

Le but de ce TD est de vous familiariser avec les différentes méthodes d'optimisation qui seront utilisées par la suite, d'en voir les limites et de comprendre dans quelles situations elles peuvent échouer.

## Descente de gradient

### Sur des fonctions à une seule variable

Nous allons dans un premier temps implémenter la descente de gradient pour des fonctions à une seule variable relativement simples.

L'idée de la descente de gradient est de partir d'un point  $x_0$  sur la courbe de la fonction  $f$  dont on veut trouver le minimum et de calculer la dérivée  $f'(x_0)$  en ce point pour savoir dans quelle direction le minimum se trouve à priori:

- Si  $f'(x_0) > 0$ , la fonction est croissante et donc à priori le minimum est plus petit que  $x_0$ .
- en revanche si  $f'(x_0) < 0$ , la fonction est décroissante et le minimum plus grand que  $x_0$ .

On se déplace ensuite sur la courbe de  $f$  d'un petit pas  $\eta$  (appelé *learning rate* dans le contexte du machine learning) pour arriver à un nouveau point et on recommence  $x_{n+1} = x_n - \eta f'(x_n)$ . On s'arrête quand la valeur de  $f'(x_n)$  est suffisamment proche de 0 : ceci peut être spécifié par un paramètre  $\epsilon$ , l'algorithme s'arrête alors quand  $|f'(x_n)| \leq \epsilon$ .

Ouvrez un nouveau cahier dans lequel vous complétez le code suivant:

```
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp

def gradient_descent(f,x0,eta,epsilon):
    x = sp.symbols("x")
    df = sp.lambdify(x,sp.diff(f(x),x)) # la dérivée de f (fonctionne si f n'est pas trop compliquée)
    # Votre code ici
    return result
```

Vous pourrez tester votre code avec les paramètres  $g = \text{lambda } x : x**2-x+1, \text{eta}=0.1 \text{ et } x0=10 \text{ et } \text{epsilon}=0.1$

Exportez votre code en allant dans "File-> Download .py".

Essayez maintenant avec la fonction  $f = \text{lambda } x: 0.01*(x**4+2*x**3-12*x**2-2*x+6)$  et les mêmes paramètres que précédemment. Cette fonction a deux minima en  $x_1 \approx -3.28$  et  $x_2 \approx 1.86$ . Si vous avez implémenté l'algorithme tel que donné ci-dessus, il aura normalement trouvé une valeur proche de  $x_2$ , mais la valeur en  $x_1$  est plus faible. L'algorithme a donc renvoyé un minimum local au lieu du minimum global.

Vous pouvez réessayer sur  $g$  avec  $\eta = .2, x_0 = 4$  et  $\epsilon = 0.1$ . Pourquoi le résultat est-il différent ?

Trouvez des conditions initiales avec lesquelles l'algorithme ne termine pas pour la fonction  $f(x) = x^2$ .

### Sur des fonctions multivariées

Soit  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , son gradient, noté  $\nabla f(\vec{x})$  est le vecteur

$$\begin{pmatrix} \frac{\partial f \vec{x}}{\partial x_1} \\ \vdots \\ \frac{\partial f \vec{x}}{\partial x_n} \end{pmatrix}$$

On peut voir le gradient comme la pente de la courbe dessinée par  $f$  selon chacune des directions.

Par exemple pour  $f(x,y) = x^2 + y^2$ :

$$\nabla f(\vec{x}) = \begin{pmatrix} \frac{\partial f \vec{x}}{\partial x_1} \\ \frac{\partial f \vec{x}}{\partial x_n} \end{pmatrix} = \begin{pmatrix} 2x \\ 2y \end{pmatrix}$$

L'algorithme de descente de gradient sur des fonctions à plusieurs variables est l'analogue de celui sur des fonctions à une seule variable: au lieu de prendre la dérivée, on prend  $\nabla f$ , et donc  $x_{n+1} = x_n - \nabla f(x_n)$ .

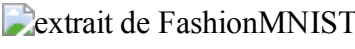
## Apprentissage par descente de gradient : approximations linéaires

### Descente de Gradient

Au lieu d'avoir une fonction  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  bien définie, on a maintenant une information partielle sur celle-ci : des observations  $y_i = f(\vec{x}_i)$  avec  $i \in \{1, \dots, n\}$ . Si l'on souhaite approximer  $f$  par une fonction affine, on doit trouver des paramètres  $\vec{w}$  et une constante  $b$  tels que  $\vec{w} \cdot \vec{x}_i + b$  soit proche de  $y_i$  pour tout  $i$ .

Prenons la fonction  $S(\vec{w}) = \frac{1}{n} \sum_i (y_i - \vec{w} \cdot \vec{x}_i - b)^2$ , une telle fonction est appelée fonction de perte (loss function en anglais) : plus sa valeur est proche de 0, plus l'approximation est proche de  $f$  sur les observations.

À l'aide de la méthode du gradient, on va essayer d'apprendre à reconnaître des habits. Pour cela on va se servir du jeu de données [FashionMNIST](#), qui consiste en des images d'habits de taille 28x28 et étiquetées en 10 catégories. Voici un extrait de ce jeu de données :



On peut facilement importer ce jeu de données avec pytorch:

```
from torchvision import datasets
import matplotlib.pyplot as plt

training_data = datasets.FashionMNIST(".",download=True,train=True)
test_data = datasets.FashionMNIST(".",download=True,train=False)
```

On a en fait importé deux jeux de données ici : un jeu d'entrainement dans `training_data`, et un jeu de test dans `test_data`. Chacun des deux est constitué d'un tableau de taille 60000 et 10000 respectivement, contenant un couple (image,label).

Vous pouvez afficher les afficher, par exemple pour le 1001ème élément :

```
plt.imshow(training_data[1000][0])
```

Afin de pouvoir effectuer du calcul matriciel sur ces images, on va les transformer en des tableaux unidimensionnels:

```
test_data = [(np.ndarray.flatten(np.array(img))/255,cat) for img,cat in test_data]
training_data = [(np.ndarray.flatten(np.array(img))/255,cat) for img,cat in training_data]
np.random.shuffle(training_data) # on mélange les données d'entraînement qui étaient classées
```

Dans un premier temps écrivez une fonction `propagate(w,b,training_data)` qui renvoie un couple `(cout,gradw,gradb)` contenant le coût  $S(w)$  et le gradient de  $S$  en  $w$  et en  $b$ . On pourra commencer par calculer, sur papier, l'expression du gradient. Pour le calcul matriciel on se servira de [Numpy](#).

Une fois la fonction `propagate` écrite, écrivez une fonction `optimize_gd(w,b,training_data,rate=5e-3,maxiter=1000,info=10)` qui implémente la descente de gradient et renvoie un triplet `(w,b,costa)` contenant les valeurs de  $\vec{w}$  et  $b$  ainsi qu'un tableau contenant le coût à chaque itération. La fonction termine soit quand on a atteint le nombre maximal d'itérations `maxiter`, soit quand la norme du gradient est suffisamment proche de 0 (en pratique ce sera souvent le premier cas).

**L'exécution peut prendre un certain temps, on fera donc en sorte que la fonction affiche le coût renvoyé par `propagate` toutes les `info` itérations (si le coût ne décroît pas, le `rate` n'est pas bien choisi).**

*On utilisera uniquement les 10000 premiers éléments de l'ensemble d'entraînement, et on lancera dans un premier temps l'algorithme avec un petit nombre maximum d'itérations.*

Déposez le fichier contenant les deux fonctions.

**Toutes les fonctions demandées dans ce TD dont le nom commencera par `optimize_` devront renvoyer un triplet `(w,b,costa)`.**

### Test du modèle

On va maintenant tester les paramètres obtenus sur le jeu de données de test `test_data`, il suffit pour cela de calculer pour une entrée  $\vec{w} \cdot \vec{x} + b$  et de voir de quel entier entre 0 et 9 sa valeur est proche. En pratique, il suffira de vérifier si on est à plus de 0.5 de la valeur attendue.

On veut en particulier connaître la proportion d'entrées correctement évaluées par ces paramètres. Vous allez donc écrire la fonction `test_params(w,b,test_data)` qui renvoie cette proportion. Vos chargés de TD ont obtenu  $\approx 0.33$  avec 1000 itérations (ce n'est pas si ridicule pour une approximation aussi simple, nous verrons dans un prochain TD comment obtenir de bien meilleurs résultats sur ce jeu de données).

*Note: On aurait pu ajouter une dimension contenant toujours 1 aux vecteurs représentant les images afin de se passer de  $b$ , cela aurait simplifié toutes les expressions et le code.*

### Descente de gradient stochastique

#### Online

Comme vous l'avez peut être remarqué, la méthode de la descente de gradient peut être lente : à chaque itération il faut évaluer les paramètres  $w$  sur toutes les observations  $(x_i, y_i)$ , une variante de la descente de gradient consiste à ne considérer qu'une observation à chaque itération.

Si on pose  $S(\vec{w}) = \sum_i Q_i(\vec{w})$ , la descente de gradient se fait alors avec

$$w_{n+1}^- = \vec{w}_n - \eta \nabla Q_i(\vec{w}_n).$$

On a donc une mise à jour des poids  $w$  par observation. On peut ainsi parcourir plusieurs fois le jeu de données pour calculer petit à petit le coût  $S(\vec{w})$  en additionnant les  $Q_i(\vec{w})$  calculés à chaque étape.

Écrivez la version stochastique `optimize_sgd(w,b,training_data,rate=5e-4,maxiter=40,info=1)` de la fonction `optimize_gd` (si l'on a écrit la fonction `propagate` de manière suffisamment générique, on pourra utiliser les *slices* de python pour se simplifier le travail) . On fera attention à bien calculer le coût, `maxiter` représentera ici le nombre de fois où l'on itère sur tout le jeu de données.

Jouez un peu avec les paramètres `rate,maxiter`. Le résultat obtenu avec cette méthode par vos chargés de TD est  $\approx 0.33$  pour 1000 itérations sur les 10000 premiers éléments.

**NOTE** : On utilise ici le terme d'itération pour un passage sur tout le jeu de données, mais en anglais le terme utilisé est *epoch* et le terme *iteration* est réservé à chaque mise à jour du vecteur  $\vec{w}$ .

#### Mini-batch

Un variante intermédiaire entre la descente de gradient stochastique et la descente de gradient classique est d'utiliser des sous-lots du jeu d'entraînement au lieu d'utiliser une seule observation.

Écrivez la fonction `optimize_sgdn(w,b,training_data,batch_size=100,rate=5e-4,maxiter=40,info=1)`, la version mini-batch de `optimize_sgd`.

Vos chargés de TD ont obtenu des résultats proches de  $\approx 0.33$ .

### Comparaison de convergence

Comparez les vitesses de convergence des différentes méthodes précédentes en affichant le graphe des coûts par itérations. Vous pourrez utiliser la fonction suivante :

```
def graph(**kwargs):
    colors = "grayrwykw"
    fig, ax = plt.subplots()

    for name,color in zip(kwargs,colors):
        ax.semilogx(kwargs[name],color,label=name)
    ax.legend()
    plt.show()

# exemple d'utilisation
graph(toto1=[1,2,3],toto2=[3,4,5])
# le label de [1,2,3] est "toto1"
```

Il n'y a pas vraiment de différence sur notre exemple.

## Moment de Nesterov, ADAM, NADAM

Le problème de la descente de gradient stochastique/classique est qu'elle peut très facilement rester bloquée sur un minimum local ou faire des bonds d'un côté à l'autre d'une vallée en se dirigeant très lentement vers le bas. Pour éviter cela, on peut s'inspirer de la physique et lui donner une vélocité et donc un moment. L'idée étant que l'on arrivera sur un minimum local avec une certaine vélocité et que l'on arrivera à le "dépasser" et qu'au fur et à mesure de la descente de la vallée on gagnera en vitesse dans cette direction pour y arriver plus rapidement.

### La méthode du moment

La manière classique d'implémenter le moment est de commencer avec une vitesse  $v = 0$  et de la mettre à jour à chaque étape :

$$v_{n+1} = \mu w_n - \eta \nabla Q(\vec{w}_n)$$

Le paramètre  $\mu$  représente un coefficient de friction, il sert à ce qu'il puisse y avoir une stabilisation au minimum, on le choisit en général égale à 0.9 ou 0.95. La mise à jour des poids  $w$  se fait alors grâce à la vitesse :

$$w_{n+1}^- = \vec{w}_n + v_{n+1}$$

Implémentez `def optimize_sgd_moment(w,b,training_data,rate=1e-3,mu=0.95,maxiter=40,info=1)`.

Vos chargés de TD ont obtenu des résultats proches de  $\approx 0.36$  : légèrement mieux que la descente de gradient classique, pour un temps de calcul bien inférieur.

### La méthode du moment de Nesterov

La méthode du moment de Nesterov diffère de la méthode du moment par le fait que l'on va d'abord faire un bond dans la direction de la vitesse précédente :

$$\vec{w}_n' = w_n + \mu \vec{w}_n$$

puis mettre à jour avec la nouvelle vitesse calculée en ce point :

$$v_{n+1} = \mu \vec{w}_n' - \eta \nabla S(\vec{w}_n')$$

$$w_{n+1}^- = \vec{w}_n + v_{n+1}$$

Implémentez :

- Soit `def optimize_sg_nesterov(w,b,training_data,rate=1e-5,mu=0.95,maxiter=500,info=10)` pour la descente de gradient (non-stochastique) .
- Soit `def optimize_sgd_nesterov(w,b,training_data,rate=1e-5,mu=0.95,maxiter=40,info=1)` pour la descente de gradient stochastique .

Vos chargés de TD ont obtenu  $\approx 0.36$  pour le moment de Nesterov stochastique et  $\approx 0.35$  pour le moment de Nesterov avec descente de gradient classique.

Notez que la méthode du moment de Nesterov peut ne pas converger quand utilisée avec une descente de gradient stochastique.

#### Adam, Nadam

La méthode Adam a été décrite en cours (on peut également la retrouver [ici](#)).

Implémentez la descente de gradient stochastique avec Adam `def optimize_sgd_adam(w,b,training_data,rate=1e-3,beta1=0.9,beta2=0.999,epsilon=1e-8,maxiter=10,info=1)`. Résultat obtenu par vos chargés de TD  $\approx .36$ .

**(Bonus)** Implémentez la descente de gradient avec NAdam (l'article qui introduit cette méthode peut être trouvé [ici](#)).

### Learning Rate automatique

On a pour l'instant fixé à l'avance un `rate` pour chacune des méthodes d'optimisation comme on l'a vu au début du TD avec l'exemple du polynôme, un rate fixe ne garantit pas la convergence de la descente de gradien.

On peut cependant adapter le `rate` en fonction de la position à laquelle on est. Il existe d'ailleurs des conditions garantissant la convergence si le pas est bien choisi à chaque étape.

Etant donné une fonction  $f$  que l'on cherche à minimiser, les **conditions fortes de Wolfe** sont les suivantes :

- Condition de décroissance :

$$f\Big(\vec{x}_n - \eta_n \nabla f(\vec{x}_n)\Big) \leq f(\vec{x}_n) - c_1 \eta_n \nabla f(\vec{x}_n)^T \nabla f(\vec{x}_n)$$

Cette condition permet de s'assurer que le  $\eta_n$  choisi fait suffisamment décroître la valeur de  $f$ .

- Condition de courbure :

$$\left| \nabla f(\vec{x}_n)^T \nabla f\Big(\vec{x}_n - \eta_n \nabla f(\vec{x}_n)\Big) \right| \leq c_2 \left| \nabla f(\vec{x}_n)^T \nabla f(\vec{x}_n) \right|$$

Cette condition permet de s'assurer que le  $\eta_n$  choisi fait suffisamment décroître le gradient.

où  $0 < c_1 < c_2 < 1$ . Les constantes  $c_1$  et  $c_2$  peuvent être choisies égales à  $10^{-4}$  et 0.1 respectivement.

Qu'il faille faire une recherche pour trouver une bonne valeur de  $\eta_n$  à chaque étape fait que cette méthode est peu utilisée en pratique : la recherche de  $\eta_n$  est elle même une étape de minimisation, même si l'on peut se contenter d'avoir une valeur approximative qui satisfait tout de même les conditions.

## Conclusion

### Comparaison de convergence

Refaites une comparaison des vitesses de convergence avec un graphique. Que pouvez-vous dire sur les dernières méthodes en particulier ? Est-il toujours nécessaire d'itérer sur tout le jeu de données à votre avis ?

Déposez votre code.

Le nom du fichier à déposer | [Choisissez File](#) | [No file chosen](#) | [Déposer](#)

téléchargement du fichier : TD1\_gradientdescent1)jpyyb (136475 octets) réalisé le 14/02/2020 à 22:34:51