# 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 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 $\delta x$ 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 $\phi_k$ 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 $\phi$ 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 $\phi$ 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 $\phi$, 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 $\phi'$ 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 $\infty$-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 $\gamma$ 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 $\gamma_0$ 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 $\gamma_0$ 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 $\nabla f$, 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 $\phi$ ist.

3. Bestimmen sie die Funktion $h$ in $\phi(x) = x + h(x)f(x)$ derart, dass unter den Bedingungen von 2. die Vorschrift $\phi$ 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 $\xi$ 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 $\xi$ 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 $\eta$ 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 $Q_i$s 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 $Q_i$s 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* $\eta_k := \frac{1}{km}$ *linear im Erwartungswert des quadrierten Fehlers, d.h. es gilt*

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

*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

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

beziehungsweise

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

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

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

Für $k = 1$ (und damit $\eta_k = \frac{1}{m}$) gilt die Abschätzung da

$$a_2 \leq (1-2)a_1 + \frac{1}{2}\frac{1}{m^2}M = (-1)\frac{1}{2}\|x_i - x^*\|^2 + \frac{1}{2}\frac{M}{m^2} < \frac{M}{2m^2} \leq \frac{c}{2}.$$

Für $k \geq 2$ gilt mit $\eta_k = \frac{1}{mk}$

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

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 $\epsilon$ 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 $e_n \leq qe_{n-1} \leq q^n e_0$ 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 $e_n \leq ae_{n-1}^2 \leq a^n e_0^{2n}$, 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 $\epsilon$s 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

# Ein NN Beispiel

Zur Illustration der Konzepte, betrachten wir ein Beispiel in dem ein einfaches *Neuronales Netz* für die Klassifizierung von Pinguinen anhand vierer Merkmale aufgesetzt, trainiert und auf Funktion geprüft wird.

## 5.1 Der PENGUINS Datensatz

Die Grundlage ist der *Pinguin Datensatz*, der eine gern genommene Grundlage für die Illustration in der Datenanalyse ist. Die Daten wurden von Kristen Gorman erhoben und beinhalten 4 verschiedene Merkmale (engl. *features*) von einer Stichprobe von insgesamt 344[1]Pinguinen die 3 verschiedenen Spezies zugeordnet werden können oder sollen (Fachbegriff hier: *targets*). Im Beispiel werden die Klassen mit `0, 1, 2` codiert und beschreiben die Unterarten *Adele*, *Gentoo* und *Chinstrap* der Pinguine. Die Merkmale sind gemessene Länge und Höhe des Schnabels (hier *bill*), die Länge der Flosse (*flipper*) sowie das Köpergewicht [2] (*body mass*).

Wir stellen uns die Frage:

> Können wir aus den Merkmalen (*features*) die Klasse (*target*) erkennen und wie machen wir gegebenenfalls die Zuordnung?

In höheren Dimensionen ist schon die graphische Darstellung der Daten ein Problem. Wir können aber alle möglichen 2er Kombinationen der Daten in 2D plots visualisieren.

Ein Blick auf die Diagonale zeigt schon, dass manche Merkmale besser geeignet als andere sind, um die Spezies zu unterscheiden. Allerdings reichen (in dieser

[1] allerdings mit 2 unvollständigen Datenpunkten, die ich entfernt habe für unsere Beispiele

[2] Im Originaldatensatz ist das Gewicht in Gramm angegeben, um die Daten innerhalb einer 10er Skala zu haben, habe ich das Gewicht auf in kg umgerechnet

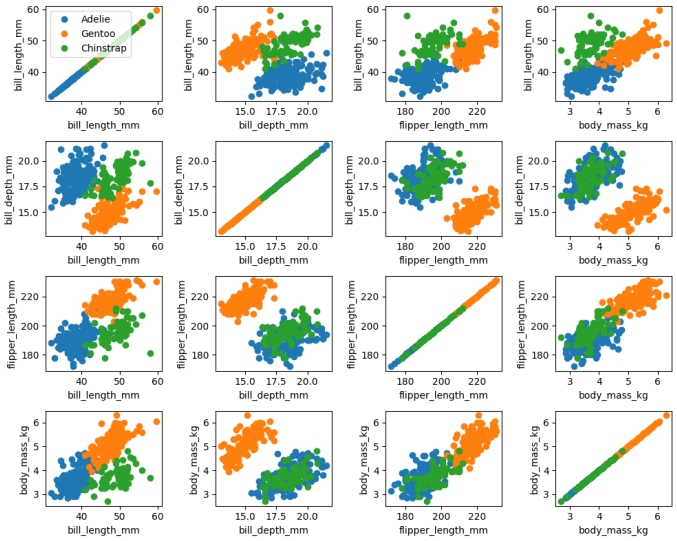


Figure 5.1: Pinguin Datenset 2D plots

linearen Darstellung) zwei Merkmale nicht aus, um eine eindeutige Diskriminierung zu erreichen.

## 5.2 Ein *2*-Layer Neuronales Netz zur Klassifizierung

Wir definieren ein neuronales Netzes $\mathcal{N}$ mit einer *hidden layer* als

$$\eta_i = \mathcal{N}(x_i) := \tanh(A_2 \tanh(A_1 x_i + b_1) + b_2),$$

das für einen Datenpunkt $x_i \in \mathbb{R}^{n_0}$ einen Ergebniswert $\eta_i \in \mathbb{R}^{n_2}$ liefert. Die sogenannten Gewichte $A_1 \in \mathbb{R}^{n_1 \times n_0}$, $b_1 \in \mathbb{R}^{n_1}$, $A_2 \in \mathbb{R}^{n_2, n_1}$, $b_2 \in \mathbb{R}^{n_2}$ parametrisieren diese Funktion. Eine Schicht besteht aus der einer affin-linearen Abbildung und einer *Aktivierungsfunktion* die hier als tanh gewählt wird und die komponentenweise angewendet wird.

Wir werden $n_0 = 4$ (soviele Merkmale als Eingang) und $n_2 = 1$ (eine Entscheidungsvariable als Ausgang) setzen und das Netzwerk so trainieren, dass anhand der gemessenen Daten $x_i$ die bekannte Pinguin Population `penguin-data.json` in zwei Gruppen aufgeteilt werden, wobei in der ersten Gruppe eine Spezies enthalten ist und in der anderen die beiden anderen Spezies.

Dazu kann eine Funktion $\ell\colon X \mapsto \{-1, 1\}$ definiert werden, die die bekannten Pinguine $x_i$ aus dem Datensatz $X$ ihrer Gruppe zuordnet. Dann können die

Koeffizienten von $\mathcal{N}$ über das Optimierungsproblem

$$\frac{1}{|X|}\sum_{x_i \in X} \|\ell(x_i) - \mathcal{N}(x_i)\|^2 \to \min_{A_1, b_1, A_2, b_2}$$

mittels des *stochastischen (batch) Gradientenabstiegs* bestimmt werden.

Zur Optimierung wird typischerweise ein Teil (z.B. 90%) der Datenpunkte verwendet über die mehrfach (in sogenannten *epochs*) iteriert wird.

Danach kann mittels der verbliebenen Datenpunkte getestet werden, wie gut das Netzwerk Daten interpretiert, die es noch nicht "gesehen" hat.

## 5.3 Beispiel Implementierung

Für ein 2-layer Netzwerk zur Klassifizierung der Pinguine.

Hier ein python file oder ein ipython file sowie die Pinguin Daten zum Direktdownload.

### 5.3.1 Setup

Wir importieren die benötigten Module und laden die Daten.

```
# import the required modules
import json
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import approx_fprime  # we will use this function to compute gradients

# load the data
with open('penguin-data.json', 'r') as f:
    datadict = json.load(f)
# turn it into a numpy array
data = np.array(datadict['data'])
data = data - data.mean(axis=0)  # center the data
# extract the labels
lbls = np.array(datadict['target'])
```

In diesem Beispiel unterscheiden wir nur zwei Gruppen. Wir teilen die ersetzen die eigentlichen *labels* `[0, 1, 2]` durch die zwei *lables* `[-1, 1]`.

```
# a dictionary that maps the labels(=targets) of the data into labels {1, -1}
# that will use for distinction of two groups
mplbldict = {0: np.array([1]),
             1: np.array([1]),
             2: np.array([-1])}
print('our two groups: \n', [f'{datadict["target_names"][lblid]} --> {mplbldict[lblid].item()}' f
```

Als nächstes legen wir die Dimensionen der *layers* fest und damit auch die Größe der Gewichtsmatrizen. Bei unserem 2-layer Netzwerk, bleibt uns da nur die Größe der mittleren Schicht, da die Eingangsdimension durch die Daten und die Ausgangsdimension durch unsere Wahl, wie wir entscheiden wollen, bereits festgelegt ist.

```
# sizes of the layers
sxz, sxo, sxt = data.shape[1], 2, mplbldict[0].size
# defines also the sizes of the weightmatrices
```

Zuletzt noch die Parameter, die das *training* definieren.

- `batchsize` – über wieviele Samples wird der stochastische Gradient bestimmt
- `lr` – *learning rate* – die Schrittweite
- `epochs` – wie oft wird über die Daten iteriert

und dann wie gross der Anteil und was die Indizes der Trainings– beziehungsweise Testdaten sind

```
# parameters for the training -- these worked fine for me
batchsize = 30  # how many samples for the stochastic gradients
lr = 0.125  # learning rate
epochs = 1000  # how many gradient steps

# the data
traindataratio = .9     # the ratio of training data vs. test data
ndata = data.shape[0]   # number of datapoints
trnds = int(ndata*traindataratio)
allidx = np.arange(ndata)                                    # indices of all data
trnidx = np.random.choice(allidx, trnds, replace=False)      # training ids
tstidx = np.setdiff1d(allidx, trnidx)                        # test ids
```

### 5.3.2 Neural Network Evaluation Setup

Hier definieren wir das Netzwerk als Funktion der Parameter und die *loss function*, die misst wie gut das Netzwerk die Daten wiedergibt und die Grundlage fuer die Optimierung ist.

```
def fwdnn(xzero, Aone=None, bone=None, Atwo=None, btwo=None):
    ''' definition/(forward)evaluation of a neural networks of two layers

    '''
    xone = np.tanh(Aone @ xzero + bone)
    xtwo = np.tanh(Atwo @ xone + btwo)
    return xtwo
```

```
def sqrdloss(weightsvector, features=None, labels=None):
    ''' compute the sqrd `loss`
```

```
    || NN(x_i) - y_i ||^2

    given the vector of weights for a given data point (features)
    and the corresponding label
    '''

    Aone, bone, Atwo, btwo = wvec_to_wmats(weightsvector)
    # compute the prediction
    nnpred = fwdnn(features, Aone=Aone, bone=bone, Atwo=Atwo, btwo=btwo)
    return np.linalg.norm(nnpred - labels)**2
```

An sich liegen die Parameter als Matrizen vor. Da jedoch die Theorie (und auch die praktische Implementierung) einen Parameter**vektor** voraussetzt, entrollen wir die Matrizen und stecken sie in einen grossen Vektor. Dann muessen wir noch an der richtigen Stelle wieder die Matrizen aus dem Vektor extrahieren; was die folgende Funktion realisiert.

```
def wvec_to_wmats(wvec):
    ''' helper to turn the vector of weights into the system matrices

    '''
    Aone = wvec[:sxz*sxo].reshape((sxo, sxz))
    cidx = sxz*sxo
    bone = wvec[cidx:cidx+sxo]
    cidx = cidx + sxo
    Atwo = wvec[cidx:cidx+sxo*sxt].reshape((sxt, sxo))
    cidx = cidx + sxo*sxt
    btwo = wvec[cidx:]
    if Aone.size + bone.size + Atwo.size + btwo.size == wvec.size:
        return Aone, bone, Atwo, btwo
    else:
        raise UserWarning('mismatch weightsvector/matrices')
```

### 5.3.3 Das Training

Der Parametervektor ("die Gewichte") werden zufällig initialisiert und dann mit dem stochastischen Gradienten in mehreren Epochen optimiert.

**Bemerkung**: Hier benutzen wir `scipy.optimize.approx_fprime` um den Gradienten numerisch zu bestimmen. Das ist hochgradig ineffizient. "Richtige" Implementierungen von *Machine Learning* Bibliotheken benutzen anstelle *Automatisches Differenzieren* für eine sowohl schnelle und als auch akkurate Berechnung des Gradienten.

```
# initialization of the weights
wini = np.random.randn(sxo*sxz + sxo + sxt*sxo + sxt)
```

```
gradnrml = []  # list of norm of grads for plotting later

cwghts = wini  # the current state of the weight vector
for kkk in range(epochs):
    cids = np.random.choice(trnidx, batchsize, replace=False)
    cgrad = np.zeros(wini.shape)
    for cid in cids:
        itrgts = data[cid, :]
        ilabls = mplbldict[lbls[cid]]
        cgrad = cgrad + approx_fprime(cwghts, sqrdloss, 1e-8,
                                      itrgts, ilabls)
    cwghts = cwghts - lr*1/batchsize*cgrad  # the upgrade
    gradnrml.append(1/batchsize*np.linalg.norm(cgrad))
    if np.mod(kkk, 50) == 0:
        print(f'k={kkk}: norm of gradient: {np.linalg.norm(cgrad)}')
```

```
plt.figure()
plt.semilogy(gradnrml, label='norm of gradient estimate')
plt.xlabel('$k$-th stochastic gradient step')
plt.legend()
plt.show()
```

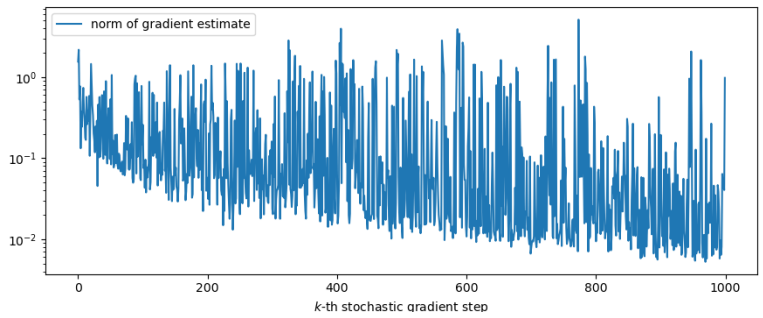


Figure 5.2: Beispiel Konvergenz des Stochastischen Gradienten

Wir koennen eine gewisse Konvergenz beobachten (sichtbar an der unteren Kante) aber auch ein typisches stochastisches Verhalten.

### 5.3.4 Das Auswerten

Wir nehmen das Ergebnis der letzten Iteration als *beste Parameter*, definieren damit das Neuronale Netz, und testen auf den übriggebliebenen Daten das Ergebnis.

```
optwghts = cwghts  # the optimal weights
Aonex, bonex, Atwox, btwox = wvec_to_wmats(optwghts)
```

```
print('***** testing the classification *****')
faillst = []
for cti in tstidx:  # iteration over the test data points
    itrgt = data[cti, :]
    ilbl = mplbldict[lbls[cti]]
    # the prediction of the neural network
    nnlbl = fwdnn(itrgt, Aone=Aonex, bone=bonex, Atwo=Atwox, btwo=btwox)
    sccs = np.sign(ilbl) == np.sign(nnlbl)
    print(f'label: {ilbl.item()} -- nn: {nnlbl.item():.4f} -- success: {sccs}')
    if not sccs:
        faillst.append((cti, ilbl.item(), nnlbl.item(),
                        datadict['target_names'][lbls[cti]]))
    else:
        pass
```

```
print('\n***** Results *****')
print(f'{100-len(faillst)/tstidx.size*100:.0f}% was classified correctly')
print('***** Misses *****')
if len(faillst) == 0:
    print('None')
else:
    for cfl in faillst:
        cid, lbl, nnlbl, name = cfl
        print(f'ID: {cid} ({name} pinguin) was missclassified ' +
              f'with score {nnlbl:.4f} vs. {lbl}')
```

# Chapter 6

# Singulärwert Zerlegung

Die Singulärwertzerlegung ist ein Universalwerkzeug der Datenanalyse und Modellsynthese. Die wesentliche Eigenschaft ist die Quantifizierung wesentlicher und redundanter Anteile in Daten oder Operatoren.

Die direkte Anwendung ist die *Principal Component Analysis*, die orthogonale Dimensionen in multivariablen Daten identifiziert, die nach der Stärke der Varianz sortiert sind. So kann diese erste *principal component* als die *reichhaltigste* Datenrichtung interpretiert werden und die letzten Richtungen (insbesondere wenn die Varianz komplett verschwindet) als wenig aussagekräftig (und insbesondere redundant) identifiziert werden.

Andere Anwendungen ist die Lösung von überbestimmten Gleichungssystemen (wie sie in der linearen Regression vorkommen) oder das Entfernen von *Rauschen* aus Daten.

## 6.1 Definition und Eigenschaften

**Theorem 6.1** (Singulärwertzerlegung (SVD))**.** *Sei $A \in \mathbb{C}^{m\times n}$, $m \geq n$. Dann existieren orthogonale Matrizen $U \in \mathbb{C}^{m\times m}$ und $V \in \mathbb{C}^{n\times n}$ und eine Matrix $\Sigma \in \mathbb{R}^{m\times n}$ der Form*

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \sigma_n \\ 0 & 0 & \dots & 0 \\ \vdots & \ddots & & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

*mit reellen sogenannten* Singulärwerten

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

*sodass gilt*

$$A = U\Sigma V^*$$

*wobei gilt* $V^* = \overline{V^T}$ *(transponiert und komplex konjugiert).*

Ein paar Bemerkungen.

- Ist $A$ reell, können auch $U$ und $V$ reell gewählt werden.
- Ein Beweis ist in (Bollhöfer and Mehrmann 2004, Satz 14.14) zu finden.
- Die Annahme $m \geq n$ war nur nötig um für die Matrix $\Sigma$ keine Fallunterscheidung zu machen. (Für $m \leq n$ "steht der Nullblock rechts von den Singulärwerten"). Insbesondere gilt $A^* = V\Sigma U^*$ ist eine SVD von $A^*$.
- Eine Illustration der Zerlegung ist Abbildung 6.1 zu sehen.

Wir machen einige Überlegungen im Hinblick auf große Matrizen. Sei dazu $m > n$, $A \in \mathbb{C}^{m\times n}$ und $A = U\Sigma V^*$ eine SVD wie in Theorem 6.1. Sei nun

$$U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}$$

partitioniert sodass $U_1$ die ersten $n$ Spalten von $U$ enthält.

Dann gilt (nach der Matrix-Multiplikations Regel *Zeile mal Spalte* die Teile $U_2$ und $V_2$ immer mit dem Nullblock in $\Sigma$ multipliziert werden) dass

$$A = U\Sigma V = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \hat{\Sigma} \\ 0 \end{bmatrix} V^* = U_1 \hat{\Sigma} V^*$$

Es genügt also nur die ersten $m$ Spalten von $U$ zu berechnen. Das ist die sogenannte **slim SVD**.

Hat, darüberhinaus, die Matrix $A$ keinen vollen Rang, also $\mathrm{Rg}(A) = r < n$, dann:

- ist $\sigma_i = 0$, für alle $i = r+1, \ldots, n$, (wir erinnern uns, dass die Singulärwerte nach Größe sortiert sind)
- die Matrix $\hat{\Sigma}$ hat $n - r$ Nullzeilen
- für die Zerlegung sind nur die ersten $r$ Spalten von $U$ und $V$ relevant – die sogenannte **Kompakte SVD**.

In der Datenapproximation ist außerdem die **truncated SVD** von Interesse. Dazu sei $\hat{r} < r$ ein beliebig gewählter Index. Dann werden alle Singulärwerte, $\sigma_i = 0$, für alle $i = \hat{r} + 1, \ldots, n$, abgeschnitten – das heißt null gesetzt und die entsprechende *kompakte SVD* berechnet.

Hier gilt nun nicht mehr die Gleichheit in der Zerlegung. Vielmehr gilt

$$A \approx A_{\hat{r}}$$

wobei $A_{\hat{r}}$ aus der *truncated SVD* von $A$ erzeugt wurde. Allerdings ist diese Approximation von $A$ durch optimal in dem Sinne, dass es keine Matrix vom

Rang $\hat{r} \geq r = \mathrm{Rg}(A)$ gibt, die $A$ (in der *induzierten* euklidischen Norm) besser approximiert. Es gilt

$$\min_{B \in \mathbb{C}^{m \times n}, \mathrm{Rg}(B) = \hat{r}} \|A - B\|_2 = \|A - A_{\hat{r}}\|_2 = \sigma_{\hat{r}+1};$$

(vgl. Satz 14.15, Bollhöfer and Mehrmann 2004).

Zum Abschluss noch der Zusammenhang zur *linearen Ausgleichsrechnung*. Die Lösung $w$ des Problems der *linearen Ausgleichsrechnung* war entweder als Lösung eines Optimierungsproblems

$$\min_w \|Aw - y\|^2$$

oder als Lösung des linearen Gleichungssystems

$$A^T A w = y.$$

Ist $A = U\Sigma V^* = U_1 \hat{\Sigma} V^*$ (slim) "SV-zerlegt", dann gilt

$$A^* A w = V \hat{\Sigma}^* U_1^* U_1 \hat{\Sigma} V^* w = V \hat{\Sigma}^2 V^* w$$

und damit

$$A^* A w = A^* y \quad \Leftrightarrow \quad V \hat{\Sigma}^2 V^* w = V \hat{\Sigma}^* U_1^* y \quad \Leftrightarrow \quad w = V \hat{\Sigma}^{-1} U_1^* y$$

was wir (mit den Matrizen der vollen SVD) als

$$w = V \Sigma^+ U^* y$$

schreiben, wobei

$$\Sigma^+ = \begin{bmatrix} \hat{\Sigma}^{-1} \\ 0_{m-n \times n} \end{bmatrix}$$

.

**Bemerkung**: $\Sigma^+$ kann auch definiert werden, wenn $\hat{\Sigma}$ nicht invertierbar ist (weil manche Diagonaleinträge null sind). Dann wird $\hat{\Sigma}^+$ betrachtet, bei welcher nur die $\sigma_i > 0$ invertiert werden und die anderen $\sigma_i = 0$ belassen werden. Das definiert eine sogenannte *verallgemeinerte Inverse* und löst auch das Optimierungsproblem falls $A$ keinen vollen Rang hat.

## 6.2 Numerische Berechnung

Die praktische Berechnung der Singulärwertzerlegung einer Matrix $A \in \mathbb{R}^{m \times n}$ verlangt einen gesamten Grundkurs in *numerischer Mathematik*.

In direkter Weise könnten die Singulärwerte und –vektoren über das Eigenwertproblem für $AA^T$ oder $A^T A$ bestimmt werden. Das ist nicht so schlecht, wie mit dem Argument, *dass sich mit dem quadrieren der Matrizen auch die Konditionszahl quadriert*, gerne nahegelegt wird [1], da

[1] denn die Kondition des Eigenwertproblems ist direkt proportional zur Kondition der Matrix; vgl. (Satz 5.9, Richter and Wick 2017)

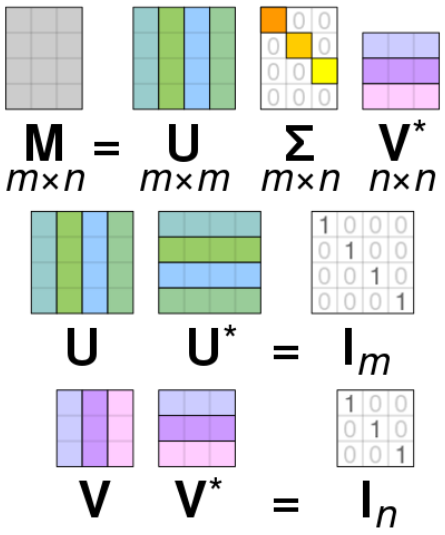


Figure 6.1: Illustration der SVD. Bitte beachten, der $*$ bedeutet hier transponiert und komplex konjugiert. By Cmglee - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=67853297

- wenn $n \ll m$ oder $m \ll n$, dann ist $A^T A$ oder $AA^T$ wesentlich kleiner als $A$
- das Eigenwertproblem symmetrisch ist, was gut ausgenutzt werden kann
- wenn $A$ sehr gross aber *dünnbesetzt* (engl. *sparse*) ist, dann können die Eigenwerte durch effiziente *sparse matrix-vector* Multiplikationen angenähert werden
- es können ohne weiteres nur eine Anzahl von Singulärwerten berechnet werden

sodass für *sparse* Matrizen diese Methode der de-facto Standard ist[2].

Für normale Matrizen kommt jedoch der folgende Algorithmus, der mehrere wunderbar effiziente Algorithmen elegant kombiniert besser in Betracht:

1. Betrachte
$$M = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \in \mathbb{R}^{n+m \times n+m},$$
deren positiven Eigenwerte mit den (positiven) Singulärwerten von $A$ übereinstimmen.
2. Bringe $M$ durch *Householder transformationen* in *Hessenberg*-Form, also
$$H = QMQ^T$$

[2] und beispielsweise die Methode, die in `scipy.sparse.linalg.svd` implementiert ist

mit $Q$ orthogonal. Wegen Orthogonalität ist das eine Ähnlichkeitstransformation ($H$ hat die gleichen Eigenwerte wie $M$) und wegen Symmetrie von $M$ ist auch $H$ symmetrisch und damit *tridiagonal*.
3. Berechne die positiven Eigenwerte von $H$ mittels der *QR-Iteration*, die für *Hessenbergmatrizen* sehr effizient implementiert werden kann.

**Der Standard**[3] funktioniert wie folgt:

1. Berechne eine orthogonale Transformation auf eine *bidiagonale* $B = U_A^T A V_A$.
2. Berechne eine SVD von $B = U_B \Sigma V_B^T$ (das wird effizient mit einem *divide and conquer* Algorithmus von Gu und Eisenstat getan)
3. Erhalte die gesuchte SVD als $A = (U_A U_B)\Sigma(V_A V_B)^T$.

## 6.3 Aufgaben

### 6.3.1 Norm und Orthogonale Transformation

Sei $Q \in \mathbb{R}^{n\times n}$ eine orthogonale Matrix und sei $y \in \mathbb{R}^n$. Zeigen Sie, dass

$$\|y\|^2 = \|Qy\|^2$$

gilt.

### 6.3.2 Kleinste Quadrate und Mittelwert

Zeigen sie, dass der *kleinste Quadrate* Ansatz zur Approximation einer Datenwolke

$$(x_i, y_i), \quad i = 1, 2, \ldots, N,$$

mittels einer konstanten Funktion $f(x) = w_1$ auf $w_1$ auf den Mittelwert der $y_i$ führt.

### 6.3.3 QR Zerlegung und Kleinstes Quadrate Problem

Sei $A \in \mathbb{R}^{m,n}$, $m > n$, $A$ hat vollen Rank und sei

$$[Q_1 \quad Q_2] \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} = A$$

eine QR-Zerlegung von $A$ (d.h., dass $Q$ unitär ist und $\hat{R}$ eine (im Falle, dass $A$ vollen Rang hat invertierbare) obere Dreiecksmatrix. Zeigen sie, dass die Lösung von

$$\hat{R}w = Q_1^T y$$

[3] z.B. die LAPACK routinen, die die Basis bspw. von `numpy.linalg.svd` aber auch von Matlab's SVD ist

ein kritischer Punkt (d.h. der Gradient $\nabla_w$ verschwindet) von

$$w \mapsto \frac{1}{2}\|Aw - y\|^2$$

ist, also $w = \hat{R}^{-1}Q_1^T y$ eine Lösung des Optimierungsproblems darstellt. Vergleichen Sie mit der SVD Lösung aus der Vorlesung.

### 6.3.4 Eigenwerte Symmetrischer Matrizen

Zeigen Sie, dass Eigenwerte symmetrischer reeller Matrizen $A \in \mathbb{R}^{n\times n}$ immer reell sind.

### 6.3.5 Singulärwertzerlegung und Eigenwerte I

Zeigen Sie, dass die quadrierten Singulärwerte einer Matrix $A \in \mathbb{R}^{m\times n}$, $m > n$, genau die Eigenwerte der Matrix $A^TA$ sind und beschreiben Sie in welcher Beziehung sie mit den Eigenwerten von $AA^T$ stehen. **Hinweis**: hier ist "$m > n$" wichtig.

### 6.3.6 Singulärwertzerlegung und Eigenwerte II

Weisen Sie nach, dass die positiven Eigenwerte von

$$\begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$$

genau die *nicht-null* Singulärwerte von $A$ sind.

### 6.3.7 Truncated SVD

1. Berechnen und plotten sie die Singulärwerte einer $4000 \times 1000$ Matrix mit zufälligen Einträgen und die einer Matrix mit "echten" Daten (hier Simulationsdaten einer Stroemungssimulation)[4]. Berechnen sie den Fehler der *truncated SVD* $\|A - A_{\hat{r}}\|$ für $\hat{r} = 10, 20, 40$ für beide Matrizen.
2. Was lässt sich bezüglich einer Kompression der Daten mittels SVD für die beiden Matrizen sagen. (Vergleichen sie die plots der Singulärwerte und beziehen sie sich auf die gegebene Formel für die Differenz).
3. Für die "echten" Daten: Speichern sie die Faktoren der bei $\hat{r} = 40$ abgeschnittenen SVD und vergleichen Sie den Speicherbedarf der Faktoren und der eigentlichen Matrix.

Beispielcode:

```
import numpy as np
import scipy.linalg as spla
import matplotlib.pyplot as plt
```

[4] Download bitte hier – Achtung das sind 370MB

```
randmat = np.random.randn(4000, 1000)

rndU, rndS, rndV = spla.svd(randmat)

print('U-dims: ', rndU.shape)
print('V-dims: ', rndV.shape)
print('S-dims: ', rndS.shape)

plt.figure(1)
plt.semilogy(rndS, '.', label='Singulaerwerte (random Matrix)')

realdatamat = np.load('velfielddata.npy')

# # Das hier ist eine aufwaendige Operation
rlU, rlS, rlV = spla.svd(realdatamat, full_matrices=False)
# # auf keinen Fall `full_matrices=False` vergessen

print('U-dims: ', rlU.shape)
print('V-dims: ', rlV.shape)
print('S-dims: ', rlS.shape)

plt.figure(1)
plt.semilogy(rlS, '.', label='Singulaerwerte (Daten Matrix)')

plt.legend()
plt.show()
```

**Hinweis**: Es gibt viele verschiedene Normen für Vektoren und Matrizen. Sie dürfen einfach mit `np.linalg.norm` arbeiten. Gerne aber mal in die Dokumentation schauen *welche* Norm berechnet wird.

# Chapter 7

# PCA und weitere SVD Anwendungen

## 7.1 Proper-Orthogonal Decomposition – POD

Die POD Methode ist ein Ansatz um den hohen Rechen– und Speicheraufwand in der Simulation von hochdimensionalen (d.h. viele Variable umfassenden) Simulationen von dynamischen Systemen abzumildern. Grob gesagt funktioniert POD wie folgt.

Es sei ein dynamisches System

$$\dot{y}(t) = f(t, y(t)), \quad y(0) = y_0 \in \mathbb{R}^m$$

gegeben, das die Entwicklung eines Zustandes $y(t) \in \mathbb{R}^m$ über die Zeit $t > 0$ beschreibt. Je größer die Dimension $m$ ist, desto aufwändiger ist das numerische berechnen (bzw. approximieren) der Werte von $y$.

Die Idee von POD ist

1. anzunehmen, dass die Zustände $y(t)$ mit weniger als $m$ Koordinaten beschrieben werden können, also

   $$y(t) \approx V\hat{y}(t)$$

   mit einer Basismatrix $U_r \in \mathbb{R}^{m\times r}$, $r \leq m$, und reduzierten Koordinaten $\hat{y}(t) \in \mathbb{R}^r$

2. die Matrix $U_r$ aus der Rang-$r$-Bestapproximation[1] der Datenmatrix

   $$Y = \begin{bmatrix} y(t_1) & y(t_2) & \dots & y(t_k) \end{bmatrix},$$

[1] Die *truncated* SVD ergibt auch die optimale Approximation in der *Frobenius*-norm, was hier die naheliegende Norm ist, da die einzelnen Einträge (also die Daten selbst) verglichen werden (und nicht irgendwelche Eigenwerte)

als die Matrix der ersten $r$ Singulärvektoren zu bestimmen.

3. und dann das System auf den Spann von $U_r$ (also auf $r$ Dimensionen) zu projizieren.

## 7.2 Simultane Diagonalisierung

Sind zwei symmetrische positiv definite Matrizen $P \in \mathbb{R}^{n\times n}$ und $Q \in \mathbb{R}^{n\times n}$ gegeben, so gibt es immer eine invertierbare Matrix $T \in \mathbb{R}^{n\times n}$, sodass die transformierten Matrizen

$$\tilde{P} := TPT^* =: D, \quad \tilde{Q} := T^{-*}QT^{-1} = D$$

identisch und diagonal sind. Eine Möglichkeit, die Existenz von $T$ zu beweisen (und auch eine numerisch zu berechnen) funktioniert über die SVD (vgl. die bald erscheinende Übungsaufgabe).

## 7.3 PCA

*Principal Component Analysis* ist ein Ansatz aus der Statistik, multivariate Daten so zu transformieren, dass

- die einzelnen Komponenten (empirisch)[2] nicht mehr korreliert sind
- die Varianz sich hierarchisch absteigend in den ersten Komponenten konzentriert.

Weil ich den Ansatz gerne *ad hoc* also am Problem entlang motivieren und einführen will, vorweg schon mal die bevorstehenden Schritte

1. Zentrierung/Skalierung der Daten.
2. Berechnung der Varianzen im Standard Koordinatensystem.
3. Überlegung, dass Daten in einem anderen Koordinatensystem eventuell besser dargestellt werden.
4. Berechnung eines optimalen Koordinatenvektors mittels SVD.

Wir nehmen noch einmal die Covid-Daten her, vergessen kurz, dass es sich um eine Zeitreihe handelt und betrachten sie als Datenpunkte $(x_i, y_i)$, $i = 1, \dots, N$, im zweidimensionalen Raum mit Koordinaten $x$ und $y$.

Als erstes werden die Daten **zentriert** indem in jeder Komponente der Mittelwert

$$x_c = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad y_c = \frac{1}{N}\sum_{i=1}^{N} y_i.$$

abgezogen wird und dann noch mit dem inversen des Mittelwerts skaliert.

[2] Die Zufallsvariable die hinter den Daten steckt wird dabei **nicht** notwendigerweise dekorreliert – insbesondere, wenn neue Daten hinzu kommen, muss die PCA wiederholt werden. Allerdings gibt es auch entsprechende asymptotische statistische Aussagen und Methoden, eine PCA *aufzudatieren*.

Also, die Daten werden durch $(\frac{x_i-\bar{x}}{\bar{x}}, \frac{y_i-\bar{y}}{\bar{y}})$ ersetzt.

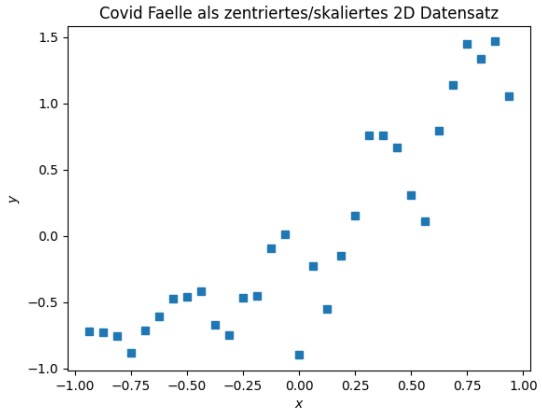


Figure 7.1: Fallzahlen von Sars-CoV-2 in Bayern im Oktober 2020 – zentriert

### 7.3.1 Variationskoeffizienten

Als nächstes kann Jan sich fragen, wie gut die Daten durch ihren Mittelwert beschrieben werden und die Varianzen berechnen, die für zentrierte Daten so aussehen

$$s_x^2 = \frac{1}{N-1}\sum_{i=1}^{N} x_i^2, \quad s_y^2 = \frac{1}{N-1}\sum_{i=1}^{N} y_i^2.$$

Im gegebenen Fall bekommen wir

$$s_x^2 \approx 0.32 \quad s_y^2 \approx 0.57$$

und schließen daraus, dass in $y$ Richtung *viel passiert* und in $x$ Richtung *nicht ganz so viel*. Das ist jeder Hinsicht nicht befriedigend, wir können weder

- Redundanzen ausmachen (eine Dimension der Daten vielleicht weniger wichtig?) noch
- dominierende Richtungen feststellen (obwohl dem Bild nach so eine offenbar existiert)

und müssen konstatieren, dass die Repräsentation der Daten im $(x, y)$ Koordinatensystem nicht optimal ist.

Die Frage ist also, ob es ein Koordinatensystem gibt, dass die Daten besser darstellt.

Ein Koordinatensystem ist nichts anderes als eine Basis. Und die Koordinaten eines Datenpunktes sind die Komponenten des entsprechenden Vektors in dieser Basis. Typischerweise sind Koordinatensysteme orthogonal (das heißt eine Orthogonalbasis) und häufig noch orientiert (die Basisvektoren haben eine bestimmte Reihenfolge und eine bestimmte Richtung).

### 7.3.2 Koordinatenwechsel

Sei nun also $\{b_1, b_2\} \subset \mathbb{R}^2$ eine orthogonale Basis.

Wie allgemein gebräuchlich, sagen wir *orthogonal*, meinen aber *orthonormal*. In jedem Falle soll gelten

$$b_1^T b_1 = 1, \quad b_2^T b_2 = 1, \quad b_1^T b_2 = b_2^T b_1 = 0.$$

Wir können also alle Datenpunkte $\mathbf{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$ in der neuen Basis darstellen mit eindeutig bestimmten Koeffizienten $\alpha_{i1}$ und $\alpha_{i2}$ mittels

$$\mathbf{x}_i = \alpha_{i1} b_1 + \alpha_{i2} b_2.$$

Für orthogonale Basen sind die Koeffizienten durch *testen* mit dem Basisvektor einfach zu berechnen:

$$\begin{aligned} b_1^T \mathbf{x}_i &= b_1^T(\alpha_{i1} b_1 + \alpha_{i2} b_2) = \alpha_{i1} b_1^T b_1 + \alpha_{i2} b_1^T b_2 = \alpha_{i1} \cdot 1 + \alpha_{i2} \cdot 0 = \alpha_{i1}, \\ b_2^T \mathbf{x}_i &= b_2^T(\alpha_{i1} b_1 + \alpha_{i2} b_2) = \alpha_{i1} b_1^T b_2 + \alpha_{i2} b_2^T b_2 = \alpha_{i1} \cdot 0 + \alpha_{i2} \cdot 1 = \alpha_{i2}. \end{aligned}$$

Es gilt also

$$\alpha_{i1} = b_1^T \mathbf{x} = b_1^T \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \quad \alpha_{i2} = b_2^T \mathbf{x}_i = b_2^T \begin{bmatrix} x_i \\ y_i \end{bmatrix}.$$

Damit, können wir jeden Datenpunkt $\mathbf{x}_i = (x_i, y_i)$ in den neuen Koordinaten $(\alpha_{i1}, \alpha_{i2})$ ausdrücken.

Zunächst halten wir fest, dass auch in den neuen Koordinaten die Daten zentriert sind. Es gilt nämlich, dass

$$\begin{aligned} \frac{1}{N}\sum_{i=1}^N \alpha_{ji} = \frac{1}{N}\sum_{i=1}^N b_j^T \mathbf{x}_i = \frac{1}{N} b_j^T \sum_{i=1}^N \begin{bmatrix} x_i \\ y_i \end{bmatrix} &= \frac{1}{N} b_j^T \begin{bmatrix} \sum_{i=1}^N x_i \\ \sum_{i=1}^N y_i \end{bmatrix} \\ &= b_j^T \begin{bmatrix} \frac{1}{N}\sum_{i=1}^N x_i \\ \frac{1}{N}\sum_{i=1}^N y_i \end{bmatrix} = b_j^T \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0, \end{aligned}$$

für $j = 1, 2$.

Desweiteren gilt wegen der Orthogonalität von $B = [b_1\ b_2] \in \mathbb{R}^{2\times 2}$, dass

$$x_i^2 + y_i^2 = \|\mathbf{x}_i\|^2 = \|B^T\mathbf{x}_i\|^2 = \| \begin{bmatrix} b_1^T \\ b_2^T \end{bmatrix} \mathbf{x}_i\|^2 = \| \begin{bmatrix} b_1^T\mathbf{x} \\ b_2^T\mathbf{x} \end{bmatrix} \|^2 = \| \begin{bmatrix} \alpha_{i1} \\ \alpha_{i2} \end{bmatrix} \|^2 = \alpha_{i1}^2 + \alpha_{i2}^2$$

woraus wir folgern, dass in jedem orthogonalen Koordinatensystem, die Summe der beiden Varianzen die gleiche ist:

$$s_x^2 + s_y^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i^2 + y_i^2) = \frac{1}{N-1}\sum_{i=1}^{N}(\alpha_{i1}^2 + \alpha_{i2}^2) =: s_1^2 + s_2^2.$$

Das bedeutet, dass durch die Wahl des Koordinatensystems die Varianz als Summe nicht verändert wird. Allerdings können wir das System so wählen, dass eine der Varianzen in Achsenrichtung maximal wird (und die übrige(n) entsprechend klein).

Analog gilt für den eigentlichen Mittelwert der (nichtzentrierten) Daten, dass die Norm gleich bleibt. In der Tat, für die *neuen* Koordinaten des Mittelwerts gilt in der Norm

$$\| \begin{bmatrix} \alpha_{c1} \\ \alpha_{c2} \end{bmatrix} \| = \|B^T \begin{bmatrix} x_c \\ y_c \end{bmatrix} \| = \| \begin{bmatrix} x_c \\ y_c \end{bmatrix} \|.$$

### 7.3.3 Maximierung der Varianz in (Haupt)-Achsenrichtung

Wir wollen nun also $b_1 \in \mathbb{R}^2$, mit $\|b_1\| = 1$ so wählen, dass

$$s_1^2 = \frac{1}{N-1}\sum_{i=1}^{N}\alpha_{i1}^2$$

maximal wird. Mit der Matrix $\mathbf{X}$ aller Daten

$$\mathbf{X} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \in \mathbb{R}^{N\times 2}$$

können wir die Varianz in $b_1$-Richtung kompakt schreiben als

$$s_1^2 = \frac{1}{N-1}\sum_{i=1}^{N}\alpha_{i1}^2 = \frac{1}{N-1}\sum_{i=1}^{N}(b_1^T\mathbf{x}_i)^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\mathbf{x}_i^T b_1)^2 = \frac{1}{N-1}\|\mathbf{X}b_1\|^2$$

Wir sind also ein weiteres mal bei einem Optimierungsproblem (diesmal mit Nebenbedingung) angelangt:

$$\max_{b\in\mathbb{R}^2,\,\|b\|=1} \|\mathbf{X}b\|^2 \tag{7.1}$$

**Lemma 7.1** (Maximale Varianz)**.** *Die Lösung des Varianz-Maximierungsproblem* (7.1) *ist mit $b = v_1$ gegeben, wobei $v_1$ der erste (rechte) Singulärvektor von* $\mathbf{X}$ *ist:*

$$\mathbf{X} = U\Sigma V^T = U\Sigma \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix}.$$

*Proof.* Ein etwas indirekter Beweis basiert auf der Feststellung dass in (7.1) genau die 2-Norm der Matrix $X$ gesucht ist und dass bei $v_1$ das Maximum realisiert wird. □

Damit rechnen wir auch direkt nach, dass im neuen Koordinatensystem $\{b_1, b_2\} = \{v_1, v_2\}$ die Varianzen $s_1^2$ und $s_2^2$ (bis auf einen Faktor von $\frac{1}{N-1}$) genau die quadrierten Singulärwerte von $\mathbf{X}$ sind:

$$(N-1)s_1^2 = \|\mathbf{X}v_1\|^2 = \|U\Sigma \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} v_1\|^2 = \|\Sigma \begin{bmatrix} v_1^T v_1 \\ v_2^T v_1 \end{bmatrix}\|^2 = \|\Sigma \begin{bmatrix} 1 \\ 0 \end{bmatrix}\|^2 = \sigma_1^2,$$
$$(N-1)s_2^2 = \|\mathbf{X}v_2\|^2 = \|U\Sigma \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} v_2\|^2 = \|\Sigma \begin{bmatrix} v_1^T v_2 \\ v_2^T v_2 \end{bmatrix}\|^2 = \|\Sigma \begin{bmatrix} 0 \\ 1 \end{bmatrix}\|^2 = \sigma_2^2$$

Für unser Covid Beispiel ergibt sich

$$V^T \approx \begin{bmatrix} 0.5848 & 0.8111 \\ 0.8111 & -0.5848 \end{bmatrix}$$

also

$$b_1 = v_1 = \begin{bmatrix} 0.5848 \\ 0.8111 \end{bmatrix} \quad b_2 = v_2 = \begin{bmatrix} 0.8111 \\ -0.5848 \end{bmatrix}$$

als neue Koordinatenrichtungen mit

$$s_1^2 \approx 0.85, \quad s_2^2 \approx 0.04,$$

was bereits eine deutliche Dominanz der $v_1$-Richtung – genannt *Hauptachse* – zeigt.

Im Hinblick auf Anwendungen und Eigenschaften der PCA untersuchen werden, noch ein Plot der Daten mit der $v_1$-Richtung als Linie eingezeichnet.

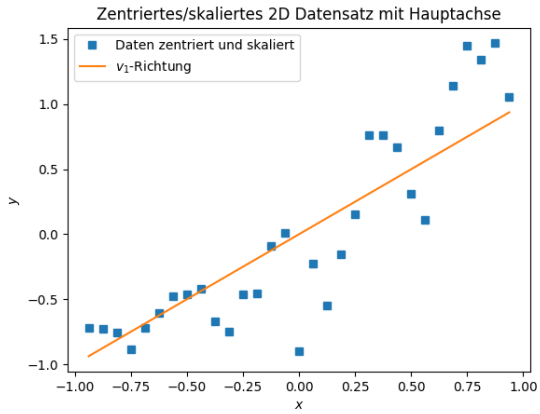


Figure 7.2: Fallzahlen von Sars-CoV-2 in Bayern im Oktober 2020 – zentriert/skaliert/Hauptachse

# Chapter 8

# Support Vector Machines

In diesem Kapitel betrachten wir, wie das Klassifizierungsproblem (erstmal bezüglich zweier Merkmale) durch eine optimale Wahl einer trennenden Hyperebene gelöst werden kann. In der Tat sind die Merkmale der SVM

- dass die *beste* Hyperebene
- effizient berechnet wird
- und das sogar für möglicherweise nichtlineare Einbettungen in höhere Dimensionen (*kernel-SVM*)

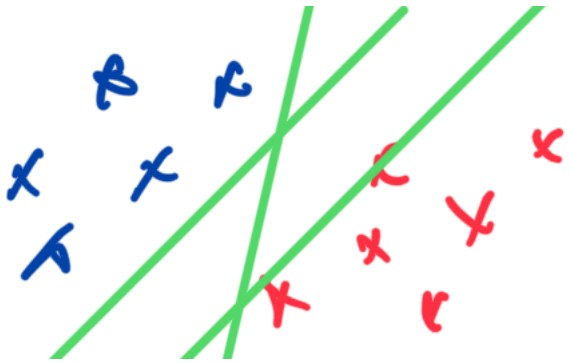

Figure 8.1: Beispiel Illustration von Punktwolken mit zwei verschiedenen Labels (hier rot und blau) und verschiedener trennender Hyperebenen

## 8.1 Problemstellung

**Definition 8.1** (Problemstellung für die SVM)**.**

1. Gegeben sei eine Wolke im $\mathbb{R}^n$ von $N$ Datenpunkten

$$\mathbb{X} := \{(x_i, y_i) \colon x_i \in \mathbb{R}^n,\, y_i \in \{-1, +1\},\, i = 1, \dots, N\}$$

   wobei $x_i$ der Datenpunkt ist und $y_i$ das zugehörige Label.

2. Eine *Hyperebene* $H \subset \mathbb{R}^n$, definiert durch den Stützvektor $b \subset \mathbb{R}^n$ und den Normalenvektor $w \in \mathbb{R}^n$, heißt **trennend**, falls

$$y_i \langle x_i - b,\, w\rangle > 0, \quad i = 1, \dots, N \tag{8.1}$$

3. Die Hyperebene $H$ heißt *Support Vector Machine* falls, $H$ die Hyperebene ist, für die der **kleinste** Abstand

$$d(H, \mathbb{X}) = \min_{s \in H, i = 1, \dots, N} \{\|s - x_i\|_2\}$$

   **maximal** wird.

Ein Paar Bemerkungen dazu.

1. Dass die Labels mit $\pm 1$ gewählt werden hat ganz praktische Gründe:
2. Zunächst lässt sich anhand des Vorzeichen des inneren Produktes $\langle x - b,\, w\rangle$ entscheiden, auf *welcher Seite* von $H$ ein Punkt $x$ liegt. (Punkte auf der gleichen Seite haben das gleiche Vorzeichen). Mit der Wahl der labels als $\pm 1$ kann das kompakt in einer einzigen Gleichung wie in (8.1) in der Definition geschrieben werden.
3. Ist die Hyperebene bekannt, können neue Datenpunkte $x$ über das Vorzeichen von $\langle x - b,\, w\rangle$ gelabelt werden – das ist die eigentliche Motivation, diese Hyperebene möglichst gut zu bestimmen.
4. Es gilt $\langle x - b,\, w\rangle = \langle x,\, w\rangle - \langle b,\, w\rangle := \langle x,\, w\rangle - \beta$. Und damit genügt es zur Definition (und zum Einsatz in der Klassifizierung), nur den Vektor $w \in \mathbb{R}^n$ und den *bias* $\beta \in \mathbb{R}$ zu bestimmen (beziehungsweise zu kennen).
5. Klassischere Methoden zur Klassifizierung mittels Hyperebene benutzen beispielsweise einfache neuronale Netze um ein passendes $(w, \beta)$ zu bestimmen.
6. Eine solche Hyperebene kann durchaus auch **nicht existieren**, dann heißen die Daten *nicht linear trennbar*. Existiert eine solche Ebene, dann existieren unendlich viele – ein weiterer Grund, die Ebene optimal (und damit hoffentlich eindeutig) zu wählen.

## 8.2 Maximierung des Minimalen Abstands

Um den Abstand maximieren zu können leiten wir eine Formel her.

Dazu sei $w \in \mathbb{R}^n$ der Normalenvektor von $H$ sei $h \in \mathbb{R}$ so, dass $b = \frac{h}{\|w\|} w$ ein Stützvektor ist. (Insbesondere kann Jan eine Hyperebene auch über den Normalenvektor und den Abstand von $H$ zum Ursprung charakterisieren). Damit

bekommen wir den Abstand von $H$ zum Ursprung als $h$ (Achtung: das $h$ kann auch negativ sein – es sagt uns wie weit müssen wir den normalisierten Vektor $w$ entlanglaufen, bis wir zu Ebene $H$ gelangen).

Zu einem beliebigen Punkt $x \in \mathbb{R}^n$ bekommen wir den Abstand zu $H$ als

- den Abstand der Ebene $H$ zur Ebene $H_x$, die parallel zu $H$ verläuft und $x$ enthält
- beziehungsweise als die Differenz der Abstände von $H_x$ und $H$ zum Ursprung

Da auch $w$ der Normalenvektor von $H_x$ ist, gilt für den Abstand $h'$, dass

$$h_x = \|x\|_2 \cos(\phi(w,x)) = \|x\|_2 \frac{\langle w,\, x\rangle}{\|w\|_2\|x\|_2} = \frac{\langle w,\, x\rangle}{\|w\|_2}$$

wobei $\cos(\phi(w,x))$ aus dem Winkel zwischen $x$ und $w$ herrührt.

Mit dieser Formel und mit $b = hw$, erkennen wir, dass der Test auf das Vorzeichen

$$\langle x{-}b,\, w\rangle = \langle x{-}\frac{h}{\|w\|_2}w,\, w\rangle = \langle x,\, w\rangle{-}h\|w\|_2 = \|w\|_2(\frac{\langle x,\, w\rangle}{\|w\|_2}{-}h) = \|w\|_2(h_x{-}h)$$

den mit $\|w\|_2$ skalierten Abstand (inklusive dem Vorzeichen) enthält, beziehungsweise, dass der Abstand als

$$y_i \frac{\langle x_i - b,\, w\rangle}{\|w\|_2} = y_i \frac{\langle x_i,\, w\rangle - \beta}{\|w\|_2}$$

(für eine trennende Hyperebene $(w,\beta)$) auch immer das richtige Vorzeichen erhält (da $\|w\|_2 > 0$. Dementsprechend kann das SVM Problem als

$$\max_{w\in\mathbb{R}^n,\,\beta\in\mathbb{R}} \min_{x_i\in\mathbb{X}} y_i \frac{\langle x_i,\, w\rangle - \beta}{\|w\|_2}$$

formuliert werden.

So ein min max Problem ist generell schwierig zu analysieren und zu berechnen. Aber wir können direkt sagen, dass die *Zulässigkeit* der Optimierung gesichert ist, da "der max-imierer" nach Möglichkeit eine trennende Hyperebene wählt und so schon mal sicherstellt, dass "der min-inimierer" nur über positive Zahlen minimiert.

Darüberhinaus, wenn eine trennendes Hyperebene existiert, sodass

$$y_i(\langle x_i,\, w\rangle - \beta) = y_i(\langle x_i,\, w\rangle - h\|w\|_2) \geq q > 0$$

dann können wir durch die Wahl von $\tilde{w} = \frac{1}{q}w$, immer erreichen, dass das Minimum $\min_{x_i\in\mathbb{X}} y_i\langle x_i,\, w\rangle - \beta = 1$ ist und das Max-Min durch ein Maximierungsproblem unter Zulässigkeitsnebenbedingungen

$$\min_{w,\beta} \frac{1}{2}\|w\|_2^2, \quad s.t. \quad y_i(\langle x_i,\, w\rangle - \beta) \geq 0,\, i = 1, \ldots, N$$

ersetzen. Dabei haben wir noch ausgenutzt, dass $\max_w \frac{1}{\|w\|} \leftrightarrow \min_w \frac{1}{2}\|w\|^2$ entspricht, um die Standardform eines *quadratischen Optimierungsproblems* unter (affin) *linearen Ungleichungsnebenbedingungen* zu erhalten.

Für solche Optimierungsprobleme kann das *duale Problem* direkt hergeleitet werden, was sich in diesem Fall als die Suche eines Vektors $a \in \mathbb{R}^n$ über das restringierte Minimierungsproblem

$$\min_a \left( \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} a_i a_k y_i y_k \langle x_i, x_k \rangle - \sum_{i=1}^{N} a_i \right) \quad s.t. \quad a \geq 0, \sum_{i=1}^{N} a_i y_i = 0$$

ergibt.

Aus der KKT Theorie kann abgeleitet werden, dass $a_i > 0$, genau dann wenn $x_i$ die Gleichheit $y_i(\langle x_i, w \rangle - \beta) = 1$ erfüllt ist, also $x_i$ ein sogenannter *support vector* ist. Ist $S$ die Menge aller Indices, die die *support vectors* indizieren, so kann die Klassifikation eines neuen Datenpunktes $x$ mittels

$$y(x) = \sum_{i \in S} a_i y_i \langle x, x_i \rangle + \gamma$$

vorgenommen werden, wobei der *bias* als

$$\gamma = \sum_{i \in S} (y_i - \sum_{k \in S} a_k y_k \langle x_i, x_k \rangle)$$

vorberechnet werden kann.

### 8.2.1 Nichtlineare Separation

Existiert auf den Originaldaten keine separierende Hyperebene (oder ist sie schwer zu berechnen) dann können die Daten in höhere Dimensionen nichtlinear eingebettet werden und dort separiert werden.

$$\Psi : \mathbb{R}^n \to \mathbb{R}^M$$

Beispiel von gestern

$$\Psi : \mathbb{R} \to \mathbb{R}^2 : x \to \Psi(x) = (x, (x-2)^2)$$

Mit der erhöhten Dimension kommen zwei Aufgaben mit einer Lösung

1. Die transformierten Daten müssen separiert und neue Daten müssen klassifiziert werden – das ist schwierig und aufwändig für $N \gg 1$.
2. Wie soll $\Psi$ gewählt werden?

Die Lösung zu beidem ist, dass

1. in der dualen Formulierung nur Skalarprodukte $\langle x_i, x_j \rangle$ beziehungsweise $\langle \Psi(x_i), \Psi(x_j) \rangle$ benötigt werden
2. die bestenfalls durch *kernel* Funktionen einfach berechnet werden können (ohne das Lifting in den hochdimensionalen Raum)

Beispiele:

- $\Psi(x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$ hat die *kernel* Funktion $k(x, y) = (x_1y_1 + x_2y_2)^2$

Damit wird zunächst das Berechnungsproblem gelöst. Das Problem, welche Funktionen gewählt werden, wird nebenbei dadurch gelindert, dass

1. die Effizienz in der Auswertung *try and error* möglich macht
2. Jan letztlich gar keine $\Psi$s mehr sucht, sondern nur noch $k$s mit bekannten Eigenschaften.

## 8.3 Aufgaben

1. Sei die Hyperebene über $w$ und $b$ gegeben. Bestimmen Sie den (möglicherweise negativen) Abstand $h \in \mathbb{R}$, sodass $hw \in H$.

# Chapter 9

# Best and Universal Approximation

Alle bisher betrachteten Approximationsprobleme waren in Bezug auf eine *2-norm* ("least squares") formuliert. Die für die Berechnung unmittelbaren Vorteile sind die

1. Differenzierbarkeit der Norm und die
2. Charakterisierung der Optimallösung über Orthogonalität.

Ein gängiges Optimierungsproblem, eine Bestapproximation zu einer stetigen Funktion $f\colon [a,b] \to \mathbb{R}$ in einer passenden Menge von Funktionen $\mathcal{G}$ bezüglich der *Supremumsnorm*[1] zu finden

$$\min_{g\in\mathcal{G}} \|f-g\|_\infty = \min_{g\in\mathcal{G}}( \max_{x\in[a,b]} |f(x)-g(x)|)$$

fällt nicht darunter. Die entstehenden Schwierigkeiten und theoretische und praktische Ansätze zur möglichen Lösung dieses sogenannten *Tschebyscheff-Approximation*-Problem, sind in (Kapitel 8.7.2, Richter and Wick 2017) gut nachzulesen.

## 9.1 Universal Approximation

Wir wollen hier nachvollziehen, dass klassische neuronale Netze, dieses Problem approximativ aber mit beliebiger Genauigkeit $\epsilon$ lösen könnten. Die Schritte da hin sind wie folgt

1. Zu einem gegebenen $f \in \mathcal{C}[a,b]$ (einer reellwertigen, stetigen Funktion auf einem endlichen und abgeschlossenen Intervall) existiert immer eine

[1] für ein kompaktes Intervall wird das zur *Maximumsnorm*

stückweise konstante Funktion $f_N$ mit endlich vielen *Sprungstellen*, sodass

$$\|f - f_N\|_\infty < \frac{\epsilon}{2}$$

2. Zu diesem $f_N$ können wir immer eine Funktion

$$s_M(x) = c_0 + \sum_{i=1}^{M} c_i \tanh(a_i(x - b_i))$$

konstruieren (durch Anpassung der Parameter $c_0$, $c_i$, $b_i$, $a_i$, $i = 1, \ldots, M$) sodass

$$\|f_N - s_M\| < \frac{\epsilon}{2}.$$

3. Wir interpretieren $s_M$ als ein neuronales Netz mit einer *hidden layer* und können konstatieren dass

$$\|f - s_M\|_\infty \leq \|f - f_N\|_\infty + \|f_N - s_M\|_\infty < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon.$$

Einige Fragen werden wir unbeantwortet lassen müssen, vor allem

- wie wählen wir $M$ (das Resultat sagt nur $M$ muss groß genug sein)

und

- wie wirkt sich in der Praxis die approximative Berechnung von $a_i$–$c_i$ auf die Approximation aus?

Dennoch gibt dieses Beispiel einen Einblick in die Funktionsweise der Approximation und das referenzierte *universal approximation theorem* ist Grundlage vieler Analyseansätze für neuronale Netzwerke.

In Schritt 1, wird die stückweise konstante Funktion $f_N$ definiert. Die Existenz folgt aus dem Satz

**Theorem 9.1** (Approximation durch stückweise konstante Funktionen)**.** *Sei $[a, b] \subset \mathbb{R}$ ein abgeschlossenes endliches Intervall. Der Abschluss bezüglich der Supremumsnorm der Menge* $\mathrm{Pc}[a, b]$ *aller stückweise konstanten Funktionen auf $[a, b]$ mit endlich vielen Sprungstellen* ***enthält*** $\mathcal{C}[a, b]$*, d.h.*

$$\mathcal{C}[a, b] \subset \mathrm{closure}_{\|\cdot\|_\infty}(\mathrm{Pc}[a, b]).$$

*Insbesondere, existiert für ein beliebiges $f \in \mathcal{C}[a, b]$ und $\varepsilon > 0$, immer ein $g \in \mathrm{Pc}[a, b]$ mit $\|f - g\|_\infty < \varepsilon$.*

*Proof.* Der Beweis ist klassisch – hier nur die relevanten und konstruktiven Elemente.

An sich muss gezeigt werden, dass es zu jeder Funktion $f \in \mathcal{C}[a, b]$ eine Folge $\{f_n\} \subset \mathrm{Pc}[a, b]$ gibt, mit $f$ als Grenzwert. Wir zeigen nur die Konstruktion eines potentiellen Folgengliedes.

Sei $f \in \mathcal{C}[a,b]$ beliebig. Da stetige Funktionen auf kompakten Mengen gleichmäßig stetig sind, gibt es zu jedem $\varepsilon > 0$ ein $\delta > 0$, sodass

$$|f(x \pm h) - f(x)| < \varepsilon$$

für alle $h < \delta$. Damit können wir zu jedem $\varepsilon$ eine Unterteilung von $(a,b]$ in $N(\delta)$ halboffene (bis auf das abgeschlossene "erste" Intervall, das $a$ enthält) disjunkte Intervalle $I_j$, $j = 1, \dots, N(\delta)$ finden, sodass

$$f_N(x) = \sum_{j=1}^{N} \chi_{I_j}(x)$$

□

mit den Indikatorfunktionen $\chi_{I_j}$ und mit

$$f_j = \frac{1}{2}(\max_{\xi \in I_j}\{f(\xi)\} + \min_{\eta \in I_j}\{f(\eta)\})$$

eine Funktion aus $\mathrm{PL}[a,b]$ darstellt, die um maximal $\varepsilon$ von $f$ abweicht. Außerdem können wir damit sicherstellen, dass

$$|f_j - f_{j+1}| < \varepsilon, \tag{9.1}$$

für alle $j = 1, \dots, N-1$ gilt, was im nächsten Schritt relevant wird.

In Schritt 2 wird nun die Funktion $f_N$ durch Linearkombinationen von transformierten tanh Funktionen approximiert:

$$f_N(x) \approx g_M(x) := c_0 + \sum_{i=1}^{M} c_i \tanh(a_i(x - b_i))$$

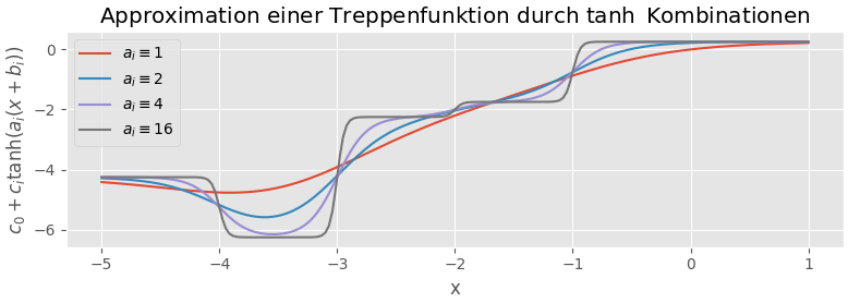


Figure 9.1: Beispiel Illustration wie eine Linearkombination von skalierten tanh Funktionen eine Treppenfunktion approximiert

Die Konstruktion und der Nachweis, dass für jedes $\varepsilon < 0$ (mit $f_N$ bereits entsprechend konstruiert) eine Differenz $\|f_N - g_M\|_\infty < \varepsilon$ möglich ist, basiert auf folgenden Argumenten:

- Allgemein gilt für die Funktion tanh, dass sie *streng monoton* ist, dass $\tanh(0) = 0$ und dass $\lim_{x\to\pm\infty} \tanh(x) = \pm 1$.
- Durch eine Skalierung von $x \leftarrow ax$ mit $a \to \infty$, passiert der übergang von $-1$ zu 1 beliebig schnell – tanh entspricht zunehmend der Treppenfunktion, die bei 0 von $-1$ auf $+1$ springt. (Jan bemerke, dass diese "Konvergenz" **nicht** bezüglich der *Supremumsnorm* stattfindet.)
- Durch einen Shift $x \leftarrow x - b$ kann der Sprung von $x = 0$ zu $x = b$ verschoben werden.
- Durch Skalierung $\tanh \leftarrow c\tanh$ kann die Höhe des Sprungs angepasst werden.

Damit (und insbesondere mit $b_i$ als die Sprungstellen von $f_N$ gewählt und $c_i$ als die Differenz zwischen den Werten an diesem Sprung), kann direkt ein $g_M$ konstruiert werden, das bis auf beliebig kleine, offene Umgebungen um die Sprungstellen dem $f_N$ beliebig nahe kommt. (Allerdings nicht in der *Supremumsnorm*). Da für genügend große $a_i$ aber sichergestellt wird, dass $g_M$ auf jedem Teilintervall zwischen den Werten von $f_N$ interpoliert, folgt aus (9.1), dass auch die punktweise Differenz kleiner als $\varepsilon$ ist. Insgesamt folgt so die Existenz von $g_M$ mit den gewünschten Eigenschaften.

Im letzten Schritt 3 interpretieren wir die $g_M$ Approximation als ein neuronales Netz.

Dazu bemerken wir, dass wir $g_M$ schreiben können als

$$g_M(x) = c_0 + c^T \tanh(Ax + b) \tag{9.2}$$

mit

$$c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_M \end{bmatrix}, \quad A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{bmatrix}, \quad b = \begin{bmatrix} -a_1 b_1 \\ -a_2 b_2 \\ \vdots \\ -a_M b_M \end{bmatrix}$$

und der *eintragsbezogenen* Interpretation der eigentlich skalaren tanh Funktion und dass

- der Anteil
$$\tanh(Ax + b)$$
eine klassische *linear layer* mit Aktivierung ist
- während der Ausgang $y = c_0 + c^T\xi$ eine einfache lineare Abbildung (ohne Aktivierung ist).

In dieser Darstellung, ist die Suche nach den Parametern für $g_M$ in den Standardroutinen von *ML* Paketen ohne weiteres möglich.

Wir schließen mit einigen allgemeinen Bemerkungen

- die Funktion tanh hat durchaus Relevanz in der Praxis. Für die Theorie, die im wesentlichen Beschränktheit und Monotonie voraussetzt, gehen auch die gerne verwendeten allgemeineren *sigmoid* Funktionen.
- die größte Lücke zur Praxis ist die Annahme $M$ *genügend groß*. Etwas unpraktisch ist auch, dass nur eine *hidden layer* verwendet wird. Neuere Arbeiten behandeln ähnliche Approximationsresultate mit mehreren Schichten.
- die theoretisch unschönste Lücke ist die Annahme, dass eine Funktion auf einer kompakten Menge approximiert wird (was beispielsweise in der Behandlung von dynamischen Systemen nachteilig ist)
- Zur Approximation von $f_N$ ist Jan geneigt, die $a_i$ einfach sehr groß zu wählen. Ist $f$ stetig, dann werden allerdings bessere Resultate (die den Verlauf von $f$ nachzeichnen) mit kleineren $a_i$ erreicht. Außerdem führen große Werte von $a$ zum sogenannten *vanishing gradients* Phänomen, da $\frac{\tanh(a(x+h))-\tanh(ax)}{h} \to 0$ für $a \to \infty$.

## 9.2 Aufgaben

Zur Approximation einer Funktion $f\colon \mathbb{R} \to \mathbb{R}$ über den Ansatz (9.2) seien Datenpunkte $\mathbb{X} = \{(x_i, y_i)\}_i$ mit $y_i = f(x_i)$ gegeben und das Optimierungsproblem

$$L(c, A, b; \mathbb{X}) = \sum_i \|y_i - g_M(c, A, b;\, x_i)\|_2^2 \to \min_{c,A,b}$$

mittels des stochastischen Gradientenabstiegs zu lösen.

1. Schreiben Sie die Koeffizienten in einen Vektor $p = (c, A, b)$ und berechnen Sie $\nabla_{c_0} L(p; \mathbf{x})$, sowie beispielhaft die Komponenten von $\nabla_p L(p; \mathbf{x})$ für einen einzelnen Datenpunkt $\mathbf{x} = x_i$ oder einen *batch* $\mathbf{x} = \mathbb{x}$ (wie es für den stochastischen Gradienten benutzt würde).

2. Formulieren Sie die Berechnung des Gradienten (bezüglich $p$) für $M = 1$ und einen Datenpunkt $x$, also für die Funktion $l = L(c_0, c_1, a_1, b_1; x)$ über *automatisches Differenzieren* im Vorwärts– und im Rückwärtsmodus.

3. Implementieren Sie die Approximation und das Training. Dafür können die Bausteine aus Beispiel Implementierung verwendet werden. Implementieren Sie auch die Gradientenberechnung zur Verwendung im stochastischen Abstiegsverfahren. Testen Sie ihre Implementierung für verschiedene $M$ und verschiedene Daten für die Funktionen

   $$f_1(x) = \chi_{[-1,0]}(x) - \chi_{(0,\frac{1}{3}]}(x) + \chi_{(\frac{1}{3},\frac{1}{2}]}(x) - \chi_{(\frac{1}{2},1]}(x), \quad f_2(x) = \sin(4x)$$

   jeweils auf dem Intervall $[-1, 1]$.

# Chapter 10

# Automatisches (Algorithmisches) Differenzieren

In der Mathematik und im Bereich der Computeralgebra ist das *automatische Differenzieren* (auch *Auto-Differenzieren*, *Autodiff* oder einfach *AD* genannt und in anderen communities als *algorithmisches Differenzieren* oder *computergestütztes Differenzieren* bezeichnet), ein Satz von Techniken zur Berechnung (der Werte(!)) der partiellen Ableitung einer durch ein Computerprogramm spezifizierten Funktion.

AD nutzt die Tatsache, dass jede Computerberechnung, egal wie kompliziert, eine Sequenz von elementaren arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division usw.) und elementaren Funktionen (exp, log, sin, cos usw.) ausführt. Durch wiederholte Anwendung der Kettenregel auf diese Operationen können partielle Ableitungen beliebiger Ordnung automatisch, genau bis zur Arbeitspräzision und mit höchstens einem kleinen konstanten Faktor mehr an arithmetischen Operationen als das ursprüngliche Programm berechnet werden.

## 10.1 Andere Differentiationsmethoden

In manchen Anwendungsfällen wird auf eine symbolische Repräsentation von Funktionen und Variablen in der computergestützten Berechnung zurückgegriffen. Sogenannte Computeralgebra Pakete wie `Maple`, `Mathematica` oder die *Python* Bibliotheken `SageMath` oder `Sympy` können dann automatisiert auch Operationen auf Funktionen wie Integralberechnung und eben auch Differentiation *exakt* ausführen. Durch die Kettenregel und unter der Massgabe, dass

alle Codes letztlich nur elementare Funktionen mit bekannten Ableitungen verschalten, wäre es auch möglich, bei entsprechender Implementierung des primären Progamms, auch automatisiert die Ableitungen zu erzeugen. Allerdings ist der Mehraufwand in der symbolischen Programmierung erheblich und die Auswertung der Ausdrücke langsam, sodass symbolische Berechnungen (und entsprechend auch die Möglichkeit der symbolischen automatischen Differentiation) nur in spezifischen und insbesondere nicht in "grossen" und "multi-purpose" Algorithmen zur Anwendung kommen.

> Auch AD rechnet symbolisch und rekursiv mit der Kettenregel. Der Unterschied ist, dass AD nur mit Funktionswerten der Ableitungen arbeitet während die symbolische Ableitung versucht, die Ableitung als Funktion zu erzeugen.

Auf der anderen Seite steht die *numerische Differentiation* (beispielsweise durch Berechnung von Differenzenquotienten). Diese Methode ist universell (Ableitungen können ohne Kenntnis dessen berechnet werden, was im Innern des Programms alles passiert) ist jedoch enorm schlecht konditioniert (da im Zähler des Differenzenquotienten fast gleich grosse Größen subtrahiert werden). Die Bestimmung einer passenden Schrittweite muss immer *ad hoc* erfolgen und macht diese Berechnung zusätzlich teuer.

Sind höhere Ableitungen gefragt, verstärken sich zudem die Komplexität (für die symbolische Berechnung) und die Fehlerverstärkung (in der numerischen Differentiation).

## 10.2 Anwendungen

Automatisches Differenzieren ist ein entscheidender Baustein im Erfolg des maschinellen Lernens. Jan könnte sagen, dass ohne AD die Optimierung der neuronalen Netze mit tausenden bis Millionen von Parametern nicht möglich wäre.

## 10.3 Vorwärts- und Rückwärtsakkumulation

### 10.3.1 Kettenregel der partiellen Ableitungen zusammengesetzter Funktionen

Grundlegend für die automatische Differentiation ist die Zerlegung von Differentialen, die durch die Kettenregel der partiellen Ableitungen zusammengesetzter Funktionen bereitgestellt wird. Für die einfache Zusammensetzung

$$
\begin{aligned}
y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3 \\
w_0 &= x \\
w_1 &= h(w_0) \\
w_2 &= g(w_1) \\
w_3 &= f(w_2) = y
\end{aligned}
$$

ergibt die Kettenregel für die Werte zu einem fixen Wert $x^*$ von $x$:

$$
\begin{aligned}
\frac{\partial y(x^*)}{\partial x} &= \left.\frac{\partial y}{\partial w_2}\right|_{w_2=g(h(x^*))} \left.\frac{\partial w_2}{\partial w_1}\right|_{w_1=h(x^*)} \left.\frac{\partial w_1}{\partial w_0}\right|_{w_0=x^*} \\
&= \frac{\partial f}{\partial w_2}(w_2^*)\left[\frac{\partial g}{\partial w_1}(w_1^*)\left[\frac{\partial h}{\partial x}(w_0^*)\right]\right]
\end{aligned}
\tag{10.1}
$$

Für multivariate Funktionen gilt die mehrdimensionale Kettenregel und das Produkt der Ableitungen wird zum Produkt der Jacobi-Matrizen.

## 10.4 AD – Vorwärtsmodus

Im sogenannten *Vorwärtsmodus* (auch *forward accumulation*) wird jede Teilfunktion im Programm so erweitert, dass mit dem Funktionswert direkt der Wert der Ableitung mitgeliefert wird. Ein Programmfluss für obiges $f$, $g$, $h$ Beispiel würde also jeweils immer zwei Berechnungen machen und die Ableitung *akkumulieren*:

| Schritt | Funktionswert | Ableitung | Akkumulation |
|---|---|---|---|
| 0 | $w_0 = x$ | $\dot{w}_0 = 1$ | $1$ |
| 1 | $(w_1, \dot{w}_1) = (h(w_0), h'(w_0))$ | $\dot{w}_1$ | $\dot{w}_1 \cdot 1$ |
| 2 | $(w_2, \dot{w}_2) = (g(w_1), g(w_1))$ | $\dot{w}_2$ | $\dot{w}_2 \cdot \dot{w}_1 \cdot 1$ |
| 3 | $(w_3, \dot{w}_3) = (f(w_1), f(w_1))$ | $\dot{w}_3$ | $\dot{w}_3 \cdot \dot{w}_2 \cdot \dot{w}_1 \cdot 1$ |

Das ergibt die Ableitung als den finalen Wert der Akkumulation. Wir bemerken, dass hierbei

- die Ableitung entlang des Programmflusses immer direkt mitbestimmt wird (deshalb *vorwärts* Modus)
- es genügt im Schritt $k$, den Wert $w_{k-1}$ und die Akkumulation zu kennen.
- Für eine Funktion $F \otimes G \otimes h \colon \mathbb{R} \to \mathbb{R}^m$, funktioniert die Berechnung ganz analog mit beispielsweise

$$(G, \partial G) \colon \mathbb{R} \to \mathbb{R}^\ell \times \mathbb{R}^\ell$$

und

$$(F, \partial F) \colon \mathbb{R}^\ell \to \mathbb{R}^m \times \mathbb{R}^{m \times \ell}$$

und der Akkumulation

$$\frac{\partial y}{\partial x}(x^*) = \partial F(w_2) \cdot \partial G(w_1) \cdot h'(w_0) \cdot 1.$$

- Für eine Funktion in mehreren Variablen, d.h. von $\mathbb{R}^n$ nach $\mathbb{R}^m$, werden die partiellen Ableitungen $\frac{\partial y}{\partial x_k}$ separat in eigenen Durchläfen berechnet:
  - damit ist an der Implementierung nichts zu ändern, nur die Akkumulation wird mit $\dot{w}_0 = \begin{bmatrix} 0 & \cdots & 1 & 0 & \cdots & 0 \end{bmatrix}^T$ initialisiert
  - eine simultane Berechnung aller Ableitungen verursacht vergleichsweise viel *overhead* (entweder müssen die verschiedenen Richtungen im Code organisiert werden oder es müssen alle Zwischenberechnungen der Ableitung gespeichert werden).

Wir halten fest, dass der *Vorwärtsmodus* gut funktioniert für skalare Eingänge unabhängig von der Zahl der Ausgänge. Wir werden lernen, dass sich beim Rückwärtmodus dieses Verhältnis umdreht. Damit:

- *Vorwärtsakkumulation*: Bevorzugt für Funktionen $f \colon \mathbb{R}^n \to \mathbb{R}^m$, wobei $n \ll m$.
- *Rückwärtsakkumulation*: Bevorzugt für Funktionen $f \colon \mathbb{R}^n \to \mathbb{R}^m$, wobei $n \gg m$.

Vor allem für neuronale Netze mit vielen Parametern und dem Fehler als (skalaren) Ausgang, ist der Rückwärtsmodus fraglos die Methode der Wahl. Hier wird dann typischerweise von *backpropagation* gesprochen, was eine Adaption der Methode an die Architektur typischer neuronaler Netze ist.

Bevor wir zum Rückwärtsmodus kommen, noch einige Praktische Bemerkungen anhand eines konkreten Beispiel.

Zunächst mal die Bemerkung, dass es vorteilhaft ist, ein Programm als ein Graph der die Abhängigkeiten der Variablen enthält, darzustellen. Dann werden insbesondere “nicht vorhandene” Abhängigkeiten vermieden und Speicher– und Rechenaufwand reduziert.

Entsprechend werden die Zwischenwerte $w_i$ nicht mehr einfach durchnummeriert, sondern es wird von Vorgängern gesprochen

> $w_j$ ist ein *Vorgänger* von $w_i$ genau dann wenn $w_i$ unmittelbar (also *explizit*) von $w_j$ abhängt.

Damit (und mit Anwendung der Kettenregel im mehrdimensionalen) wird aus $\dot{w}_i = \frac{\partial w_i}{\partial w_{\text{“}i-1\text{”}}}$ der Ausdruck

$$\dot{w}_i = \sum_{j \in \{\text{Vorgänger von i}\}} \frac{\partial w_i}{\partial w_j} \dot{w}_j$$

Für das Beispiel

$$\begin{aligned} y &= f(x_1, x_2) \\ &= x_1 x_2 + \sin x_1 \\ &=: w_1 w_2 + \sin w_1 \\ &=: w_3 + w_4 \\ &=: w_5 \end{aligned}$$

ergibt sich folgender Berechnungsgraph und Fluss in der Vorwärtsakkumulation.

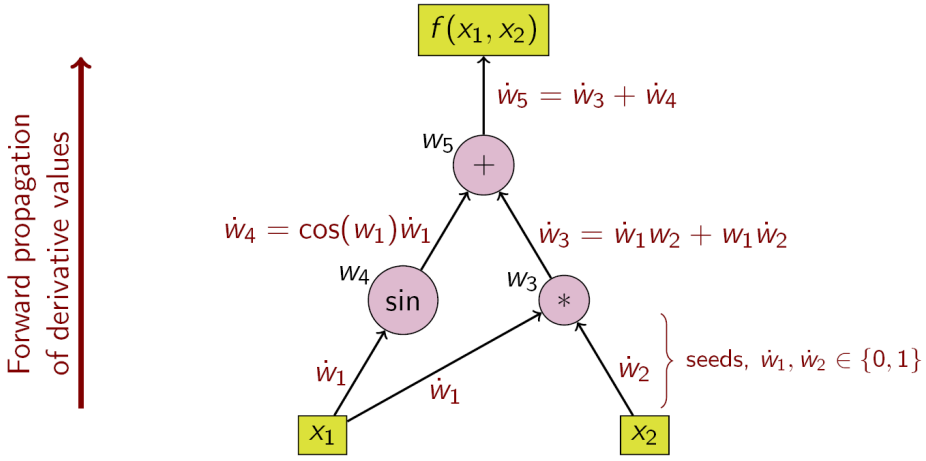


Figure 10.1: Beispiel für Vorwärtsakkumulation mit Berechnungsgraph

Mit den folgenden Schritten ergibt sich für die partielle Ableitung von $y$ bezüglich $x_1$

| Schritt | Funktionswert | Ableitung | Akkumulation |
|---|---|---|---|
| 0a | $w_1 = x_1$ | | $\dot{w}_1 = 1$ |
| 0b | $w_2 = x_2$ | | $\dot{w}_2 = 0$ |
| 1a | $w_3 = w_1 w_2$ | $\partial w_3 = \begin{bmatrix} w_2 & w_1 \end{bmatrix}$ | $\dot{w}_3 = \begin{bmatrix} w_2 & w_1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = x_2$ |
| 1b | $w_4 = \sin(w_1)$ | $\partial w_4 = \cos(w_1)$ | $\dot{w}_4 = \cos(w_1) \cdot \dot{w}_1 = \cos(x_1)$ |
| 2 | $w_5 = w_3 + w_4$ | $\partial w_5 = \begin{bmatrix} 1 & 1 \end{bmatrix}$ | $\dot{w}_5 = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} \dot{w}_3 \\ \dot{w}_4 \end{bmatrix} = x_2 + \cos(x_1)$ |

## 10.5 Rückwärtsmodus

Bei der Rückwärtsakkumulation werden die sogenannten *Adjungierten*

$$\bar{w}_i = \frac{\partial y}{\partial w_i}$$

berechnet. Jan beachte, dass immer $y$ *differenziert* wird und dass der gewünschte Ausdruck bei beispielsweise $\bar{x} = \frac{\partial y}{\partial x}$ erreicht ist.

Unter Verwendung der Kettenregel ergibt sich die folgende rekursive Formel aus dem Berechnungsgraphen:

$$\bar{w}_i = \sum_{j \in \{\text{Nachfolger von i}\}} \bar{w}_j \frac{\partial w_j}{\partial w_i}$$

Sind also *spätere* Adjungierte $\bar{w}_j$ bekannt, können *frühere* $\bar{w}_i$ bestimmt werden.

Die Rückwärtsakkumulation durchläuft die Kettenregel (wie in Gleichung (10.1) von außen nach innen oder im Falle des Berechnungsgraphen (wie in Abbildung 10.2 illustriert) von oben nach unten.

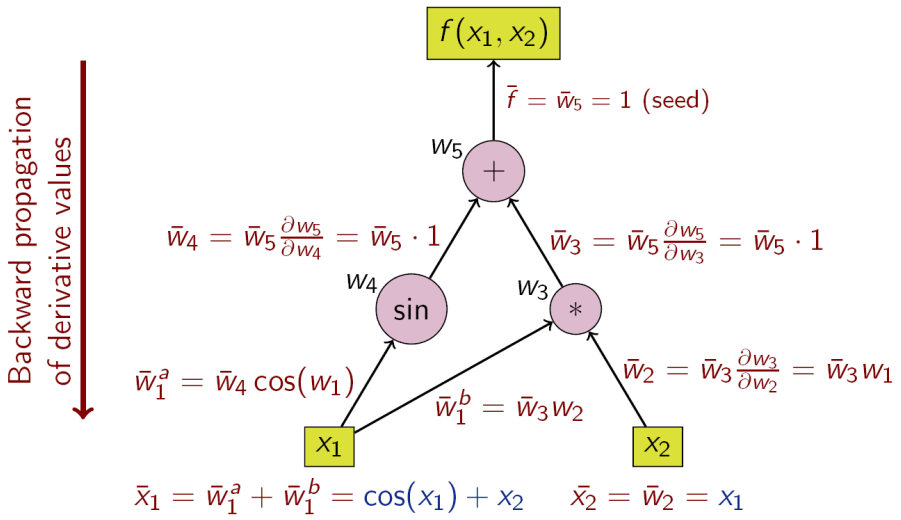


Figure 10.2: Beispiel für Rückwärtsakkumulation mit Berechnungsgraph

Die Operationen zur Berechnung der Ableitung mittels Rückwärtsakkumulation sind in der folgenden Tabelle dargestellt (beachte die umgekehrte Reihenfolge):

| Schritt | Funktionswert | Ableitung | Adjungierte |
|---|---|---|---|
| `0` | $y = w_5$ | $\partial y = 1$ | $\bar{w}_5 = 1$ |
| `1` | $w_5 = w_4 + w_3$ | $\partial w_5 = \begin{bmatrix} 1 & 1 \end{bmatrix}$ | $\bar{w}_5 = 1$ |
| `2a` | $w_4 = \sin(w_1)$ | $\partial w_4 = \cos(w_1)$ | $\bar{w}_4 = \bar{w}_5 \cdot 1 = 1$ |
| `2b` | $w_3 = w_1 w_2$ | $\partial w_3 = \begin{bmatrix} w_2 & w_1 \end{bmatrix}$ | $\bar{w}_3 = \bar{w}_5 \cdot 1 = 1$ |
| `3b` | $w_2 = x_2$ | | $\bar{w}_2 = \bar{w}_3 w_1 = x_1$ |
| `3a` | $w_1 = x_1$ | | $\bar{w}_1 = \bar{w}_3 w_2 + \bar{w}_4 \cos(w_1) = x_2 + \cos(x_1)$ |

Wir bemerken, dass in einem Durchlauf, beide partiellen Ableitungen

$$\bar{w}_2 = \frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial x_2}, \quad \bar{w}_1 = \frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial x_1}$$

direkt berechnet werden. Allerdings müssen zunächst in einem Vorwärsdurchlauf die Gradienten $\partial w_i$ berechnet und gespeichert werden. Das kann bei komplexen Programmen durchaus ein Nachteil sein. Ein Ausweg bietet *checkpointing* wo nur wenige Zwischenetappen der Werte $w_i$ gespeichert werden, aus denen bei Bedarf die Nachfolger und Gradienten zwischen den Checkpoints erzeugt werden können.

Zusätzlich muss der Programmablauf (also welche Variablen aus welchen hervorgegangen sind – die sogenannte *Wengert Liste*) verfügbar sein.

Hätte $y$ mehrere Komponenten, müsste für jede Komponente die entsprechenden adjungierten in einem neuen Durchlauf berechnet werden.

Andersherum ist es beim Vorwärtsmodus, bei welchem mehrere Komponenten im Ausgang direkt berechnet werden aber für jede Eingangsvariable die Akkumulation separat durchgeführt werden muss.

# Chapter 11

# Implementierungen, Anwendungsbeispiele, Backpropagation

## 11.1 Exkurs – Gradienten und Repräsentation

Die Berechnung von Gradienten ist ebenso wesentlich wie schwierig in numerischen Algorithmen. Viele *erfolgreiche* Algorithmen basieren auf effizient berechenbaren Darstellungen oder Approximationen des Gradienten.

Ein erstes Beispiel ist der stochastische Gradientenabstieg, der auf einen Schätzer statt des eigentlichen Gradienten baut.

In der Optimierung mit (partiellen) Differentialgleichungen wie

$$\mathcal{J}(u) = \frac{1}{2}\int_0^T \|x(s) - x^*\|^2 + \|u(s)\|^2 \,\mathrm{d}s \to \min_u \quad \text{s.t. } \dot{x}(t) = Ax(t) + Bu(t)$$

macht es einen entscheidenden Unterschied, dass der Gradient

$$\partial_u \mathcal{J}(u) = u + B^T p$$

(Jan beachte, dass $u$ eine Funktion ist, also ein $\infty$-dimensionales Objekt) über die Lösung der adjungierten Gleichung

$$-\dot{p}(t) = A^T p(t) + x(t) - x^*, \quad p(T) = 0$$

definiert ist (siehe beispielsweise (Kurdila and Zabarankin 2005, Theorem 5.5.1) für ein abstraktes Resultat und bspw. (Tröltzsch 2009, Abschnitt 5.9.1) für eine Umsetzung in einem Gradientenabstiegsverfahren mit PDEs).

Während diese Resultate in der Theorie die Wohlgestelltheit der Optimierungsprobleme helfen zu analysieren, werden Sie in der Praxis gerne verwendet weil die Lösung einer Differentialgleichung einfacher umzusetzen ist (und im Zweifel auch effizienter) als die Berechnung von sehr abstrakten Gradienten.

Um abstraktere Gradienten zu charakterisieren hilft oftmals die Definition der *totalen Ableitung* einer Funktion $f\colon X \to Y$ bei einem $x \in X$ als die lineare Abbildung $L(x)\colon X \to Y$ derart, dass

$$f(x+h) - f(x) - L(x)[h] = o(\|h\|)$$

für $h \in X$ gilt.

Seien beispielsweise die Räume als $X = Y = \mathbb{R}^{n\times n}$ gegeben und

$$f(S) = A^T S + SA - SRS + Q \tag{11.1}$$

dann hat der *Riccati* Operator $f$ mit Koeffizienten $A$, $R$, $Q \in \mathbb{R}^{n\times n}$ demnach die Realisierung des Gradienten (an der Stelle $S_0$) gegeben als

$$L(S_0)[h] = (A^T - S_0 R)h + h(A - RS_0).$$

Für neuronale Netze ist der Gradient von $f(x; \tilde{A}, b) = \tilde{A}x + b$ bezüglich der *Gewichte* $\tilde{A} \in \mathbb{R}^{m\times n-1}$ und $b \in \mathbb{R}^m$ interessant. Zunächst mal verstehen wir die affine lineare Abbildung im projektiven Raum vermöge

$$y = Ax + b \leftrightarrow \begin{bmatrix} y \\ 1 \end{bmatrix} = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}$$

und betrachten einfach $f(x; A) = Ax$ mit $A \in \mathbb{R}^{m\times n}$. Gemäß der Ableitungsformel gilt

$$f(x; A+h) - f(x; A) - hx = (A+h)x - Ax - hx = 0 = o(\|h\|)$$

also im Prinzip ist $L(A) \leftrightarrow x$ die Ableitung wenn auch nicht als Vektor sondern als

$$L_{[x]}\colon \mathbb{R}^{n\times m} \to \mathbb{R}^n\colon h \to L_{[x]}h = hx.$$

> Weil hier an der Stelle $A$ abgeleitet wird, sollte hier $L(A)$ stehen. Da bei linearen Abbildung die Ableitung überall gleich ist, gilt $L(A) \equiv L$ unabhängig von $A$. Für später wird es interessant sein, dass die Ableitung direkt mit $x$ zusammenhängt, sodass wir das $x$ in die Definition mit aufnehmen.

Diese Abbildung $L_{[x]}$ ist linear und könnte als Matrix (bzw. Tensor im Sinne einer höherdimensionalen Datenstruktur) im $\mathbb{R}^{n\times(n\times m)}$ realisiert werden.

## 11.2 Backpropagation

Die besondere Architektur neuronaler Netze, wie sie im Machine Learning verwendet werden, impliziert eine besondere Struktur in der Gradientenberechnung.

Wir betrachten ein 2-layer Netzwerk

$$s(x) = \sigma(A_2\sigma(A_1x))$$

mit den *bias*-Vektoren $b_1$ und $b_2$ wie oben erläutert bereits in den Matrizen $A_1 \in \mathbb{R}^{n_0\times n_1}$ und $A_2 \in \mathbb{R}^{n_2\times n_1}$ integriert. Für das *learning* wird zu Datenpunkten $(x, y) \in \mathbb{R}^{n_0} \times \mathbb{R}^{n_2}$ die Ableitung der Funktion

$$(A_1, A_2) \to l((x, y); A_1, A_2) = \|y - s(x)\|_2^2$$

bezüglich des Parameter"vektors" $p = (A_2, A_1)$ gesucht.

Eine erste Anwendung der Kettenregel ergibt

$$\partial_p l((x, y); p) = -2\langle y - s(x; p),\, \partial_p s(x; p)\rangle$$

sodass wir uns erstmal um die Ableitung von $s$ bezüglich $p$ kümmern können.

Bezüglich $A_2$ ergibt die Kettenregel direkt

$$\partial_{A_2} = \sigma'(A_2x_1)L_{[x_1]}$$

wobei

- $x_1 := \sigma(A_1x)$ der Zwischenwert nach der vorletzten (hier der ersten) *layer* ist,
- $\sigma'(A_2x_1) \in \mathbb{R}^{n_2'\times n_2}$ die (diagonale) Jacobimatrix der komponentenweise definierten Aktivierungsfunktion $\sigma$
- und $L_{[x_1]} \in \mathbb{R}^{n_2\times(n_2\times n_1)}$ die Ableitung von $A_2x_1$ nach $A_2$ (vgl. oben).

Bezüglich $A_1$ ergibt die Kettenregel direkt

$$\partial_{A_1} = \sigma'(A_2x_1)A_2\sigma'(A_1x)L_{[x]}$$

### 11.2.1 Rekursion

Wir bemerken, dass die $A_i$ Komponenten des Gradienten von $s$ über

$$\delta_N := \sigma'(A_Nx_{N-1}), \quad \partial_{A_N}s(x) = \delta_N L_{[x_{N-1}]}$$

für die letzte *layer* und dann rekursiv über

$$\delta_{n-1} := \delta_n A_n\sigma'(A_{n-1}x_{n-1}), \quad \partial_{A_{n-1}}s(x) = \delta_{n-1}L_{[x_{n-2}]}$$

von der letzten bis zur ersten *layer* berechnet werden können.

## 11.3 Praktische Berechnung des Gradienten

Im schönsten *machine learning*-Sprech sagt Jan hier immer Gradient obwohl eher die Ableitung oder die Jacobi Matrix gemeint ist.

Mit $\hat{y} = -2(y - s(x;p)) \in \mathbb{R}^{n_2 \times 1}$ bekommen wir aus obigen Formeln, dass

$$\begin{aligned}\partial_{A_2} l((x,y);p) = \langle \hat{y}, \partial_{A_2} s(x;p) \rangle &= \hat{y}^T \sigma'(A_2 x_1) L_{[x_1]} \\ &= \tilde{y}^T L_{[x_1]},\end{aligned}$$

mit $\tilde{y} = \sigma'(A_2 x_1)\hat{y}$, was wegen der Diagonalität von $\sigma'$ nur eine Skalierung der einzelnen Elemente von $\hat{y}$ darstellt.

Eine Betrachtung der Dimensionen $\hat{y} \in \mathbb{R}^{n_2 \times 1}$, $\sigma'(A_2 x_1) \in \mathbb{R}^{n_2 \times n_2}$ und $L_{[x_1]} \in \mathbb{R}^{n_2 \times (n_2 \times n_1)}$ ergibt für das Produkt, dass

$$\tilde{y}^T L_{[x_1]} \in \mathbb{R}^{1 \times (n_2 \times n_1)}$$

sodass wir schon fast den Gradienten zum Parameter $A_2 \in \mathbb{R}^{n_2 \times n_1}$ addieren können.

Wir berechnen die Einträge $g_{1ij}$ des Gradienten $\tilde{y}^T L_{[x_1]}$ als die 1-te (und einzige – da der Bildraum 1-dimensional ist) Komponente (bzgl. die kanonischen Basis) auf die der $ij$ kanonische Basisvektor aus dem Urbildraum abgebildet wird. Hier bietet sich als kanonische Basis $\{e^{(ij)}\} \subset \mathbb{R}^{n_2 \times n_1}$ die Menge der Matrizen an, die 0 sind bis auf eine 1 in der $i$-ten Zeile an der $j$-ten Stelle. In der Tat gilt dann

$$\mathbb{R}^{n_2 \times n_1} \ni A = [a_{ij}] \leftrightarrow A = \sum_{i,j} a_{ij} e^{(ij)}$$

und weiter

$$g_{1ij} = \tilde{y}^T L_{[x_1]} e^{(ij)} = \tilde{y}^T e^{(ij)} x_1 = \tilde{y}_i (x_1)_j$$

sodass wir als 1-te und einzige (matrixwertige) Komponente des Gradienten das äußere Produkt von $\tilde{y}$ und $x_1$ erhalten

$$(\tilde{y}^T L_{[x_1]})_1 = \tilde{y} x_1^T \in \mathbb{R}^{n_2 \times n_1}.$$

Für die Ableitung bezüglich $A_k$ in einem $N$-layer Netzwerk bekommen wir mit obiger Rekursionsformel

$$\partial_{A_k} l(x;p) = \hat{y}^T \delta_k L_{[x_{k-1}]} = \tilde{\tilde{y}} x_{k-1}^T$$

mit

$$\tilde{\tilde{y}}^T = \hat{y}^T \sigma'(A_N x_N) A_N \sigma'(A_{N-1}) A_{N-1} \cdots \sigma'(A_k x_{k-1}).$$

## 11.4 Implementierungen und Beispiele

Jan könnte als grobe Einschätzung sagen:

> Die Eleganz und Effizienz von AD für Optimierungsanwendungen ist unbestritten. Dennoch bedeutet der Einsatz eine Abwägung von Extra-Implementierungsaufwand und Selbstbeschränkung in Struktur und Funktionsumfang des Codes (das sind zwei Seiten der gleichen Medaille – auch *nonstandard but clever* Funktionen können mit entsprechendem Aufwand A-differenziert werden). Deswegen ist AD in der Forschung (hier wird der Mehraufwand gescheut) und in der industriellen Anwendung (viel legacy code mit spezifisch cleveren Funktionen) eine Randerscheinung wenn auch mit leuchtenden Erfolgsgeschichten.

Sicherlich ist es am besten im Bezug auf Aufwand und Funtionalität, den code sofort mit Augenmerk auf AD zu entwickeln.

Soll der AD *overhead* nicht selbst mitentwickelt werden, empfiehlt es sich auf Programmumgebungen zurückzugreifen, die einen *AD framework* mitbringen. Hier greift obige Abwägung – die AD verursacht kaum Mehraufwand allerdings muss der eigene Code an die vorgegebene Struktur und Funktionalität angepasst sein.

Für bestehenden Code gibt es zwei Herangehensweisen (neben der "Um"entwicklung).

1. *precompiling* – "AD ready" Code wird automatisiert aus bestehendem Code generiert.
2. *overloading* – Es werden AD Variablentypen und Klassen definiert und die Grundoperationen *überladen* sodass mit minimalen Änderungen ein bestehender Code die AD Funktionalität bekommt.

### 11.4.1 Anwendungsbeispiele

- Algorithmic differentiation of an industrial airfoil design tool coupled with the adjoint CFD method – ein Beispiel in dem ein kompletter Simulationscode *automatisch differenziert* wurde um Tragflügel zu optimieren.

### 11.4.2 Implementierungen

- CASADI – eine *AD aware* (für `python` und `Matlab/Octave`) Umgebung zur Optimierung (auch mit Differentialgleichungen)
- PyTorch – eine *machine learning toolbox* mit AD integriert zur Gradientenberechnung
- ADOL-C – ein *overloading* framework für `C` und `C++`
- Tapenade – ein *precompiler* der aus Code *AD Code* generiert

## 11.5 Aufgaben

### 11.5.1 Ableitung und Newton der Riccati Gleichung

Weisen Sie nach, dass der Gradient der (matrixwertigen) Riccati Abbildung (11.1) tatsächlich die gegebene Form hat und formulieren Sie damit ein Newton Verfahren zur Lösung der Riccati Gleichung $f(S) = 0$.

### 11.5.2 Ableitung von $Ax$ nach der Matrix

Sei $A = [a_{ij}] \in \mathbb{R}^{m\times n}$. Bestimmen Sie partielle Ableitung der Funktion $f_x(A) = A(x)$ nach einer Komponente

$$\frac{\partial f_x}{\partial a_{ij}}(A)$$

und geben Sie davon ausgehend eine Darstellung der totalen Ableitung $\partial f_x(A)\colon \mathbb{R}^{m\times n} \to \mathbb{R}^m$ an

### 11.5.3 Backward Propagation

Leiten Sie aus $\mathbf{A} = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix}$ und $\mathbf{x} = (x, 1)$ sowie $\mathbf{y} = (y, 1)$ an der Stelle $[A\, b] = [A_0\, b_0]$ die Formel für den Gradienten

$$\partial_{[A\,b]}(\frac{1}{2}\|\mathbf{y} - \sigma(\mathbf{A}\mathbf{x})\|_2^2)\Big|_{[A_0\,b_0]} = \Big(\sigma'(A_0x + b_0)(y - \sigma(A_0x + b_0))\Big)\,\mathbf{x}^T$$

Bestätigen Sie sie numerisch, beispielsweise mit Hilfe der Code snippets aus Beispiel Implementierung, für ein Beispiel Netzwerk $s\colon \mathbb{R}^3 \to \mathbb{R}^2\colon \xi \to \sigma(\tilde{A}\xi + b)$ mit $\sigma = \tanh$ an der Stelle $A_0 = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$ und $b_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ beim Datenpunkt $(\tilde{x}, y) = ((1, 1, 1), (1, 1))$.

### 11.5.4 AD and Autograd in `pytorch`

Arbeiten Sie sie sich durch die `pytorch` tutorials

1. Introduction to `torch.autograd`
2. AD with `torch.autograd`

und erweitern Sie die Codes um den Gradienten (in Bezug auf $A \in \mathbb{R}^{2\times 2}$) von $x \to x^T Ax$ für $x \in \mathbb{R}^2$ an der Stelle $A_0 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ zu bestimmen.

# Chapter 12

# 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

Bollhöfer, M., Mehrmann, V.: Numerische Mathematik. Eine projektorientierte Einführung für Ingenieure, Mathematiker und Naturwissenschaftler. Vieweg (2004)
Byrne, C.L.: Lecture notes on iterative optimization algorithms, https://faculty.uml.edu/cbyrne/IOIPNotesOct2014.pdf, (2014)
Kurdila, A.J., Zabarankin, M.: Convex functional analysis. Birkhäuser (2005)
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)
Tröltzsch, F.: Optimale steuerung partieller differentialgleichungen. Vieweg+Teubner, Wiesbaden, Germany (2009)