranforderungen und eine höhere Rechenlast mit sich bringt.
- In iterativen Verfahren können Speicher und Rechenaufwand leicht gespart werden, indem die Iteration in einem frühen Stadium gestoppt wird - auf Kosten einer weniger genauen Lösungsapproximation.

Beide, irgendwie trivialen Beobachtungen sind grundlegende Bestandteile des Trainings neuronaler Netzwerke. Erstens wurde beobachtet, dass Niedrigpräzisionsarithmetik Rechenkosten sparen kann, mit nur geringen Auswirkungen auf die Genauigkeit. Zweitens ist das Training ein iterativer Prozess mit oft langsamer Konvergenz, sodass der richtige Zeitpunkt für einen vorzeitigen Abbruch des Trainings entscheidend ist.

Definition 2.1 (Absolute und relative Fehler)**.** Sei $x \in \mathbb{R}$ die interessierende Größe und $\tilde{x} \in \mathbb{R}$ eine Annäherung daran. Dann wird der *absolute Fehler* definiert als $|\delta x| := |\tilde{x} - x|$ und der *relative Fehler* als $\frac{|\delta x|}{|x|} = \frac{|\tilde{x}-x|}{|x|}$.

Generell wird der relative Fehler bevorzugt, da er den gemessenen Fehler in den richtigen Bezug setzt. Zum Beispiel kann ein absoluter Fehler von 10 km/h je nach Kontext groß oder klein sein. Andererseits erfordert der relative Fehler die Kenntnis des tatsächlichen

Werts und die Division durch einen Wert nahe 0 kann die Fehlerschätzung verstärken.

Als Nächstes definieren wir die *Kondition* eines Problems A und analog eines Algorithmus (der das Problem löst). Dafür lassen wir x einen Parameter/Eingabe des Problems sein und $y = A(x)$ die entsprechende Lösung/Ausgabe. Die Kondition ist ein Maß dafür, inwieweit eine Änderung $x + \delta x$ in der Eingabe die resultierende relative Änderung in der Ausgabe beeinflusst. Dafür betrachten wir

$$\delta y = \tilde{y} - y = A(\tilde{x}) - A(x) = A(x + \delta x) - A(x)$$

welches nach Division durch $y = A(x)$ und Erweiterung durch $x\,\delta x$ wird zu

$$\frac{\delta y}{y} = \frac{A(x + \delta x) - A(x)}{\delta x} \frac{x}{A(x)} \frac{\delta x}{x}.$$

Für infinitesimal kleine δx wird der Differenzenquotient $\frac{A(x+\delta x)-A(x)}{\delta x}$ zur Ableitung $\frac{\partial A}{\partial x}(x)$, so dass wir die Kondition des Problems/Algorithmus bei x abschätzen können durch

$$\frac{|\delta y|}{|y|} \leq |\frac{\partial A}{\partial x}(x)| \frac{|x|}{|A(x)|} \frac{|\delta x|}{|x|} =: \kappa_{A,x} \frac{|\delta x|}{|x|}. \tag{2.1}$$

und nennen $\kappa_{A,x}$ die Konditionszahl.

Für vektorwertige Probleme/Algorithmen können wir die Konditionszahl darüber definieren, wie eine Differenz in der j-ten Eingabekomponente x_j die i-te Komponente $y_i = A_i(x)$ der Ausgabe beeinflusst.

Definition 2.2 (Konditionszahl)**.** Für ein Problem/Algorithmus $A\colon \mathbb{R}^n \to \mathbb{R}^m$ nennen wir

$$(\kappa_{A,x})_{ij} := \frac{\partial A_i}{\partial x_j}(x) \frac{x_j}{A_i(x)}$$

die partielle *Konditionszahl* des Problems. Ein Problem wird als *gut konditioniert* bezeichnet, wenn $|(\kappa_{A,x})_{ij}| \approx 1$ und als *schlecht konditioniert*, wenn $|(\kappa_{A,x})_{ij} \gg 1$, für alle $i = 1, \dots, m$ und $j = 1, \dots, m$.

Anstatt die skalaren Komponentenfunktionen von $A\colon \mathbb{R}^n \to \mathbb{R}^m$ zu verwenden, kann man die Berechnungen, die zu (2.1) geführt haben, mit vektorwertigen Größen in den entsprechenden Normen wiederholen.

2.1 Übungen

1. Leiten Sie die *Konditionszahl* wie in (2.1) für eine vektorwertige Funktion $A\colon \mathbb{R}^n \to \mathbb{R}^m$ ab. Wo spielt eine Matrixnorm eine Rolle?

2. Leiten Sie die Konditionszahl einer invertierbaren Matrix M ab, d.h. die Kondition des Problems $x \to y = M^{-1}x$, nach demselben Verfahren. Wo spielt die Matrixnorm eine Rolle?

Referenzen