Optimisation TD 2
Méthode de Levenberg-Marquardt

C. Frindel

4 Novembre 2016

1 Introduction

Le but de ce TP est d'implémenter en python la méthode d'optimisation de Levenberg-Marquardt. Il s'agit d'une méthode d'optimisation "astucieuse" qui combine les avantages des méthodes de descente d'ordre 1 et d'ordre 2 étudiées lors du TP précédent.

Rappel: A l'ordre 1, la direction de descente d_1 d'une fonction f au point x est donnée par la formule suivante : $d_1 = -\nabla_f(x)$.
A l'ordre 2, la direction de descente d_2 d'une fonction f au point x est donnée par la formule suivante : $d_2 = -\frac{\nabla_f(x)}{\nabla_f^2(x)}$.

La méthode de Levenberg-Marquardt consiste à introduire un paramètre ($\lambda > 0$) qui favorise, lorsqu'il est judicieux de le faire, l'influence de l'ordre 1 par rapport à l'ordre 2. Il pondère l'influence des termes diagonaux de la matrice Hessienne, $H = \nabla_f^2(x)$, pour définir une nouvelle matrice H^{LM} :

$$\begin{aligned} H_{ii}^{LM} &= H_{ii}(1+\lambda) \\ H_{ij}^{LM} &= H_{ij} \end{aligned}$$

C'est donc la matrice H^{LM} qui est utilisée pour calculer la direction de descente : $d_{LM} = -\frac{\nabla_f(x)}{H_f^{LM}}$. Si λ est proche de zéro, H^{LM} tend vers l'expression d'une Hessienne classique, et donc $d_{LM} \to d_2$; et si λ est très grand, les termes diagonaux de H^{LM} sont prépondérants, ce qui revient à faire un pas dans la direction de plus forte pente ($d_{LM} \to d_1$). La valeur de λ évoluera en fonction de la proximité au minimum de la fonction de coût.

2 Régression non-linéaire

Dans ce TP, cette méthode sera appliquée à une régression non-linéaire. On suppose qu'on possède des données (observations) sous la forme de doublet (x_i, y_i). On cherche les paramètres d'une certaine fonction g qui approxime au mieux ces données. On suppose que g est connue et dépendante d'un vecteur de paramètres $\mathbf{a} \in \mathbb{R}^n$. Pour obtenir la régression non-linéaire, on va chercher à minimiser (en optimisant $\mathbf{a}$) la somme des erreurs au carré entre les observations y_i et la courbe $g(\mathbf{a})$ aux points x_i :

$$f(\mathbf{a}) = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - g(x_i, \mathbf{a})\right)^2$$

Les termes du gradient de f selon $\mathbf{a}$ s'écrivent comme suit :

$$\frac{\partial f}{\partial a_k} = -\sum_{i=1}^{N}\left(y_i - g(x_i, \mathbf{a})\right)\frac{\partial g}{\partial a_k}(x_i, \mathbf{a})$$

Les dérivées d'ordre 2 de f selon $\mathbf{a}$ s'écrivent tels que :

$$\frac{\partial^2 f}{\partial a_l \partial a_k} = \sum_{i=1}^{N} \left[\frac{\partial g}{\partial a_k}(x_i, \mathbf{a}) \frac{\partial g}{\partial a_l}(x_i, \mathbf{a}) - (y_i - g(x_i, \mathbf{a})) \frac{\partial^2 g}{\partial a_l \partial a_k}(x_i, \mathbf{a}) \right]$$

Dans un contexte de régression non-linéaire, il est possible d'utiliser l'approximation de Gauss-Newton qui permet d'éviter le calcul des dérivées secondes. En effet, cette approximation permet de déduire les dérivées d'ordre 2 à partir des dérivées d'ordre 1.

Approximation de Gauss-Newton : nous supposerons la fonction g suffisamment régulière et le terme $(y_i - g(x_i, \mathbf{a}))$ suffisamment faible pour pouvoir approximer les dérivées secondes par :

$$\frac{\partial^2 f}{\partial a_k \partial a_l} = \sum_{i=1}^{N} \frac{\partial g}{\partial a_k}(x_i, \mathbf{a}) \frac{\partial g}{\partial a_l}(x_i, \mathbf{a})$$

Remarquons ici que les dérivées secondes s'expriment uniquement comme un produit des dérivées d'ordre 1. Le calcul de la Hessienne ne sera donc pas nécessaire dans ce TP. Attention, cette approximation n'est valable que dans un contexte de minimisation de moindres carrés (régressions non-linéaires par exemple).

3 Cas mono-exponentiel

On va supposer que $g(x) = e^{-ax}$ avec $a \in \mathbb{R}$. On va donc faire une simple régression non-linéaire sur un espace de dimension 1.

1. Rappeler brièvement les avantages/inconvénients des méthodes de descente d'ordre 1 et 2 vues lors du précédent TP.
2. Écrire la fonction g qui prend en argument x et a.
3. Créer un jeu de données bruité sur l'intervalle $x \in [0, 3]$, avec $a = 2.0$, tel que $y = g(a, x) + bN(0, 1)$ avec b un paramètre faisant varier l'amplitude d'un bruit de distribution normale. Prendre un pas de 0.01 pour x. Utiliser la fonction `randn` de la librairie `numpy.random`.
4. Tracer les données simulées avec la fonction `pyplot` de la librairie `matplotlib`.
5. Écrire la fonction de coût f qui prend en entrée x, y et a
6. Écrire la fonction qui retourne le gradient de f par rapport à a
7. Écrire la fonction qui retourne les dérivées d'ordre 2 de f. Utiliser l'approximation de Gauss-Newton.
8. Implémenter l'algorithme de Levenberg-Marquardt :
   - Initialisation de a à 1.5
   - Initialisation de λ (Marquardt conseille à 0.001 pour commencer)
   - Tant que les critères d'arrêt ne sont pas pas satisfaits (vous inspirer du TP précédent) :
     - Calcul du gradient pour le a courant
     - Calcul de la dérivée seconde
     * Calcul de H^{LM}
     - Calcul de la direction de descente d_{LM}
     - Calcul du nouveau coût $f(a^{(k+1)}) = f(a^{(k)} + d_{LM})$
     - Si ce coût est plus faible que le coût courant, faire $a^{(k+1)} = a^{(k)} + d_{LM}$ et diviser λ par 10
     - Sinon multiplier λ par 10 et reprendre à $(*)$

9. Étudier l'évolution des paramètres de l'algorithme (λ, norme du gradient, valeur de la f) au fur et à mesure des itérations. Commenter.

10. Faire varier l'amplitude du bruit dans le jeu de données créé (paramètre b). Étudier l'influence sur la convergence de l'algorithme.

11. Tracer sur le même graphe le jeu de données ainsi que la fonction $g(x, a)$ optimale pour les valeurs de bruit testées.

Note : Cette méthode est suffisamment simple et robuste pour être utilisée avec plusieurs types de "fit" (i.e d'autres fonctions g). Elle peut donc être utilisée telle quelle pour de vraies données expérimentales.

4 Cas bi-exponentiel

On étudie maintenant le cas ou f va de $\mathbb{R}^2$ dans R i.e. pour une fonction g ayant deux paramètres en entrée. On pose $g(x, \mathbf{a}) = x^{a_1} e^{-a_2 x}$ avec, bien sûr, $\mathbf{a} = [a_1, a_2]$.

1. Définir comme précédemment la fonction g, un jeu de données bruité (pour $x \in [0, 5]$, $a_1 = 2.0$ et $a_2 = 3.0$), et la fonction f.

2. Calculer le gradient de f par rapport à $\mathbf{a}$.

3. Calculer la matrice des dérivées secondes avec l'approximation de Gauss-Newton. Indication: on peut utiliser la fonction dot de `numpy` pour calculer un produit scalaire.

4. Faire tourner l'algorithme de Levenberg-Marquardt pour en déduire le vecteur $\mathbf{a}$ optimal (initialisation de a à [1.5,1.5]). Indication: pour calculer l'inverse d'une matrice, utiliser la fonction `inv()` du package `numpy.linalg`.

5. Étudier l'évolution des paramètres de l'algorithme (λ, norme du gradient, valeur de la f) au fur et à mesure des itérations. Commenter et tester pour plusieurs amplitudes de bruit.

6. Afficher les résultats avec `matplotlib` pour plusieurs amplitudes de bruit.

7. Conclure sur les avantages de la méthode de Levenberg-Marquardt en faisant le lien avec la question 3.1.