TME 3 - Descente de gradient

1 Algorithme de descente de gradient

L'algorithme de descente de gradient permet d'approcher le minimum d'une fonction convexe de manière itérée selon la formule de mise-à-jour de x à l'instant t suivante :$\mathbf{x_{t+1}} = \mathbf{x_t} - \epsilon * \nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x_t})$. La structure `Fonction` ci-dessous permet d'enregistrer les 3 informations nécessaires à l'optimisation de la fonction : la fonction elle-même, son gradient et la dimension de son entrée.

```
from collections import namedtuple
Fonction = namedtuple("Fonction",["f","grad","dim"])  #declaration de la structure
def f_id(x) : return x                                #definition de l afonction
def f_id_grad(x) : return 1                           #definition du gradient
identite = Fonction(f_id,f_id_grad,1)                 #construire la structure
identite.f(1), identite.grad(1), identite.dim #utiliser la structure

def v2m(x):
    return x.reshape((1,x.size)) if len(x.shape)==1 else x
```

Coder une fonction `optimize(fonc,eps=0.01,max_iter=100,xinit=None)` qui permet d'optimiser une fonction contenue dans `fonc`, avec un pas de descente `epsilon`, qui réalise `max_iter` itérations avec un point initial (si donné) `xinit` sinon tiré au hasard. Cette fonction doit rendre un triplet `(log_x,log_f,log_grad)`, `log_x` étant la matrice des points parcourus lors de l'optimisation, `log_f` des valeurs de la fonction et `log_grad` des valeurs du gradient.

Conventions : dans toute la suite des TMEs, nous considérerons des fonctions qui prennent en paramètre une matrice d'exemples. Attention en python, il y a de grandes confusions possibles entre matrice et vecteur ! Un vecteur de taille n à une propriété `shape` du type $(n,)$, alors qu'une matrice du type (n, d) (potentiellement $d = 1$), les multiplications ne se déroulent pas de la même façon (élément par élément ou multiplication matricielle...). Nous travaillerons dans toute la suite qu'avec des matrices, en ligne chaque exemple, en colonne les différentes dimensions. Prévoyez vos fonctions pour des matrices, pas pour des vecteurs (impossibilité pour un vecteur de faire $v[:, 0]$ par exemple pour obtenir la première dimension de tous les exemples). N'hésitez pas à utiliser la fonction `v2m` ci-dessus qui permet de transformer un vecteur en matrice en cas de besoin (au début de chacune de vos fonctions).
Vous aurez entre autre besoin des fonctions numpy : `np.vstack((m1,m2))` qui permet "d'empiler" les 2 matrices verticalement (l'une sur l'autre), et de la fonction `np.hstack((m1,m2))` qui permet de les empiler horizontalement cote à cote.

2 Applications

Testez votre implémentation sur 3 fonctions :

- en 1d sur la fonction $f(x) = x.\cos(x)$
- en 1d sur la fonction $-\log(x) + x^2$
- en 2d sur la fonction Rosenbrock (ou banana), définie par : $f(x_1, x_2) = 100*(x_2-x_1^2)^2+(1-x_1)^2$.

Tracer les courbes d'évolution de f en fonction du nombre d'itérations. Tracer (en 3d ou en 2d) la surface de la fonction, et la trajectoire de l'optimisation. Vous pouvez utilisez le code suivant pour la visualisation 3d.

```
1    grid,xx,yy = make_grid(xmin=-1,xmax=1,ymin=-1,ymax=1)
2    fig = plt.figure()
3    ax = fig.gca(projection='3d')
4    surf = ax.plot_surface(xx, yy, mafonction.f(grid).reshape(xx.shape), rstride=1,
         cstride=1, cmap=cm.gist_rainbow, linewidth=0, antialiased=False)
5    fig.colorbar(surf)
6    ax.plot(log_x[:,0],log_x[:,1],log_f.ravel(),color='black')
7    plt.show()
```

Régression logistique Implémentez la résolution pour la régression logistique. Testez sur les données artificielles et sur les données réelles MNIST (reconnaissance des chiffres).