Numerik des Maschinellen Lernens

Jan Heiland

TU Ilmenau – Sommersemester 2024

Contents

Vorwort **5**

1 Einführung **7**
1.1 Was ist ein Algorithmus 7
1.2 Konsistenz, Stabilität, Genauigkeit 9
1.3 Rechenkomplexität 9
1.4 Literatur 11
1.5 Übungen 11

2 Fehler und Konditionierung **13**
2.1 Fehler 13
2.2 Kondition 14
2.3 Kondition der Grundrechenarten 15
2.4 Übungen 16

3 Iterative Methoden **17**
3.1 Iterative Methoden als Fixpunktiteration 18
3.2 Gradientenabstiegsverfahren 20
3.3 Auxiliary Function Methods 22
3.4 Übungen 23

4 Stochastisches Gradientenverfahren **25**
4.1 Motivation und Algorithmus 25
4.2 Stochastisches Abstiegsverfahren 26
4.3 Konvergenzanalyse 28
4.4 Übungen 31

5 Nachklapp **33**

Referenzen **35**

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 Analysis* sind.

1.1 Was ist ein Algorithmus

Interessanterweise ist der Begriff *Algorithmus* zugleich intuitiv und abstrakt. Es bedurfte großer Anstrengungen, um eine allgemeine und wohlgestellte 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, genau dann 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 noch nicht einmal definiert, was eine Turing-Maschine ist.

Eine *Turing-Maschine* kann als eine Maschine beschrieben werden, die ein Band von Anweisungen liest und auf dieses Band schreiben kann. Abhängig davon, was sie liest, kann sie vorwärts bewegen, rückwärts bewegen oder anhalten (wenn das Band 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 *durchführbar*.
- Der Algorithmus erfordert eine endliche Menge an Speicher.
- Er wird nach endlich vielen Schritten beendet.
- In jedem Schritt ist der nächste Schritt eindeutig definiert (*Determiniertheit*).
- Für denselben Anfangszustand wird er im selben Endzustand anhalten (*Bestimmtheit*).

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 informellere Verweise 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* (oft auch als *Gleitkommaarithmetik* bezeichnet) auftreten.

> Man könnte sagen, dass ein Algorithmus konsistent ist, wenn *er das Richtige tut* und dass er stabil ist, *wenn er trotz beliebiger kleiner 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 *Konsistenzordnung* 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 zusätzlich 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 – wir werden stets die Ausführung einer Grundrechenart als eine

Operation zählen.

> Die Definition einer *elementaren Operation* auf einem Computer ist nicht universal, da viele Faktoren hier reinspielen. Gerne werden *FLOP* s angeführt, was für *floating point operations* steht. Allerdings ist es wiederum sehr verschieden auf verschiedenen Prozessoren wieviele FLOPs für eine Multiplikation oder Addition gebraucht werden.

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 unpräzise.

1. Das oft verwendete "="-Zeichen ist informell und keineswegs eine Gleichheit.
2. Was der Grenzwert a ist, wird selten 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 auszuwerten, 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 die Algorithmen

```
'''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 Literatur

- (Nocedal and Wright 2006): Ein gut lesbares Buch zur Optimierung.

1.5 Übungen

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

2. Zeigen Sie, dass es für $f \in O(g)$ mit $f \geq 0$ und $g > 0$ eine Konstante C gibt, sodass $f(n) = h(n) + Cg(n)$ mit $h \in o(g)$. *Bemerkung: diese Relation ist die Rechtfertigung für die eigentlich inkorrekte Schreibweise* $f = O(g)$.

3. Ermitteln Sie experimentell *die Ordnung* (d.h. den Exponent x in $O(n^x)$) und *die Konstante* C (s.o.) für die Laufzeit $t(n)$ der in `scipy.linalg.cholesky` implementierten Cholesky Zerlegung der Bandmatrix `A_n` aus dem folgenden Code Beispiel

```
import numpy as np
from scipy.linalg import cholesky
from time import time
n = 10                                  # example problem size
A_n = -1*np.diag(np.ones(n-1), -1) + \  # a tridiagonal band matrix
    2*np.diag(np.ones(n), 0) + \
    -1*np.diag(np.ones(n-1), 1)
tic = time()                            # start the timer
_ = cholesky(A_n)                       # perform the computation
toc = time()                            # stop the timer
print(f'n: {n} -- t_n: {toc-tic:.4e}')
```

Hinweis: Hier geht es um die Methodik und um eine sinnvolle Interpretation der Ergebnisse. Es kann gut sein, dass die Ergebnisse auf verschiedenen Rechnern verschieden ausfallen. Außerdem können für große n (wenn der Exponent und die Konstante am besten sichtbar sind) auf einmal bspw. ein zu voller Arbeitsspeicher die Berechnung negativ beeinflussen.

4. Diskutieren Sie, wie Laufzeitmessungen (bspw. zur Komplexitätsanalyse eines Verfahrens) aufgesetzt werden sollten, um reproduzierbare Ergebnisse zu erhalten. Was sollte dokumentiert werden, damit dritte Personen die Ergebnisse einordnen und ggf. reproduzieren können.

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

Beide, irgendwie trivialen Beobachtungen sind grundlegende Bestandteile des Trainings neuronaler Netzwerke. Erstens wurde beobachtet, dass Zahldarstellungen mit *einfacher Genauigkeit* (im Vergleich zum gängigen *double precision*) 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.

2.1 Fehler

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.

2.2 Kondition

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

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

Wir 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.

2.3 Kondition der Grundrechenarten

Da einfach jede Operation von Zahlen auf dem Computer auf die Grundrechenarten zurückgeht, ist es wichtig sich zu vergegenwärtigen wie sich diese Basisoperationen in Bezug auf kleine Fehler verhalten.

2.3.1 Addition

```
def A(x, y):
    return x+y

x, tx, y = 1.02, 1.021, -1.00
z = A(x, y)
tz = A(tx, y)
relerrx = (tx - x)/x        # here: 0.00098039
relerrz = (tz - z)/z        # here: 0.04999999
kondAx = relerrz/relerrx    # here: 50.9999999
```

In diesem Code Beispiel liegt der relative Fehler in x bei etwa 0.01% und im Ausgang z bei etwa 5%, was einer etwa 50-fachen Verstärkung entspricht. Für die Konditionszahl der Addition $A_y\colon x \to y + x$ gilt:

$$\kappa_{A_y;x} = \frac{|x|}{|x+y|} = \frac{1}{|1+\frac{y}{x}|}.$$

Diese Konditionszahl kann offenbar beliebig groß werden, wenn x nah an $-y$ liegt. Jan spricht von Auslöschung und tatsächlich lässt sich nachvollziehen, dass in diesem Fall die vorderen (korrekten) Stellen einer Zahl von einander abgezogen werden und die hinteren (möglicherweise inkorrekten) Stellen übrig bleiben.

Praktisch gesagt: Hantiert Jan mit Addition großer Zahlen um ein kleines Ergebnis erzielen ist das numerisch sehr ungünstig.

2.3.2 Multiplikation

```
def M(x, y):
    return x*y

x, tx, y = 1.02, 1.021, -1.00
z = M(x, y)
tz = M(tx, y)
relerrx = (tx - x)/x        # here: 0.00098039
relerrz = (tz - z)/z        # here: 0.00098039
kondMx = relerrz/relerrx    # here: 1.0
```

Das Ergebnis `1.0` für die empirisch ermittelte Konditionszahl war kein Zufall. Es gilt im Allgemeinen für $M_y \colon x \to yx$ dass

$$\kappa_{M_y;x} = |y| \frac{|x|}{|xy|} = 1.$$

Die Multiplikation ist also generell gut konditioniert.

2.3.3 Wurzelziehen

Das Berechnen der Quadratwurzel $W \colon x \to \sqrt{x}$ hat die Konditionszahl $\frac{1}{2}$. Bei Konditionszahlen kleiner als 1 verringert sich der relative Fehler, Jan spricht von *Fehlerdämpfung*.

2.4 Übungen

1. Leiten Sie die *Konditionszahl* wie in (2.1) für eine vektorwertige Funktion $A \colon \mathbb{R}^n \to \mathbb{R}^m$ her. Wo spielt eine Matrixnorm eine Rolle?
2. Leiten Sie mit dem selben Verfahren die Konditionszahl einer invertierbaren Matrix M her, d.h. die Kondition des Problems $x \to y = M^{-1}x$. Wo spielt die Matrixnorm eine Rolle?
3. Leiten Sie die Konditionszahlen für die Operationen *Division* und *Quadratwurzelziehen* her.
4. Veranschaulichen Sie an der Darstellung des Vektors $P = [1, 1]$ in der Standardbasis $\{[1, 0], [0, 1]\}$ und in der Basis $\{[1, 0], [1, 0.1]\}$ unter Verweis auf die Kondition der Addition, warum *orthogonale Basen* als *gut konditioniert* gelten.

Chapter 3

Iterative Methoden

Allgemein nennen wir ein Verfahren, das sukzessive (also *iterativ*) eine Lösung z über eine iterativ definierte Folge $x_{k+1} = \phi_k(x_k)$ annähert ein *iteratives Verfahren*.

Hierbei können z, x_k, x_{k+1} skalare, Vektoren oder auch unendlich dimensionale Objekte sein und ϕ_k ist die Verfahrensfunktion, die das Verfahren beschreibt. Oftmals ist das Verfahren immer das gleiche egal bei welchem Schritt k Jan gerade ist, weshalb auch oft einfach ϕ geschrieben wird.

Bekannte Beispiele sind iterative Verfahren zur

- Lösung linearer Gleichungssysteme (z.B. *Gauss-Seidel*)
- Lösung nichtlinearer Gleichungssysteme (z.B. *Newton*)
- Optimierung (z.B. von ML Modellen mittels *Gradientenabstieg*)

Der Einfachheit halber betrachten wir zunächst $z,\ x_k,\ x_{k+1} \in \mathbb{R}$. Die Erweiterung der Definitionen erfolgt dann über die Formulierung mit Hilfe passender Normen anstelle des Betrags.

Definition 3.1 (Konvergenz einer Iteration)**.** Eine Iteration die eine Folge $(x_k)_{k\in\mathbb{N}} \subset \mathbb{R}$ produziert, heißt *konvergent der Ordnung* p (gegen $z \in \mathbb{R}$) mit $p \geq 1$, falls eine Konstante $c > 0$ existiert sodass

$$|x_{k+1} - z| \leq c|x_k - z|^p, \tag{3.1}$$

für $k = 1, 2, ...$

Ist $p = 1$, so ist $0 < c < 1$ notwendig für Konvergenz, genannt *lineare Konvergenz* und das kleinste c, das (3.1) erfüllt, heißt *(lineare) Konvergenzrate*.

Gilt $p = 1$ und gilt $|x_{k+1} - z| \leq c_k|x_k - z|^p$ mit $c_k \to 0$ für $k \to \infty$ heißt die Konvergenz *superlinear*.

Wiederum gelten Konvergenzaussagen eigentlich für die Kombination aus Methode und Problem. Dennoch ist es allgemeine Praxis, beispielsweise zu sagen, dass das *Newton-Verfahren quadratisch konvergiert.*

Wir stellen fest, dass im Limit (und wenn vor allem $\phi_k \equiv \phi$ ist) gelten muss, dass

$$x = \phi(x),$$

die Lösung (bzw. das was berechnet wurde) ein *Fixpunkt* der Verfahrensfunktion ist.

In der Tat lassen sich viele iterative Methoden als Fixpunktiteration formulieren und mittels Fixpunktsätzen analysieren. Im ersten Teil dieses Kapitels, werden wir Fixpunktmethoden betrachten.

Als eine Verallgemeinerung, z.B. für den Fall dass ϕ tatsächlich von k abhängen soll oder dass kein Fixpunkt sondern beispielsweise ein Minimum angenähert werden soll, werden wir außerdem sogenannte *Auxiliary Function Methods* einführen und anschauen.

3.1 Iterative Methoden als Fixpunktiteration

Um eine iterative Vorschrift, beschrieben durch ϕ, als (konvergente) Fixpunktiteration zu charakterisieren, sind zwei wesentliche Bedingungen nachzuweisen

1. die gesuchte Lösung z ist ein Fixpunkt des Verfahrens, also $\phi(z) = z$.
2. Für einen Startwert x_0, konvergiert die Folge $x_{k+1} := \phi(x_k)$, $k = 1, 2, ...,$ gegen z.

Dazu kommen Betrachtungen von Konditionierung, Stabilität und Konvergenzordnung.

Wir beginnen mit etwas analytischer Betrachtung. Sei $g\colon \mathbb{R} \to \mathbb{R}$ stetig differenzierbar und sei $z \in \mathbb{R}$ ein Fixpunkt von g. Dann gilt, dass

$$\lim_{x \to z} \frac{g(x) - g(z)}{x - z} = \lim_{x \to z} \frac{g(x) - z}{x - z} = g'(z)$$

und damit, dass für ein x_k in einer Umgebung U um z gilt, dass

$$|g(x_k) - z| \leq c|x_k - z|$$

mit $c = \sup_{x \in U} |g'(x)|$. Daraus können wir direkt ableiten, dass

- wenn $|g'(z)| < 1$ ist, dann ist die Vorschrift $x_{k+1} = \phi(x_k) := g(x_k)$ *lokal* linear konvergent
- wenn $g'(z) = 0$ dann sogar *superlinear*
- wenn $|g'(z)| > 1$ ist, dann divergiert die Folge weg von z (und der Fixpunkt wird *abstoßend* genannt).

Für höhere Konvergenzordnungen wird diese Beobachtung im folgenden Satz verallgemeinert.

Theorem 3.1 (Konvergenz höherer Ordnung bei glatter Fixpunktiteration)**.** *Sei $g\colon D \subset \mathbb{R} \to \mathbb{R}$ p-mal stetig differenzierbar, sei $z \in D$ ein Fixpunkt von g. Dann konvergiert die Fixpunktiteration $x_{k+1} = g(x_k)$* lokal *mit Ordnung p, genau dann wenn*

$$g'(z) = \cdots g^{(p-1)}(z) = 0, \quad g^{(p)} \neq 0.$$

Proof. Siehe (Richter and Wick 2017, Thm. 6.31) □

> Das *genau dann wenn* in Satz 3.1 ist so zu verstehen, dass die Konvergenzordnung genau gleich p ist, was insbesondere beinhaltet, dass wenn $g^{(p)} = 0$ ist, die Ordnung eventuell grösser als p ist. (Jan ist verleitet zu denken, dass in diesem Fall die Iteration nicht (oder mit einer niedrigeren Ordnung) konvergieren würde).

Ist die Iterationsvorschrift linear (wie bei der iterativen Lösung linearer Gleichungssysteme), so ist die erste Ableitung ϕ' konstant (und gleich der Vorschrift selbst) und alle weiteren Ableitungen sind 0. Dementsprechend, können wir

- maximal lineare Konvergenz erwarten
- (die aber beispielsweise durch dynamische Anpassung von Parametern auf superlinear verbessert werden kann)
- dafür aber vergleichsweise direkte Verallgemeinerungen zu mehrdimensionalen und sogar ∞-dimensionalen Problemstellungen.

Zur Illustration betrachten wir den *Landweber-Algorithmus* zur näherungsweisen Lösung von "$Ax = b$". Dieser Algorithmus wird zwar insbesondere nicht verwendet um ein lineares Gleichungssystem zu lösen, durch die Formulierung für möglicherweise überbestimmte Systeme und die Verbindung zur iterativen Optimierung hat er aber praktische Anwendungen in *compressed sensing* und auch beim *supervised learning* gefunden; vgl. wikipedia:Landweber_iteration.

Definition 3.2 (Landweber Iteration)**.** Sei $A \in \mathbb{R}^{m\times n}$ und $b \in \mathbb{R}^m$. Dann ist, ausgehend von einem Startwert $x_0 \in \mathbb{R}^n$, die *Landweber Iteration* definiert über

$$x_{k+1} = x_k - \gamma A^T(Ax_k - b),$$

wobei der Parameter γ als $0 < \gamma < \frac{2}{\|A\|_2}$ gewählt wird.

Zur Illustration der Argumente, die die Konvergenz einer Fixpunktiteration mit linearer Verfahrensfunktion herleiten, zeigen wir die Konvergenz im Spezialfall, dass $Ax = b$ ein reguläres lineares Gleichungssystem ist.

Theorem 3.2 (Konvergenz der Landweber Iteration)**.** *Unter den Voraussetzungen von Definition 3.2 und für $m = n$ und $A \in \mathbb{R}^{n\times n}$ regulär, konvergiert die Landweber Iteration linear für einen beliebigen Startwert x_0.*

Proof. Ist das Gleichungssystem $Az = b$ eindeutig lösbar, bekommen wir direkt, dass

$$\begin{aligned} x_{k+1} - z &= x_k - \gamma A^T(Ax_k - b) - z \\ &= x_k - \gamma A^T A x_k - \gamma A^T b - z \\ &= (I - \gamma A^T A)x_k - \gamma A^T A z - z \\ &= (I - \gamma A^T A)(x_k - z) \end{aligned}$$

Damit ergibt eine Abschätzung in der 2-Norm und der induzierten Matrixnorm, dass

$$\|x_{k+1} - z\|_2 \leq \|I - \gamma A^T A\|_2 \|x_k - z\|_2$$

gilt, was lineare Konvergenz mit der Rate $c = \|I - \gamma A^T A\|_2$ bedeutet, wobei $c < 1$ gilt nach der getroffenen Voraussetzung, dass $0 < \gamma < \frac{2}{\|A^T A\|_2}$ ist. □

Das Prinzip dieser Beweise ist festzustellen, dass die Verfahrensfunktion in der Nähe des Fixpunkts eine *Kontraktion* ist, d.h. Lipschitzstetig mit Konstante $L < 1$.

3.2 Gradientenabstiegsverfahren

Anstelle der Nullstellensuche behandeln wir jetzt die Aufgabe

$$f(x) \to \min_{x \in \mathbb{R}^n}$$

für eine Funktion $f \colon \mathbb{R}^n \to \mathbb{R}$, also die Aufgabe ein $z^* \in \mathbb{R}^n$ zu finden, für welches der Wert von f minimal wird.

Ist f differenzierbar (der Einfachheit halber nehmen wir an, dass *totale* Differenzierbarkeit vorliegt; es würde aber Differenzierbarkeit in einer beliebigen Richtung, also *Gateaux*-Differenzierbarkeit, genügen), so gilt, dass in einem Punkt x_0, der Gradient $\nabla f(x_0)$ (ein Vektor im $\mathbb{R}^n$) in die Richtung des stärksten Wachstums zeigt und der negative Gradient $-\nabla f(x_0)$ in die Richtung, in der f kleiner wird.

Auf der Suche nach einem Minimum könnten wir also ausnutzen, dass

$$f(x_0 - \gamma_0 \nabla f(x_0)) := f(x_1) < f(x_0)$$

falls γ_0 nur genügend klein ist und $\nabla f(x_0) \neq 0$.

Was ist, wenn $\nabla f(x_0) = 0$ ist und warum gibt es andernfalls so ein γ_0 und wie könnten wir es systematisch bestimmen?

Diese Beobachtung am nächsten Punkt x_1 wiederholt, führt auf des *Gradientenabstiegsverfahren*.

Definition 3.3 (Gradientenabstiegsverfahren)**.** Sei $f\colon \mathbb{R}^n \to \mathbb{R}$ differenzierbar, dann heißt die Iteration

$$x_{k+1} := x_k - \gamma_k \nabla f(x_k) \tag{3.2}$$

für passend gewählte $\gamma_k > 0$, das Gradientenabstiegsverfahren zur Berechnung eines Minimums von f.

Lemma 3.1 (Gradientenabstieg als konvergente Fixpunkt Iteration)**.** *Sei $f\colon D \subset \mathbb{R}^n \to \mathbb{R}$ konvex und zweimal stetig differenzierbar auf D offen. Ist $z^* \in D$ ein Minimum von f und sei $\overline{\lambda}$ die* Lipschitz-Konstante *von ∇f, dann definiert (3.2) mit $\gamma_k \equiv \frac{2}{\overline{\lambda}}$ eine konvergente Fixpunktiteration für $\phi(x) = x - \gamma \nabla f(x)$ mit einem lokalen Minimum z^* von f als Fixpunkt.*

Proof. Vorlesung. □

Die vorhergegangenen Überlegungen gingen von z^* innerhalb eines offenen Definitionsbereichs D von f aus, wo ein Minimum durch $\nabla f(x^*) = 0$ und $f(x^*) \leq f(x)$ für alle x aus einer Umgebung von x^* gegeben ist.

Ein typischer Anwendungsfall ist jedoch, dass x^* in einem zulässigen Bereich C liegen muss, der eine echte Teilmenge von D ist. Dann besteht die Möglichkeit, dass ein (lokales) Minimum am Rand des Bereichs C vorliegt (wo die Funktion f zwar weiter fällt, aber "das Ende" der Zulässigkeit erreicht ist).

Ist $C \subset \mathbb{R}^n$ konvex und abgeschlossen, so gilt folgendes allgemeine Resultat (dessen Argumente und Voraussetzungen auch leicht auf beispielsweise Funktionen auf allgemeinen Hilberträumen oder Mengen die nur lokal konvex sind angepasst werden können).

Theorem 3.3 (Projiziertes Gradientenabstiegsverfahren)**.** *Sei $C \subset \mathbb{R}^n$ konvex und abgeschlossen, dann ist die Projektion $P_C\colon \mathbb{R}^n \to C$ mittels*

$$P_C(x) := x^*,$$

wobei x^* das Minimierungsproblem

$$\min_{z \in C} \|x - z\|_2$$

löst, wohldefiniert.

Sei ferner $f\colon D \to \mathbb{R}$, mit $C \subset D$, konvex und differenzierbar mit Lipschitzstetigem Gradienten mit Konstante L. Dann konvergiert das projizierte Gradientenabstiegsverfahren

$$x_{k+1} := P_C(x_k - \gamma_k \nabla f(x_k)) \tag{3.3}$$

für jeden Anfangswert $x_0 \in D$ und beliebige Wahl von $\gamma_k < \frac{1}{L}$ zu einem lokalen Minimum $z^* \in C$ von f.

Proof. Technisch… □

3.3 Auxiliary Function Methods

In manchen Fällen ist es hilfreich, wenn das Problem selbst iterativ definiert wird. Dann wird in jedem Schritt ein vereinfachtes Problem gelöst und mit der gewonnenen Information, kann das Problem dem eigentlichen aber schwierigen Originalproblem näher gebracht werden.

Als Beispiel betrachten wir das Problem

$$f(x) = x_1^2 + x_2^2 \to \min_{x \in D \subset \mathbb{R}^2}, \quad \text{wobei } D := \{x \in \mathbb{R}^2 \,|\, x_1 + x_2 \geq 0\}. \tag{3.4}$$

Zwar ist hier das projizierte Gradientenabstiegsverfahren unmittelbar anwendbar, wir werden aber sehen, dass wir mit einer Hilfsfunktion, sogar die analytische Lösung direkt ablesen können.

Für $k = 1, 2, ...$, sei das Hilfsproblem definiert als

$$B_k(x) := x_1^2 + x_2^2 - \frac{1}{k} \log(x_1 + x_2 - 1) \to \inf_C, \tag{3.5}$$

wobei $C = \{x \,|\, x_1 + x_2 > 0\}$. Aus dem "0-setzen" der partiellen Ableitungen von B_k, bekommen wir

$$x_{k,1} = x_{k,2} = \frac{1}{4} + \frac{1}{4}\sqrt{1 + \frac{4}{k}},$$

also eine Folge, die zum Minimum des eigentlichen Problems konvergiert.

Zur Analyse solcher Verfahren, allgemein geschrieben als

$$G_k(x) = f(x) + g_k(x) \to \min_C, \quad k = 1, 2, ... \tag{3.6}$$

werden die folgenden zwei Bedingungen gerne herangenommen:

1. Die Iteration (3.6) heißt *auxiliary function* (AF) Methode, falls $g_k(x) \geq 0$, für alle $k \in \mathbb{N}$ und $x \in C$, $g_k(x_{k-1}) = 0$.
2. Die Iteration gehört zur *SUMMA* Klasse, falls $G_k(x) - G_k(x_k) \geq g_{k+1}(x)$.

Unter der 1. Annahme gilt sofort, dass

$$f(x_k) \leq f(x_k) + g_k(x_k) = G_k(x_k) \leq G_k(x_{k-1}) = f(x_{k-1}) + g_k(x_{k-1}) = f(x_{k-1}),$$

also dass die Folge $\{f(x_k)\}_{k \in \mathbb{N}}$ monoton fallend ist.

Aus der 2. Annahme folgt dann, dass $f(x_k) \to \beta^* = \inf_{x \in c} f(x)$, für $k \to \infty$, was sich wie folgt argumentieren läßt:

Angenommen, $f(x_k) \to \beta > \beta^*$, dann existiert ein $z \in C$, sodass $\beta > f(z) \geq \beta^*$. Dann ist, der 2. 2. Annahme nach,

$$\begin{aligned} g_k(z) - g_{k+1}(z) &= g_k(z) - (G_k(z) - G_k(x_k)) \\ &= g_k(z) - (f(z) + g_k(z) - f(x_k) - g_k(x_k)) \\ &\geq f(z) - \beta + g_k(x_k) \geq f(z) - \beta > 0, \end{aligned}$$

was impliziert, dass $0 \leq g_{k+1}(z) < g_k(z)+c$, für alle k und eine konstantes $c > 0$, was ein Widerspruch ist.

Wir rechnen nach, dass (3.5) die Annahmen 1. und 2. erfüllt (allerdings erst nach einigen äquivalenten Umformungen).

Zunächst halten wir fest, dass die Iteration in (3.5) geschrieben werden kann als

$$B_k(x) = f(x) + \frac{1}{k}b(x) \to \min \tag{3.7}$$

was, da eine Skalierung das Minimum nicht änder ebensowenig wie die Addition eines konstanten Termes (konstant bezüglich x), äquivalent ist zu

$$G_k(x) = f(x) + g_k(x)$$

mit

$$g_k(x) = [(k-1)f(x) + b(x)] - [(k-1)f(x_{k-1}) + b(x_{k-1})].$$

Wir rechnen direkt nach, dass $g_k(x) \geq 0$ ist (folgt daraus, dass x_{k-1} optimal für G_{k-1} ist), dass $g_k(x_{k-1}) = 0$ ist, und dass $G_k(x) - G_k(x_k) = g_{k+1}(x)$ ist (dafür muss ein bisschen umgeformt werden), sodass die Voraussetzungen für AF und SUMMA erfüllt sind.

Zum Abschluss einige Bemerkungen

- das allgemeine b in (3.7) und im speziellen in (3.5) ist eine sogenannte *barrier* Funktion, die beispielsweise einen zulässigen Bereich als $C = \{x \,|\, b(x) < \infty\}$ definiert.
- weitere Methoden der Optimierung, die in die betrachteten (AF) Klassen fallen sind beispielsweise *Majorization Minimization*, *Expectation Maximization*, *Proximal Minimization* oder *Regularized Gradient Descent*.
- Eine schöne Einführung und Übersicht liefert das Skript *Lecture Notes on Iterative Optimization Algorithms* (Byrne 2014).

3.4 Übungen

1. Bestimmen Sie die Konvergenzordnung und die Rate für das Intervallschachtelungsverfahren zur Nullstellenberechnung.

2. Benutzen Sie Satz 3.1 um zu zeigen, dass aus f zweimal stetig differenzierbar und $f(z) = 0$, $f'(z) \neq 0$ für ein $z \in D(f)$ folgt, dass das Newton-Verfahren zur Berechnung von z lokal quadratisch konvergiert. *Hinweis*: Hier ist es wichtig zunächst zu verstehen, was die Funktion f ist und was die Verfahrensfunktion ϕ ist.

3. Bestimmen sie die Funktion h in $\phi(x) = x + h(x)f(x)$ derart, dass unter den Bedingungen von 2. die Vorschrift ϕ einen Fixpunkt in z hat und derart, dass die Iteration quadratisch konvergiert.

4. Erklären Sie an Hand von Satz 3.1 (und den vorhergegangenen Überlegungen) warum Newton für das Problem *finde x, so dass $x^2 = 0$ ist* **nicht** quadratisch (aber doch superlinear) konvergiert.
5. Beweisen Sie, dass für $0 < \gamma < \frac{2}{\|A^T A\|_2}$ gilt, dass$\|I - \gamma A^T A\|_2 < 1$ für beliebige $A \in \mathbb{R}^{m \times n}$.
6. Rechnen Sie nach, dass die Landweber Iteration aus Definition 3.2 einem gedämpften Gradientenabstiegsverfahren für $\|Ax - b\|_2^2 \to \min_{x \in \mathbb{R}^m}$ entspricht.
7. Implementieren Sie das projizierte Gradientenabstiegsverfahren für (3.4) und das *nichtprojizierte* aber an k angepasste Gradientenabstiegsverfahren für (3.5). Vergleichen Sie die Konvergenz für verschiedene Startwerte.

Chapter 4

Stochastisches Gradientenverfahren

Das stochastische Gradientenverfahren formuliert den Fall, dass im k-ten Schritt anstelle des eigentlichen Gradienten $\nabla f(x_k) \in \mathbb{R}^n$ eine Schätzung $g(x_k, \xi) \in \mathbb{R}^n$ vorliegt, die eine zufällige Komponente in Form einer Zufallsvariable ξ hat. Dabei wird angenommen, dass $g(x_k, \xi)$ *erwartungstreu* ist, das heißt

$$\mathbb{E}_\xi[g(x_k, \xi)] = \nabla f(x_k),$$

wobei $\mathbb{E}_\xi$ den Erwartungswert bezüglich der Variablen ξ beschreibt.

4.1 Motivation und Algorithmus

Im *Maschinellen Lernen* oder allgemeiner in der *nichtlinearen Regression* spielt die Minimierung von Zielfunktionalen in Summenform

$$Q(w) = \frac{1}{N} \sum_{i=1}^{N} Q_i(w)$$

eine Rolle, wobei der Parametervektor $w \in \mathbb{R}^n$, der Q minimiert, gefunden oder geschätzt werden soll. Jede der Summandenfunktionen Q_i ist typischerweise assoziiert mit einem i-ten Datenpunkt (einer Beobachtung) beispielsweise aus einer Menge von Trainingsdaten.

Sei beispielsweise eine parametrisierte nichtlineare Funktion $T_w \colon \mathbb{R}^m \to \mathbb{R}^n$ gegeben die durch Optimierung eines Parametervektors w an Datenpunkte $(x_i, y_i) \subset \mathbb{R}^n \times \mathbb{R}^m$, $i = 1, \ldots, N$, *gefittet* werden soll, ist die *mittlere quadratische Abweichung*

$$\mathsf{MSE}\,(w) := \frac{1}{N} \sum_{i=1}^{N} \|N(x_i) - y_i\|_2^2$$

genannt *mean squared error*, ein naheliegendes und oft gewähltes Optimierungskriterium.

Um obige Kriterien zu minimieren, würde ein sogenannter Gradientenabstiegsverfahren den folgenden Minimierungsschritt

$$w^{k+1} := w^k - \eta \nabla Q(w^k) = w^k - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla Q_i(w^k),$$

iterativ anwenden, wobei η die Schrittweite ist, die besonders in der *ML* community oft auch *learning rate* genannt wird.

Die Berechnung der Abstiegsrichtung erfordert hier also in jedem Schritt die Bestimmung von N Gradienten $\nabla Q_i(w^k)$ der Summandenfunktionen. Wenn N groß ist, also beispielsweise viele Datenpunkte in einer Regression beachtet werden sollen, dann ist die Berechnung entsprechend aufwändig.

Andererseits entspricht die Abstiegsrichtung

$$\frac{1}{N} \sum_{i=1}^{N} \nabla Q_i(w^k)$$

dem Mittelwert der Gradienten aller Q_is am Punkt w_k, der durch ein kleineres Sample

$$\frac{1}{N} \sum_{i=1}^{N} \nabla Q_i(w^k) \approx \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} \nabla Q_j(w^k),$$

angenähert werden könnte, wobei $\mathcal{J} \subset \{1, \dots, N\}$ eine Indexmenge ist, die den *batch* der zur Approximation gewählten Q_is beschreibt.

4.2 Stochastisches Abstiegsverfahren

Beim stochastischen (oder "Online") Gradientenabstieg wird der wahre Gradient von $Q(w^k)$ durch einen Gradienten bei einer einzelnen Probe angenähert:

$$w^{k+1} = w^k - \eta \nabla Q_j(w^k),$$

mit $j \in \{1, \dots, N\}$ zufällig gewählt (ohne zurücklegen).

Während der Algorithmus den Trainingssatz durchläuft, führt er die obige Aktualisierung für jede Trainingsprobe durch. Es können mehrere Durchgänge (sogenannte *epochs*) über den Trainingssatz gemacht werden, bis der Algorithmus konvergiert. Wenn dies getan wird, können die Daten für jeden Durchlauf gemischt werden, um Zyklen zu vermeiden. Typische Implementierungen verwenden zudem eine adaptive Lernrate, damit der Algorithmus überhaupt oder schneller konvergiert.

Die wesentlichen Schritte als Algorithmus sehen wie folgt aus:

```
#####################################################
# The basic steps of a stochastic gradient method #
#####################################################

w = ...  # initialize the weight vector
eta = ... # choose the learning rate
I = [1, 2, ..., N]  # the full index set

for k in range(number_epochs):
    J = shuffle(I)  # shuffle the indices
    for j in J:
        # compute the gradient of Qj at current w
        gradjk = nabla(Q(j, w))
        # update the w vector
        w = w - eta*gradjk
    if convergence_criterion:
       break
#################################################
```

Die Konvergenz des *stochastischen Gradientenabstiegsverfahren* als Kombination von *stochastischer Approximation* und *numerischer Optimierung* ist gut verstanden. Allgemein und unter bestimmten Voraussetzung lässt sich sagen, dass das stochastische Verfahren ähnlich konvergiert wie das *exakte Verfahren* mit der Einschränkung, dass die Konvergenz *fast sicher* stattfindet.

In der Praxis hat sich der Kompromiss etabliert, der anstelle des Gradienten eines einzelnen Punktes $\nabla Q_j(w_k)$, den Abstieg aus dem Mittelwert über mehrere Samples berechnet, also (wie oben beschrieben)

$$\frac{1}{N}\sum_{i=1}^{N}\nabla Q_i(w^k) \approx \frac{1}{|\mathcal{J}|}\sum_{j\in\mathcal{J}}\nabla Q_j(w^k).$$

Im Algorithmus wird dann anstelle der zufälligen Indices $j \in \{1,\dots,N\}$, über zufällig zusammengestellte Indexmengen $\mathcal{J} \subset \{1,\dots,N\}$ iteriert.

Da die einzelnen Gradienten $\nabla Q_j(w^K)$ unabhängig voneinander berechnet werden können, kann so ein *batch* Verfahren effizient auf Computern mit mehreren Prozessoren realisiert werden. Die Konvergenztheorie ist nicht wesentlich verschieden vom eigentlichen *stochastischen Gradientenabstiegsverfahren*, allerdings erscheint die beobachte Konvergenz weniger erratisch, da der Mittelwert statistische Ausreißer ausmitteln kann.

4.3 Konvergenzanalyse

Wir betrachten den einfachsten Fall wie im obigen Algorithmus beschrieben, dass im k-ten Schritt, die Schätzung

$$g(x_k, \xi) = g(x_k, i(\xi)) =: \nabla Q_{i(\xi)}(x_k)$$

also dass der Gradient von $\frac{1}{N}\nabla \sum Q_i$ geschätzt wird durch den Gradienten der $i(\xi)$-ten Komponentenfunktion, wobei $i(\xi)$ zufällig aus der Indexmenge $I = \{1, 2, \dots, N\}$ gezogen wird.

> Im folgenden Beweis wird verwendet werden, dass *zurückgelegt* wird, also dass im k-ten Schritt alle möglichen Indizes gezogen werden können. Das ist notwendig um zu schlussfolgern, dass
>
> $$\mathbb{E}_{i(\xi)}[g(x_k, k_\xi)] = \nabla Q(x_k)$$
>
> In der Praxis (und oben im Algorithmus) wir **nicht** zurückgelegt, es gilt also $I_{k+1} = I_k \setminus \{k_\xi\}$. Der Grund dafür ist, dass gerne gesichert wird, dass auch in wenig Iterationsschritten alle Datenpunkte *besucht* werden.

Und die Iteration lautet

$$x_{k+1} = x_k - \eta_k g(x_k, i(\xi)).$$

Theorem 4.1 (Konvergenz des stochastischen Gradientenabstiegsverfahren)**.** *Sei* $Q := \frac{1}{N}\sum_{i=1}^N Q_i$ *zweimal stetig differenzierbar und* streng konvex *mit* Modulus $m > 0$ *und es gebe eine Konstante* M *mit* $\frac{1}{N}\sum_{i=1}^N \|\nabla Q_i\|_2^2 \leq M$*. Ferner sei* x^* *das Minimum von* Q*. Dann konvergiert das einfache stochastische Gradientenabstiegsverfahren mit* $\eta_k \leq \frac{1}{km}$ *linear im Erwartungswert des quadrierten Fehlers, d.h. es gilt*

$$a_{k+1} := \frac{1}{2}\mathbb{E}[\|x_{k+1} - x^*\|^2] \leq \frac{C}{k+1}$$

für eine Konstante C*.*

Proof. Streng konvex mit Modulus $m > 0$ bedeutet, dass alle Eigenwerte der Hessematrix H_Q größer als m sind. Insbesondere gilt, dass

$$Q(z) \leq Q(x) + \nabla Q(x)^T(z - x) + \frac{1}{2}m\|z - x\|^2$$

für alle z und x aus dem Definitionsbereich von Q.

Zunächst erhalten wir aus der Definition der 2-norm, dass

$$\begin{aligned}\frac{1}{2}\|x_{k+1}-x^*\|^2 &= \frac{1}{2}\|x_k-\eta_k\nabla Q_{i(k;\xi)}(x_k)-x^*\|^2\\ &= \frac{1}{2}\|x_k-x^*\|^2-\eta_k\nabla Q_{i(k;\xi)}(x_k)^T(x_k-x^*)+\eta_k^2\|\nabla Q_{i(k;\xi)}(x_k)\|^2\end{aligned}$$

Im nächsten Schritt nehmen wir den Erwartungswert dieser Terme. Dabei ist zu beachten, dass auch die x_k zufällig (aus der Sequenz der zufällig gezogenen Richtungen) erzeugt wurden. Dementsprechend müssen wir zwischen $\mathbb{E}$ (als Erwartungswert bezüglich aller bisherigen zufälligen Ereignisse für $\ell = 0, 1, \dots, k-1$) und zwischen $\mathbb{E}_{i(k;\xi)}$ (was wir im k-ten Schritt bezüglich der aktuellen Auswahl der Richtung erwarten können) unterscheiden.

In jedem Fall ist der Erwartungswert eine lineare Abbildung, sodass wir die einzelnen Terme der Summe separat betrachten können.

Für den Mischterm erhalten wir

$$\eta_k\mathbb{E}[\nabla Q_{i(k;\xi)}(x_k)^T(x_k-x^*)] = \eta_k\mathbb{E}[\mathbb{E}_{i(k;\xi)}[\nabla Q_{i(k;\xi)}(x_k)^T(x_k-x^*)\,|\,x_k]]$$

wobei der innere Term die Erwartung ist unter der Bedingung das x_k eingetreten ist (folgt aus dem Satz der *iterated expectation*). Da im inneren Term nur noch die Wahl von i zufällig ist und wegen der Linearität des Erwartungswertes bekommen wir

$$\begin{aligned}\mathbb{E}_{i(k;\xi)}[\nabla Q_{i(k;\xi)}(x_k)^T(x_k-x^*)\,|\,x_k] &= \mathbb{E}_{i(k;\xi)}[\nabla Q_{i(k;\xi)}(x_k)^T\,|\,x_k](x_k-x^*)\\ &= \nabla Q(x_k)^T(x_k-x^*).\end{aligned}$$

sodass mit der m-Konvexität gilt dass

$$\mathbb{E}[\nabla Q_{i(k;\xi)}(x_k)^T(x_k-x^*)] \geq m\mathbb{E}[\|x_k-x^*\|^2]; \tag{4.1}$$

vergleiche die Übungsaufgabe unten.

> Diese Manipulation mit den Erwartungswerten ist der formale Ausdruck dafür, dass, egal woher das x_k kam, die zufällige Wahl der aktuellen Richtung führt im statistischen Mittel auf $\nabla Q(x_k)$.

Mit gleichen Argumenten und der Annahme der Beschränktheit $\|\nabla Q(x)\|^2 < M$, bekommen wir für die erwartete quadratische Abweichung a_k, dass

$$\frac{1}{2}\|x_{k+1}-x^*\|^2 \leq (1-2\eta_k m)\frac{1}{2}\|x_k-x^*\| + \frac{1}{2}\eta_k^2M^2$$

beziehungsweise

$$a_{k+1} \leq (1-2\eta_k m)a_k + \frac{1}{2}\eta_k^2M.$$

Insbesondere wegen des konstanten Terms in der Fehlerrekursion, bedarf es bis zur $1/k$-Konvergenz weiterer Abschätzungen. Wir zeigen induktiv, dass für $\eta_k < \frac{1}{km}$ gilt, dass

$$a_{k+1} \leq \frac{c}{2(k+1)}, \quad c = \max\{\|x_1 - x^*\|^2, \frac{M^2}{m^2}\}.$$

Für $k = 0$ gilt die Abschätzung direkt. Für $k \geq 1$ gilt mit $\eta_k < \frac{1}{mk}$

$$\begin{aligned} a_{k+1} \leq & (1 - 2m\eta_k)a_k + \frac{1}{2}\eta_k^2 M \\ \leq & (1 - \frac{2}{k})a_k + \frac{1}{2}\frac{M}{k^2 m^2} \\ \leq & (1 - \frac{2}{k})\frac{c}{2} + \frac{1}{2}\frac{1}{k^2}\frac{c}{2} \\ = & \frac{k^2 - 1}{k^2}\frac{c}{2(k+1)} \leq \frac{c}{2(k+1)} \end{aligned}$$

sodass der Beweis erbracht ist mit $C := \frac{c}{2}$. □

Zum Abschluss schätzen wir noch aus der erhaltenen Konvergenzart und –rate, wie lange iteriert werden muss um den Fehler unter einen vorgegebenen Wert ϵ zu bekommen.

Dazu sei e_n der Fehler nach der n-ten Iteration und entsprechend e_0 der Fehler zum Startwert.

1. Für lineare Konvergenz gilt $e_n \leq q e_{n-1} \leq q^n e_0$ und damit

$$q^n e_0 = e_n < \epsilon \quad \leftrightarrow \quad n > \frac{\log \epsilon - \log e_0}{\log q}$$

2. Für quadratische Konvergenz folgt aus $e_n \leq a e_{n-1}^2 \leq a^n e_0^{2n}$, dass

$$a^n e_0^{2n} = e_n < \epsilon \quad \leftrightarrow \quad n > \frac{\log \epsilon}{\log a + 2 \log e_0}$$

3. Für "$1/k$" mit $e_n \leq \frac{C}{n}$ gilt dann

$$e_n < \epsilon \leftrightarrow n > \frac{C}{\epsilon}$$

Wir lesen ab, dass für lineare Konvergenz der Startwert nur entscheidend für die Anzahl der Iteration, während für quadratische Konvergenz $\log a + 2 \log e_0 < 0 \leftrightarrow a < \sqrt{e}_0$ wichtig ist um überhaupt Konvergenz zu haben. Abschließend zu bemerken ist dass, unter den getätigten Annahmen, für den stochastischen Gradientenabstieg der **quadratische Fehler** mit "1/k"

konvergiert. Dementsprechend muss entsprechend von $n \sim \frac{1}{\epsilon^2}$ ausgegangen werden.

Diese schlechte Konvergenz ist auch ein Grund dafür, dass das Lernen von neuronalen Netzen sehr rechenintensiv ist. Abhilfe schaffen hier Algorithmen, die Richtungsinformationen *2. Ordnung* einbeziehen (z.B. über ein Momentum wie im *ADAM* Algorithmus) sowie der Rückgriff auf *low-precision* Arithmetik (was naheliegend ist, wenn kleine ϵs ohnehin quasi unerreichbar sind).

4.4 Übungen

1. Eine Funktion heißt L-glatt (L-*smooth*) wenn sie stetig differenzierbar ist und der Gradient *Lipschitz*-stetig mit Konstante L ist. Zeigen Sie, dass für ein L-glatte Funktion, die zweimal differenzierbar ist, gilt:
   1. $f(y) \leq f(x) + \nabla f(x)^T(y-x) + \frac{L}{2}\|y-x\|^2$ für alle x und y aus dem Definitionsbereich.
   2. $-LI \leq H_f(x) \leq LI$ für alle x, für die Hesse-Matrix H_f und für "$\leq$" im Sinne der *Loewner-Halbordnung* definiter Matrizen.
2. Zeigen Sie, dass aus m-*Konvexität* von $Q\colon \mathbb{R}^n \to \mathbb{R}$ und $\mathbb{E}_\xi[g(\xi)] = \nabla Q(x)$ folgt, dass im Minimum x^* von Q gilt, dass
   $$\mathbb{E}_\xi[g(\xi)^T(x-x^*)] \geq Q(x) - Q(x^*) + \frac{m}{2}\|x-x^*\|^2 \geq m\|x-x^*\|^2,$$
   für alle x.
3. ((super)-Quadratische Kovergenz für glatte konvexe Funktionen) Sei $f\colon \mathbb{R}^n \to \mathbb{R}$ konvex und L-glatt und sei x^* die Lösung von $f(x) \to \min$. Zeigen Sie, dass Gradientenverfahren mit der Schrittweite $\frac{1}{L}$ eine Folge $\{x_k\}_{k\in\mathbb{N}} \subset \mathbb{R}^n$ erzeugt für die gilt
   $$f(x_N) - f(x^*) \leq \frac{L}{2T}\|x_0 - x^*\|^2, \quad N = 1, 2, \dots .$$
4. Berechnen sie näherungsweise den Gradienten der Beispielfunktion
   $$f(x_1, x_2, x_3) = \sin(x_1) + x_3\cos(x_2) - 2x_2 + x_1^2 + x_2^2 + x_3^2$$
   im Punkt $(x_1, x_2, x_3) = (1, 1, 1)$, indem sie die partiellen Ableitungen durch den Differenzenquotienten, z.B.,
   $$\frac{\partial g}{\partial x_2}(1,1,1) \approx \frac{g(1, 1+h, 1) - g(1,1,1)}{h}$$
   für $h \in \{10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}\}$ berechnen. Berechnen sie auch die Norm der Differenz zum exakten Wert von $\nabla g(1,1,1)$ (s.o.) und interpretieren sie die Ergebnisse.

Hier schon mal ein Codegerüst.

```
import numpy as np

def gfun(x):
    return np.sin(x[0]) + x[2]*np.cos(x[1]) \
        - 2*x[1] + x[0]**2 + x[1]**2 \
        + x[2]**2

def gradg(x):
    return np.array([np.NaN,
                     -x[2]*np.sin(x[1]) - 2 + 2*x[1],
                     np.NaN]).reshape((3, 1))

# Inkrement
h = 1e-3

# der x-wert und das h-Inkrement in der zweiten Komponente
xzero = np.ones((3, 1))
xzeroh = xzero + np.array([0, h, 0]).reshape((3, 1))

# partieller Differenzenquotient
dgdxtwo = 1/h*(gfun(xzeroh) - gfun(xzero))
# Alle partiellen Ableitungen ergeben den Gradienten
hgrad = np.array([np.NaN, dgdxtwo, np.NaN]).reshape((3, 1))

# Analytisch bestimmter Gradient
gradx = gradg(xzero)

# Die Differenz in der Norm
hdiff = np.linalg.norm((hgrad-gradx)[1])
# bitte alle Kompenenten berechnen
# und dann die Norm ueber den ganzen Vektor nehmen

print(f'h={h:.2e}: diff in gradient {hdiff.flatten()[0]:.2e}')
```

Chapter 5

Nachklapp

Ein paar lose Beispiele wo Numerik und maschinelles Lernen sich treffen.

- Iterative Methoden
  - Konvergenz/Konvergenzraten
  - stochastische Konvergenz
  - lokale Extrema
  - randomisierte Methoden
- Optimierung/Ausgleichsrechnung
- Approximationstheorie
  - Universal Approximation Theorem
- Stabilität und Fehleranalyse
  - mixed precision Arithmetik
- Numerische lineare Algebra
  - PCA
  - Support Vector Machines
  - Empfehlungssysteme
- Automatisches Differenzieren
  - *backward propagation* zur Gradientenberechnung

Referenzen

Byrne, C.L.: Lecture notes on iterative optimization algorithms, https://faculty.uml.edu/cbyrne/IOIPNotesOct2014.pdf, (2014)

Nocedal, J., Wright, S.J.: Numerical optimization. Springer (2006)

Richter, T., Wick, T.: Einführung in die numerische Mathematik. Begriffe, Konzepte und zahlreiche Anwendungsbeispiele. Heidelberg: Springer Spektrum (2017)