# Optimization for AI

*Rade Kutil, 11. Januar 2026*

# 1 Einführung

Optimierung wird in der Mathematik oft auch als *Programm/Programming* bezeichnet. Die generelle Definition von Optimierung einer Funktion $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ ist:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in X} f(\boldsymbol{x}), \qquad X \subseteq \mathbb{R}^n$$

Es gibt mehrere Unterscheidungskriterien, jeweils von einfach bis schwierig:

- Art der Funktion $f$

  **Linear** $f(\boldsymbol{x}) = A\boldsymbol{x}$. Optimierung macht nur in Zusammenhang mit Constraints Sinn.

  **Least Squares** $f(\boldsymbol{x}) = \|A\boldsymbol{x} + \boldsymbol{b}\|_2^2$.

  **Quadratische Programmierung** $f(\boldsymbol{x}) = \boldsymbol{x}^\mathsf{T} A\boldsymbol{x} + \boldsymbol{b}^\mathsf{T}\boldsymbol{x}$.

  **Konvexe Optimierung** $f(\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y}) \leq \alpha f(\boldsymbol{x}) + (1-\alpha) f(\boldsymbol{y})$. Auch die Definitionsmenge $X$ muss üblicherweise konvex sein.

  **Generell nichtlinear** Ein lokales Minimum ist nicht automatisch ein globales Minimum.

- Verfügbarkeit von Gradienten

  **Keine Ableitung** Nur Funktionswerte verfügbar.

  **Erste Ableitung** Gradient-Descent.

  **Zweite Ableitung** Hesse-Matrix. Newton-Methode.

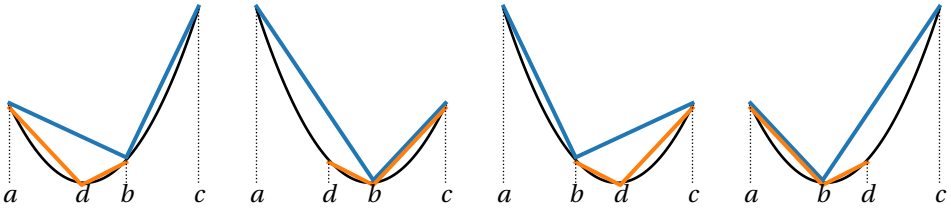
- Constraints (Nebenbedingungen)

  **Unconstrained**

  **Equality Constraints** $g_i(\boldsymbol{x}) = 0$. Lagrange-Multiplikator.

  **Inequality Constraints** $h_i(\boldsymbol{x}) \geq 0$. Karush-Kuhn-Tucker (KKT).

All diese Möglichkeiten gibt es in vielen Kombinationen. Daher ist es schwierig, eine vernünftige Reihenfolge aller Methoden zu finden.




(a) $b-a > c-b$, $f(d) < f(b)$ (b) $b-a > c-b$, $f(d) > f(b)$ (c) $b-a < c-b$, $f(d) < f(b)$ (d) $b-a < c-b$, $f(d) > f(b)$

Abbildung 1: Golden-Section-Search. Blau: altes Tripel $(a,b,c)$, orange: neues Tripel.

## 2 Golden-Section-Search

Eindimensional, konvex, keine Ableitung, unconstrained.

Dieses Verfahren ist ähnlich zur Nullstellensuche mittels Binary-Search (Bisektionsverfahren). Dabei wird ein Intervall $[a,b]$ mit entweder $f(a) < 0$ und $f(b) > 0$ (oder umgekehrt) in zwei Hälften $[a, \frac{a+b}{2}]$ und $[\frac{a+b}{2}, b]$ geteilt. Wenn $f(\frac{a+b}{2}) > 0$ ist, wird mit ersterem Intervall rekursiv fortgefahren, ansonsten mit zweiterem (oder umgekehrt).

Wenn nicht eine Nullstelle sondern ein Minimum gesucht wird, reicht ein Intervall mit einem Mittelpunkt nicht aus, weil aus $f(\frac{a+b}{2}) < f(a) < f(b)$ nicht ersichtlich ist, ob das Minimum links oder rechts von $\frac{a+b}{2}$ liegt. Stattdessen braucht man ein Tripel $(a,b,c)$ mit $a < b < c$, $f(b) < f(a)$, $f(b) < f(c)$.

Es gibt nun zwei Möglichkeiten: $b-a > c-b$, also das linke Teilintervall ist größer, oder umgekehrt. Wir betrachten vorerst nur die erste Möglichkeit. Es wird nun ein $d$ mit $a < d < b$ gewählt. Ist nun $f(d) < f(b)$ wird mit dem Tripel $(a,d,b)$ fortgefahren (Abbildung 1 (a)). Ist $f(d) > f(b)$ wird mit $(d,b,c)$ fortgefahren (Abbildung 1 (b)). Dabei wäre von Vorteil, wenn die Verhältnisse der Teilintervalle gleich bleiben. Das Verhältnis von Gesamtintervall zu größeren Teilintervall ist für $(a,b,c)$ und die möglichen neuen Tripel $(a,d,b)$ und $(d,b,c)$ sollte dann gleich sein:

$$\frac{c-a}{b-a} = \varphi \qquad \frac{b-a}{d-a} = \varphi \qquad \frac{c-d}{c-b} = \varphi$$

$\varphi$ ist dann der goldene Schnitt, für den gilt:

$$\varphi = \frac{1+\sqrt{5}}{2} \qquad \varphi^2 = \frac{(1+\sqrt{5})^2}{4} = \frac{1+2\sqrt{5}+5}{4} = \frac{3+\sqrt{5}}{2} = 1+\varphi$$

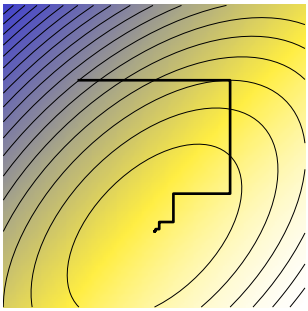


Abbildung 2: Coordinate-Descent

*Beweis:* Wenn $b$ und $d$ so gewählt wird, dass die ersten zwei Gleichungen gelten, dann gilt auch die dritte:

$$\frac{c-d}{c-b} = \frac{(c-a)-(d-a)}{(c-a)-(b-a)} \overset{/(b-a)}{=\!=} \frac{\varphi - 1/\varphi}{\varphi - 1} \overset{\cdot\varphi}{=\!=} \frac{\varphi^2-1}{\varphi^2-\varphi} = \frac{1+\varphi-1}{1+\varphi-\varphi} = \varphi$$

Wenn $b-a < c-b$ ist, also das rechte Teilintervall größer, dann gilt das gleiche, nur mit $a$ und $c$ vertauscht, sodass sich Abbildung 1 (c) und (d) ergibt.

Also: Entweder ist $b = a + (c-a)/\varphi$ (wenn $b-a > c-b$) und dann

$$d = a + \frac{b-a}{\varphi} = a + \frac{c-a}{\varphi^2} \overset{\varphi^2-\varphi=1}{=\!=} \frac{a\varphi^2 + (c-a)(\varphi^2-\varphi)}{\varphi^2} = c + \frac{a-c}{\varphi}$$

oder $b = c + (a-c)/\varphi$ (wenn $b-a < c-b$) und dann $d = a + (c-a)/\varphi$.

# 3 Coordinate Descent

| Konvex/generell, keine (komplette) Ableitung, unconstrained. |
|---|

Hier wird in jeder Iteration eine Koordinate (oder mehrere) ausgewählt. Danach wird eine eindimensionale Optimierung entlang dieser Koordinate durchgeführt.

Beispiel (siehe Abbildung 2): Für die Funktion

$$f(x,y) = x^2 + y^2 - xy$$

wird abwechselnd die $y$- und $x$-Koordinate ausgewählt. Dann wird die jeweils andere Koordinate als konstant betrachtet. Wir minimieren die Funktion eindimensional durch Ableiten und Nullsetzen:

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}x} = 2x - y = 0 \qquad \Rightarrow \quad x = \frac{y}{2}$$

Für die andere Koordinate:

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}y} = 2y - x = 0 \qquad \Rightarrow \quad y = \frac{x}{2}$$

Mögliche Varianten:

- Statt Ableitung-Nullsetzen z.B. Golden-Section-Search oder ähnliches.
- Nicht unbedingt volle eindimensionale Iterationen für jede Koordinaten-Iteration. Z.B. nur ein Schritt in die Koordinaten-Richtung, dann schon die andere Koordinate.
- Auswahl mehrerer Koordinaten. Wenn z.B. $f(\boldsymbol{x}, \boldsymbol{y})$ nach $\boldsymbol{x}$ und $\boldsymbol{y}$ ableitbar und nullsetzbar ist, aber nicht nach $(\boldsymbol{x}, \boldsymbol{y})$ gemeinsam. Das nennt man dann üblicherweise *alternierende* Optimierung.
- Optimierung in jeder Koordinate versuchen. Dann die beste Koordinate auswählen.

# 4 Pattern-Search

Konvex/generell, keine Ableitung, unconstrained.

Hier werden um einen Suchpunkt in einem Muster um den Suchpunkt mehrere Punkte evaluiert. Der Suchpunkt wird dann auf den Punkt mit dem kleinsten Funktionswert verschoben. Hat der Suchpunkt selbst den kleinsten Funktionswert, wird das Muster verkleinert.

Siehe Abbildung 3. Der Algorithmus ist:

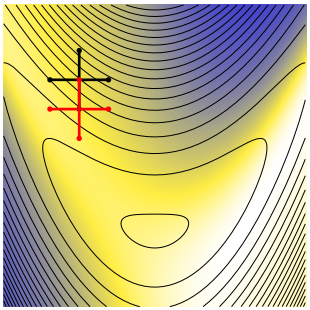
$P = \text{pattern}(\boldsymbol{x})$ (Beispiel: $P = \boldsymbol{x} + s \cdot \{(\pm 1, 0, \ldots), (0, \pm 1, \ldots), \ldots\}$)
$f(\boldsymbol{x}) < \min f(P) \quad \Rightarrow \quad \underset{\rightarrow}{s} = s/2$
$\boldsymbol{u} \in P \wedge f(\boldsymbol{u}) = \min f(P \cup \{\boldsymbol{x}\}) \quad \Rightarrow \quad \underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{u}$

# 5 Nelder-Mead

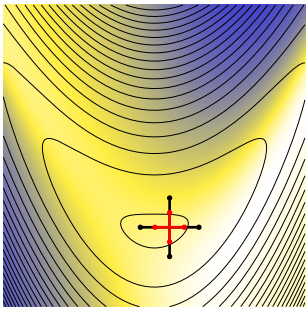
Konvex/generell, keine Ableitung, unconstrained.

Der aktuelle Zustand im Nelder-Mead-Algorithmus ist ein Simplex in $n$ Dimensionen, das sind $n+1$ Ecken $X = \{\boldsymbol{x}_i \mid i \in 1 \ldots n+1\}$. Die Ecken werden nach Funktionswert sortiert:

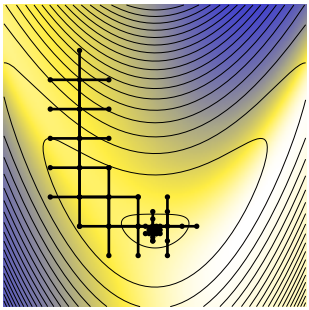
$$f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \ldots \leq f(\boldsymbol{x}_{n+1})$$



(a) Randpunkt minimal

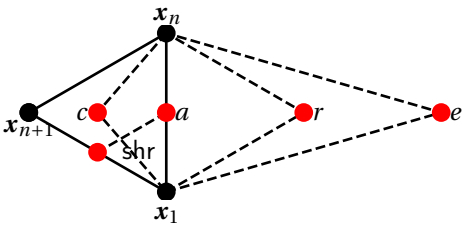


(b) Suchpunkt minimal

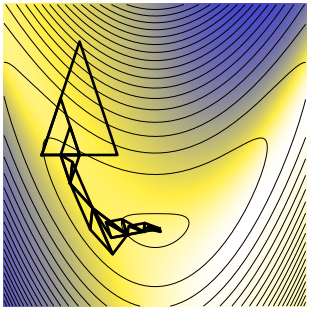


(c) Iteration

Abbildung 3: Pattern-Search. Schwarz: Ausgangszustand, rot: Folgezustand.




(a) Diagramm: Zentroid $a$, Contraction $c$, Reflection $r$, Expansion $e$, Shrink shr



(b) Plot

Abbildung 4: Nelder-Mead-Algorithmus

Dann wird eines der Operationen Reflection, Expansion, Contract, Shrink durchgeführt, mit Hilfe des Zentroids der Ecken ohne der Ecke mit dem größten Funktionswert (Mittelpunkt von $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$).

Der Algorithmus in Abbildung 4 und Abbildung 5 probiert zuerst die Reflection der schlechtesten Ecke $\boldsymbol{x}_{n+1}$ am Zentroid der anderen $n$ Ecken aus. Führt diese zu einem besseren Funktionswert als die beste bisherige Ecke $\boldsymbol{x}_1$, wird zusätzlich noch eine Expansion in die selber Richtung probiert. Falls diese noch besser ist, wird sie an Stelle der schlechtesten Ecke übernommen. Falls die Reflection wenigstens besser als die bisher zweitschlechteste Ecke ist, wird sie übernommen. Andernfalls wird eine Contraction zwischen $\boldsymbol{x}_{n+1}$ und dem Zentroid übernommen, falls sie eine Verbesserung gegenüber der Reflection und der schlechtesten

$\boldsymbol{a} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i$ Zentroid
$\boldsymbol{r} = \boldsymbol{a} + q_r(\boldsymbol{a} - \boldsymbol{x}_{n+1})$ Reflection, $q_r = 1$
$\boldsymbol{e} = \boldsymbol{a} + q_e(\boldsymbol{a} - \boldsymbol{x}_{n+1})$ Expansion, $q_e = 2$
$\boldsymbol{c} = \boldsymbol{a} - q_c(\boldsymbol{a} - \boldsymbol{x}_{n+1})$ Contraction, $q_c = 0.5$
$f(\boldsymbol{e}) < f(\boldsymbol{r}) < f(\boldsymbol{x}_1) \Rightarrow$
$\underset{\rightarrow}{X} = \text{sort}_f\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{e}\}$
else $f(\boldsymbol{r}) < f(\boldsymbol{x}_n) \Rightarrow$
$\underset{\rightarrow}{X} = \text{sort}_f\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{r}\}$
else $f(\boldsymbol{c}) < \min(f(\boldsymbol{r}), f(\boldsymbol{x}_{n+1})) \Rightarrow$
$\underset{\rightarrow}{X} = \text{sort}_f\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{c}\}$
else $\Rightarrow$
$\underset{\rightarrow}{X} = \text{sort}_f(X + \boldsymbol{x}_1)/2$ Shrink

Abbildung 5: Nelder-Mead-Algorithmus. $\text{sort}_f$ sortiert aufsteigend nach dem Funktionswert $f(\boldsymbol{x}_i)$.

Ecke darstellt. Falls so keine bessere Ecke gefunden wird, wird der ganze Simplex in Richtung der besten Ecke geschrumpft.

Es gibt einige modifizierte Varianten des Algorithmus mit anderen Konstanten $q_r, q_e, q_c$ und leicht anderen Bedingungen/Entscheidungen.

# 6 Line-Search

Generell, keine/erste Ableitung, unconstrained.

Line-Search ist ein Hilfsmittel, um bei gegebener Optimierungs-Richtung $\boldsymbol{d}$ eine geeignete Schrittweite $\alpha$ zu finden.

$$\alpha = \arg\min_{\alpha} f(\underset{\rightarrow}{\boldsymbol{x}}) = \arg\min_{\alpha} f(\boldsymbol{x} + \alpha\boldsymbol{d})$$

Sehr oft ist die Optimierungsrichtung $\boldsymbol{d} = -\nabla f(\boldsymbol{x})$ (Gradient-Descent, Abschnitt 7). Es reicht aber, wenn in einem (kleinen) Bereich $X$ um $\boldsymbol{x} \in X$ gilt:

$$\forall \boldsymbol{y} \in X : f(\boldsymbol{y}) - f(\boldsymbol{x}) \geq -\boldsymbol{d}^{\mathsf{T}}(\boldsymbol{y} - \boldsymbol{x})$$

Diese Definition funktioniert auch an Stellen, an denen $f$ nicht differenzierbar ist. Siehe Abbildung 6

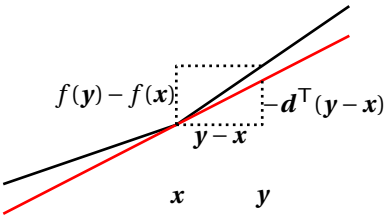



Abbildung 6: Subgradient. Schwarz: Eine Funktion $f$, die an $\boldsymbol{x}$ nicht differenzierbar ist. Rot: Die Gerade (Ebene, Hyperebene), die durch $-\boldsymbol{d}$ aufgespannt wird.

## 6.1 Exact Line-Search

Durch übliches Ableiten (nach $\alpha$) und Nullsetzen kann man die optimale Schrittweite ermitteln:

$$0 = \frac{\mathrm{d}f(\boldsymbol{x}+\alpha\boldsymbol{d})}{\mathrm{d}\alpha} = \nabla f(\boldsymbol{x}+\alpha\boldsymbol{d})^\top \frac{\mathrm{d}(\boldsymbol{x}+\alpha\boldsymbol{d})}{\mathrm{d}\alpha} = \nabla f(\boldsymbol{x}+\alpha\boldsymbol{d})^\top \boldsymbol{d} =: g(\alpha)$$

$\boldsymbol{d}$ und $\boldsymbol{x}$ sind Konstanten in der Nullstellensuche von $g(\alpha)$. $g(\alpha) = 0$ ist univariat und oft ein Polynom, und lässt sich daher mit bekannten Methoden lösen. Man beachte, dass $g(\alpha) = 0 \Leftrightarrow \nabla f(\underset{\rightarrow}{\boldsymbol{x}})^\top \nabla \boldsymbol{d} = 0$ ist, d.h. man sucht eine Stelle entlang der Geraden $\boldsymbol{x}+\alpha\boldsymbol{d}$, an der der Gradient orthogonal zur aktuellen Optimierungsrichtung ist.

## 6.2 Numeric Line-Search

Benutze eine eindimensionale Methode (z.B. Golden-Section-Search), um $f(\boldsymbol{x}+\alpha\boldsymbol{d})$ zu minimieren.

## 6.3 Backtracking Line-Search

Es ist nicht unbedingt notwendig, im Line-Search wirklich das (ein) Minimum zu finden. Es reicht, wenn der Funktionswert ausreichend kleiner wird. Das ist für ein genügend kleines $\alpha$ sicher der Fall. Dafür reicht es, $\alpha$ sukzessive zu verkleinern.

$$\underset{\rightarrow}{\alpha} = \rho\alpha \qquad 0 < \rho < 1$$

Wiederhole das, bis z.B. $f(\boldsymbol{x}+\alpha\boldsymbol{d}) < f(\boldsymbol{x})$.

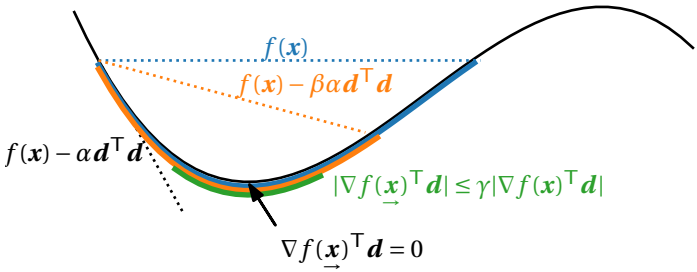



Abbildung 7: Wolfe-Conditions. Blau: Erlaubte Region für normales Backtracking. Orange: Für die erste Wolfe-Condition. Grün: Für die zweite Wolfe-Condition.

### 6.4 Wolfe-Conditions

Die Bedingung $f(\boldsymbol{x}+\alpha\boldsymbol{d}) < f(\boldsymbol{x})$ garantiert leider keine Konvergenz. Die Funktionsreduktion könnte einfach immer kleiner werden. Daher wird folgende *erste Wolfe-Condition* oder *Armijo-Regel* eingeführt:

$$f(\boldsymbol{x}+\alpha\boldsymbol{d}) \leq f(\boldsymbol{x}) + \beta\alpha\boldsymbol{d}^\top\nabla f(\boldsymbol{x}) = f(\boldsymbol{x}) - \beta\alpha\boldsymbol{d}^\top\boldsymbol{d} \qquad 0 < \beta < 1$$

Beachte: Nicht in allen Optimierungsansätzen ist $\boldsymbol{d} = -\nabla f(\boldsymbol{x})$. Man kann grundsätzlich jede Richtung mit $\boldsymbol{d}^\top\nabla f(\boldsymbol{x}) < 0$ wählen.

Die zweite Wolfe-Condition benutzt, dass im Idealfall $\nabla f(\underset{\rightarrow}{\boldsymbol{x}})^\top\nabla f(\boldsymbol{x}) = 0$ sein soll. Annähernd soll das Produkt zumindest kleiner als das aktuelle werden:

$$|\nabla f(\underset{\rightarrow}{\boldsymbol{x}})^\top\boldsymbol{d}| \leq \gamma|\nabla f(\boldsymbol{x})^\top\boldsymbol{d}| \qquad 0 < \gamma < 1$$

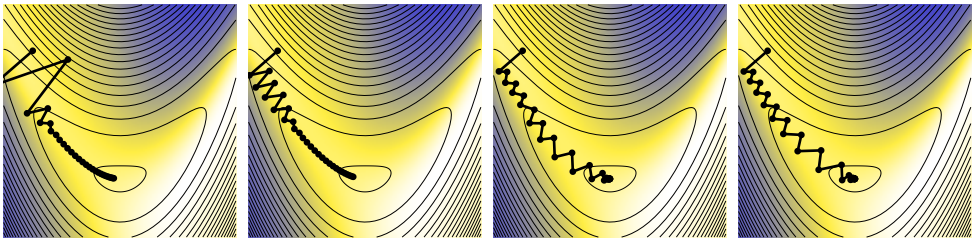
Siehe Abbildung 7.

## 7 Gradient-Descent

Generell, erste Ableitung, unconstrained.

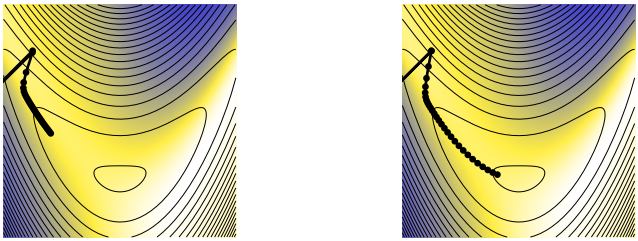
Gradient-Descent-Methoden ermitteln die Richtung des steilsten Abstiegs der Funktionswertes, was dem negativen Gradienten entspricht, und geht dann einen Schritt in diese Richtung, wobei die Schrittweite mit einem Parameter $\alpha$ kontrolliert wird.

$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha\nabla f(\boldsymbol{x}) = \boldsymbol{x} + \alpha\boldsymbol{d}$$



(a) ohne Line-Search (b) Backtracking (c) Wolfe-Condition 1 (d) Wolfe-Condition 2

Abbildung 8: Gradient-Descent



(a) Adagrad (b) RMSProp

Abbildung 9: Gradient-Descent mit Koordinaten-adaptiven Lernraten

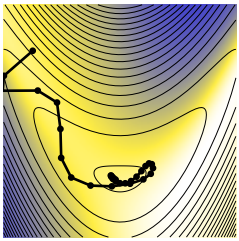
Es gibt viele Varianten und Methoden, $\alpha$ festzulegen, und die Schrittrichtung zu optimieren.

Abbildung 8 zeigt das Verhalten von Gradient-Descent mit diversen Line-Search-Varianten.

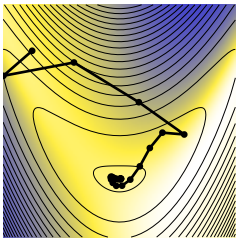
## 7.1 Adagrad

Oft benötigen manche Parameter eine andere Lernrate $\alpha$ als andere. Die Methode der adaptive Gradients (Adagrad) löst das, indem für jede Komponente $\boldsymbol{x}$ ein adaptiver Lernfaktor aus der Summe der bisherigen quadrierten Gradienten-Komponenten berechnet wird. Das führt allerdings zu immer kleiner werdenden Lernschritten, oft noch bevor das Minimum erreicht wird. Siehe Abbildung 9 (a).

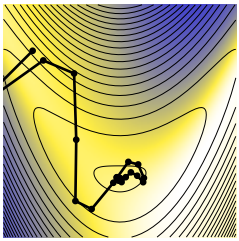
$$\underset{\rightarrow i}{x} = x_i + \frac{\alpha}{\epsilon + \sqrt{s_i}} d_i$$
$$\underset{\circ i}{s} = 0$$
$$s_i = \underset{\leftarrow i}{s} + d_i^2$$



(a) Momentum



(b) Nesterov



(c) Adam

Abbildung 10: Gradient-Descent mit Momentum-Methoden

$\epsilon$ ist dabei eine sehr kleine Zahl, und dient nur dazu, Divisionen durch 0 zu vermeiden. $\alpha$ ist wie üblich die generelle Lernrate.

## 7.2 RMSProp

Die RMSProp-Methode versucht daher, einen abklingenden Mittelwert der quadrierten Gradienten-Komponenten zu berechnen. Siehe Abbildung 9 (b).

$$\underset{\rightarrow i}{x} = x_i + \frac{\alpha}{\epsilon + \sqrt{s_i}} d_i$$
$$\underset{\circ i}{s} = 0$$
$$s_i = \gamma \underset{\leftarrow i}{s} + (1-\gamma) d_i^2 \qquad \gamma = 0.9$$

## 7.3 Momentum

Da die Konvergenz entlang eines flachen Tales bei den bisherigen Methoden relativ langsam ist, versucht man mit Hilfe eines Bewegungsimpulses (Momentum) eine Beschleunigung zu erzielen, vergleichbar mit einem Ball, der das Tal entlang rollt und dabei beschleunigt. Siehe Abbildung 10 (a).

$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \underset{\rightarrow}{\boldsymbol{v}}$$
$$\underset{\rightarrow}{\boldsymbol{v}} = \beta \boldsymbol{v} + \alpha \boldsymbol{d}$$

## 7.4 Nesterov-Momentum

Ein Problem bei der normalen Momentums-Methode ist das Überschießen an der Miminums-Stelle. Die Nesterov-Momentum-Methoden versucht, das in den Griff zu bekommen, indem statt des aktuellen Gradienten den an der potentiell nächsten Iterationsstelle $\boldsymbol{x} + \beta \boldsymbol{v}$ zu verwenden. Siehe Abbildung 10 (b).

$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \underset{\rightarrow}{\boldsymbol{v}}$$
$$\underset{\rightarrow}{\boldsymbol{v}} = \beta \boldsymbol{v} - \alpha \nabla f(\boldsymbol{x} + \beta \boldsymbol{v})$$

## 7.5 Adam

Die Adaptive-Momentum-Estimation-Methode (Adam) vereint die Momentum- und RMSProp-Methode. Außerdem werden korrigierte Varianten von $\boldsymbol{s}$ und $\boldsymbol{v}$ berechnet ($\hat{\boldsymbol{s}}$, $\hat{\boldsymbol{v}}$), die das Problem der Null-Initialisierung von $\boldsymbol{s}$ und $\boldsymbol{v}$ ausgleichen sollen. Siehe Abbildung 10 (c).

$$\underset{\rightarrow i}{x} = x_i + \frac{\alpha}{\epsilon + \sqrt{\hat{s}_i}} \underset{\rightarrow i}{\hat{v}}$$
$$\underset{\circ i}{s} = 0$$
$$s_i = \gamma \underset{\leftarrow i}{s} + (1-\gamma) d_i^2 \qquad \gamma = 0.9$$
$$\hat{\boldsymbol{s}} = \boldsymbol{s} / (1 - \gamma^k)$$
$$\underset{\rightarrow}{\boldsymbol{v}} = \beta \boldsymbol{v} + (1-\beta) \boldsymbol{d}$$
$$\underset{\rightarrow}{\hat{\boldsymbol{v}}} = \underset{\rightarrow}{\boldsymbol{v}} / (1 - \beta^k)$$
$$\underset{\circ}{k} = 0 \quad k = \underset{\leftarrow}{k} + 1$$

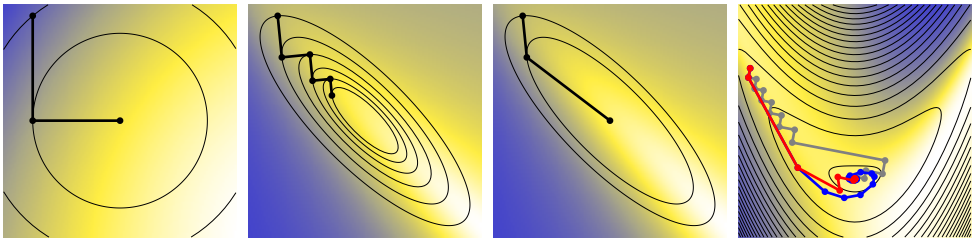
Für konstantes $d_i^2$ gilt

$$s_i = d_i^2 (1 - \gamma^k) \qquad \text{weil} \qquad \underset{\circ i}{s} = d_i^2 (1 - \gamma^0) = 0 \checkmark$$
$$\text{und} \quad \underset{\rightarrow i}{s} = \gamma s_i + (1-\gamma) d_i^2 = \gamma d_i^2 (1 - \gamma^k) + (1-\gamma) d_i^2$$
$$= d_i^2 (\gamma - \gamma^{k+1} + 1 - \gamma) = d_i^2 (1 - \gamma^{k+1}) \checkmark$$

Daher ist $s_i / (1 - \gamma^k)$ eine bessere Schätzung für $d_i^2$, vor allem für kleine $k$ für die ersten Iterationen. Für $\boldsymbol{v}$ gilt dasselbe.

## 7.6 Conjugate Gradient

| Generell/quadratisch, erste Ableitung, unconstrained. |
|---|



(a) Orthogonale Richtungen für $A = I$: $\boldsymbol{x}^\mathsf{T} A\boldsymbol{x} = \boldsymbol{x}^\mathsf{T}\boldsymbol{x}$

(b) Orthogonale Richtungen für $A$ nicht-diagonal

(c) Konjugierte Richtungen

(d) Nicht-quadratische Funktion $f$. Grau: normaler Gradient-Descent, blau: Fletcher-Reeves, rot: Polak-Ribière.

Abbildung 11: Conjugate-Gradient

Quadratische Funktionen haben die Form

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^\mathsf{T} A\boldsymbol{x} + \boldsymbol{b}^\mathsf{T}\boldsymbol{x} + c\,,$$

wobei $A$ eine symmetrische positiv definite Matrix ist, für die $\boldsymbol{x}^\mathsf{T} A\boldsymbol{x} > 0$ für alle $\boldsymbol{x} \neq 0$ gilt. Für solche Matrizen gibt es eine (invertierbare) Transformationsmatrix $B$ mit $A = B^\mathsf{T} B$. Es gilt:

$$\nabla f(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}\,.$$

Wir verwenden hier *exact Line-Search*. In diesem Fall sind aufeinander folgende Gradienten orthogonal (siehe Abschnitt 6.1). Wenn $A = I$ (Identitätsmatrix) ist, können wir das Minimum in $n$ Schritten ($n$ ist die Dimension, hier 2) finden (siehe Abbildung 11(a)), selbst wenn orthogonale Richtungen unabhängig vom Gradienten vorgegeben sind. Wenn $A$ nicht-diagonal ist, führt das hingegen zu einem Zick-Zack-Pfad (siehe Abbildung 11(b)). Die Idee ist nun, eine Art Orthogonalität auf einem durch $B$ transformierten Koordinatensystem einzuführen (siehe Abbildung 11(c)). Das nennt man dann *konjugiert* statt orthogonal.

Zwei Vektoren $\boldsymbol{x}$ und $\boldsymbol{y}$ sind *konjugiert* bezüglich $A$, wenn

$$0 = \boldsymbol{x}^\mathsf{T} A\boldsymbol{y} = \boldsymbol{x}^\mathsf{T} B^\mathsf{T} B\boldsymbol{y} = (B\boldsymbol{x})^\mathsf{T}(B\boldsymbol{y})\,,$$

also wenn die transformierten Vektoren orthogonal sind.

Wir starten wie üblich mit einer Optimierungsrichtung

$$\underset{\circ}{\boldsymbol{d}} = -\nabla f(\boldsymbol{x}), \qquad \underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \alpha \boldsymbol{d}.$$

In folgenden Iterationen soll die Richtung $\boldsymbol{d}$ nun so gewählt werden, dass sie zur vorherigen konjugiert ist. Und zwar wählen wir den Ansatz

$$\boldsymbol{d} = -\nabla f(\boldsymbol{x}) + \beta \underset{\leftarrow}{\boldsymbol{d}}.$$

$\beta$ muss dann wie folgt berechnet werden:

$$\boldsymbol{d}^\mathsf{T} A \underset{\leftarrow}{\boldsymbol{d}} = 0 \quad \Rightarrow \quad \beta = \frac{\nabla f(\boldsymbol{x})^\mathsf{T} A \underset{\leftarrow}{\boldsymbol{d}}}{\underset{\leftarrow}{\boldsymbol{d}}^\mathsf{T} A \underset{\leftarrow}{\boldsymbol{d}}}$$

Für nicht-quadratische Funktionen kann diese Methode auch verwendet werden, weil die meisten Funktionen lokal durch eine quadratische Funktion approximiert werden können, zumindest am Konvergenzpunkt. Leider ist in diesem Fall aber $A$ nicht bekannt, bzw. wäre nur mittels der zweiten Ableitung eruierbar. Es gibt aber äquivalente Formeln für $\beta$, die auch ohne Kenntnis von $A$ berechnet werden können. Diese dienen als gut funktionierende Annäherungen, und zwar einmal nach *Fletcher-Reeves*:

$$\beta = \frac{\nabla f(\boldsymbol{x})^\mathsf{T} \nabla f(\boldsymbol{x})}{\nabla f(\underset{\leftarrow}{\boldsymbol{x}})^\mathsf{T} \nabla f(\underset{\leftarrow}{\boldsymbol{x}})},$$

oder nach *Polak-Ribière*:

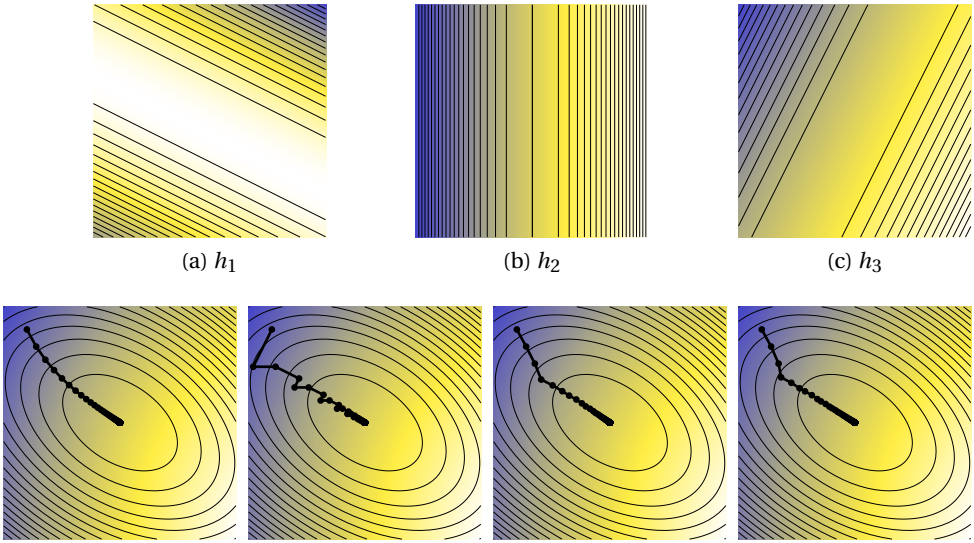
$$\beta = \frac{\nabla f(\boldsymbol{x})^\mathsf{T} (\nabla f(\boldsymbol{x}) - \nabla f(\underset{\leftarrow}{\boldsymbol{x}}))}{\nabla f(\underset{\leftarrow}{\boldsymbol{x}})^\mathsf{T} \nabla f(\underset{\leftarrow}{\boldsymbol{x}})},$$

wobei bei letzterem Schema $\beta$ nach unten mit 0 beschränkt werden sollte (Reset). Einen Vergleich zeigt Abbildung 11(d).

## 7.7 Stochastic Gradient-Descent

Oft ist die Zielfunktion $f(\boldsymbol{x})$ als Summe (bzw. Mittelwert) von $n$ einzelnen Teilfunktionen $g$ gegeben, die von Datenpunkten $\boldsymbol{y}_i$ abhängen. Dann ergibt sich:

$$f(\boldsymbol{x}) = \frac{1}{n} \sum_i g(\boldsymbol{x}, \boldsymbol{y}_i) = \frac{1}{n} \sum_i h_i(\boldsymbol{x})$$



(a) $h_1$ (b) $h_2$ (c) $h_3$

(d) Full Gradient-Descent (e) Stochastic Gradient-Descent (f) SAG (g) SVRG

Abbildung 12: Stochastic gradient descent

Zum Beispiel ist beim Trainieren von künstlichen neuronalen Netzen $\boldsymbol{y}$ ein Satz von Trainingsdaten (Batch), die sich in Inputs $\boldsymbol{y}_{\text{in}}$ und gewünschten Outputs $\boldsymbol{y}_{\text{out}}$ unterteilen. Die Zielfunktion ist dann die Summe aller Abweichungen der Prediction-Funktion $\boldsymbol{p}$ und den Outputs aus $\boldsymbol{y}_{\text{out}}$. $\boldsymbol{p}(\boldsymbol{x}, \boldsymbol{y}_{\text{in},i})$ berechnet mit Hilfe der Parameter (Gewichte) des neuronalen Netzes aus den Input-Daten $\boldsymbol{y}_{\text{in},i}$ eine Annäherung an den gewünschten Output $\boldsymbol{y}_{\text{out},i}$.

$$g(\boldsymbol{x}, \boldsymbol{y}_i) = |\boldsymbol{p}(\boldsymbol{x}, \boldsymbol{y}_{\text{in},i}) - \boldsymbol{y}_{\text{out},i}|^2$$

Als Beispiel wählen wir $\boldsymbol{p}(\boldsymbol{x}, y_{\text{in}}) = x_1 y_{\text{in}} + x_2$, $\boldsymbol{y}_{\text{in}} = (2, 0, -0.5)$, $\boldsymbol{y}_{\text{out}} = (5, 1, 0)$, was zur Lösung $\boldsymbol{x} = (2, 1)$ führen sollte. Die einzelnen Fehlerfunktionen $h_1$, $h_2$, $h_3$ sind in Abbildung 12(a-c) dargestellt.

Üblicherweise müsste man hier den Gradienten von $f$ berechnen, der sich aus der Summe der Gradienten von $g(\boldsymbol{x}, \boldsymbol{y}_i)$ für alle $i$ zusammensetzt.

$$\boldsymbol{d} = -\nabla f(\boldsymbol{x}) = -\nabla \frac{1}{n} \sum_i g(\boldsymbol{x}, \boldsymbol{y}_i) = -\frac{1}{n} \sum_i \nabla_{\boldsymbol{x}} g(\boldsymbol{x}, \boldsymbol{y}_i) = -\frac{1}{n} \sum_i \nabla h_i(\boldsymbol{x})$$

Stattdessen führt man bei der Stochastic Gradient-Descent-Methode den Update-Schritt für jeden der $n$ Einträge in den Trainingsdaten durch.

$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha \nabla g(\boldsymbol{x}, \boldsymbol{y}_i)$$
$$\underset{\circ}{i} = 0 \qquad \underset{\rightarrow}{i} = (i+1) \bmod n$$

Den Vergleich zwischen „Full Gradient-Descent" und Stochastic Gradient-Descent sieht man in Abbildung 12(d) und (e).

Der Nachteil dabei ist, dass je nach (zufälliger) Sortierung der Trainingsdaten $\boldsymbol{x}$ in beinahe zufällige Richtungen springen kann, die sich nur durch spätere Iterationen wieder ausgleichen. Der Vorteil ist allerdings, dass die Berechnung der Gradienten viel schneller ist, und daher viel mehr Iterationen pro Zeit ausgeführt werden können.

Um die Zufälligkeit dieser Methode doch etwas einzuschränken, kann man die Trainingsdaten (Batch, Größe $n$) in sogenannte *Minibatches* der Größe $b$ unterteilen und die Gradienten über die Minibatches summieren.

$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha \sum_{k=0}^{b-1} \nabla g(\boldsymbol{x}, \boldsymbol{y}_{(i+k) \bmod n})$$
$$\underset{\circ}{i} = 0 \qquad \underset{\rightarrow}{i} = (i+b) \bmod n$$

## 7.8 Stochastic Variance-Reduction

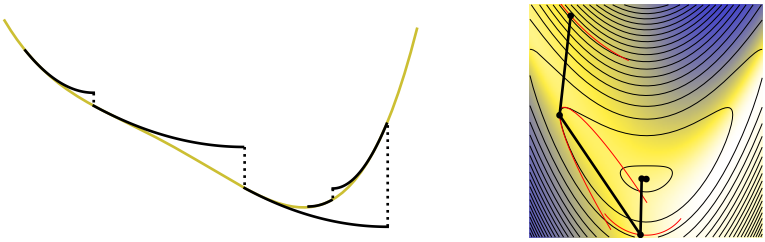
Eine andere Methode, das zufällige Herumspringen, also die Varianz der Gradienten zu reduzieren, ist *Variance-Reduction*. Dazu gibt es zwei Ansätze. Der erste ist, eine Tabelle von Gradienten für alle $h_i$ aufzubauen. Immer wenn ein neuer Gradient $\nabla h_i$ berechnet wird, überschreibt dieser den jeweiligen Eintrag in der Tabelle. Für den Optimierungsschritt wird dann der Mittelwert der Tabelle verwendet. Die Berechnung dieses Mittelwerts kann man übrigens umgehen, indem man bei jedem Update der Tabelle auch den gespeicherten Mittelwert updatet. Folgender Algorithmus nennt sich SAG:

$$\underset{\circ}{i} = 0 \qquad \underset{\rightarrow}{i} = (i+1) \bmod n$$
$$\boldsymbol{g}_i = \nabla h_i(\boldsymbol{x})$$
$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha \frac{1}{n} \sum_j \boldsymbol{g}_j$$

Das Ergebnis ist eine glattere Konvergenz, wie man in Abbildung 12(f).

Der zweite Ansatz ist, in regelmäßigen Abständen (z.B. nach $n$ Schritten) einen vollen Gradienten $\nabla f(\boldsymbol{x})$ zu berechnen, und in den folgenden Schritten einen Optimierungsschritt zu machen, der diesen vollen Gradienten ergänzt. Folgender Algorithmus nennt sich SVRG:

$$\underset{\circ}{i} = 0 \qquad \underset{\rightarrow}{i} = (i+1) \bmod n$$



(a) eindimensional (b) zweidimensional

Abbildung 13: Newton-Methode

$$\tilde{\boldsymbol{x}} = \begin{cases} \boldsymbol{x} & i = 0 \\ \underset{\leftarrow}{\tilde{\boldsymbol{x}}} & i > 0 \end{cases} \qquad \boldsymbol{g} = \nabla f(\tilde{x})$$
$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha(\nabla h_i(\boldsymbol{x}) - \nabla h_i(\tilde{\boldsymbol{x}}) + \boldsymbol{g})$$

Das Ergebnis ist in Abbildung 12(g) zu sehen.

# 8 Newton-Methode

Generell, zweite Ableitung, unconstrained.

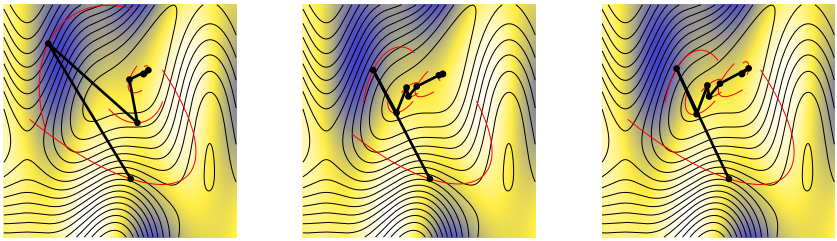
Die Newton-Methode gibt es in zwei Varianten: Einmal zur Nullstellensuche in eindimensionalen Funktionen $f : \mathbb{R} \longrightarrow \mathbb{R}$ (oder $\boldsymbol{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$). Dabei approximiert man die Funktion an der Stelle $x$ als Gerade mit Steigung $f'(x)$ und berechnet die Nullstelle $\underset{\rightarrow}{x}$ dieser Gerade, also

$$0 = y = f(x) + (\underset{\rightarrow}{x} - x) f'(x) \qquad \Rightarrow \quad \underset{\rightarrow}{x} = x - \frac{f(x)}{f'(x)},$$

und fährt an dieser Stelle mit neuen Iterationen fort.

Wir beschäftigen uns mit der Variante zur Funktionsoptimierung. Dabei wird die Funktion an der Stelle $\boldsymbol{x}$ als Paraboloid oder quadratische Funktion (siehe Abschnitt 7.6) approximiert, und davon die Minimumsstelle $\underset{\rightarrow}{\boldsymbol{x}}$ berechnet. Für die Parameter dieser Funktionen ergibt sich durch zweimaliges Ableiten nach $\boldsymbol{d}$ (dem Abstand von $\boldsymbol{x}$):

$$f(\boldsymbol{x} + \boldsymbol{d}) \approx \frac{1}{2} \boldsymbol{d}^\mathsf{T} A \boldsymbol{d} + \boldsymbol{b}^\mathsf{T} \boldsymbol{d} + c \qquad \Rightarrow \quad f(\boldsymbol{x}) = c$$



(a) Gauss-Newton (b) Levenberg-Marquard (c) Powells Dog-Leg

Abbildung 14: Least-Squares-Methoden

$$\nabla f(\boldsymbol{x}+\boldsymbol{d}) = A\boldsymbol{d}+\boldsymbol{b} \quad \Rightarrow \quad \nabla f(\boldsymbol{x}) = \boldsymbol{b}$$
$$\nabla^2 f(\boldsymbol{x}+\boldsymbol{d}) = A \quad \Rightarrow \quad \nabla^2 f(\boldsymbol{x}) = A$$

Siehe auch Abschnitt 26. Durch Nullsetzen der ersten Ableitung ergibt sich

$$0 = \nabla f(\boldsymbol{x}+\boldsymbol{d}) = \nabla^2 f(\boldsymbol{x})\boldsymbol{d} + \nabla f(\boldsymbol{x}) \quad \Rightarrow \quad \nabla^2 f(\boldsymbol{x})\boldsymbol{d} = -\nabla f(\boldsymbol{x}).$$

Das ist ein Gleichungssystem, das mit Hilfe von Gleichungssystemlösern (Cholesky-Faktorisierung) nach $\boldsymbol{d}$ gelöst wird.

Abbildung 13 zeigt Ausführungen der Newton-Methode. Man beachte das Überschießen an Stellen mit verringerter Krümmung, was in vielen Fällen Backtracking-Line-Search notwendig macht.

# 9 Gauss-Newton-Methode

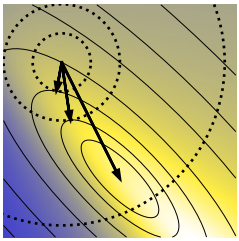
Nichtlineare Least-Squares, erste Ableitung, unconstrained.

Bei Least-Squares-Problemen ist die zu optimierende Funktion als Summe von quadrierten Funktionen gegeben.

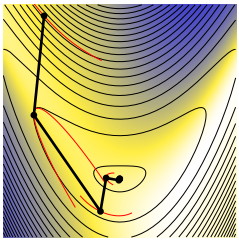
$$f(\boldsymbol{x}) = \sum_i g_i^2(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x})^\mathsf{T}\boldsymbol{g}(\boldsymbol{x})$$

Wie in Abschnitt 7.7 ist das häufigste Beispiel die Anpassung einer Prediction-Funktion:

$$g_i^2(\boldsymbol{x}) = \|\boldsymbol{p}(\boldsymbol{x}, \boldsymbol{y}_{\text{in},i}) - \boldsymbol{y}_{\text{out},i}\|^2$$



(a) Minima sind abhängig vom Radius der Trust-Region



(b) Newton-Methode mit Trust-Region

Abbildung 15: Trust-Region

$\boldsymbol{g}$ wird nun als lineare Funktion approximiert:

$$\boldsymbol{g}(\boldsymbol{x}+\boldsymbol{d}) = \nabla \boldsymbol{g}(\boldsymbol{x})\boldsymbol{d} + \boldsymbol{g}(\boldsymbol{x})$$

Hier ist $\nabla \boldsymbol{g}$ die *Jacobi-Matrix* (siehe Abschnitt 26). Dann ist:

$$f(\boldsymbol{x}+\boldsymbol{d}) = \boldsymbol{g}(\boldsymbol{x}+\boldsymbol{d})^{\mathsf{T}}\boldsymbol{g}(\boldsymbol{x}+\boldsymbol{d}) \approx \boldsymbol{d}^{\mathsf{T}}\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\nabla \boldsymbol{g}(\boldsymbol{x})\boldsymbol{d} + 2\boldsymbol{d}^{\mathsf{T}}\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\boldsymbol{g}(\boldsymbol{x})$$

Und mittels Ableiten und Nullsetzen erhalten wir:

$$\nabla_{\boldsymbol{d}} f(\boldsymbol{x}+\boldsymbol{d}) \approx 2\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\nabla \boldsymbol{g}(\boldsymbol{x})\boldsymbol{d} + 2\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\boldsymbol{g}(\boldsymbol{x}) = 0$$

$$\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\nabla \boldsymbol{g}(\boldsymbol{x})\boldsymbol{d} = -\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}}\boldsymbol{g}(\boldsymbol{x})$$

Ähnlich wie bei der Newton-Methode müssen wir also ein Gleichungssystem lösen. Abbildung 14(a) zeigt einen Optimierungslauf.

# 10 Trust-Region

Trust-Region-Methoden sind in gewisser Weise das Gegenstück zu Line-Search. Während bei Line-Search zuerst die Optimierungsrichtung feststeht, und danach eine gute Schrittweite gesucht wird, wird bei Trust-Region zuerst die (maximale) Schrittweite festgelegt (der Radius der Trust-Region), und danach die beste Optimierungsrichtung. Das führt zu einem Optimierungsproblem mit Constraints (Nebenbedingungen), welche wir uns im Detail erst später anschauen.

$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \arg\min_{\boldsymbol{d}} \hat{f}(\boldsymbol{x}+\boldsymbol{d}) \quad \text{mit} \quad |\boldsymbol{d}|^2 < r^2$$

Dass die Optimierungsrichtung dabei abhängig von der Größe der Trust-Region ist, sieht man in Abbildung 15(a).

## 10.1 Newton-Methode mit Trust-Region

Generell, zweite Ableitung, unconstrained.

Man verwendet dabei üblicherweise nicht die originale Funktion $f$, sondern eine lokale Approximation, die sich leichter minimieren lässt. Meist wird dabei eine quadratische Funktion, basierend auf der Hesse-Matrix (also eine Newton-Methode) verwendet.

$$\hat{f}(\boldsymbol{x}+\boldsymbol{d}) = \frac{1}{2}\boldsymbol{d}^{\mathsf{T}} A \boldsymbol{d} + \boldsymbol{b}^{\mathsf{T}}\boldsymbol{d}, \qquad \boldsymbol{b} = \nabla f(\boldsymbol{x}), \quad A = \nabla^2 f(\boldsymbol{x})$$

Es gibt dann einen *Lagrange-Multiplikator* $\lambda$, sodass man stattdessen die Constraint-lose Funktion

$$\hat{f}_\lambda(\boldsymbol{x}+\boldsymbol{d}) = \hat{f}(\boldsymbol{x}+\boldsymbol{d}) + \lambda(\boldsymbol{d}^{\mathsf{T}}\boldsymbol{d} - r^2)$$

minimieren kann. Durch Ableiten und Nullsetzen erhält man:

$$\nabla_{\boldsymbol{d}} \hat{f}_\lambda(\boldsymbol{x}+\boldsymbol{d}) = A\boldsymbol{d} + \boldsymbol{b} + 2\lambda\boldsymbol{d} = (A + 2\lambda I)\boldsymbol{d} + \boldsymbol{b} = 0\,,$$

wobei $I$ die Identitätsmatrix (Diagonale 1) ist. Das heißt, dass man das für die Newton-Methode zu lösende Gleichungssystem modifiziert, indem man konstante Werte $2\lambda$ zur Diagonale von $A$ addiert. Ist $\lambda = 0$ erhält man die Newton-Methode. Das kann man verwenden, wenn die Trust-Region das Minimum der quadratischen Approximation enthält. Je größer $\lambda$ wird, desto kleiner wird der Radius der Trust-Region, und desto mehr nähert sich $\boldsymbol{d}$ der Gradienten-Richtung. Abbildung 15(a) zeigt mehrere Lösungen für verschiedene $\lambda$.

Die nächste Frage ist nun, welchen Wert $\lambda$ haben sollte. Die Lösung ist üblicherweise, mit einem bestimmen kleinen $\lambda$ zu starten, und dann wiederholt zu erhöhen, falls der neue Funktionswert nicht kleiner ist als der aktuelle. Das ist eine Form von Backtracking. Wenn der Funktionswert verkleinert werden kann, wird $\lambda$ entweder wieder auf den ursprünglichen Wert zurückgesetzt, oder verkleinert. Das gibt folgenden Algorithmus:

Löse $(\nabla^2 f(\boldsymbol{x}) + \lambda I)\boldsymbol{d} = -\nabla f(\boldsymbol{x})$
if $f(\boldsymbol{x}+\boldsymbol{d}) < f(\boldsymbol{x})$
  $\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{d}$
  $\underset{\rightarrow}{\lambda} = \underset{\circ}{\lambda}$  oder  $\underset{\rightarrow}{\lambda} = \lambda/\rho$
sonst
  $\underset{\rightarrow}{\lambda} = \rho\lambda$  $\rho = 2$

Ein Optimierungslauf ist in Abbildung 15(b) zu sehen. Vergleiche dazu Abbildung 13(b), wo im zweiten Schritt der Funktionswert schlechter wird.

## 10.2 Levenberg-Marquard

| Nichtlineare Least-Squares, erste Ableitung, unconstrained. |
|---|

Die Levenberg-Marquard-Methode erweitert die Gauss-Newton-Methode um eine Trust-Region. Es ergibt sich also:

$$(\nabla \boldsymbol{g}(\boldsymbol{x})^\mathsf{T} \nabla \boldsymbol{g}(\boldsymbol{x}) + \lambda I)\boldsymbol{d} = -\nabla \boldsymbol{g}(\boldsymbol{x})^\mathsf{T} \boldsymbol{g}(\boldsymbol{x})$$

Und die Steuerung von $\lambda$ wie im Algorithmus in Abschnitt 10. Abbildung 14(b) zeigt einen Testlauf.

## 10.3 Powells Dog-Leg

| Nichtlineare Least-Squares, erste Ableitung, unconstrained. |
|---|

Eine andere Methode, die Gauss-Newton-Methode mit Trust-Region zu vereinen, ist Powells Dog-Leg. Dabei wird zuerst $\boldsymbol{d}_{\mathrm{GN}}$ wie in der Gauss-Newton-Methode berechnet. Außerdem der Gradient-Descent-Schritt $\boldsymbol{d}_{\mathrm{GD}} = -\nabla \boldsymbol{g}(\boldsymbol{x})^\mathsf{T} \boldsymbol{g}(\boldsymbol{x})$.

Danach wird in Richtung von $\boldsymbol{d}_{\mathrm{GD}}$ das Minimum gesucht:

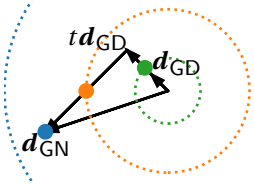
$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t} f(\boldsymbol{x} + t\boldsymbol{d}_{\mathrm{GD}}) &= \nabla f(\boldsymbol{x} + t\boldsymbol{d}_{\mathrm{GD}})^\mathsf{T} \boldsymbol{d}_{\mathrm{GD}} \\
&= 2t \underbrace{\boldsymbol{d}_{\mathrm{GD}}^\mathsf{T} \nabla \boldsymbol{g}(\boldsymbol{x})^\mathsf{T} \nabla \boldsymbol{g}(\boldsymbol{x}) \boldsymbol{d}_{\mathrm{GD}}}_{\|\nabla \boldsymbol{g}(x)\boldsymbol{d}_{\mathrm{GD}}\|^2} + 2 \underbrace{\boldsymbol{g}(\boldsymbol{x})^\mathsf{T} \nabla \boldsymbol{g}(\boldsymbol{x})}_{-\boldsymbol{d}_{\mathrm{GD}}^\mathsf{T}} \boldsymbol{d}_{\mathrm{GD}} = 0 \\
t &= \frac{\|\boldsymbol{d}_{\mathrm{GD}}\|^2}{\|\nabla \boldsymbol{g}(x)\boldsymbol{d}_{\mathrm{GD}}\|^2}
\end{aligned}$$

Wenn nun der Gauss-Newton-Schritt innerhalb der Trust-Region liegt, wird dieser durchgeführt. Wenn nicht, und der Gradient-Descent-Schritt liegt innerhalb, dann wird zwischen den zweien interpoliert ($\boldsymbol{d} = \beta \boldsymbol{d}_{\mathrm{GN}} + (1-\beta) t \boldsymbol{d}_{\mathrm{GD}}$), und zwar so, dass der Interpolationspunkt $\boldsymbol{d}$ genau die Distanz $r$ (Trust-Region-Radius) hat. Das ergibt eine quadratische Gleichung in $\beta$ ($\beta^2 \|\boldsymbol{d}_{\mathrm{GN}} - t\boldsymbol{d}_{\mathrm{GD}}\|^2 + \beta \cdot 2(\boldsymbol{d}_{\mathrm{GN}} - t\boldsymbol{d}_{\mathrm{GD}})^\mathsf{T} t\boldsymbol{d}_{\mathrm{GD}} + \|t\boldsymbol{d}_{\mathrm{GD}}\|^2 - r^2 = 0$). Andernfalls wird der Gradient-Descent-Schritt auf Länge $r$ gekürzt/verlängert.

Abbildung 16 zeigt den Algorithmus. Abbildung 14(c) zeigt einen Testlauf.

Berechne $\boldsymbol{d}_{\text{GN}}$ (wie in Abschnitt 9)
$\boldsymbol{d}_{\text{GD}} = -\nabla \boldsymbol{g}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{g}(\boldsymbol{x}), \quad t = \|\boldsymbol{d}_{\text{GD}}\|^2 / \|\nabla \boldsymbol{g}(x) \boldsymbol{d}_{\text{GD}}\|^2$
if $\|\boldsymbol{d}_{\text{GN}}\|^2 < r^2$
    $\boldsymbol{d} = \boldsymbol{d}_{\text{GN}}$
else if $\|t\boldsymbol{d}_{\text{GD}}\|^2 < r^2$
    Berechne $\beta$ (quadratische Gleichung)
    $\boldsymbol{d} = \beta \boldsymbol{d}_{\text{GN}} + (1-\beta) t \boldsymbol{d}_{\text{GD}}$  sodass  $\|d\|^2 = r^2$
else
    $\boldsymbol{d} = r \boldsymbol{d}_{\text{GD}} / \|\boldsymbol{d}_{\text{GD}}\|$
if $f(\boldsymbol{x} + \boldsymbol{d}) < f(\boldsymbol{x})$
    $\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{d}$
    $\underset{\rightarrow}{r} = \rho_1 r \qquad \rho_1 = 2$
else
    $\underset{\rightarrow}{r} = \rho_2 r \qquad \rho_2 = 0.8$

(a) Algorithmus




(b) Schema

Abbildung 16: Powells Dog-Leg

# 11 Quasi-Newton-Methoden

Bei Quasi-Newton-Methoden wird die Hesse-Matrix $\nabla^2 f(\boldsymbol{x})$ approximiert, ohne die zweite Ableitung bilden zu müssen. Meist werden dazu aufeinander folgende erste Ableitungen benutzt.

## 11.1 BFGS (Broyden-Fletcher-Goldfarb-Shanno)

Bei dieser Methode wird die inverse Hesse-Matrix approximiert, die wir hier $G$ nennen. Aus dem sonst benutzten zu lösenden Gleichungssystem wird dann:

$$\nabla^2 f(\boldsymbol{x}) \boldsymbol{d} = -\nabla f(\boldsymbol{x}) \quad \Rightarrow \quad \boldsymbol{d} = -\underbrace{(\nabla^2 f(\boldsymbol{x}))^{-1}}_{=:G} \nabla f(\boldsymbol{x})$$

Wir betrachten nun die Approximation der Funktion $f$ an der Stelle $\underset{\rightarrow}{\boldsymbol{x}}$ durch die übliche quadratische Funktion $\hat{f}$. Der Gradient von $\hat{f}$ ist bekannterweise:

$$\nabla \hat{f}(\underset{\rightarrow}{\boldsymbol{x}} + \boldsymbol{q}) = \underset{\rightarrow}{G}^{-1} \boldsymbol{q} + \nabla f(\underset{\rightarrow}{\boldsymbol{x}})$$

Wir wollen nun $G$ so wählen, dass der Gradient an beiden Stellen $\underset{\rightarrow}{\boldsymbol{x}}$ und $\boldsymbol{x}$ dem echten Gradienten entspricht:

$$\nabla \hat{f}(\underset{\rightarrow}{\boldsymbol{x}}) = \nabla f(\underset{\rightarrow}{\boldsymbol{x}}) \qquad \nabla \hat{f}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$$

Ersteres ist (mit $\boldsymbol{q} = 0$) offenbar erfüllt. Für die zweite Bedingung ergibt sich mit $\boldsymbol{x} = \underset{\rightarrow}{\boldsymbol{x}} - \boldsymbol{d}$:

$$\nabla f(\boldsymbol{x}) = -\underset{\rightarrow}{G}^{-1}\boldsymbol{d} + \nabla f(\underset{\rightarrow}{\boldsymbol{x}}) \qquad \underset{\rightarrow}{G}^{-1}\boldsymbol{d} = \nabla f(\underset{\rightarrow}{\boldsymbol{x}}) - \nabla f(\boldsymbol{x}) =: \boldsymbol{e} \qquad \underset{\rightarrow}{G}\boldsymbol{e} = \boldsymbol{d}$$

Wir suchen also ein $\underset{\rightarrow}{G}$, das dem alten $G$ ähnlich ist, aber diese Bedingung erfüllt.

$$\underset{\rightarrow}{G} = \arg\min_{\underline{G}} \|\underline{G} - G\| \quad \text{mit} \quad \underline{G} = \underline{G}^{\mathsf{T}}, \underline{G}\boldsymbol{e} = \boldsymbol{d}$$

Durch geeignete Wahl der Matrix-Norm $\|\cdot\|$ kann man zeigen, dass folgender Update diese Bedingungen erfüllt.

$$\underset{\rightarrow}{G} = \left(I - \frac{\boldsymbol{e}\boldsymbol{d}^{\mathsf{T}}}{\boldsymbol{d}^{\mathsf{T}}\boldsymbol{e}}\right)^{\mathsf{T}} G \left(I - \frac{\boldsymbol{e}\boldsymbol{d}^{\mathsf{T}}}{\boldsymbol{d}^{\mathsf{T}}\boldsymbol{e}}\right) + \frac{\boldsymbol{d}\boldsymbol{d}^{\mathsf{T}}}{\boldsymbol{d}^{\mathsf{T}}\boldsymbol{e}}$$

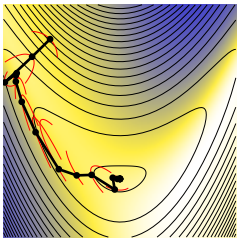
Durch Multiplizieren mit $\boldsymbol{e}$ von rechts sieht man schnell, dass $\underset{\rightarrow}{G}\boldsymbol{e} = \boldsymbol{d}$ ist. $G$ soll aber auch positiv definit sein. Dafür muss man garantieren, dass:

$$\boldsymbol{d}^{\mathsf{T}}\boldsymbol{e} > 0 \quad \Leftrightarrow \quad \boldsymbol{d}^{\mathsf{T}}\nabla f(\underset{\rightarrow}{\boldsymbol{x}}) > \boldsymbol{d}^{\mathsf{T}}\nabla f(\boldsymbol{x}) \left(= -\nabla f(\boldsymbol{x})^{\mathsf{T}} H \nabla f(\boldsymbol{x}) < 0\right)$$

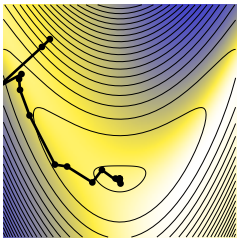
Das erreichen wir durch Line-Search mit den Wolfe-Conditions (beginnend immer mit $\alpha = 1$). Der Algorithmus sieht dann so aus:

$\underset{\circ}{G} = \alpha I \qquad \Rightarrow \underset{\circ}{\boldsymbol{d}} = -\alpha \nabla f(\underset{\circ}{\boldsymbol{x}})$
$\boldsymbol{d} =$ Line-Search in Richtung $-G\nabla f(\boldsymbol{x})$
$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{d}$
$\boldsymbol{e} = \nabla f(\underset{\rightarrow}{\boldsymbol{x}}) - \nabla f(\boldsymbol{x})$
$\underset{\rightarrow}{G} =$ BFGS-Update$(G, \boldsymbol{d}, \boldsymbol{e})$

Abbildung 17(a) zeigt einen Testlauf.



(a) BFGS



(b) L-BFGS ($m = 5$)

Abbildung 17: Quasi-Newton-Methoden

## 11.2 L-BFGS (Limited-Memory-BFGS)

Für hoch-dimensionale Probleme wird das Berechnen der approximierten Hesse-Matrix ein Problem in Bezug auf rechnerische Komplexität und Speicherbedarf. Daher wird hier versucht, das Produkt $G\nabla f(\boldsymbol{x})$ durch Produkte von Vektoren zu implementieren. Da dieses Produkt wiederum Produkte mit vorangegangenen $\underset{\leftarrow i}{G}$ beinhaltet, ergibt sich ein rekursives Schema, das nach $m$ Schritten mit einer skalierten Identitätsmatrix statt $\underset{\leftarrow m}{G}$ abgebrochen wird. Für die rekursiven Iterationen benötigt man einen Puffer der letzten $m$ Vektoren $\boldsymbol{d}$ und $\boldsymbol{e}$. Es ergibt sich der Algorithmus in Abbildung 18. Abbildung 17(b) zeigt einen Testlauf.

# 12 Regularization

Unter Regularization versteht man Methoden, eine zu optimierende Funktion so zu modifizieren, dass die Optimierungsresultate robuster gegen Ungenauigkeiten oder Variabilitäten der Funktionsparameter sind.

## 12.1 Ridge/Tikhonov

Die *Ridge-Regression* oder *Tikhonov-Regularization* ist am besten für lineare Least-Squares-Probleme zu erklären:

$$f(\boldsymbol{x}) = \|A\boldsymbol{x} - \boldsymbol{y}\|^2$$

Durch Ableiten und Nullsetzen bekommt man die sogenannte Normalgleichung:

$$\nabla f(\boldsymbol{x}) = 2A^{\mathsf{T}}(A\boldsymbol{x} - \boldsymbol{y}) = \boldsymbol{0} \qquad \Rightarrow \qquad A^{\mathsf{T}}A\boldsymbol{x} = A^{\mathsf{T}}\boldsymbol{y}$$

LBFGS($\boldsymbol{p}, i$) :=
  if $i = m \vee \underset{\leftarrow}{i} = \circ$
    return $\frac{\underset{\leftarrow}{\boldsymbol{d}}^\top \underset{\leftarrow}{\boldsymbol{e}}}{\underset{\leftarrow}{\boldsymbol{e}}^\top \underset{\leftarrow}{\boldsymbol{e}}} \boldsymbol{p}$
  $a = \frac{\underset{\leftarrow i}{\boldsymbol{d}}^\top \boldsymbol{p}}{\underset{\leftarrow i}{\boldsymbol{d}}^\top \underset{\leftarrow i}{\boldsymbol{e}}}$
  $\boldsymbol{q} = \text{LBFGS}(\boldsymbol{p} - a \underset{\leftarrow i}{\boldsymbol{e}}, i+1)$
  return $\boldsymbol{q} + \left( a - \frac{\underset{\leftarrow i}{\boldsymbol{e}}^\top \boldsymbol{q}}{\underset{\leftarrow i}{\boldsymbol{d}}^\top \underset{\leftarrow i}{\boldsymbol{e}}} \right) \underset{\leftarrow i}{\boldsymbol{d}}$

$\underset{\circ}{\boldsymbol{d}} = -\alpha \nabla f(\underset{\circ}{\boldsymbol{x}})$
$\underset{\circ}{\boldsymbol{d}} = \text{Line-Search in Richtung -LBFGS}(\nabla f(\boldsymbol{x}), 1)$
$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{d}$
$\boldsymbol{e} = \nabla f(\underset{\rightarrow}{\boldsymbol{x}}) - \nabla f(\boldsymbol{x})$

Abbildung 18: L-BFGS-Algorithmus

Wie stabil die Lösungen dieser Gleichung sind, hängt von der Matrix $A^\top A$ ab. Wenn diese knapp singulär (nicht-invertierbar) ist, dann reagiert die Lösung sehr stark auf leicht veränderte Koeffizienten:

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1.01 \end{pmatrix} \quad \Rightarrow$$

$$A^\top A \begin{pmatrix} 101 \\ -100 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad A^\top A \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1.005 \end{pmatrix}, \quad A^\top A \begin{pmatrix} -100 \\ 100 \end{pmatrix} = \begin{pmatrix} 1 \\ 1.01 \end{pmatrix}$$

Dieses Verhalten wird auch durch die *Konditionszahl* von $A^\top A$ beschrieben (größter durch kleinster Singulärwert). Sie ist für diese Beispielmatrix > 16000 (die Identitätsmatrix hat 1). So ein Problem nennt man *ill-posed*.

Um das unter Kontrolle zu bekommen, modellieren wir $A$ als Erwartungswert einer randomisierten Matrix, indem wir eine Matrix von unabhängigen Zufallsvariablen addieren, mit Erwartungswert E = 0, Varianz V = $\sigma^2$ und Kovarianz Cov = 0 für verschiedene Zufallsvariablen:

$$\tilde{A} = A + N \quad \text{mit} \quad \text{E}(N_{i,j}) = 0, \quad \text{V}(N_{i,j}) = \text{E}(N_{i,j}^2) = \sigma^2, \quad \text{Cov}(N_{i,j}, N_{k,l \neq i,j}) = 0$$

Dann ergibt sich:

$$\mathrm{E}(\tilde{A}) = A, \quad \mathrm{E}(\tilde{A}^\mathsf{T}\tilde{A})_{i,j} = \mathrm{E}\left(\sum_k (A+N)_{i,k}(A+N)_{k,j}\right) = (A^\mathsf{T}A)_{i,j} + \begin{cases}\sigma^2 & i=j\\ 0 & i\neq j\end{cases}$$

Das heißt, es wird ein konstanter Wert auf die Diagonale addiert. Dementsprechend ist die Konditionszahl von z.B. $A^\mathsf{T}A + 0.1I$ gleich 41.2. Die Lösungen der Gleichung sind viel stabiler:

$$(A^\mathsf{T}A + 0.1I)\begin{pmatrix}0.25\\0.23\end{pmatrix} = \begin{pmatrix}0.987\\0.990\end{pmatrix}, \quad (A^\mathsf{T}A + 0.1I)\begin{pmatrix}0.25\\0.24\end{pmatrix} = \begin{pmatrix}1.01\\1.01\end{pmatrix}$$

Diese Lösungsgleichung löst auch das modifizierte Optimierungsproblem mit dem Tikhonov-Regularization-Term $\|x\|^2$.

$$\min f(\boldsymbol{x}) + \lambda\|\boldsymbol{x}\|^2$$

Denn durch Ableiten und Nullsetzen erhalten wird die gleiche modifizierte Normalgleichung:

$$\nabla(f(\boldsymbol{x}) + \lambda\|\boldsymbol{x}\|^2) = 2A^\mathsf{T}A\boldsymbol{x} - 2A^\mathsf{T}\boldsymbol{y} + 2\lambda\boldsymbol{x} = 2(A^\mathsf{T}A + \lambda I)\boldsymbol{x} - 2A^\mathsf{T}\boldsymbol{y} = \boldsymbol{0}$$

Für nichtlineare Optimierungen kann die gleiche Art Regularisierung verwendet werden. Man beachte, dass die Newton-Methode mit Trust-Region die gleiche Form hat.

## 12.2 LASSO (least absolute shrinkage and selection operator)

Ähnlich zur Ridge-Regularization, wo die $\ell_2$-Norm verwendet wird, wird bei LASSO die $\ell_1$-Norm verwendet.

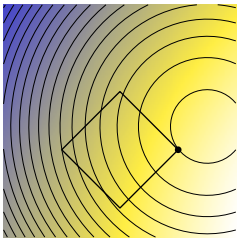
$$\min f(\boldsymbol{x}) + \lambda\|\boldsymbol{x}\|_1, \qquad \|\boldsymbol{x}\|_1 = \sum_i |x_i|$$

Das Wichtigste bei dieser Regularisierung ist, dass die Lösung dazu tendiert, *sparse* zu sein, d.h. dass viele Koeffizienten des Lösungsvektors 0 sind. Der Grund dafür ist, dass der Schnitt zwischen Höhenlinien von $f(\boldsymbol{x})$ und $\lambda\|\boldsymbol{x}\|$ aufgrund letzterer geometrischen Form häufig auf Koordinatenachsen liegt. Siehe Abbildung 19.

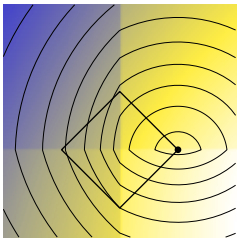
Die starke (und übliche) Version von LASSO ist die Optimierung mit Constraint:

$$\min f(\boldsymbol{x}) \;\text{ mit }\; \|\boldsymbol{x}\|_1 \leq R$$

Durch Anwendung des Langrage-Multiplikators $\lambda$ kommt man dann auf obige Regularisierung. Bei den Optimierungsschritten muss man aber darauf achten, dass $\vec{\boldsymbol{x}}$ tatsächlich die Bedingung $\|\boldsymbol{x}\|_1 \leq R$ erfüllt. Siehe dazu Abschnitt 13.



(a) Schnitt zwischen Höhenlinie von $f(\boldsymbol{x})$ und $\|\boldsymbol{x}\| = 1$



(b) Minimum von $f(\boldsymbol{x}) + \|\boldsymbol{x}\|_1$

Abbildung 19: LASSO

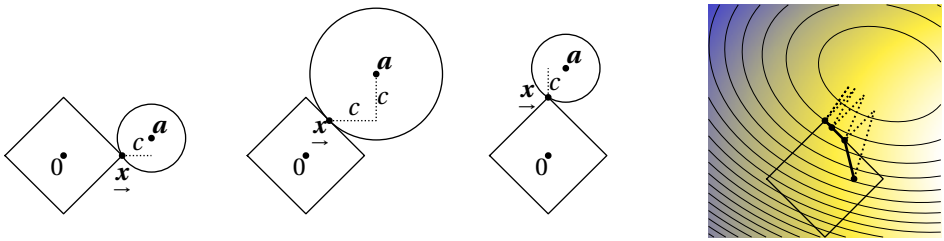
# 13 Projected Gradient-Descent

Die Projected Gradient-Descent-Methode ist eine Variante des Gradient-Descent, bei der garantiert wird, dass die $\ell_1$-Bedingung $\|\boldsymbol{x}\|_1 \leq R$ erfüllt ist. Es wird also jener Punkt $\underset{\rightarrow}{\boldsymbol{x}}$ gesucht, der dem normalen Gradient-Descent-Schritt $\boldsymbol{x} + \alpha \boldsymbol{d}$ am nächsten liegt, aber die Bedingung erfüllt. Sie löst also das LASSO-Problem:

$$\underset{\rightarrow}{\boldsymbol{x}} = \arg \min_{\|\boldsymbol{u}\|_1 \leq R} \|\boldsymbol{u} - (\boldsymbol{x} + \alpha \boldsymbol{d})\|^2$$

Wenn $\boldsymbol{x} + \alpha \boldsymbol{d} \leq R$ ist, dann wird natürlich einfach $\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{u} = \boldsymbol{x} + \alpha \boldsymbol{d}$ gewählt. Falls es aber außerhalb liegt, muss ein Punkt am Rand $\|\underset{\rightarrow}{\boldsymbol{x}}\|_1 = R$ gewählt werden. Man sieht nun, dass manche Koordinaten von $\boldsymbol{x}$ gleich 0 werden, während die anderen Koordinaten alle den gleichen Abstand $c$ von $x_i + \alpha d_i$ haben, weil der Vektor-Abstand normal auf den Rand steht und $R = \sum |x_i| = \sum q_i x_i$ mit $|q_i| = 1$, und $\boldsymbol{q}$ ist ein Normalvektor. Außerdem sind die Koordinaten, die nicht 0 werden, jene, die betragsmäßig größer sind. Wenn $b_i$ jene absoluten Koordinaten von $\boldsymbol{x} + \alpha \boldsymbol{d}$ sind, die nicht 0 werden, dann ist $\sum |x_i| = R$ und $|x_i| = b_i - c$. Daraus ergibt sich $c = \frac{1}{k}((\sum b_i) - R)$. Weil noch $c < b_k$ sein muss, sortiert der Algorithmus die $b_i$ und findet das größte $k$ sodass dies noch erfüllt ist:

$$\begin{aligned}
&\boldsymbol{a} = \boldsymbol{x} + \alpha \boldsymbol{d} \\
&\boldsymbol{b} = \text{sort}_{\text{absteigend}} |\boldsymbol{a}| \\
&k^* = \max \left\{ k \;\middle|\; \tfrac{1}{k} \left( \left( \textstyle\sum_{i=1}^{k} b_i \right) - R \right) < b_k \right\} \\
&c = \tfrac{1}{k^*} \left( \left( \textstyle\sum_{i=1}^{k^*} b_i \right) - R \right) \\
&\underset{\rightarrow}{x_i} = \begin{cases} a_i - \text{sign}(a_i) c & |a_i| > c \\ 0 & |a_i| \leq c \end{cases}
\end{aligned}$$




(a) $\boldsymbol{a} = \boldsymbol{b} = (1.5, 0.3)$, $c = \frac{1}{1}(1.5-1) = 0.5$, $\underset{\rightarrow}{\boldsymbol{x}} = (a_1 - c, 0) = (1, 0)$

(b) $\boldsymbol{a} = (1.2, 1.4)$, $\boldsymbol{b} = (1.4, 1.2)$, $c = \frac{1}{2}(1.2+1.4-1) = 0.8$, $\underset{\rightarrow}{\boldsymbol{x}} = (a_1 - c, a_2 - c) = (0.4, 0.6)$

(c) $\boldsymbol{a} = (0.3, 1.5)$, $\boldsymbol{b} = (1.5, 0.3)$, $c = \frac{1}{1}(1.5-1) = 0.5$, $\underset{\rightarrow}{\boldsymbol{x}} = (0, a_2 - c) = (0, 1)$

(d) Testlauf mit GD-Schritten plus Projektion (punktiert)

Abbildung 20: Projected Gradient-Descent

Abbildung 20(a-c) zeigt ein 2D-Beispiel. Ein Beispiel höherer Dimension für $\boldsymbol{a} = (0.6, -1.5, 1.8, -1.1)$ und $R = 1$ wäre:

| $i$ | $b_i$ | $c$ | $\underset{\rightarrow}{x}_i$ |
|---|---|---|---|
| 1 | 1.8 | $(1.8-1)/1 = 1$ | $0 \quad (0.6 < 1.15)$ |
| 2 | 1.5 | $\mathbf{(1.8+1.5-1)/2 = 1.15}$ | $-1.5 + 1.15 = -0.35$ |
| 3 | 1.1 | $(1.8+1.5+1.1-1)/3 = 1.133 \mathbf{> 1.1}$ | $1.8 - 1.15 = 0.65$ |
| 4 | 0.6 | | $0 \quad (\lvert -1.1 \rvert < 1.15)$ |

Abbildung 20(d) zeigt einen Testlauf mit Gradient-Descent-Schritt ($\boldsymbol{a}$) und Projektion ($\underset{\rightarrow}{\boldsymbol{x}}$) in jedem Schritt.

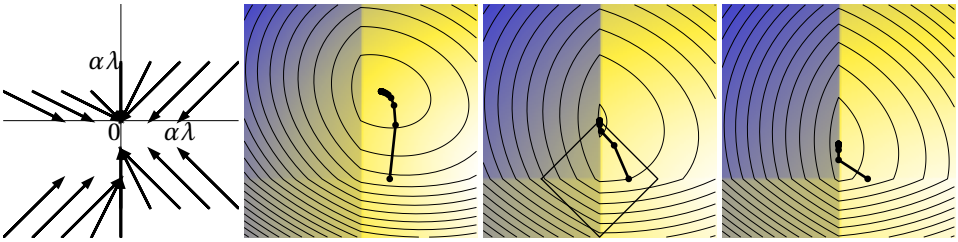
# 14 Proximal Gradient-Descent

Das schwächere $\ell_1$-Problem

$$\min f(\boldsymbol{x}) + \lambda \|\boldsymbol{x}\|_1$$

wird von der *Proximal Gradient-Descent*-Methode gelöst. Dabei wird der Gradient-Descent-Schritt mit einem *proximal operator* $\text{prox}_\varphi$ (für eine Regularization-Funktion $\varphi$, meist $\varphi(\boldsymbol{u}) = \lambda \|\boldsymbol{u}\|_1$) korrigiert.

$$\underset{\rightarrow}{\boldsymbol{x}} = \text{prox}_{\alpha\lambda\|\cdot\|_1}(\underline{\boldsymbol{x}}) = \arg\min_{\boldsymbol{u}} \frac{1}{2}\|\boldsymbol{u} - \underline{\boldsymbol{x}}\|^2 + \alpha\lambda\|\boldsymbol{u}\|_1 \qquad \underline{\boldsymbol{x}} := \boldsymbol{x} + \alpha\boldsymbol{d}$$




(a) Soft-Thresholding (b) $\lambda = 1$ (c) $\lambda = 2.4$ (d) $\lambda = 3.4$

Abbildung 21: Proximal Gradient-Descent

Durch Ableiten und Nullsetzen (nach $\boldsymbol{u}$) bekommen wir folgende Gleichung:

$$\boldsymbol{u} - \underset{\rightarrow}{\boldsymbol{x}} + \alpha\lambda\nabla\|\boldsymbol{u}\|_1 = 0, \qquad (\nabla\|\boldsymbol{u}\|_1)_i = \begin{cases} 1 & u_i > 0 \\ [-1,1] & u_i = 0 \\ -1 & u_i < 0 \end{cases}$$

wobei $\|\boldsymbol{u}\|_1$ nicht überall differenzierbar ist. Es gibt an den Stellen mit $u_i = 0$ eine Subgradienten (siehe Abbildung 6), wodurch ein gewisser Freiheitsgrad entsteht. Dieser ermöglicht die Lösung:

$$\underset{\rightarrow}{x_i} = u_i = \underset{\rightarrow}{x_i} + \begin{cases} -\alpha\lambda & \underset{\rightarrow}{x_i} > \alpha\lambda \\ -\underset{\rightarrow}{x_i} & -\alpha\lambda \leq \underset{\rightarrow}{x_i} \leq \alpha\lambda \\ +\alpha\lambda & \underset{\rightarrow}{x_i} < -\alpha\lambda \end{cases} = \begin{cases} \underset{\rightarrow}{x_i} - \alpha\lambda & \underset{\rightarrow}{x_i} > \alpha\lambda \\ 0 & -\alpha\lambda \leq \underset{\rightarrow}{x_i} \leq \alpha\lambda \\ \underset{\rightarrow}{x_i} + \alpha\lambda & \underset{\rightarrow}{x_i} < -\alpha\lambda \end{cases}$$

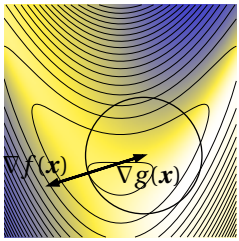
Eine andere Wahl würde nicht funktionieren, weil wenn z.B. $0 < \underset{\rightarrow}{x_i} < \alpha\lambda$ und $u_i > 0$ wäre, dann müsste $u_i = \underset{\rightarrow}{x_i} - \alpha\lambda < 0$ sein (Widerspruch), und umgekehrt. Diesen Update-Schritt nennt man auch *soft threshold* (Abbildung 21(a)).

Der Update-Schritt ist sinnvoll, denn wir können zeigen, dass ein Optimum $\boldsymbol{x}^*$ durch den Update-Schritt nicht verändert wird (Fixpunkt), und umgekehrt ein Fixpunkt auch ein Optimum ist. Für ein Optimum $\boldsymbol{x}^*$ gilt nämlich, dass die Ableitung von $f(\boldsymbol{x}) + \lambda\|\boldsymbol{x}\|_1$ gleich 0 ist:

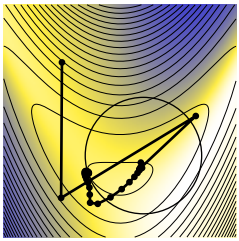
$$\nabla f(\boldsymbol{x}^*) + \lambda\nabla\|\boldsymbol{x}^*\|_1 = \boldsymbol{0} \quad \Rightarrow \quad \frac{\mathrm{d}f(\boldsymbol{x}^*)}{\mathrm{d}x_i} = -\lambda\frac{\mathrm{d}\|\boldsymbol{x}^*\|_1}{\mathrm{d}x_i^*} = \begin{cases} -\lambda & x_i^* > 0 \\ \lambda[-1,1] & x_i^* = 0 \\ +\lambda & x_i^* < 0 \end{cases}$$

Für den Fall $x_i^* > 0$ bleibt $\underset{\rightarrow}{x_i} = x_i^*$, weil:

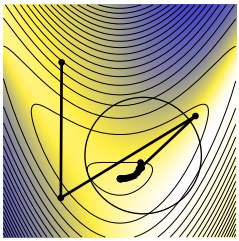
$$x_i^* > 0 \quad \Leftrightarrow \quad \underset{\rightarrow}{x_i} = x_i^* - \alpha\frac{\mathrm{d}f(\boldsymbol{x}^*)}{\mathrm{d}x_i} = x_i^* + \alpha\lambda > \alpha\lambda \quad \Leftrightarrow \quad \underset{\rightarrow}{x_i} = \underset{\rightarrow}{x_i} - \alpha\lambda = x_i^*$$




(a) $\nabla f(\boldsymbol{x}) = \lambda \nabla g(\boldsymbol{x})$



(b) Equality constraint



(c) Inequality constraint

Abbildung 22: Gradient-Descent mit Constraints

Für $x_i^* < 0$ gilt das analog. Und für $x_i^* = 0$ gilt ebenfalls:

$$x_i^* = 0 \quad \Leftrightarrow \quad \underset{\rightarrow}{x}_i \in x_i^* + \alpha\lambda[-1,1] \quad \Leftrightarrow \quad -\alpha\lambda \leq \underset{\rightarrow}{x}_i \leq \alpha\lambda \quad \Leftrightarrow \quad \underset{\rightarrow}{x}_i = 0 = x_i^*$$

Abbildung 21(b-d) zeigt drei Testläufe für die gleiche Funktion $f$ wie in Abbildung 20, nur eben mit $\lambda\|\boldsymbol{x}\|_1$ addiert. (c) findet die gleiche Lösung wie die Projected-Gradient-Descent-Methode, was zeigt, dass es immer ein bestimmtes $\lambda$ gibt, sodass das schwache LASSO-Problem die Lösung des starken findet. In diesem Fall ist übrigens $\lambda = -\frac{\mathrm{d}f((0,1))}{\mathrm{d}x_1}$.

# 15 Constraints

Constraints (Nebenbedingungen) schränken den Wertebereich der erlaubten Lösungsvektoren ein. Es gibt Gleichheits- und Ungleichheits-Constraints. Für ein einzelnes Gleichheits-Constraint müssen wir folgendes Problem lösen:

$$\min f(\boldsymbol{x}) \quad \text{mit} \quad g(\boldsymbol{x}) = 0$$

wobei ein Constraint einer anderen Form (z.B. $\|\boldsymbol{x}\|^2 = 5$) immer auf diese Form gebracht werden kann (z.B. $\|x\|^2 - 5 = 0$). Es kann dann gezeigt werden, dass die Gradienten von $f$ und $g$ an einer Optimumsstelle $\boldsymbol{x}$ parallel sind, d.h. dass es ein $\lambda \in \mathbb{R}$ (*Lagrange-Multiplikator*) gibt sodass:

$$\nabla_{\boldsymbol{x}}(f(\boldsymbol{x}) - \lambda g(\boldsymbol{x})) = 0, \quad g(\boldsymbol{x}) = 0$$

Siehe Abbildung 22(a). Die Bedingung $g(\boldsymbol{x}) = 0$ muss nach wie vor berücksichtigt werden, weil man diese zusätzliche Gleichung benötigt, um die zusätzliche Variable $\lambda$ zu fixieren.

Wenn es mehr als ein Constraint gibt, also $g_i(\boldsymbol{x}) = 0$, dann kann man die $g_i$ als Vektor $\boldsymbol{g}$ schreiben:

$$\min f(\boldsymbol{x}) \text{ mit } \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0}$$

Für jedes Constraint wird ein eigener Lagrange-Multiplikator $\lambda_i$ eingeführt. Wenn man diese $\lambda_i$ ebenfalls als Vektor $\boldsymbol{\lambda}$ schreibt, dann gilt für ein Optimum $\boldsymbol{x}$:

$$\nabla_{\boldsymbol{x}}(f(\boldsymbol{x}) - \boldsymbol{\lambda}^\top \boldsymbol{g}(\boldsymbol{x})) = 0, \quad \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0}$$

Als Beispiel verwenden wir das Constraint $\|\boldsymbol{x} - (0.4, 0.4)\|^2 = 1$, und damit $g(\boldsymbol{x}) = 1 - \|\boldsymbol{x} - (0.4, 0.4)\|^2$. Um per Gradient-Descent nicht nur $f(\boldsymbol{x})$ minimal werden zu lassen, sondern auch $\lambda$ so zu updaten, dass $g(\boldsymbol{x}) = 0$ wird, wollen wir zusätzlich $\frac{1}{2} g(\boldsymbol{x})^2$ minimieren, um den Update-Schritt $-\alpha g(\boldsymbol{x}) \nabla g(\boldsymbol{x})$ in den Update-Schritt $-\alpha(\nabla f(\boldsymbol{x}) - \lambda \nabla g(\boldsymbol{x}))$ integrieren, was mit einem Update von $\lambda$ durch $-g(\boldsymbol{x})$ gemacht wird. Das sich dadurch ergebende Schema ist:

$$\lambda = \underset{\leftarrow}{\lambda} - g(\boldsymbol{x})$$
$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha(\nabla f(\boldsymbol{x}) - \lambda \nabla g(\boldsymbol{x}))$$

Ein Testlauf ($\alpha = 0.1$) ist in Abbildung 22(b) zu sehen. Das Schema ist aber sehr instabil und tendiert zu Schwingungsverhalten um $g(\boldsymbol{x}) = 0$. Bessere Verfahren kommen später.

Wenn zusätzlich Ungleichheitsconstraints nötig sind,

$$\min f(\boldsymbol{x}) \text{ mit } \boldsymbol{g}(\boldsymbol{x}) = 0, \boldsymbol{h}(\boldsymbol{x}) \geq \boldsymbol{0}$$

dann wird das Schema nach Karush-Kuhn-Tucker (KKT) angewendet. Es gibt für die Ungleichheitsconstraints zusätzliche Lagrange-Multiplikatoren $\mu_i$. An einer Optimumsstelle $\boldsymbol{x}$ gilt:

$$\nabla_{\boldsymbol{x}}(f(\boldsymbol{x}) - \boldsymbol{\lambda}^\top \boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\mu}^\top \boldsymbol{h}(\boldsymbol{x})) = 0, \quad \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0}, \boldsymbol{h}(\boldsymbol{x}) \geq \boldsymbol{0}, \quad \boldsymbol{\mu} \geq \boldsymbol{0}, \quad \boldsymbol{\mu}^\top \boldsymbol{h}(\boldsymbol{x}) = 0$$

Auch hier braucht man die ursprünglichen Constraints, um die zusätzlichen Variablen zu fixieren. Außerdem müssen noch die $\mu_i \geq 0$ sein (vierte Bedingung). Und die fünfte Bedingung bedeutet praktisch nur, dass entweder $\mu_i = 0$ ist oder $h_i(\boldsymbol{x}) = 0$ oder beide. Aus $h_i(\boldsymbol{x}) \geq 0$ und $\mu_i \geq 0$ folgt nämlich:

$$\boldsymbol{\mu}^\top \boldsymbol{h}(\boldsymbol{x}) = 0 \quad \Leftrightarrow \quad \forall i : \mu_i h_i(\boldsymbol{x}) = 0 \quad \Leftrightarrow \quad \forall i : \mu_i = 0 \vee h_i(\boldsymbol{x}) = 0$$

Die KKT-Bedingungen gelten genau genommen aber nur, wenn für die Optimumsstelle die LICQ-Bedingung gilt (linear independence constraint qualification). Dafür müssen die $g_i$-Gradienten sowie die *aktiven* $h_i$-Gradienten (wo $h_i(\boldsymbol{x}) = 0$, also Minimum am Rand des erlaubten Bereichs)

$$\{\nabla g_i(\boldsymbol{x})\} \cup \{\nabla h_i(\boldsymbol{x}) \mid h_i(\boldsymbol{x}) = 0\} \qquad \text{linear unabhängig sein.}$$

Lineare Unabhängigkeit bedeutet, dass eine Linearkombination von Vektoren $\boldsymbol{v}_i$: $\sum_i a_i \boldsymbol{v}_i$ nur 0 werden kann, wenn alle $a_i = 0$ sind. Oder anders gesagt: Kein Vektor liegt in der Hyperebene, die durch die jeweils anderen Vektoren aufgespannt wird.

Abbildung 22(c) zeigt einen Testlauf für ein ähnliches Gradient-Descent-Schema wie bei den Equality-Constraints, hier für $\|\boldsymbol{x} - (0.4, 0.4)\|^2 \leq 1$, also $h(\boldsymbol{x}) = 1 - \|\boldsymbol{x} - (0.4, 0.4)\|^2$, wobei wir $\mu \geq 0$ im Update-Schritt garantieren:

$$\mu = \max(\underset{\leftarrow}{\mu} - g(\boldsymbol{x}), 0)$$
$$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} - \alpha(\nabla f(\boldsymbol{x}) - \mu \nabla g(\boldsymbol{x}))$$

# 16 Simplex-Methode

Das Optimieren einer linearen Funktion in $\boldsymbol{x}$ macht nur mit Constraints Sinn, da sonst ein Funktionswert $-\infty$ erreicht werden kann. Die *Standard-Form* für lineare Optimierung ist:

$$\min \boldsymbol{c}^\mathsf{T} \boldsymbol{x} \text{ mit } A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}$$

Hier hat $A$ $m$ Zeilen (Anzahl der Constraints) und $n$ Spalten (Anzahl der Variablen), wobei üblicherweise $n > m$ ist.

Um Constraints in Nicht-Standard-Form auf Standard-Form zu bringen, verwendet man zwei Methoden. Erstens werden generelle lineare Ungleichungen über sogenannte Slack-Variablen $\boldsymbol{y}$ in absolute Ungleichungen (Vergleich mit 0) plus Gleichungen verwandelt:

$$① \; A\boldsymbol{x} \leq \boldsymbol{b} \Leftrightarrow A\boldsymbol{x} + \boldsymbol{y} = \boldsymbol{b}, \boldsymbol{y} \geq \boldsymbol{0} \quad \text{oder} \quad ② \; A\boldsymbol{x} \geq \boldsymbol{b} \Leftrightarrow A\boldsymbol{x} - \boldsymbol{y} = \boldsymbol{b}, \boldsymbol{y} \geq \boldsymbol{0}$$

Zweitens werden nicht-eingeschränkte Variablen (also $\boldsymbol{x}$) in einen positiven $\boldsymbol{x}^+$ und negativen Teil $\boldsymbol{x}^-$ aufgespalten. Mit $\boldsymbol{x} = \boldsymbol{x}^+ - \boldsymbol{x}^-$, wobei $\boldsymbol{x}^+ = \max(\boldsymbol{x}, \boldsymbol{0})$, $\boldsymbol{x}^- = \max(-\boldsymbol{x}, \boldsymbol{0})$ ist, ergibt sich:

$$① \min \begin{pmatrix} \boldsymbol{c} \\ -\boldsymbol{c} \\ \boldsymbol{0} \end{pmatrix}^\mathsf{T} \begin{pmatrix} \boldsymbol{x}^+ \\ \boldsymbol{x}^- \\ \boldsymbol{y} \end{pmatrix} \text{ mit } \begin{pmatrix} A & -A & I \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^+ \\ \boldsymbol{x}^- \\ \boldsymbol{y} \end{pmatrix} = \boldsymbol{b}, \quad \begin{pmatrix} \boldsymbol{x}^+ \\ \boldsymbol{x}^- \\ \boldsymbol{y} \end{pmatrix} \geq \boldsymbol{0}$$

was die Standard-Form hat. Im Fall ② muss es natürlich $-I$ heißen. Falls die Bedingungen eine gemischte Form haben, können diese Transformationen auch zeilenweise angewendet werden.

Die Karush-Kuhn-Tucker-Bedingungen (KKT) für ein Optimum der Standardform sind:

$$\nabla_{\boldsymbol{x}}(\boldsymbol{c}^\top\boldsymbol{x} - \boldsymbol{\lambda}^\top(A\boldsymbol{x} - \boldsymbol{b}) - \boldsymbol{\mu}^\top\boldsymbol{x}) = \boldsymbol{0} \quad \Leftrightarrow \quad A^\top\boldsymbol{\lambda} + \boldsymbol{\mu} = \boldsymbol{c}$$

$$A\boldsymbol{x} = \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0},\ \boldsymbol{\mu} \geq \boldsymbol{0},\ \boldsymbol{\mu}^\top\boldsymbol{x} = 0$$

Die Menge der erlaubten Punkte, die die Gleichungen und Ungleichungen erfüllen ergeben ein Polytop, das von Hyperebenen begrenzt wird. Es lässt sich nun zeigen (fundamentaler Satz der linearen Optimierung), dass alle Eckpunkte dieses Polytops durch eine Auswahl $\boldsymbol{\kappa}$ von $m$ Variablen-Indizes bestimmt ist. Es seien $\bar{\boldsymbol{\kappa}}$ die Indizes, die nicht in $\boldsymbol{\kappa}$ sind: $\bar{\boldsymbol{\kappa}} = \{k \mid k \notin \boldsymbol{\kappa}\}$. Es seien $A_{\boldsymbol{\kappa}}$, $\boldsymbol{x}_{\boldsymbol{\kappa}}$, etc. die Matrizen und Vektoren, die auf die Indizes $\boldsymbol{\kappa}$ beschränkt sind. Dann gilt:

$$\boldsymbol{x}_{\bar{\boldsymbol{\kappa}}} = \boldsymbol{0}, \quad A\boldsymbol{x} = A_{\boldsymbol{\kappa}}\boldsymbol{x}_{\boldsymbol{\kappa}} = \boldsymbol{b}, \quad \boldsymbol{x}_{\boldsymbol{\kappa}} = A_{\boldsymbol{\kappa}}^{-1}\boldsymbol{b}$$

$$\boldsymbol{\mu}_{\boldsymbol{\kappa}} = \boldsymbol{0}, \quad A_{\boldsymbol{\kappa}}^\top\boldsymbol{\lambda} = \boldsymbol{c}_{\boldsymbol{\kappa}}, \quad A_{\bar{\boldsymbol{\kappa}}}^\top\boldsymbol{\lambda} + \boldsymbol{\mu}_{\bar{\boldsymbol{\kappa}}} = \boldsymbol{c}_{\bar{\boldsymbol{\kappa}}} \quad \Rightarrow \quad \boldsymbol{\lambda} = \left(A_{\boldsymbol{\kappa}}^\top\right)^{-1}\boldsymbol{c}_{\boldsymbol{\kappa}}, \quad \boldsymbol{\mu}_{\bar{\boldsymbol{\kappa}}} = \boldsymbol{c}_{\bar{\boldsymbol{\kappa}}} - A_{\bar{\boldsymbol{\kappa}}}^\top\boldsymbol{\lambda}$$

Wenn nun eine Menge $\boldsymbol{\kappa}$ gefunden wird, wo $\boldsymbol{\mu}_{\bar{\boldsymbol{\kappa}}} \geq \boldsymbol{0}$ ist, dann hat man das Minimum gefunden. Es wird daher ein *Entering-Index* $q \in \bar{\boldsymbol{\kappa}}$ mit $\mu_q < 0$ ausgewählt. Die Veränderung der Gleichung durch Hinzunahme des $x_q > 0$ ist dann

$$A_{\boldsymbol{\kappa}}(\boldsymbol{x}_{\boldsymbol{\kappa}} - \boldsymbol{\Delta}) + A_q x_q = \boldsymbol{b}, \quad \boldsymbol{\Delta} = A_{\boldsymbol{\kappa}}^{-1}A_q x_q, \quad \boldsymbol{x}_{\boldsymbol{\kappa}} - \boldsymbol{\Delta} \geq \boldsymbol{0},$$

wobei $\boldsymbol{x}_{\boldsymbol{\kappa}}$ für einen Index $p$ gleich 0 werden soll, also $x_p - \Delta_p = 0$. $p$ ist der *Leaving-Index* und ergibt sich aus:

$$p = \arg\min_{\substack{k \in \boldsymbol{\kappa} \\ d_k > 0}} \frac{x_k}{d_k}, \quad \boldsymbol{d} = A_{\boldsymbol{\kappa}}^{-1}A_q$$

Für die nächste Iteration gilt:

$$\underset{\rightarrow}{\boldsymbol{\kappa}} = \boldsymbol{\kappa} \cup \{q\} \setminus \{p\}$$

Für die initiale Index-Auswahl werden für $\boldsymbol{\kappa}$ meist die Indizes der Slack-Variablen ($\boldsymbol{y}$) verwendet. Wegen ① $A\boldsymbol{x} = I\boldsymbol{y} = \boldsymbol{b}$ gilt $\boldsymbol{x} \geq \boldsymbol{0}$, wenn $\boldsymbol{b} \geq \boldsymbol{0}$ ist, bzw. wegen ② $A\boldsymbol{x} = -I\boldsymbol{y} = \boldsymbol{b}$ gilt $\boldsymbol{x} \geq \boldsymbol{0}$, wenn $\boldsymbol{b} \leq \boldsymbol{0}$. Wenn das nicht erfüllt ist, muss man ein $\boldsymbol{\kappa}$ suchen, das $A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}$ erfüllt. Dafür löst man das Optimierungsproblem

$$① \min \boldsymbol{1}^\top\boldsymbol{z} \text{ mit } \begin{pmatrix} A & -I \end{pmatrix}\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{z} \end{pmatrix} = \boldsymbol{b}, \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{z} \end{pmatrix} \geq \boldsymbol{0},$$

das eine Lösung mit $\boldsymbol{z} = \boldsymbol{0}$ haben muss, wenn das ursprüngliche Problem überhaupt eine Lösung hat. $\boldsymbol{1}$ ist ein Vektor aus nur 1-en. Man startet dabei mit den

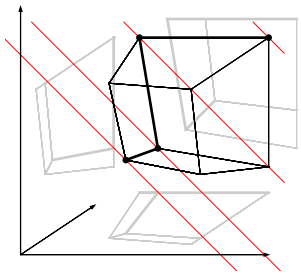


Abbildung 23: Simplex-Algorithmus. Beginnend rechts oben: $\boldsymbol{\kappa} = (6,7,8) \mapsto (3,6,7) \mapsto (3,4,6) \mapsto (3,4,5)$.

Indizes von $\boldsymbol{z}$ für $\boldsymbol{\kappa}$, was $\boldsymbol{z} \geq 0$ garantiert, wenn $\boldsymbol{b} \leq \mathbf{0}$ ist. Für ② muss man $I$ statt $-I$ einsetzen. Das $\boldsymbol{\kappa}$ der Lösung beinhaltet dann keinen $\boldsymbol{z}$-Index mehr (wegen $\boldsymbol{z} = \mathbf{0}$), erfüllt die Bedingung $A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq \mathbf{0}$ des ursprünglichen Problems, und kann daher als initiales $\boldsymbol{\kappa}$ für die Optimierung verwendet werden. Für zeilenweise gemischte Bedingungen ① und ② kann man natürlich statt $-I$ oder $I$ eine Diagonalmatrix mit entsprechenden Einträgen $-1$ oder $+1$ verwenden. Wenn nur manche Koeffizienten von $\boldsymbol{b}$ das falsche Vorzeichen haben, kann man $\boldsymbol{z}$ auch auf diese Indizes kürzen und statt $-I$ oder $I$ nur jene Spalten verwenden, die diesen Indizes entsprechen. Es gibt auch die Möglichkeit, beide Optimierungsprobleme gleichzeitig zu lösen, indem man $\min \boldsymbol{c}^\mathsf{T}\boldsymbol{x} + \omega \mathbf{1}^\mathsf{T}\boldsymbol{z}$ löst für $\omega$ groß genug.

Abbildung 23 zeigt einen Testlauf für

$$\min \begin{pmatrix} 1 \\ 1 \\ 0.5 \end{pmatrix}^\mathsf{T} \boldsymbol{x} \text{ mit } \begin{pmatrix} 1 & 1 & 3 \\ 1 & 6 & 1 \\ 6 & 1 & 1 \end{pmatrix} \boldsymbol{x} \geq \begin{pmatrix} -2.5 \\ -2.5 \\ -2.5 \end{pmatrix}.$$

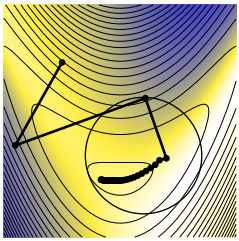
# 17 Penalty-Methode

Die Penalty-Methode ist eine einfache Möglichkeit, allgemeine Constraint-Probleme

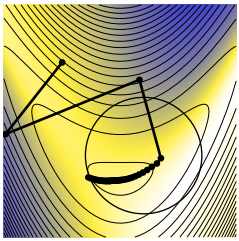
$$\min f(\boldsymbol{x}) \text{ mit } \boldsymbol{g}(\boldsymbol{x}) = \mathbf{0},\ \boldsymbol{h}(\boldsymbol{x}) \geq \mathbf{0}$$

zu lösen. Hier werden die Constraint-Funktionen einfach mit einem Faktor gewichtet zur zu optimierenden Funktion addiert:

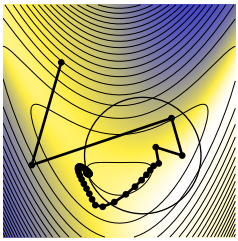
$$\min f(\boldsymbol{x}) + l\|\boldsymbol{g}(\boldsymbol{x})\|^2 + m\|\min(\boldsymbol{h}(\boldsymbol{x}), \mathbf{0})\|^2$$



(a) $\ell_2$ penalty



(b) $\ell_1$ penalty



(c) Augmented Lagrange ($l = 0.5$)

Abbildung 24: Penalty-Methode mit $\underset{\circ}{l} = 0.1$, $\underset{\rightarrow}{l} = 1.05l$.

Da dadurch die Constraints nicht exakt erfüllt würden, muss man während der Optimierungsschritte die Faktoren $l, m$ sukzessive erhöhen und gegen $\infty$ wachsen lassen.

Es sind auch andere Penalty-Funktionen als $\|\cdot\|^2$ möglich, z.B. die $\ell_1$-Norm $\|\cdot\|_1$. Diese ist oft effizienter.

Abbildung 24 zeigt Testläufe. Die Methode ist sehr sensitiv in Bezug auf die Wahl von $l$ und wie stark $l$ anwächst. Für große $l$ wird die Methode instabil und benötigt Backtracking.

# 18 Augmented Lagrange-Methode

Die Augmented Lagrange-Methode ist eine Erweiterung der Penalty-Methode um Lagrange-Multiplikatoren. Zu optimieren ist:

$$\min f(\boldsymbol{x}) \quad \text{mit} \quad \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0}$$

Wir addieren dazu jetzt einen Lagrange-Term $\boldsymbol{\lambda}^\mathsf{T}\boldsymbol{g}(\boldsymbol{x})$ und setzen die Ableitung auf 0:

$$\boldsymbol{0} = \nabla_{\boldsymbol{x}} \left( f(\boldsymbol{x}) - \boldsymbol{\lambda}^\mathsf{T}\boldsymbol{g}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{l}^\mathsf{T}\boldsymbol{g}(\boldsymbol{x})^2 \right) = \nabla f(\boldsymbol{x}) - (\boldsymbol{\lambda} - \boldsymbol{l}\boldsymbol{g}(\boldsymbol{x}))^\mathsf{T} \nabla \boldsymbol{g}(\boldsymbol{x})$$

Daraus ergibt sich, dass $\boldsymbol{\lambda} - \boldsymbol{l}\boldsymbol{g}(\boldsymbol{x})$ der optimale Lagrange-Multiplikator wäre. Daher wählen wir das Update-Schema:

$$\underset{\rightarrow}{\boldsymbol{\lambda}} = \boldsymbol{\lambda} - \boldsymbol{l}\boldsymbol{g}(\boldsymbol{x})$$

Dafür muss man das $l$ jetzt nicht mehr erhöhen.

# 19 Alternating Direction Multiplier-Methode

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) + g(\boldsymbol{x}) \quad \longrightarrow \quad \min_{\boldsymbol{x},\boldsymbol{y}} f(\boldsymbol{x}) + g(\boldsymbol{y}) \;\text{ mit }\; \boldsymbol{x} = \boldsymbol{y}$$

Lösen mit Augmented Lagrange-Methode abwechselnd

$$\underset{\rightarrow}{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} f(\boldsymbol{x}) \;\text{ mit }\; \boldsymbol{x} \approx \boldsymbol{y}$$
$$\underset{\rightarrow}{\boldsymbol{y}} = \arg\min_{\boldsymbol{y}} g(\boldsymbol{y}) \;\text{ mit }\; \boldsymbol{y} \approx \underset{\rightarrow}{\boldsymbol{x}}$$

# 20 Active-Set-Methode

Bei nichtlinearen Funktionen mit linearen Constraints kann man ähnliche Methoden anwenden wie im Simplex-Verfahren. Wir formulieren die Vorgehensweise vorerst für quadratische Funktionen.

$$\min \frac{1}{2}\boldsymbol{x}^\top G\boldsymbol{x} + \boldsymbol{c}^\top \boldsymbol{x} \;\text{ mit }\; A\boldsymbol{x} \geq \boldsymbol{b}$$

Entsprechend der Index-Menge $\boldsymbol{\kappa}$ des Simplex-Verfahrens gibt es hier die Active-Set $\boldsymbol{\alpha}$, also die Indizes, für die die Ungleichung zur Gleichung wird:

$$A_{\boldsymbol{\alpha}}\boldsymbol{x} = \boldsymbol{b}_{\boldsymbol{\alpha}}$$

Für ein festgelegtes $\boldsymbol{\alpha}$ und einen Vektor $\boldsymbol{d}$, der den Abstand zum aktuellen Optimierungspunkt $\boldsymbol{x}$ darstellt, ergibt sich das Optimierungsproblem:

$$\min_{\boldsymbol{d}} \frac{1}{2}(\boldsymbol{x}+\boldsymbol{d})^\top G(\boldsymbol{x}+\boldsymbol{d}) + \boldsymbol{c}^\top(\boldsymbol{x}+\boldsymbol{d}) \;\text{ mit }\; A_{\boldsymbol{\alpha}}(\boldsymbol{x}+\boldsymbol{d}) = \boldsymbol{b}_{\boldsymbol{\alpha}}$$

Wir führen nun Lagrange-Multiplikatoren $\boldsymbol{\lambda}$ ein und setzen den Gradienten null:

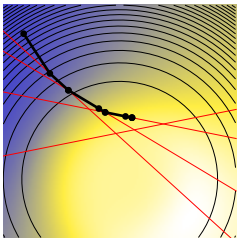
$$G(\boldsymbol{x}+\boldsymbol{d}) + \boldsymbol{c} - A_{\boldsymbol{\alpha}}^\top \boldsymbol{\lambda} = \boldsymbol{0}, \qquad A_{\boldsymbol{\alpha}}(\boldsymbol{x}+\boldsymbol{d}) = \boldsymbol{b}_{\boldsymbol{\alpha}}$$

Das kann man umformen in eine einzige Vektor-Gleichung in $\boldsymbol{d}$ und $\boldsymbol{\lambda}$,

$$\begin{pmatrix} G & -A_{\boldsymbol{\alpha}}^\top \\ A_{\boldsymbol{\alpha}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{d} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} -G\boldsymbol{x} - \boldsymbol{c} \\ -A_{\boldsymbol{\alpha}}\boldsymbol{x} + \boldsymbol{b}_{\boldsymbol{\alpha}} \end{pmatrix},$$

und daraus $\boldsymbol{d}$ und $\boldsymbol{\lambda}$ ausrechnen. Wir finden jetzt einen Entering-Index $q$ und Verkürzung von $\boldsymbol{d}$, sodass die Ungleichungen in $\bar{\boldsymbol{\alpha}}$ erfüllt sind:

$$A_i(\boldsymbol{x}+\beta_i\boldsymbol{d}) = b_i \quad \Rightarrow \quad \beta_i = \frac{b_i - A_i\boldsymbol{x}}{A_i\boldsymbol{d}}, \qquad q = \arg\min_{\substack{i\in\bar{\boldsymbol{\alpha}} \\ 0\leq\beta_i<1}} \beta_i$$



$$\min f(\boldsymbol{x}) = (\boldsymbol{x}^\mathsf{T}\boldsymbol{x})^2 \text{ mit}$$
$$\begin{pmatrix} 1.0 & 1.1 \\ 0.6 & 1.0 \\ 0.2 & 1.0 \\ -0.2 & 1.0 \end{pmatrix} \boldsymbol{x} \geq \begin{pmatrix} 0.8 \\ 1.0 \\ 1.1 \\ 0.8 \end{pmatrix}$$

Abbildung 25: Active-Set-Methode

Der Leaving-Index $p$ ist jener mit dem negativsten $\lambda_i$:

$$p = \arg\min_{\substack{i \in \boldsymbol{\alpha} \\ \lambda_i < 0}} \lambda_i$$

Die Iteration ist dann:

$\underset{\rightarrow}{\boldsymbol{\alpha}} = \boldsymbol{\alpha} \cup \{q\} \setminus \{p\}$ sofern $p$ und/oder $q$ gefunden wurden
$\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \beta_q \boldsymbol{d}$ oder $\underset{\rightarrow}{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{d}$, falls kein $q$ gefunden wurde

Diese Vorgehensweise kann man auch auf nicht-quadratische Funktionen anwenden: Dabei ersetzt man die Funktion durch durch einen lokale Approximation in Form einer quadratischen Funktion ($G = \nabla^2 f(\boldsymbol{x})$, $G\boldsymbol{x} + \boldsymbol{c} = \nabla f(\boldsymbol{x})$). Das führt zu einer Newton-Methode mit Active-Set, was auch als *Sequential-Quadratic-Programming* (SQP) bekannt ist. Abbildung 25 zeigt einen Testlauf.

Wenn man die Funktion lokal durch eine lineare Funktion approximiert, führt das zur Simplex-Methode mit Neuberechnung des $\boldsymbol{c}$-Vektors in jedem Schritt, was auch als *Sequential-Linear-Programming* (SLP) bekannt ist.

# 21 Lineare Interior-Point-Methode

Bei Interior-Point-Methoden versucht man, in den Iterationen den Rand des erlaubten Bereichs nicht zu touchieren, um die potentiell exponentielle Komplexität beim Hin- und Herspringen zwischen den Ungleichungs-Schnittpunkten zu vermeiden.

Wir suchen (vorerst) die Lösung für das lineare Optimierungsproblem:

$$\min \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \text{ mit } A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}$$

Die KKT-Bedingungen dafür sind:

$$A^\mathsf{T}\boldsymbol{\lambda} + \boldsymbol{\mu} = \boldsymbol{c},\ A\boldsymbol{x} = \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0},\ \boldsymbol{\mu} \geq \boldsymbol{0},\ \boldsymbol{\mu}^\mathsf{T}\boldsymbol{x} = 0$$

Es gibt zu diesem Optimierungsproblem ein duales Problem mit variierendem $\boldsymbol{\lambda}$:

$$\min -\boldsymbol{b}^\mathsf{T}\boldsymbol{\lambda} \;\text{ mit }\; \boldsymbol{c} - A^\mathsf{T}\boldsymbol{\lambda} \geq \mathbf{0}$$

Die KKT-Bedingungen dafür mit $\boldsymbol{x}$ als Lagrange-Multiplikator ergeben:

$$\nabla_{\boldsymbol{\lambda}}(-\boldsymbol{b}^\mathsf{T}\boldsymbol{\lambda} - \boldsymbol{x}^\mathsf{T}(\boldsymbol{c} - A^\mathsf{T}\boldsymbol{\lambda})) = \mathbf{0} \quad \Leftrightarrow \quad A\boldsymbol{x} = \boldsymbol{b}$$

Und wenn $\boldsymbol{\mu} = \boldsymbol{c} - A^\mathsf{T}\boldsymbol{\lambda}$ gesetzt wird, sind die KKT-Bedingungen genau gleich. Es wird damit also die selbe Lösung gefunden.

Wir versuchen nun, beide Probleme (primal und dual) gemeinsam zu lösen, indem $\boldsymbol{x}$, $\boldsymbol{\lambda}$ und $\boldsymbol{\mu}$ als variabel gelten und minimiert werden. In den Iterationen darf $\mu_i x_i > 0$ sein, das Produkt wird durch die Optimierungsschritte = 0 gemacht. Wir können dafür eine Newton-Optimierung formulieren mit:

$$\nabla g(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{pmatrix} A^\mathsf{T}\boldsymbol{\lambda} + \boldsymbol{\mu} - \boldsymbol{c} \\ A\boldsymbol{x} - \boldsymbol{b} \\ (\mu_i x_i \mid i) \end{pmatrix} \quad \Rightarrow \quad \nabla^2 g(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{pmatrix} \mathbf{0} & A^\mathsf{T} & I \\ A & \mathbf{0} & \mathbf{0} \\ \operatorname{diag}(\boldsymbol{\mu}) & \mathbf{0} & \operatorname{diag}(\boldsymbol{x}) \end{pmatrix}$$

Damit suchen wir die Lösung $\nabla g = \mathbf{0}$ mittels:

$$(\nabla^2 g) \begin{pmatrix} \boldsymbol{d_x} \\ \boldsymbol{d_\lambda} \\ \boldsymbol{d_\mu} \end{pmatrix} = -\nabla g$$

Die Lösung würde möglicherweise $\boldsymbol{x} \geq 0$ und $\boldsymbol{\mu} \geq 0$ verletzen. Außerdem wollen wir $\mu_i x_i = 0$ vermeiden. Daher verwenden wir eine modifizierte Bedingung:

$$\forall i : \mu_i x_i = \sigma u, \qquad u = \frac{\boldsymbol{\mu}^\mathsf{T}\boldsymbol{x}}{n}$$

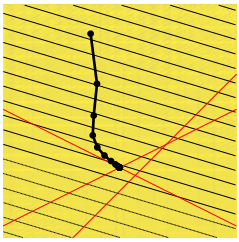
Dabei ist $u$ das *Dualitäts-Maß*, und $\sigma$ ist der *Reduktionsfaktor*. Die Newton-Iteration ist dann:

$$(\nabla^2 g)\boldsymbol{d} = -\nabla g + \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma u \mathbf{1} \end{pmatrix}$$

Die Iteration benötigt auch Backtracking, um $\boldsymbol{x} \geq 0$ und $\boldsymbol{\mu} \geq 0$ zu garantieren.

Eine andere Art, zum äquivalenten Ergebnis zu kommen, ist, einen *Log-Barrier*-Term einzuführen:

$$\min \boldsymbol{c}^\mathsf{T}\boldsymbol{x} - \tau \sum \ln x_i \;\text{ mit }\; A\boldsymbol{x} = \boldsymbol{b}$$



$$\min 0.3x_1 + x_2 \text{ mit}$$
$$\begin{pmatrix} 0.5 & 1.0 \\ -0.5 & 1.0 \\ -1.0 & 1.0 \end{pmatrix} \boldsymbol{x} \geq \begin{pmatrix} 0.2 \\ 0.2 \\ -0.2 \end{pmatrix}$$

Abbildung 26: Lineare Interior-Point-Methode

Die KKT-Bedingungen dafür sind:

$$A^\top \boldsymbol{\lambda} + \left( \frac{\tau}{x_i} \middle| i \right) = \boldsymbol{c}$$

Mit $\mu_i = \tau / x_i$ ergibt sich die äquivalente Bedingung

$$\mu_i x_i = \tau = \sigma u .$$

Die exakten Lösungen von

$$A^\top \boldsymbol{\lambda} + \boldsymbol{\mu} = c, \quad A\boldsymbol{x} = b, \qquad \forall i : \mu_i x_i = \tau ,$$

für variierendes $\tau$ ergeben einen *Pfad* zur Lösung des Optimierungsproblems, das bei $\tau = 0$ erreicht wird.

Die meisten Interior-Point-Methoden sind *Pfad-Folger*-Methoden, d.h. sie suchen eine (möglichst große) Schrittweite $\alpha$, sodass die nächste Iteration in der *Nachbarschaft* des Pfades liegt

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \\ \rightarrow \end{pmatrix} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix} + \alpha \begin{pmatrix} \boldsymbol{d_x} \\ \boldsymbol{d_\lambda} \\ \boldsymbol{d_\mu} \end{pmatrix}, \qquad \underset{\rightarrow}{\mu_i} \underset{\rightarrow}{x_i} \geq \sigma \underset{\rightarrow}{u} .$$

Abbildung 26 zeigt einen Testlauf.

# 22 Nichtlineare Interior-Point-Methode

Hier haben wir eine nicht-lineare Zielfunktion und nicht-lineare Constraints:

$$\min f(\boldsymbol{x}) \text{ mit } \boldsymbol{g}(\boldsymbol{x}) = 0, \boldsymbol{h}(\boldsymbol{x}) \geq \boldsymbol{0}$$

Um die nichtlinearen Ungleichungen in einfache lineare zu verwandeln, verwenden wir wieder Slack-Variablen $\boldsymbol{y}$:

$$\min f(\boldsymbol{x}) \;\text{ mit }\; \boldsymbol{g}(\boldsymbol{x}) = 0, \boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y} = \boldsymbol{0}, \boldsymbol{y} \geq \boldsymbol{0}$$

Die KKT-Bedingungen dafür sind:

$$\nabla_{\boldsymbol{x}} \Big( f(\boldsymbol{x}) - \boldsymbol{\lambda}^\top \boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\mu}^\top (\boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y}) \Big) = \boldsymbol{0} \quad \Leftrightarrow \quad \nabla f(\boldsymbol{x}) - \nabla \boldsymbol{g}(\boldsymbol{x})^\top \boldsymbol{\lambda} - \nabla \boldsymbol{h}(\boldsymbol{x})^\top \boldsymbol{\mu} = \boldsymbol{0}$$

$$\boldsymbol{g}(\boldsymbol{x}) = 0,\ \boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y} = \boldsymbol{0},\ \boldsymbol{y} \geq \boldsymbol{0},\ \boldsymbol{\mu} \geq \boldsymbol{0}$$

$$\boldsymbol{\mu}^\top \boldsymbol{y} = 0 \quad \longrightarrow \quad \forall i : \mu_i y_i = \tau = \sigma u, \quad u = \frac{\mu^\top \boldsymbol{y}}{n}, \sigma \in [0,1)$$

Die letzte KKT-Bedingung $\boldsymbol{\mu}^\top \boldsymbol{y} = 0$ haben wir hier wieder modifiziert, um einen „Interior Point“ zu bekommen. Dann betrachten wir sowohl $\boldsymbol{x}$ als auch $\boldsymbol{y}$, $\boldsymbol{\lambda}$ und $\boldsymbol{\mu}$ als Variablen und führen einen Newton-Schritt durch. Hier muss man explizit die KKT-Gleichungen nach allen Variablen ableiten. Es ergibt sich:

$$\begin{pmatrix} \nabla^2 f(\boldsymbol{x}) - \nabla(\nabla \boldsymbol{g}^\top \boldsymbol{\lambda}) - \nabla(\nabla \boldsymbol{h}^\top \boldsymbol{\mu}) & \boldsymbol{0} & -\nabla \boldsymbol{g}^\top & -\nabla \boldsymbol{h}^\top \\ \boldsymbol{0} & \operatorname{diag} \boldsymbol{\mu} & \boldsymbol{0} & \operatorname{diag} \boldsymbol{y} \\ \nabla \boldsymbol{g} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \nabla \boldsymbol{h} & -I & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{d_x} \\ \boldsymbol{d_y} \\ \boldsymbol{d_\lambda} \\ \boldsymbol{d_\mu} \end{pmatrix}$$

$$= - \begin{pmatrix} \nabla f(\boldsymbol{x}) - \nabla \boldsymbol{g}(\boldsymbol{x})^\top \boldsymbol{\lambda} - \nabla \boldsymbol{h}(\boldsymbol{x})^\top \boldsymbol{\mu} \\ (\mu_i y_i - \tau \mid i) \\ \boldsymbol{g}(\boldsymbol{x}) \\ \boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y} \end{pmatrix},$$

wobei

$$\nabla(\nabla \boldsymbol{g}^\top \boldsymbol{\lambda}) = \sum_j \nabla^2 g_j \lambda_j$$

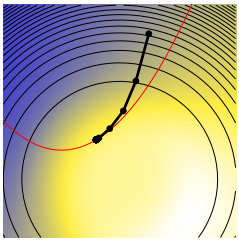
Auch hier käme man wieder zum selben Ergebnis, wenn man statt $f(\boldsymbol{x})$ eine Funktion mit Log-Barrier-Penalty minimiert:

$$\min f(\boldsymbol{x}) - \tau \sum_j \ln h_j(\boldsymbol{x}) \;\text{ mit }\; \boldsymbol{g}(\boldsymbol{x}) = 0, \boldsymbol{h}(\boldsymbol{x}) \geq \boldsymbol{0}$$

Die Iteration ist dann

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix}_{\rightarrow} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix} + \alpha \begin{pmatrix} \boldsymbol{d_x} \\ \boldsymbol{d_y} \\ \boldsymbol{d_\lambda} \\ \boldsymbol{d_\mu} \end{pmatrix}, \qquad \underset{\rightarrow}{\mu_i} \underset{\rightarrow}{y_i} \geq \sigma \underset{\rightarrow}{u}$$

Auch hier soll wieder die *Pfad-Folger*-Methode verwendet werden. Abbildung 27 zeigt einen Testlauf.



$$\min(\boldsymbol{x}^\mathsf{T}\boldsymbol{x})^2 \text{ mit}$$
$$x_1 \geq 0.5x_0^2 + x_0 + 1$$

Abbildung 27: Nichtlineare Interior-Point-Methode

# 23 Multi-Objective Optimization (Mehrzieloptimierung)

Wenn eine mehrwertige Funktion $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x}))$ optimiert werden soll, widersprechen sich üblicherweise die Optimierungsziele. Wenn $f_1(\boldsymbol{x})$ minimal ist, ist $f_2(\boldsymbol{x})$ nicht optimal, usw.

Wenn man nun $f_i(\boldsymbol{x})$ optimiert, während man alle anderen Funktionen festhält, also

$$\min \lambda_i f_i(\boldsymbol{x}) \text{ mit } \forall j \neq i : f_j(\boldsymbol{x}) = b_j\,,$$

wobei das $\lambda_i$ nur aus Symmetriegründen eingeführt wird, dann bekommt man mittels Lagrange-Multiplikatoren $\lambda_j$:

$$\nabla\left(\lambda_i f_i(\boldsymbol{x}) + \sum_{j\neq i} \lambda_j (f_j(\boldsymbol{x}) - b_j)\right) = \boldsymbol{0} \quad \Leftrightarrow \quad \nabla \boldsymbol{\lambda}^\mathsf{T} \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0} \quad \Leftrightarrow \quad \sum_j \lambda_j \nabla f_j(\boldsymbol{x}) = \boldsymbol{0}$$

Für nur zwei Funktionen heißt das

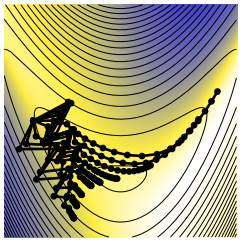
$$\nabla f_1(\boldsymbol{x}) = -\lambda \nabla f_2(\boldsymbol{x})\,,$$

also dass die Gradienten der Funktionen parallel sein müssen. Die Funktionswerte an den Stellen, an denen das gilt, nennt man **Pareto-Front**.

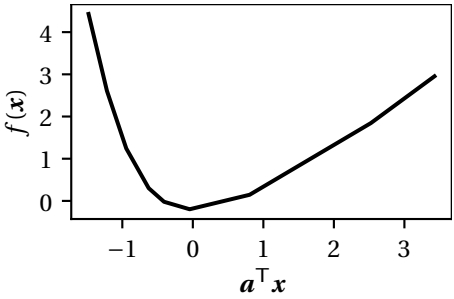
Setzen wir als Beispiel $f_2(\boldsymbol{x}) = \boldsymbol{a}^\mathsf{T}\boldsymbol{x}$. Dann ist $\nabla f_2(\boldsymbol{x}) = \boldsymbol{a}$ und:

$$\nabla f_1(\boldsymbol{x}) = -\lambda \boldsymbol{a}$$

Zum Testen führen wir Gradient-Descent für $f(\boldsymbol{x}) + \lambda \boldsymbol{a}^\mathsf{T}\boldsymbol{x}$ durch, wobei der Gradient $\nabla f(\boldsymbol{x}) + \lambda \boldsymbol{a}$ ist. Es sei $\boldsymbol{a} = \boldsymbol{1} = (1,1)$. Für verschiedene $\lambda$ sieht man das Ergebnis in Abbildung 28(a). Die Pareto-Front ist in Abbildung 28(b) dargestellt, wobei der linke Teil der übliche für positive $\lambda$ ist, wo eine Verbesserung von $\boldsymbol{a}^\mathsf{T}\boldsymbol{x}$ zu einer



(a) $\lambda = -3, -2, -1, 0, 1, 2, 4, 6, 8$




(b) Pareto-Front

Abbildung 28: Multi-Objective Optimization: $\min f(\boldsymbol{x}) + \lambda \boldsymbol{a}^\top \boldsymbol{x}$

Verschlechterung von $f(\boldsymbol{x})$ führt, und umgekehrt. Der rechte Teil für negative $\lambda$ *maximiert* $\boldsymbol{a}^\top \boldsymbol{x}$, während $f(\boldsymbol{x})$ minimiert wird.

Ein Anwendungsbeipiel für die Pareto-Front in der Bild- und Videokompression ist **Rate-Distorion-Optimierung**, wo man die Bitrate einer Datei minimieren möchte, aber auch die Distortion, also die Kompressionsartefakte, die durch die Kompression im Bild entstehen. Das Bild wird dabei in Blöcke $B_k$ zerlegt, die getrennt zu $B'_k = C(B_k, R_k)$ komprimiert werden, wobei $R_k$ die Anzahl der Bits des codierten Blocks ist. Die Distortion ist dann

$$D_k(R_k) = \|B'_k - B_k\|^2 .$$

Minimiert wird nun die Gesamt-Distortion und die Gesamt-Bitrate:

$$f_1(\boldsymbol{R}) = \sum_k D_k(R_k) , \qquad f_2(\boldsymbol{R}) = \sum_k R_k$$

Es folgt

$$\nabla f_1(\boldsymbol{R}) = (D'_k(R_k) \mid k) = -\lambda \nabla f_2(\boldsymbol{R}) = -\lambda \mathbf{1} ,$$

was bedeutet, dass alle Distortion-Steigungen der Blöcke den gleichen Wert haben müssen: $\forall k : D'_k(R_k) = -\lambda$. Durch Wahl einer Steigung $-\lambda$ bekommt man in Summe eine optimale Rate und Distortion des Gesamtbildes an der Pareto-Front.

# 24 Konvergenz

Die Iterationen $\boldsymbol{x}$ eines Algorithmus konvergieren mit einer *Konvergenzordnung* $q$ und einer *Konvergenzrate* $r$ zu einer Lösung $\boldsymbol{x}^*$, wenn

$$\frac{\|\underset{\rightarrow}{\boldsymbol{x}} - \boldsymbol{x}^*\|}{\|\boldsymbol{x} - \boldsymbol{x}^*\|^q} \longrightarrow r$$

- $q = 1, r < 1$: Lineare Konvergenz. Beispiel: $\underset{\rightarrow}{x} = \frac{x}{2}$ oder $\underset{k\rightarrow}{x} = 2^{-k}$.
- $q = 1, r = 1$: Sublineare Konvergenz. Beispiel: $\underset{k\rightarrow}{x} = \frac{1}{k}$.
- $q = 1, r = 0$: Superlineare Konvergenz. Beispiel: $\underset{k\rightarrow}{x} = k^{-k}$ (Quasi-Newton-Methoden).
- $q = 2$: Quadratische Konvergenz. Beispiel: $\underset{k\rightarrow}{x} = 2^{-2^k}$ (Newton-Methoden).

Für die Konvergenz vieler Algorithmen wird die *Lipschitz*-Stetigkeit gefordert, d.h. dass es eine Konstante $L$ gibt, sodass bei einer Abweichung $\Delta$ von einem Punkt $\boldsymbol{x}$ der Funktionswert nicht mehr als $L\Delta$ abweichen darf:

$$\forall \boldsymbol{x}_1, \boldsymbol{x}_2 : \|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)\| \leq L\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$$

Für den Beweis der quadratischen Konvergenz der Newton-Methode setzt man oft die Lipschitz-Stetigkeit der Hesse-Matrix $\nabla^2 f(\boldsymbol{x})$ voraus. (Man kann auch andere Voraussetzungen fordern.) Für $\|\nabla^2 f(\boldsymbol{x})\|$ braucht man natürlich eine geeignete Matrixnorm. Dann ergibt sich für eine Optimumsstelle $\boldsymbol{x}^*$:

$$\begin{aligned}
\|\underset{\rightarrow}{\boldsymbol{x}} - \boldsymbol{x}^*\| &= \|\boldsymbol{x} - (\nabla^2 f(\boldsymbol{x}))^{-1}\nabla f(\boldsymbol{x}) - \boldsymbol{x}^*\| \\
&= \|(\nabla^2 f(\boldsymbol{x}))^{-1}(\nabla^2 f(\boldsymbol{x})(\boldsymbol{x} - \boldsymbol{x}^*) - \nabla f(\boldsymbol{x}))\| \\
&\leq \|(\nabla^2 f(\boldsymbol{x}))^{-1}\| \cdot \|\nabla^2 f(\boldsymbol{x})(\boldsymbol{x} - \boldsymbol{x}^*) - \nabla f(\boldsymbol{x})\| \\
&\qquad \left[\nabla f(\boldsymbol{x}^*) = 0 \quad \Rightarrow \quad \nabla f(\boldsymbol{x}) = \nabla f(\boldsymbol{x}^* + t(\boldsymbol{x} - \boldsymbol{x}^*))\Big|_0^1\right. \\
&\qquad \left. = \int_0^1 \nabla^2 f(\boldsymbol{x}^* + t(\boldsymbol{x} - \boldsymbol{x}^*))(\boldsymbol{x} - \boldsymbol{x}^*)\,\mathrm{d}t\right] \\
&= \|(\nabla^2 f(\boldsymbol{x}))^{-1}\| \cdot \left\| \int_0^1 (\nabla^2 f(\boldsymbol{x}) - \nabla^2 f(\boldsymbol{x}^* + t(\boldsymbol{x} - \boldsymbol{x}^*))(\boldsymbol{x} - \boldsymbol{x}^*)\,\mathrm{d}t \right\| \\
&\leq \|(\nabla^2 f(\boldsymbol{x}))^{-1}\| \cdot \int_0^1 \underbrace{\|\nabla^2 f(\boldsymbol{x}) - \nabla^2 f(\boldsymbol{x}^* + t(\boldsymbol{x} - \boldsymbol{x}^*))\|}_{\leq Lt\|\boldsymbol{x} - \boldsymbol{x}^*\|} \cdot \|\boldsymbol{x} - \boldsymbol{x}^*\|\,\mathrm{d}t
\end{aligned}$$

$$\leq \|(\nabla^2 f(\boldsymbol{x}))^{-1}\| \cdot L \underbrace{\int_0^1 t \,\mathrm{d}t}_{=\frac{1}{2}} \|\boldsymbol{x} - \boldsymbol{x}^*\|^2$$

Wenn nun also in einem Bereich um die Optimumsstelle $\boldsymbol{x}^*$ die Norm der inversen Hesse-Matrix, die Lipschitz-Konstante $L$, sowie der aktuelle Abstand zur Optimumsstelle $\|\boldsymbol{x} - \boldsymbol{x}^*\|$ klein genug sind, konvergiert die Methode quadratisch.

# 25 Weitere Optimierungs-Methoden: Überblick

### Combinatorial Optimization

- Travelling-Salesman, Minimum-Spanning-Tree, Knapsack-Problem

Methoden:

- Zurückführen auf lineare Optimierung (Simplex)
- Heuristik (in üblichen Fällen gut, aber nicht optimal)
- Approximations-Methoden (im schlimmsten Fall gut)
  - Greedy-Algorithmus
  - Local-Search (aktuelle Lösung iterativ verbessern)
  - Dynamic-Programming (Optimum aus Optimum von überlappenden rekursiven Teilproblemen)
  - Kontinuierliche Optimierung plus Diskretisierung (Relaxation)
  - Parameterized-Approximation
- Constraint-Programming (Suchbaum frühestmöglich abschneiden, logische Programmiersprachen, Backtracking)

### Integer-Programming

- meist lineare Optimierung, aber ganzzahlige Lösung gefordert
- Bei total unimodularen Matrizen (alle quadratischen Submatrizen haben Determinante $\{1, 0, -1\}$) ist ganzzahlige Lösung garantiert
- Exakte Methoden (für wenige Variablen)
- Cutting-Plane, Branch-and-Cut (Relaxation plus anschließende Nachbarschaftssuche)

### Globale Optimierung

- Inner/Outer-Approximation (Approximation der Funktion durch Polyeder)
- Suchbäume
- Intervall-Arithmetik
- Algebraische Geometrie
- Stochastisch: Monte-Carlo-Sampling
- Heuristiken: Ant-Colony-Optimization, Simulated-Annealing, genetische Algorithmen, Graduated-Optimization, …

### Statistical-Learning

- Empirical-Risk-Minimization
- Stochastic-Approximation
- Robbins-Monro
- Kiefer-Wolfowitz
- Expectation-Maximization

## 26 Differenzieren von Vektoren und Matrizen

Um einen Vektor oder eine Matrix zu konstruieren, verwenden wir hier die Notation

$$\boldsymbol{a} = (a_i \mid i = 1 \dots n) = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix},$$

$$A = (A_{i,j} \mid i = 1 \dots m, j = 1 \dots n) = \begin{pmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{pmatrix}.$$

Der Gradient einer Funktion ist dann

$$\nabla f(\boldsymbol{x}) = \left( \frac{\mathrm{d}f}{\mathrm{d}x_i}(\boldsymbol{x}) \middle| i = 1 \dots n \right).$$

Der Gradient von inneren Produkten:

$$\nabla \boldsymbol{a}^\mathsf{T}\boldsymbol{x} = \left( \frac{\mathrm{d}}{\mathrm{d}x_j} \sum a_i x_i \middle| j \right) = (a_j \mid j) = \boldsymbol{a}$$

$$\nabla \left( |\boldsymbol{x}|^2 \right) = \nabla \left( \boldsymbol{x}^\mathsf{T}\boldsymbol{x} \right) = \left( \frac{\mathrm{d}}{\mathrm{d}x_j} \sum x_i^2 \middle| j \right) = (2x_j \mid j) = 2\boldsymbol{x}$$

Gradient einer quadratischen Form:

$$\begin{aligned} \nabla \boldsymbol{x}^\mathsf{T} A \boldsymbol{x} &= \left( \frac{\mathrm{d}}{\mathrm{d}x_k} \sum_{i,j} x_i A_{i,j} x_j \middle| k \right) = \left( \sum_i x_i A_{i,k} + \sum_j A_{k,j} x_j \middle| k \right) \\ &= \left( \sum_i (A_{i,k} + A_{k,i}) x_i \middle| k \right) = (A^\mathsf{T} + A)\boldsymbol{x} \overset{A \text{ symmetrisch}}{=\!=\!=} 2A\boldsymbol{x} \end{aligned}$$

Jacobi-Matrizen:

$$\nabla \boldsymbol{f}(\boldsymbol{x}) = \left( \frac{\mathrm{d}f_i}{\mathrm{d}x_j} \middle| i = 1 \dots m, j = 1 \dots n \right) = \left( \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{x})}{\mathrm{d}x_1} \dots \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{x})}{\mathrm{d}x_n} \right) = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\mathsf{T} \\ \vdots \\ \nabla f_m(\boldsymbol{x})^\mathsf{T} \end{pmatrix}$$

$$\nabla A\boldsymbol{x} = \left( \frac{\mathrm{d}}{\mathrm{d}x_k} \sum A_{i,j} x_j \middle| i, k \right) = (A_{i,k} \mid i, k) = A$$

Hesse-Matrix:

$$\nabla^2 f(\boldsymbol{x}) = \nabla(\nabla f(\boldsymbol{x})) = \left( \frac{\mathrm{d}^2 f}{\mathrm{d}x_i \, \mathrm{d}x_j}(\boldsymbol{x}) \middle| i, j \right)$$

$$\nabla^2 \boldsymbol{x}^\mathsf{T} A \boldsymbol{x} = \nabla(A^\mathsf{T} + A)\boldsymbol{x} = A^\mathsf{T} + A \overset{A \text{ symmetrisch}}{=\!=\!=} 2A$$