TABLE OF CONTENTS

1 DESCENTE DE GRADIENT STOCHASTIQUE

La Descente de Gradient Stochastique est une variante stochastique de la descente de gradient, qui permet d'utiliser un algorithme efficace pour minimiser les fonctions de coût sous la forme d'une somme:

$$Q(\mathbf{w}) = \sum_{i=1}^{d} Q_i(\mathbf{w}) \ , \tag{1}$$

En utilisant la descente de gradient standard, Q peut être minimisée par l'itération suivante:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \eta \nabla Q \\ &= \mathbf{w}_t - \eta \sum_{i=1}^{d} \nabla Q_i(\mathbf{w}_t)\ , \end{aligned}$$

où $\eta > 0$ est la taille du *pas*. Cette méthode d'itération par traitement de lots (*batch*) permet de calculer le coût total à chaque pas avec un temps de calcul qui est proportionnel à la taille d de l'ensemble d'entraînement.

De manière similaire, dans la descente de gradient stochastique, Q est minimisée en utilisant

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla Q_i(\mathbf{w}_t) \ , \tag{1}$$

en mettant à jour les poids $\mathbf{w}$ à chaque itération en utilisant juste un échantillon i choisi aléatoirement de l'ensemble d'entraînement. Ceci est extrêmement efficace pour les grandes tailles d'échantillons, parce que cela permet d'obtenir un temps de calcul de chaque itération indépendant de d. Un autre avantage est que cela permet de traiter les échantillons à la volée, comme une tâche d'apprentissage en ligne.

En pratique, au lieu d'un η fixé, l'algorithme décroît la taille du pas η_t pour améliorer la convergence.

On va implémenter une version avec pas constant.

En pratique, au lieu d'un η fixé, l'algorithme décroît la taille du pas η_t pour améliorer la convergence.

On va implémenter une version avec pas constant.

In [1]:

```
import torch
import torch.utils.data
import numpy as np
```

Dans cette implémentation Q_i a la forme

$$Q_i(\mathbf{w}) = \|\mathbf{y}_i - f_{\mathbf{w}}(\mathbf{x}_i)\| \ , \tag{1}$$

où $f_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}^m$ est une fonction *modèle* pour notre jeu de données, paramétrisée par $\mathbf{w}$; $\mathbf{x}_i \in \mathbb{R}^n$ et $\mathbf{y}_i \in \mathbb{R}^m$ sont la paire d'entrées/sorties pour l'$i$ème échantillon de l'ensemble d'apprentissage. Trouver les paramètres $\mathbf{w}$ qui minimisent $Q(\mathbf{w}) = \sum_{i=1}^{d} Q_i(\mathbf{w})$ et qui ajuste la fonction modèle $f_{\mathbf{w}}$ à nos données.

On va le tester grâce à l'ajustement de courbe(*curve fitting*) suivante :

$$f_{\mathbf{w}}(x) = w_1 x^2 + w_2 x + w_3 \tag{2}$$

In [2]:

```python
def f(w,x):
    return w[0] * x * x + w[1] * x + w[2]
```

In [2]:

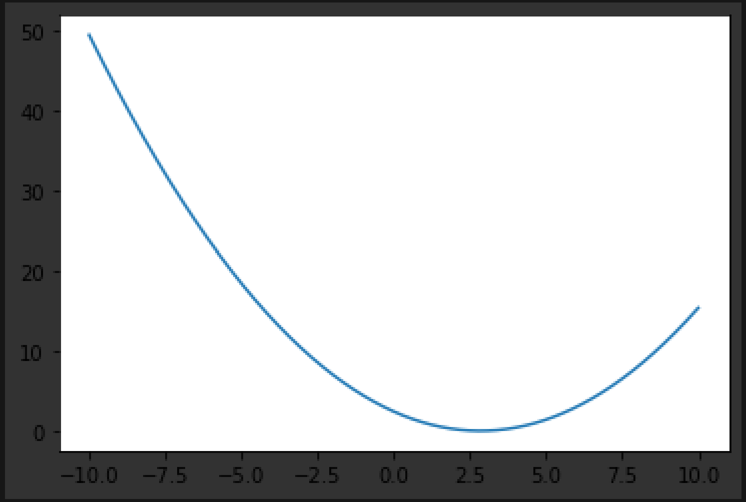
```python
def f(w,x):
    return w[0] * x * x + w[1] * x + w[2]
```

Commençons par définir notre jeu de données

In [6]:

```
wt = [0.3,-1.7,2.5] # Ce sont les paramètres que l'on va essayer de retro
xtrain = np.arange(-10,10,0.01,dtype='f')
ytrain = (lambda x : f(wt,x))(xtrain)

%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(xtrain,ytrain);
```



Pour traiter notre ensemble d'entraînement il faut mettre ensemble entrées(*input*) et sorties(*target*), via la fonction `concatenate`.

Typesetting math: 100%

Pour traiter notre ensemble d'entraînement il faut mettre ensemble entrées(*input*) et sorties(*target*), via la fonction `concatenate`.

In [7]:
```
xtrainnp = np.array([xtrain])
ytrainnp = np.array([ytrain])
dataTconcat = np.concatenate((xtrainnp,ytrainnp))
points = np.transpose(dataTconcat)
points
```

Out[7]:
```
array([[-10.      ,  49.5     ],
       [ -9.99    ,  49.423027],
       [ -9.98    ,  49.346115],
       ...,
       [  9.970457,  15.373228],
       [  9.980457,  15.416082],
       [  9.990458,  15.458996]], dtype=float32)
```

Typesetting math: 100%

On initialise les paramètres w

On initialise les paramètres w

```
In [4]: w0 = torch.zeros(3,requires_grad=True)
```

Dans une première version de la *SGD* naïve, on va itérer sur tous les échantillons et incrémenter notre vecteur de poids sur chacun d'eux.

On aura une fonction :

```
def sgd(f,w,t,eta=0.00001, epsilon=0.001,nepoch=100)
```

où `eta` est la taille du pas, `epsilon` la limite de convergence du coût, et `nepoch` le nombre maximum de fois où l'on aura parcouru tous les échantillons.

EXERCICE 1

Compléter le code ci-dessous pour faire une descente de gradient

EXERCICE 1

Compléter le code ci-dessous pour faire une descente de gradient

In [8]:

```python
# Installer tqdm et éventuellement ipywidgets
# from tqdm import tqdm # Si pas ipywidgets
from tqdm.notebook import tqdm

def sgd(f,w,t,eta=0.0001, epsilon=0.01,nepoch=20):
    bar = tqdm(total=nepoch) #
    epoch = 0
    x = t[:,0]
    y = t[:,1]
    while(epoch < nepoch):
        loss = torch.mean(torch.pow(y-f(w,x),2))
        loss.backward()
        w.data.sub_(eta * w.grad)
        w.grad.data.zero_()
        bar.update() # Affichage
```

In [9]:

```
w0 = torch.zeros(3,requires_grad=True)
sgd(f,w0,torch.tensor(points),eta=0.0002,epsilon=0.1,nepoch=10000)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
convergence au bout de 9340 epochs.
```

On vérifie que l'on s'est rapproché des paramètres du modèle.

On vérifie que l'on s'est rapproché des paramètres du modèle.

In [10]:

```
print(wt)
print(w0)
torch.norm(torch.tensor(wt)-w0)
```

```
[0.3, -1.7, 2.5]
tensor([ 0.3079, -1.7000,  2.0258], requires_grad=True)
```

Out[10]:

```
tensor(0.4743, grad_fn=<NormBackward0>)
```

EXERCICE 2

Légère simplification, utiliser la fonction de coût toute faite `torch.nn.MSELoss()`

```
criterion = torch.nn.MSELoss()
```

Ecrire la version du sgd avec cette fonction de coût "prête à l'emploi"

EXERCICE 2

Légère simplification, utiliser la fonction de coût toute faite `torch.nn.MSELoss()`

```
criterion = torch.nn.MSELoss()
```

Ecrire la version du sgd avec cette fonction de coût "prête à l'emploi"

In [ ]:

```
def sgd(f,w,t,eta=0.0001, epsilon=0.01,nepoch=20):
    bar = tqdm(total=nepoch) #
    epoch = 0
    criterion = torch.nn.MSELoss()
    x = t[:,0]
    y = t[:,1]
    while(epoch < nepoch):
```

In [ ]:

```
w0 = torch.zeros(3,requires_grad=True)
sgd(f,w0,torch.tensor(points),eta=0.0002,epsilon=0.1,nepoch=10000)
```

2 MINI-BATCH

On va maintenant mettre en placer la vraie version *minibatch* du SGD. Pour ce faire, on utilise directement les facilités du `DataLoader` de `pytorch`, qui combine un jeu de données et un échantilloneur.

In [11]:

```
trainloader = torch.utils.data.DataLoader(points,batch_size=10,shuffle=Tr
```

EXERCICE 3

énumérer tous les éléments de chaque batch avec le print suivant

EXERCICE 3

énumérer tous les éléments de chaque batch avec le print suivant

In [ ]:

```
# print("batch %i, échantillon %i : x = %f, y = %f" % (i,j,x.item(),y.ite
```

EXERCICE 4

Réécrire la descente de gradient version mini-batch en utilisant le `trainloader`. Attention pour le calcul du coût il faut maintenant cumuler sur tous les batch à chaque époque. Bien réfléchir à la bonne stratégie.

In [12]:

```
def sgd(f,w,t,eta=0.0001, epsilon=0.01,nepoch=200,minibatch=10):
    epoch = 0
    bar = tqdm(total=nepoch)
    criterion = torch.nn.MSELoss()
    trainloader = torch.utils.data.DataLoader(t,batch_size=minibatch,shuf
    while(epoch < nepoch):
        running_loss = 0.
        for data in trainloader:
            x , y = data[:,0],data[:,1]
            loss = criterion(f(w,x),y)
            loss.backward()
            w.data.sub_(eta * w.grad)
            w.grad.data.zero_()
            running_loss = running_loss + loss.item() * len(data)
        running_loss = running_loss / len(t)
        bar.update()
        bar.set_description("loss %f" % running_loss)
        epoch += 1
        if (running_loss < epsilon):
            print("convergence au bout de %i epochs." % epoch)
            break
```

In [14]:

```
w0 = torch.zeros(3,requires_grad=True)
sgd(f,w0,torch.tensor(points),eta=02,epsilon=0.1,nepoch=50)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
convergence au bout de 48 epochs.
```

In [14]:

```
w0 = torch.zeros(3,requires_grad=True)
sgd(f,w0,torch.tensor(points),eta=02,epsilon=0.1,nepoch=50)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
convergence au bout de 48 epochs.
```

In [15]:

```
print(wt)
print(w0)
torch.norm(torch.tensor(wt)-w0)
```

```
[0.3, -1.7, 2.5]
tensor([ 0.3072, -1.7006,  2.0476], requires_grad=True)
```

Out[15]:

```
tensor(0.4525, grad_fn=<NormBackward0>)
```

Maintenant il nous reste plus qu'à utiliser les optimiseurs de `torch.optim` comme `Adam`, `RMSProp` etc.

il faut charger l'optimiseur avec les paramètres w

```
optimizer = torch.optim.Adam([w], lr=lr)
```

Après chaque rétropropagation, on peut appeler `optimizer.step()`: cela mettra effecturea directement le "pas" de descente et remet à zéro les gradients, on n'a plus besoin de le faire.

EXERCICE 5

Version du sgd avec l'optimiseur RMSprop.

In [16]:

```python
def sgd(f,w,t,lr=0.01, epsilon=0.01,nepoch=200,minibatch=10):
    epoch = 0
    bar = tqdm(total=nepoch)
    optimizer = torch.optim.RMSprop([w], lr=lr)
    trainloader = torch.utils.data.DataLoader(t,batch_size=minibatch,shuf
    criterion = torch.nn.MSELoss()
    while (epoch < nepoch):
        running_loss = 0.0
        for i, data in enumerate(trainloader):
            optimizer.zero_grad()
            x, y = data[:,0],data[:,1]
            loss = criterion(f(w,x),y)
            running_loss = running_loss + loss.item() * len(data)
            loss.backward()
            optimizer.step()
        running_loss = running_loss / len(t)
        bar.update()
        bar.set_description("loss %f" % running_loss)
        epoch += 1
        if (loss.item() < epsilon):
            print("convergence au bout de %i itérations." % epoch)
            break
```

In [19]:

```
w0 = torch.zeros(3,requires_grad=True)
sgd(f,w0,torch.tensor(points),epsilon=1e-7,nepoch=50)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
convergence au bout de 23 itérations.
```

In [19]:

```
w0 = torch.zeros(3,requires_grad=True)
sgd(f,w0,torch.tensor(points),epsilon=1e-7,nepoch=50)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
convergence au bout de 23 itérations.
```

In [20]:

```
print(wt)
print(w0)
torch.norm(torch.tensor(wt)-w0)
```

```
[0.3, -1.7, 2.5]
tensor([ 0.3000, -1.7000,  2.4999], requires_grad=True)
```

Out[20]:

```
tensor(0.0001, grad_fn=<NormBackward0>)
```

EXERCICE 6

Réessayer les sgd avec l'option shuffle=False dans le trainloader. Que constatez-vous ? Quelle explication ?

EXERCICE 7

- Comparer dans la sgd la fonction de coût $L(X, Y)$ sur l'ensemble des points avec la somme des fonctions de coûts des batchs $\sum L(X_i, Y_i)$ déj). Comment expliquer la différence ? Est-ce grave docteur ? (indice, penser à l'inégalité triangulaire)
- Même question mais pour la norme du gradient.