TABLE OF CONTENTS

1 DESCENTE DE GRADIENT

L'algorithme de la descente de gradient est un algorithme d'optimisation pour trouver un minimum local d'une fonction scalaire à partir d'un point donné, en effectuant de pas successifs dans la direction de l'inverse du gradient.

Pour une fonction $f : \mathbb{R}^n \to \mathbb{R}$, partant d'un point $\mathbf{x}_0$, la méthode calcule les points successifs dans le domaine de la fonction

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta (\nabla f)_{\mathbf{x}_n} \ , \tag{1}$$

où

$\eta > 0$ est une taille de /pas/ suffisamment petite et and $(\nabla f)_{\mathbf{x}_n}$ est le gradient de f évaluée au point $\mathbf{x}_n$. Les valeurs successives de la fonction

$$f(\mathbf{x}_0) \geq f(\mathbf{x}_1) \geq f(\mathbf{x}_2) \geq \dots \tag{1}$$

vont décroître globalement et la séquence $\mathbf{x}_n$ converge habituellement vers un minimum local.

En pratique utiliser un pas de taille fixe η est particulièrement inefficace et la plupart des algorithmes vont plutôt chercher à l'adapter à chaque itération.

Le code suivant implémente la descente de gradient avec un pas de taille fixe s'arrétant quand la norme du gradient descend en dessous d'un certain seuil.

Attention par défaut, pytorch *accumule* les gradients à chaque passe inverse! C'est pourquoi il faut le remettre à zéro à chaque itération.

Commençons par importer les suspects usuels

Typesetting math: 100%

Commençons par importer les suspects usuels

In [1]:

```python
import torch
import numpy as np
import math
```

Illustrons l'accumulation du gradient

Illustrons l'accumulation du gradient

In [2]:
```python
x1 = torch.empty(2, requires_grad=True)
x1
```

Out[2]:
```
tensor([1.1210e-44, 0.0000e+00], requires_grad=True)
```

Illustrons l'accumulation du gradient

In [2]:

```
x1 = torch.empty(2, requires_grad=True)
x1
```

Out[2]:

```
tensor([1.1210e-44, 0.0000e+00], requires_grad=True)
```

In [3]:

```
f1 = torch.pow(x1[0],2)
f1
```

Out[3]:

```
tensor(0., grad_fn=<PowBackward0>)
```

Illustrons l'accumulation du gradient

In [2]:
```
x1 = torch.empty(2, requires_grad=True)
x1
```

Out[2]:
```
tensor([1.1210e-44, 0.0000e+00], requires_grad=True)
```

In [3]:
```
f1 = torch.pow(x1[0],2)
f1
```

Out[3]:
```
tensor(0., grad_fn=<PowBackward0>)
```

In [4]:
```
# x1.grad.zero_()
f1.backward(retain_graph=True)
x1.grad
```

Out[4]:
```
tensor([2.2421e-44, 0.0000e+00])
```

In [5]:
```
x1.data.sub_(torch.ones(2))
```

Maintenant essayons d'implémenter une descente de gradient pour la fonction $f(X) = sin(x_1) + cos(x_2)$

Maintenant essayons d'implémenter une descente de gradient pour la fonction $f(X) = sin(x_1) + cos(x_2)$

In [6]:

```
x0 = torch.ones(2,requires_grad=True)
```

Maintenant essayons d'implémenter une descente de gradient pour la fonction $f(X) = sin(x_1) + cos(x_2)$

In [6]:
```
x0 = torch.ones(2,requires_grad=True)
```

In [7]:
```
f = torch.sin(x0[0]) + torch.cos(x0[1])
f
```

Out[7]:
```
tensor(1.3818, grad_fn=<AddBackward0>)
```

On va avoir besoin de :

```
f.backward(...) # Pour le calcul du gradient propre
nt dit
x.grad.data.zero_() # pour la remise à zéro du grad
nt après une itération
np.linalg.norm(x.grad.numpy()) # pour contrôler la 
nvergence (norme l2)
```

On veut une fonction gd qui prend en argument f, x, η, ϵ

On va avoir besoin de :

```
f.backward(...) # Pour le calcul du gradient proprement dit
x.grad.data.zero_() # pour la remise à zéro du gradient après une itération
np.linalg.norm(x.grad.numpy()) # pour contrôler la convergence (norme l2)
```

On veut une fonction gd qui prend en argument f, x, η, ϵ

In [8]:

```
def gd(f, x, eta, epsilon):
    while 1:
        f.backward(retain_graph=True)
#        print(np.linalg.norm(x.grad.numpy()))
        if (np.linalg.norm(x.grad.numpy()) < epsilon):
            break
```

In [9]:

```
gd(f, x0, 0.9, 0.00001)
```

In [9]:

```python
gd(f, x0, 0.9, 0.00001)
```

In [10]:

```python
print(x0.data)
print(f.data)
```

```
tensor([-1.5708,  3.1416])
tensor(1.3818)
```

Cette fonction ne permet pas d'avoir la valeur de f directement sur le résultat. Il vaut mieux utiliser une fonction qu'un noeud de notre graphe comme argument de notre descente de gradient.

Cette fonction ne permet pas d'avoir la valeur de f directement sur le résultat. Il vaut mieux utiliser une fonction qu'un noeud de notre graphe comme argument de notre descente de gradient.

```
In [11]:
x0 = torch.ones(2,requires_grad=True)
x0
```

```
Out[11]: tensor([1., 1.], requires_grad=True)
```

Cette fonction ne permet pas d'avoir la valeur de f directement sur le résultat. Il vaut mieux utiliser une fonction qu'un noeud de notre graphe comme argument de notre descente de gradient.

In [11]:

```
x0 = torch.ones(2,requires_grad=True)
x0
```

Out[11]:

```
tensor([1., 1.], requires_grad=True)
```

In [12]:

```
def f(x):
    return x[0].sin() + x[1].cos()
```

Cette fonction ne permet pas d'avoir la valeur de f directement sur le résultat. Il vaut mieux utiliser une fonction qu'un noeud de notre graphe comme argument de notre descente de gradient.

In [11]:

```
x0 = torch.ones(2,requires_grad=True)
x0
```

Out[11]:

```
tensor([1., 1.], requires_grad=True)
```

In [12]:

```
def f(x):
    return x[0].sin() + x[1].cos()
```

In [13]:

```
def gd(f, x, eta, epsilon):
    while 1:
        f(x).backward()
#         print(np.linalg.norm(x.grad.numpy()))
        if (np.linalg.norm(x.grad.numpy()) < epsilon):
            break
        else:
```

In [14]:

```
gd(f, x0, 0.9, 0.00001)
```

In [14]:

```
gd(f, x0, 0.9, 0.00001)
```

In [15]:

```
print(x0)
print(f(x0))
```

```
tensor([-1.5708,  3.1416], requires_grad=True)
tensor(-2., grad_fn=<AddBackward0>)
```