Textantworten Math Opt.

Aufgabe 2

Funktionen wie beispielsweise $x \rightarrow x^4$ oder $(x, y, z) \rightarrow x^4 + y^4 + z^4$ sind aufgrund des hohen Faktors der Variablen, stark nichtlinear. Insbesondere bei Werten $|x| > 1$ führen bereits kleine Veränderungen des x-Werts zu großen Veränderungen des Funktionswerts. Bei Quasi-Newton Verfahren (z.B. BFGS) führt das zu einer schlecht Konditionierten Hessematrix. Da hier die inverse der Hessematrix nicht berechnet sondern Approximiert und aus den vorherigen werten angepasst wird kommt es zur Fehlerfortpflanzung und numerischer Instabilität.

Mögliche Lösungen:

Man könnte die Eingabe Werte in die Funktion Skalieren, um die starke Nichtlinearität der Funktion abzuschwächen. Beispielsweise könnte man x durch $\acute{x} = \frac{x}{10}$ ersetzen. Ebenfalls kann man Line Search benutzen um eine Optimale Schrittweite zu wählen um zu große Schritte und damit einhergehend „Overshooting“ zu vermeiden. Eine weitere Möglichkeit wäre ein Stabileres Verfahren für stark nichtlineare Funktionen zu verwenden. L-BFGS speichert z.B. nicht alle Anpassungen der Hessematrix und verringert dadurch Fehlerfortpflanzung.

Problem in test_BFGS5:

Bei der Funktion $(x, y, z) \rightarrow 1000 * x^4 + y^4 + 0.001 * z^4$ kommt eine stark unterschiedliche Skalierung der Variablen hinzu. Das verschlechtert die Konditionierung der Funktion. Der Optimierungsalgorithmus würde stark in Richtung x konvergieren während er für y und insbesondere z viel länger braucht oder überhaupt nicht konvergiert. Dieses Problem könnte man durch Normalisierung der Faktoren lösen, indem man z.B. nur werte zwischen 0 und 1 zulässt.

Aufgabe 3

Die SQP Implementierung hat ursprünglich dafür gesorgt, dass jeder iterierte Punkt im zulässigen Bereich bleibt (durch LineSerachForFeasibility und das beschneiden der Punkte auf Bounds). Daraus folgt, dass der Algorithmus in falle eines Optima am Rand, nur sehr kleine Schritte macht bzw. generell Probleme das Optimum überhaupt zu erreichen. Ebenfalls führen die kleinen Schrittweiten zu einer instabilen Annäherung an die Hesse-Matrix da die Skalierung in den Update-Thermen sehr groß wird.

Als Lösung könnte man eine Merit Funktion oder Filter-Verfahren implementieren. Ich habe mich bei der Implementation für eine Merit-Funktion entschieden, da dieses Verfahren oft präziser ist (gute Initialisierung und Skalierung der Funktionsvariablen vorausgesetzt):

$$\phi(x) = f(x) + \rho \cdot max(0, |c(x)|)$$

Hierbei werden die Zielfunktion f(x) sowie die Nebenbedingungen c(x) mit einem adaptivem Strafparameter kombiniert. Dies erlaubt dem Algorithmus auch schritte zu gehen, die kurzfristig die

Zielfunktion verschlechtern, solange sie eine Verbesserung der Machbarkeit (Einhalten der Bedingungen) bewirken, wodurch das Optimum am Rand effizienter erreicht wird.

Aufgabe 5

Warum ist das finden von Startpunkten aufwändig?

Gerade im Bezug auf Optimierungsprobleme in einem hochdimensionalen Raum mit vielen Constrains oder in nicht-konvexen Räumen (passiert im Echten Leben tatsächlich Häufiger) Ist es schwierig einen geeigneten Startpunkt zu finden, der die Bedingungen der Constrains erfüllt (notwendig für die meisten Algorithmen). Ebenfalls kann ein schlecht gewählter Startpunkt aufgrund von starker Nichtlinearität zu konvergenzproblemen führen.

Welche Alternativen gibt es?

1.Zweistufige Optimierungsalgorithmus

- Lösen ohne Constrains mit mehreren zufälligen Startpunkten
- gefundene Minima als startpunkte einer zweiten Optimierung mit constraints und Straffunktion für verletzen der Constraints geben (Merit Funktion als Beispiel)

Nachteile/Probleme:

- Gefundene Minima könnten weit von zulässigen Mengen entfernt sein
- Konvergieren ins unendliche möglich → (adaptive(kleiner werdende falls dagegen konvergierende)) Bounds?

2.Zufalls-basierte Methoden

- Zufällige Punkte innerhalb eines definierten Bereichs generieren und nur die gültigen behalten
- Falls möglich Anpassen der Wahrscheinlichkeitsverteilung der Punkte auf Problemstruktur

Nachteile/Probleme:

- Dauert ewig bei hohen Dimensionen/vielen Constrains

3.Verkleinern der Suchmenge (Constraint reduction)

- Gleichungssystem aus Gleichheitsbedingungen aufstellen
- System lösen → Falls Unbestimmtes Gleichungssystem ergibt sich eine Lösungsmenge Falls keine Lösung: Doof :(
- Punkte (Falls Lösungsmenge, zufällige aus der Menge) in die Ungleichheitsbedingungen einsetzen
- Gültige Punkte als Startpunkte verwenden

Nachteile/Probleme:

- Gleichungssystem könnte keine Lösungen besitzen → Weiche Constrains mit Strafmethoden
- Könnte kostenintensiv zu berechnen sein

Constraint-Satisfying Gradient Descent (CSGD)

- Wählen eines Startpunkts
- Bilden einer Penaltyfunktion aus Constrains
- Optimierung hinsichtlich der Penaltyfunktion

Nachteile/Probleme

- Kann langsam sein, falls die Constraint-Verletzung schwer zu reduzieren ist

Warum sind zufällige Startpunkte oft Hilfreich?

1. Vermeiden von Lokalen Minima → Multi-Start Optimierung
2. Unbiased Exploration
3. Robustheit gegen schlechte Startwerte
4. Falls Suchmenge nicht analytisch Bestimmbar → evtl. effizienter zufällige punkte zu testen anstatt gezielt welche zu berechnen

Wie kann man das finden von Startpunkten deterministisch Gestalten? + Vorteile

Durch Constraint Reduction hat man Lösungsmengen aus denen man beispielsweise den Kleinsten Punkt (Euklidische Distanz zum Ursprung) nehmen kann. Die daraus resultierenden Vorteile wären zum einen Reproduzierbarkeit und Konsistenz und zum anderen, gegenüber dem rein zufälligem Suchen und auf Constraints prüfen, oft eine schnellere Konvergenz.

welche Operationen gehen schief,wenn man immer den gleichen Punkt deterministisch wählt?

- Keine Garantie, dass auch das globale Minimum gefunden wird → Besser Multistart mit mehreren zufälligen Werten

→ Besser: Kombination aus beidem (z.B. deterministisches Feststellen der Suchmenge und dort dann zufällige Punkte auswählen

Quellen:

WICHTIG: Constraint-Satisfaction-Problem

MICAI 2008: Advances in Artificial Intelligence S.65

Aufgabe 7

Es gibt verschiedene Möglichkeiten Pareto-Optimierung zu implementieren, hier Beispiele:

Gewichtete Summen

Hierbei werden die Zielfunktionen in gewichtete Summen mit Gewichts variablen kombiniert.

$$min(\sum_{i=1}^{m} w_i * f_i(x))$$

w := Gewicht

f(x) :=Zielfunktion

Durch das variieren der Gewichte werden unterschiedliche Punkte gefunden, aus diesen kann dann eine Pareto-front gebildet werden

Nachteile:

- Garantieren keine vollständige Pareto-front
- nicht geeignet für nicht konvexe Pareto-Fronten

ε-Constraint-Methode

Es wird nach einer funktion optimiert während die anderen Funktionen als Nebenbedingungen $f_i(x) \leq \varepsilon$ definiert werden, wobei durch Variation von ε verschiedene Pareto-optimale Lösungen entstehen

Nachteile:

- Sehr teuer (Mehrere Durchläufe, Startpunkte finden, etc.)
- Grenzen zu wählen kann schwierig sein

Vorteile:

- Gut für bi-kriterielle-Fronten
- kann nicht konvexe Fronten erfassen
- Gut wenn eine Optimierung priorisiert werden soll

Evolutionäre Algorithmen

Aus einem zufällig gewähltem Satz werden Funktionswerte mittels einer „Fitnessfunktion“ (Funktion aus den verschiedenen zu optimierenden Funktionen und Constrains) ausgewertet und verglichen. Dabei werden nur die besten in die folgende Generation übernommen und neue punkte aus den alten generiert (Mutation & Crossover). Dabei werden Werte abgeändert oder kombiniert um neue Punkte zu finden. Von diesen werden nur die besten in die nächste Generation übernommen

Nachteile:

- Teuer
- schwer zu realisieren bei vielen Constrains (→ Soft Constrains mit Penalty)
- Finden der Hyperparameter(Mutationsrate, Populationsgröße, Slektionsmechanismen, etc.) schwierig

Vorteile:

- Viele Pareto-Punkte gleichzeitig
- Gut Parallelisierbar
- Robust

Welche verfahren sind geeignet?

SQP (ε-Constraint-Methode oder gewichtete Summen)

Anpassungen:

Hauptfunktion bestimmen, andere Funktionen als Constrains < ε einbauen. Mehrere Optimierungsaufrufe mit verschiedenen Werten für ε einfügen um Pareto-Front zu bilden. (Verlauf der anderen Funktionen sollte ungefähr bekannt sein um sinnvolle ε zu wählen)

Bayesian Optimization

Anpassungen:

Für jede Zielfunktion muss ein separates GP-Modell trainiert werden, dafür müsste man eine Klasse in GP.py implementieren, die für eine gegebene Liste aus Y-Value-Arrays zu einer gegebenen Liste an x-Werten, jeweils ein GP trainiert und für jede Zielfunktion jeweils posterior Mean und Variance zurückgibt. BO.py müsste aus dieser Rückgabe dann dann den nächsten Punkt zu Auswertung bestimmen (bsp. Über das summieren und davon den niedrigsten Erwartungswert)

FeedForwardNN

Anpassungen:

Output-layer müsste an die Anzahl an zu optimierenden Funktionen angepasst werden um deren jeweilige Y Werte ausgeben zu können

Lossfunktion müsste Vektorwertig arbeiten

GradientDescent müsste ebenfalls angepasst werden, sodass mehrere Funktionen berücksichtigt werden

Nicht geeignet:

BFGS und DownhillSimplex sind nicht geeignet, da sie, selbst wenn man andere Zielfunktionen über inequality Constrains einfügt, viel zu lange brauchen würden und stark angepasst werden müssten. Algorithmen wie SQP oder BO sind deutlich besser geeignet für die Optimierung bezüglich Constraints

Nr.8

Posterior-Mean

$$\mu(x) = \widetilde{k}^T * K^{-1} * y$$

$\widetilde{k}^T$:= Vektor der Transponierten Kernelwerte zwischen dem Testpunkt und den Trainingspunkten X

K^{-1} := Inverse der Kovarianzmatrix der Trainingsdaten

y := beobachteten Werte

Die Ableitung nach X des ganzen ist demnach

$$\nabla \mu(x) = (\nabla \widetilde{k}^T) * K^{-1} * y$$

Posterior-Variance

$$\sigma^2(x) = k(x, \acute{x}) = \widetilde{k}^T * K^{-1} * \widetilde{k}$$

Die Ableitung nach x hierzu ist:

$$\nabla \sigma^2(x) = \nabla k(x, \acute{x}) = -2\widetilde{k}^T * K^{-1} * \nabla \widetilde{k}$$

Bei beiden Funktionen muss also explizit nur die Ableitung von $\widetilde{k}^T$ bestimmt und im falle der Post Variance, transponiert werden. Die Ableitung geht hierbei aus dem Kernel hervor. Der Vorteil daran ist, dass man neue Kernel direkt an diese Schnittstelle ran implementieren kann sofern man die Ableitung des neuen Kernels mit implementiert. Die Ableitung des Kerneilwertvektors selbst ist recht trivial

Nr.9

Bei Implementierungen von BO werden verschiedene GP-Kerne benutzt je nachdem was über die zu optimierende Funktion bekannt ist:

Der Initial implementiere Kernel „RBF“ ist gut geeignet wenn Zielfunktionen glatt und stetig sind, bei verrauschten oder diskontinuierlichen Funktionen ist dieser Kernel eher ungeeignet

Der hinzu implementierte „Matern“-Kernel ist deutlich robuster gegenüber Rauschen und eignet sich besser für komplexe Funktionen.

Der Periodic-Kernel eignet sich für Funktionen über die bereits bekannt ist, dass sie periodisch sind.

Je nachdem wie viel über die Zielfunktion erwartet wird kann man auch Kernel kombinieren um verschiedene Elemente der Funktion (bsp. Eine lineare und eine periodische Komponente) zu modellieren.

Meine Implementierung für Matern ist leider ~~schrottig~~ suboptimal weil ich auf externe Imports verzichten wollte (besser wäre scipy zu benutzen um modifyed_bessel_second_kind() zu ersetzen aber die Schnittstelle ist immerhin brauchbar

Nr.11

Initiale Annahme: Entscheidungsvariante von TSP ist in P

→ Entscheidungsvariante kann genutzt werden um kürzesten Weg zu finden:

Algorithmus dafür:

1.Obere und untere Schranke für Tourkosten:

- Cmax := Abstand aller Häuser zueinander
- Cmin := kleinster Abstand zwischen zwei Häusern

2.Anwenden Binärsuche auf Schranke b:

- lower = Cmin
- upper = Cmax
- Während upper-lower > 1:
  - b = (upper + lower)/2
  - Rufe Entscheidungsalgorithmus für b aufgrund
  - Falls dieser Ja ausgibt setze upper = b, sonst lower = b

Unter der Annahme, dass die Entscheidungvariante von TSP in P liegt würde dieser Algorithmus nach O(log Cmax) schritten die Optimalen Rundreise-Kosten liefern somit würde auch die Optimierungvon TSP in P liegen.