KI LaTeX DOKUMENT
Materiały do przedmiotu "Rozwiązywanie zadań odwrotnych"

Metody gradientowe - opis zagadnień związanych z klasycznymi algorytmami optymalizacyjnymi - metody L-BFGS-B, CG oraz TNC

dr inż. Konrad M. Gruszka,*

Abstract. Zapoznamy się z pozostałymi metodami przydatnymi z naszego punktu wiwdzenia dostępnymi w *scipy*.

1 Wprowadzenie

UWAGA! Jeżeli nie wiesz jak użyć minimize z *scipy.optimize* najpierw zapoznaj się z dokumentem o klasycznej metodzie BFGS

Metoda L-BFGS-B to rozwinięcie etody Broyden-Fletcher-Goldfarb-Shanno z naciskiem na optymalizację pamięci zużywanej w procesie minimalizacji funkcji kosztu, rozszerzoną o ograniczenia (B w nazwie - bounds, L - limited memory).

Użycie metody L-BFGS-B w zasadzie sprowadza się do tego samego co w przypadku czystego BFGS, z tą różnicą że w kodzie zmieniamy method="L-BFGS-B".

Dla wcześniej rozważanego problemu minimalizacji funkji:

$$f(x) = (x_0 - 1)^2 + (x_1 - 2)^2$$

korzystając z *minimize*:

```
from scipy.optimize import minimize
import numpy as np

# Funkcja celu
def func(x):
    return (x[0] - 1)**2 + (x[1] - 2)**2

# Gradient funkcji celu
def grad(x):
    return np.array([2 * (x[0] - 1), 2 * (x[1] - 2)])

# Punkt startowy
x0 = np.array([0, 0])

# Ograniczenia na zmienne
```

* Katedra Informatyki, Wydział Informatyki i Sztucznej Inteligencji (kgruszka@icis.pcz.pl)

```
bounds = [(0, 2), (1, 3)]  # x_0 w [0,2], x_1 w [1,3]

# Minimalizacja L-BFGS-B
res_lbfgsb = minimize(func, x0, method='L-BFGS-B', jac=grad, bounds=bounds, options={'disp':
↪  True})
print("L-BFGS-B:", res_lbfgsb.x)
```

Jak widatym razem wprowadziliśmy dodatkowo *bounds*. **bounds** jest to lista krotek (dolna_granica, górna_granica), gdzie każda krotka określa ograniczenia dla jednej zmiennej. Dla każdej zmiennej można ustawić:

- None jako brak ograniczenia z jednej strony.
- Konkretne wartości jako dolną i górną granicę.

Przykład użycia bounds z konkretnymi wymaganiami:
minimalizujemy funkcję:

$$f(X) = (x_0 - 3)^2 + (x_1 + 1)^2 + (x_2 - 5)^2$$

gradient tej funkcji:

$$\nabla f(x) = \begin{bmatrix} 2(x_0 - 3) \\ 2(x_1 + 1) \\ 2(x_2 - 5) \end{bmatrix}$$

Ograniczenia:
$1 \leqslant x_0 \leqslant 4$,
$-2 \leqslant x_1 \leqslant 2$,
$0 \leqslant x_2 \leqslant 6$

Aby znaleźć gradient, musimy obliczyć pochodne cząstkowe względem każdej zmiennej:

$$\frac{\partial f}{\partial dx_0} = 2(x_0 - 3)$$

$$\frac{\partial f}{\partial dx_1} = 2(x_0 + 1)$$

$$\frac{\partial f}{\partial dx_2} = 2(x_0 - 5)$$

Rozwiązanie za pomocą *minimize*:

```
import numpy as np
from scipy.optimize import minimize

# Definicja funkcji celu
def func(x):
    return (x[0] - 3)**2 + (x[1] + 1)**2 + (x[2] - 5)**2

# Gradient funkcji celu (pochodne cząstkowe)
def grad(x):
```

```
    return np.array([2 * (x[0] - 3), 2 * (x[1] + 1), 2 * (x[2] - 5)])

# Punkt startowy
x0 = np.array([0, 0, 0])

# Ograniczenia dla zmiennych
bounds = [(1, 4), (-2, 2), (0, 6)]  # x0 nalezy [1,4], x1 nalezy [-2,2], x2 nalezy [0,6]

# Minimalizacja metodą L-BFGS-B
res = minimize(func, x0, method='L-BFGS-B', jac=grad, bounds=bounds, options={'disp': True})

# Wynik
print("Minimum znalezione w:", res.x)
```

Proszę zwrócić uwagę, że gdybyśmy nie mieli ograniczeń, teoretyczne minimum funkcji wynosiłoby (3,-1,5). Warto również sprawdzić, jak będzie wyglądało rozwiązanie gdy nie podamy gradientu (jac).

2 Metoda TNC

Metoda TNC (Truncated Newton Conjugate Gradient Algorithm) jest wariantem metody Newtona zoptymalizowanym dla dużych problemów. Obsługuje ograniczenia (bounds) podobnie jak L-BFGS-B, ale stosuje inną strategię aktualizacji Hessianu.

```
import numpy as np
from scipy.optimize import minimize

# Definicja funkcji celu
def func(x):
    return (x[0] - 3)**2 + (x[1] + 1)**2 + (x[2] - 5)**2

# Gradient funkcji celu
def grad(x):
    return np.array([2 * (x[0] - 3), 2 * (x[1] + 1), 2 * (x[2] - 5)])

# Punkt startowy
x0 = np.array([0, 0, 0])

# Ograniczenia na zmienne
bounds = [(1, 4), (-2, 2), (0, 6)]  # x_0 w [1,4], x_1 w [-2,2], x_2 w [0,6]

# Minimalizacja metodą TNC
res_tnc = minimize(func, x0, method='TNC', jac=grad, bounds=bounds, options={'disp': True})

# Wynik
print("Minimum znalezione w:", res_tnc.x)
```

Metoda **TNC (Truncated Newton Conjugate Gradient)** to **Newtonowska metoda optymalizacji**, która stosuje gradienty oraz aproksymację drugich pochodnych (Hessiana), ale **bez konieczności przechowywania pełnej macierzy Hessego**. Jest to wersja metody Newtona zoptymalizowana pod kątem dużych problemów.

2.1 Idea metody TNC

Metoda TNC bazuje na metodzie Newtona do minimalizacji funkcji $f(x)$ używa rozwinięcia w szereg Taylora drugiego rzędu:

$$f(x+p) \approx f(x) + \nabla f(x)^T p + \frac{1}{2} p^T H(x) p \tag{1}$$

gdzie:

$\nabla f(x)$ to gradient funkcji celu,

$H(x)$ to macierz Hessego (druga pochodna cząstkowa),

p to kierunek poszukiwania.

Metoda Newtona wyznacza **optymalny kierunek poszukiwań** jako rozwiązanie równania:

$$H(x)p = -\nabla f(x) \tag{2}$$

co oznacza, że obliczamy:

$$p = -H(x)^{-1} \nabla f(x) \tag{3}$$

Jest to kierunek, który minimalizuje funkcję na podstawie drugiego rzędu przybliżenia.

2.2 Problem Newtona w dużych wymiarach

- Dla dużych problemów obliczenie i odwrócenie macierzy Hessego $H(x)$ jest **kosztowne obliczeniowo** (złożoność $O(n^3)$).
- Przechowywanie pełnej macierzy $H(x)$ zajmuje $O(n^2)$ pamięci, co jest problemem dla dużych wymiarów.

2.3 Jak działa metoda TNC?

Metoda **TNC** optymalizuje metodę Newtona poprzez:

(1) **Nie przechowuje pełnej macierzy Hessego** $H(x)$, ale oblicza przybliżenia kierunku Newtona.

(2) **Rozwiązuje problem Newtona iteracyjnie** za pomocą metod gradientowych sprzężonych (Conjugate Gradient – CG).

(3) **Truncation (obcinanie):** Ogranicza liczbę iteracji Conjugate Gradient, co zmniejsza obciążenie obliczeniowe.

(4) **Obsługuje ograniczenia 'bounds'**, co pozwala na minimalizację w określonych zakresach.

Kluczowy krok TNC: rozwiązanie kierunku Newtona przybliżoną metodą gradientów sprzężonych (CG)

Zamiast obliczać dokładnie $p = -H^{-1}\nabla f(x)$, metoda **TNC** przybliża to rozwiązanie **iteracyjnie** metodą gradientów sprzężonych:

$$H(x)p_k = -\nabla f(x)$$

Przybliżone rozwiązanie p_k pozwala na wykonanie kroku optymalizacji:

$$x_{k+1} = x_k + \alpha_k p_k$$

gdzie α_k to odpowiednio dobrana długość kroku.

2.4 Algorytm TNC

Krok 1: Inicjalizacja

- Wybieramy początkowy punkt x_0.
- Ustawiamy początkowy gradient $g_0 = \nabla f(x_0)$.
- Jeśli są ograniczenia 'bounds', modyfikujemy wartości x_0 do ich przedziałów.

Krok 2: Iteracyjny kierunek Newtona

- Używamy **gradientów sprzężonych (CG)** do rozwiązania układu:

$$H(x_k)p_k = -\nabla f(x_k)$$

Krok 3: Aktualizacja zmiennych

- Wykonujemy krok optymalizacyjny:

$$x_{k+1} = x_k + \alpha_k p_k$$

2.5 Implementacja w Pythonie

```python
import numpy as np
from scipy.optimize import minimize

# Definicja funkcji celu
def func(x):
    return (x[0] - 3)**2 + (x[1] + 1)**2 + (x[2] - 5)**2

# Gradient funkcji celu
def grad(x):
    return np.array([2 * (x[0] - 3), 2 * (x[1] + 1), 2 * (x[2] - 5)])

# Punkt startowy
x0 = np.array([0, 0, 0])

# Ograniczenia dla zmiennych
bounds = [(1, 4), (-2, 2), (0, 6)]  # x_0 w [1,4], x_1 w [-2,2], x_2 w [0,6]

# Minimalizacja metodą TNC
res_tnc = minimize(func, x0, method='TNC', jac=grad, bounds=bounds, options={'disp': True})

# Wynik
print("Minimum znalezione w:", res_tnc.x)
```

3 Metoda CG - Conjugate Gradient

Metoda **gradientów sprzężonych (CG - Conjugate Gradient)** jest iteracyjną techniką optymalizacji, stosowaną często do minimalizacji funkcji kwadratowych. Jest szczególnie skuteczna dla problemów o dużych wymiarach, gdzie przechowywanie pełnej macierzy Hessego jest niepraktyczne.

3.1 Problem optymalizacji

Rozważmy funkcję kwadratową:

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{4}$$

gdzie:

A to symetryczna dodatnio określona macierz,

b to wektor współczynników,

c to stała.

Gradient funkcji jest równy:

$$\nabla f(x) = Ax - b \tag{5}$$

Rozwiązanie równania $Ax = b$ daje punkt krytyczny funkcji, czyli jej minimum.

3.2 Idea metody gradientów sprzężonych

Standardowa metoda gradientowa wybiera kierunek poszukiwania równy gradientowi:

$$p_k = -\nabla f(x_k) \tag{6}$$

Jednak metoda gradientów sprzężonych wybiera kierunek w bardziej efektywny sposób:

$$p_k = -\nabla f(x_k) + \beta_k p_{k-1} \tag{7}$$

gdzie współczynnik sprzężenia β_k jest obliczany jako:

$$\beta_k = \frac{\nabla f(x_k)^T A p_{k-1}}{p_{k-1}^T A p_{k-1}} \tag{8}$$

lub w wersji bez potrzeby znajomości A:

$$\beta_k = \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} \tag{9}$$

3.3 Algorytm CG

Krok 1: Inicjalizacja

- Wybieramy początkowy punkt x_0.
- Ustawiamy początkowy gradient $g_0 = \nabla f(x_0)$.
- Inicjalizujemy pierwszy kierunek $p_0 = -g_0$.

Krok 2: Iteracyjna aktualizacja Dla każdej iteracji k:

(1) Obliczamy długość kroku α_k:

$$\alpha_k = \frac{g_k^T g_k}{p_k^T A p_k} \tag{10}$$

(2) Aktualizujemy zmienną:

$$x_{k+1} = x_k + \alpha_k p_k \tag{11}$$

(3) Obliczamy nowy gradient:

$$g_{k+1} = \nabla f(x_{k+1}) \tag{12}$$

(4) Obliczamy współczynnik sprzężenia:

$$\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k} \tag{13}$$

(5) Aktualizujemy kierunek:

$$p_{k+1} = -g_{k+1} + \beta_k p_k \tag{14}$$

(6) Jeśli $||g_{k+1}||$ jest małe, kończymy iteracje.

3.4 Implementacja w Pythonie

```python
import numpy as np
from scipy.optimize import minimize

# Definicja funkcji celu
def func(x):
    return (x[0] - 3)**2 + (x[1] + 1)**2 + (x[2] - 5)**2

# Gradient funkcji celu
def grad(x):
    return np.array([2 * (x[0] - 3), 2 * (x[1] + 1), 2 * (x[2] - 5)])

# Punkt startowy
x0 = np.array([0, 0, 0])

# Minimalizacja metodą CG
res_cg = minimize(func, x0, method='CG', jac=grad, options={'disp': True})

# Wynik
print("Minimum znalezione w:", res_cg.x)
```

UWAGA! Proszę zwrócić uwagę na obowiązkową definicję gradientu w tej metodzie!!.

4 Podsumowanie

Przedstawiono trzy metody gradientowe tj. L-BGFS-B, CG, TNC oraz sposoby ich implementacji z użyciem *scipy*. Jak widać, programy realizujące te metody są proste i nie wymagają znacznej ilości kodu do działania.