Descente de gradient

La descente de gradient (*Gradient Descent*) est un algorithme d'optimisation qui permet de rechercher le minimum d'une fonction (minimiser une fonction) qui est différentiable (dont on peut calculer la dérivée) définie sur un espace Euclidien (par exemple $\mathbb{R}^n$). Ce n'est pas un concept nouveau, il a été inventé par un mathématicien Français, Louis Augustin Cauchy (https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf) en 1847. Il est par contre d'un grand intérêt aujourd'hui notamment pour l'entrainement des réseaux profonds (deep learning).

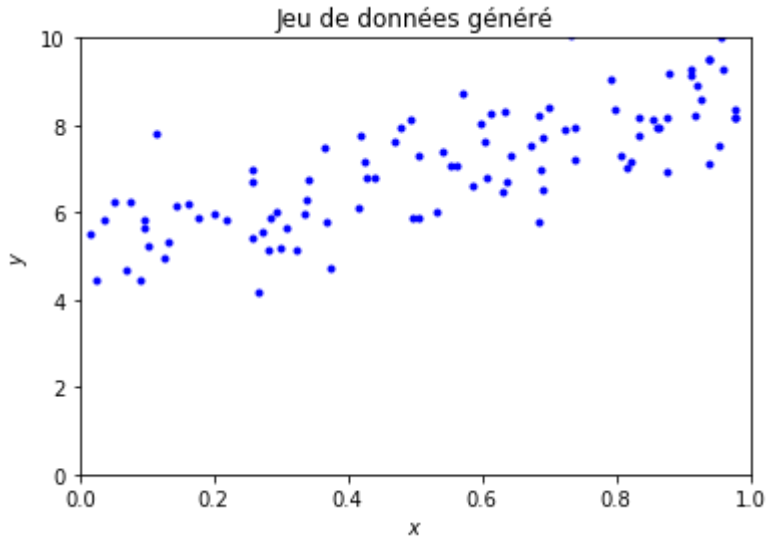
Dans la suite de ce notebook, nous nous intéressons plus particulièrement à l'apprentissage supervisé.

Cas de la régression linéaire

De manière à mieux comprendre le concept de descente de gradient, considérons un jeu de données aléatoire pour lequel nous savons que la fonction linéaire pour le créer initialement est $y = 4x + 5$.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

#definition de la fonction y=f(4X + 5)
X = np.random.rand(100,1)
y = 5 + 4*X+np.random.randn(100,1)
#X = 2 * np.random.rand(100,1)
#y = 4 +3 * X+np.random.randn(100,1)
#Visualisation du jeu de données
plt.plot(X,y,'b.')
plt.title('Jeu de données généré')
plt.xlabel("$x$", fontsize=10)
plt.ylabel("$y$", fontsize=10)
#plt.plot(X,4*X+ 5,'r-')
plt.axis([0,1,0,10])
plt.show()
```



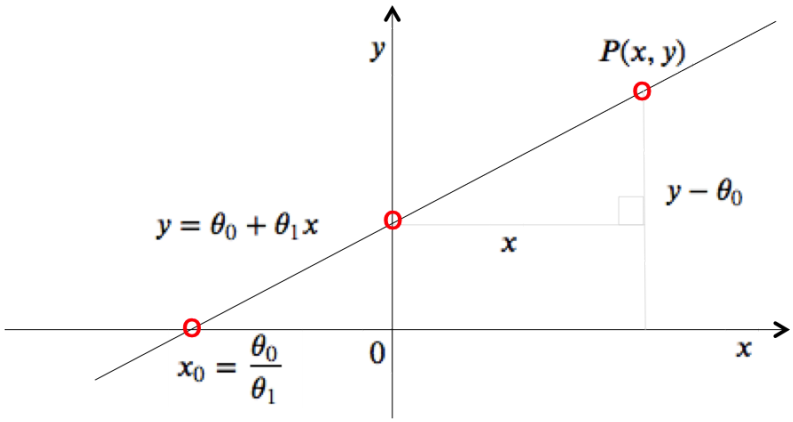
Actuellement nous connaissons plus ou moins l'équation de la droite (des valeurs aléatoires ont été ajoutées) mais l'objectif d'un algorithme d'apprentissage est, à partir d'un ensemble connu d'apprentissage de pouvoir

prédire les valeurs de nouvelles entrées. En d'autres termes, nous souhaitons construire un modèle : une hypothèse qui peut être utilisé pour estimer y en fonction de x.

Nous savons que notre hypothèse à trouver est de la forme :

$$h_\theta(x) = \theta_0 + \theta_1 x$$

où θ_0 correspond au point d'intersection avec l'axe des y et θ_1 à la pente de la droite.



Considérons la fonction predict suivante et initialisons $\theta_0 = 2$ et $\theta_1 = 1$:

In [2]:

```
def predict(X, theta0, theta1):
    return theta0 + theta1*X

print ('Valeur de X[5] : ',X[5],' - valeur réelle de y[5] : ',y[5])
theta0=2
theta1=1
print ('valeur prédite : ',predict(X[5], theta0, theta1))
```

```
Valeur de X[5] :  [0.54168813]  - valeur réelle de y[5] :  [7.3796979
2]
valeur prédite :  [2.54168813]
```

Nous pouvons constater que la valeur prédite est loin de la valeur réelle.

L'objectif est donc de déterminer les valeurs de θ_0 et θ_1, de sorte que l'erreur soit la plus faible possible.

La fonction de coût (*Cost Function*) ou de perte (*Loss Function*) permet d'évaluer les erreurs de prédiction. La fonction de coût calcule l'erreur pour une seule occurrence d'apprentissage, tandis que la fonction de perte est la moyenne des fonctions de coûts pour tous les exemples d'apprentissage. Par abus de langage, les deux sont utilisés.

Si nous avons m données dans l'ensemble d'apprentissage, nous souhaitons minimiser l'erreur. Une fonction de coût souvent utilisée est l'erreur quadratique moyenne (*Mean square error (MSE)* *https://en.wikipedia.org/wiki/Mean_squared_error (https://en.wikipedia.org/wiki/Mean_squared_error)*) :

$$J = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

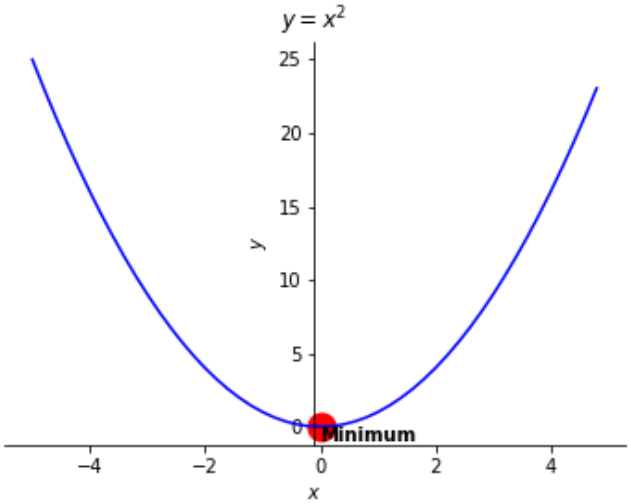
Remarque : les différences quadratiques sont préférées aux différences absolues car elles permettent de trouver une ligne de régression en calculant la dérivée première. En outre le fait de mettre au carré augmente la distance d'erreur ce qui met en évidence les mauvaises prédictions.

Dans notre cas, J dépend de θ_0 et θ_1 donc :

$$J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$
$$= \frac{1}{m} \sum_{i=1}^{m} ((\theta_0 + \theta_1 x^{(i)}) - y^{(i)})^2$$

Comment minimiser la fonction de coût ?

En fait si nous regardons J elle est de type : $y = x^2$. Nous pouvons la dessiner et le minimum est situé à la position (0,0) (point rouge sur le dessin).



Pour minimiser la fonction il faut trouver la valeur de x qui produit la valeur de y la plus basse. Pour cela on utilise l'algorithme de descente de gradient.

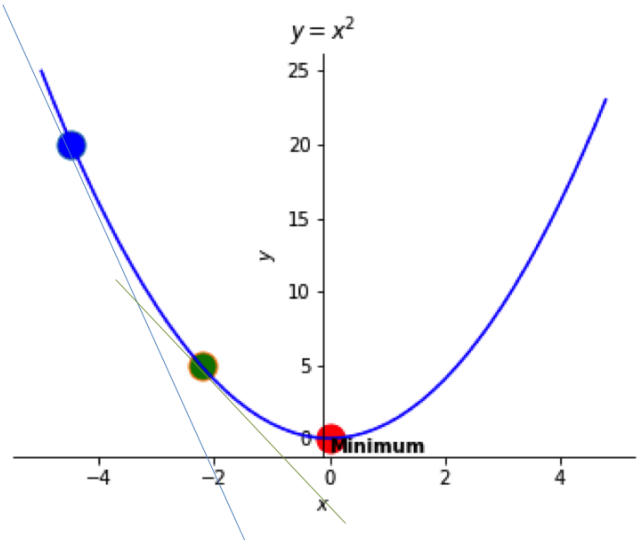
Algorithme de Descente de Gradient (*Gradient Descent*)

La descente de gradient est un algorithme, très utilisé en apprentissage, itératif d'optimisation pour trouver le minimum d'une fonction, i.e. dans notre cas la fonction de perte.

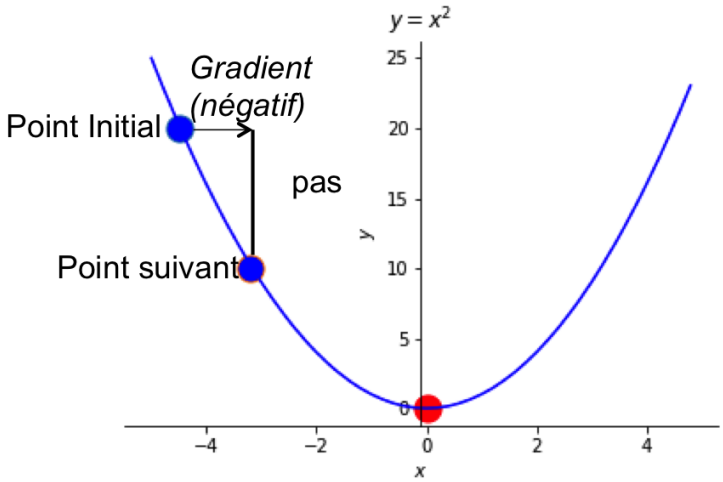
Un exemple classique pour expliquer l'approche est la suivante. Considéreons une personne sur une montage avec les yeux bandés qui souhaite descendre dans la plaine. La seule aide dont dispose la personne est l'altitude qu'elle peut relever. La personne peut commencer à descendre dans une direction aléatoire puis regarer l'altitude. Si elle est supérieure à l'altitude initiale, elle sait qu'elle est dans la mauvaise direction. Elle peut alors changer de direction et répéter le même processus. De la même manière elle peut marcher plus ou moins vite en fonction de la pente. Si la pente est raide elle descend vite et autrement plus lentement. Au bout de beaucoup d'itérations elle arrive au creux de la vallée.

Dans notre cas, les gradients indiquent la direction, l'altitude est indiquée par $J(\theta_0, \theta_1)$ et les pas sont indiqués par le taux d'apprentissage (*Learning rate*).

Pour un point donné, la valeur de du gradient (la dérivée) correspond à l'inclinaison de la pente en ce point. La figure suivante illustre la tangente de deux points. Nous pouvons constater que la pente pour le point vert est plus faible que pour le point bleu. Cela indique qu'il y a moins de pas à faire pour atteindre le minimum. Ainsi si le gradient est élevé, c'est que la pente est très pentue, s'il est faible, la pente est faible. S'il est égal à zéro il s'agit d'un minimum. Enfin s'il est négatif cela veut dire que la pente descend (dans ce cas il faut poursuivre dans cette direction) et autrement elle monte (il faut partir dans l'autre direction).



Le point bleu, dans la figure suivante, est situé à la position $x = -4.5$. Nous savons que la fonction de coût est : $y = f(x^2)$, i.e. $y = f(-4.5^2) = 20.25$. La dérivée de f est : $f' = 2x$ donc nous avons : $f'(2 \times -4.5) = -9$. Le -9 indique qu'il s'agit d'une pente forte et le fait que le nombre soit négatif indique qu'il faut continuer à se déplacer dans cette direction.



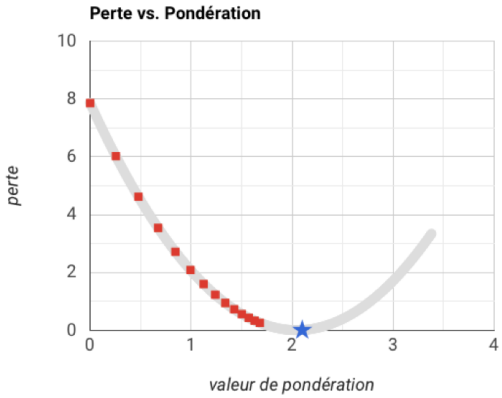
Pour résumer le gradient, i.e. la pente de la fonction de coût pour un point donné, représente la direction et le taux de variation de la fonction de coût. Suivre le gradient négatif de la fonction permet donc de la minimiser le plus rapidement possible.

Le pas pour atteindre le minimum s'appelle le taux d'apprentissage (*learning rate*) et est souvent noté η. Plus ce pas est petit plus il faudra du temps pour atteindre le point le plus bas. D'un autre côté un pas trop élevé risque de nous faire dépasser le minimum. Généralement les valeurs de η sont comprises entre 0.0001 et 1.

Le site suivant propose une application pour tester et visualiser les impacts d'un choix d'η : https://developers.google.com/machine-learning/crash-course/fitter/graph (https://developers.google.com/machine-learning/crash-course/fitter/graph).

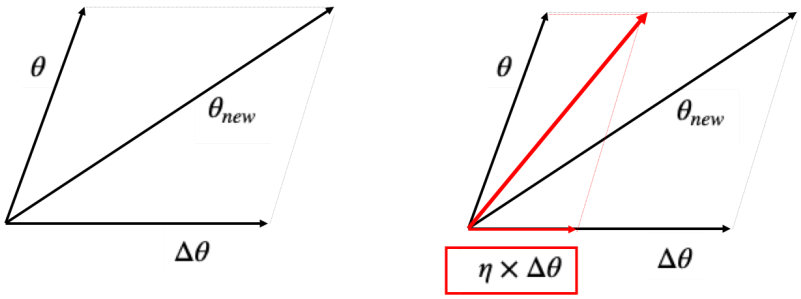
Testez différents taux d'apprentissage et voyez comment ils affectent le nombre d'étapes nécessaires pour atteindre le point minimal de la courbe de coût. Essayez d'effectuer les exercices proposés sous le graphe.

Définir le taux d'apprentissage :	0,2	
Exécuter une seule étape :	ÉTAPE	13
Réinitialiser le graphe :	RÉINITIALISER	



Il est très difficile de choisir la bonne valeur et de nombreuses expériences sont souvent nécessaires. Nous pouvons couvrir plus de domaines avec des pas plus importants (taux d'apprentissage élevé), mais nous courons le risque de dépasser les minima. D'un autre côté, les petits pas (petit taux d'apprentissage) prendront beaucoup de temps pour atteindre le point le plus bas.

Concrétement voici à quoi cela correspond. Si le vecteur des paramètres est $\theta = [\theta_0, \theta_1]$ et que les changements désirés sont $\Delta\theta = [\Delta\theta_0, \Delta\theta_1]$ alors la nouvelle position correspond à $\theta_{new} = \theta_{old} + \eta \times \Delta\theta_{old}$.



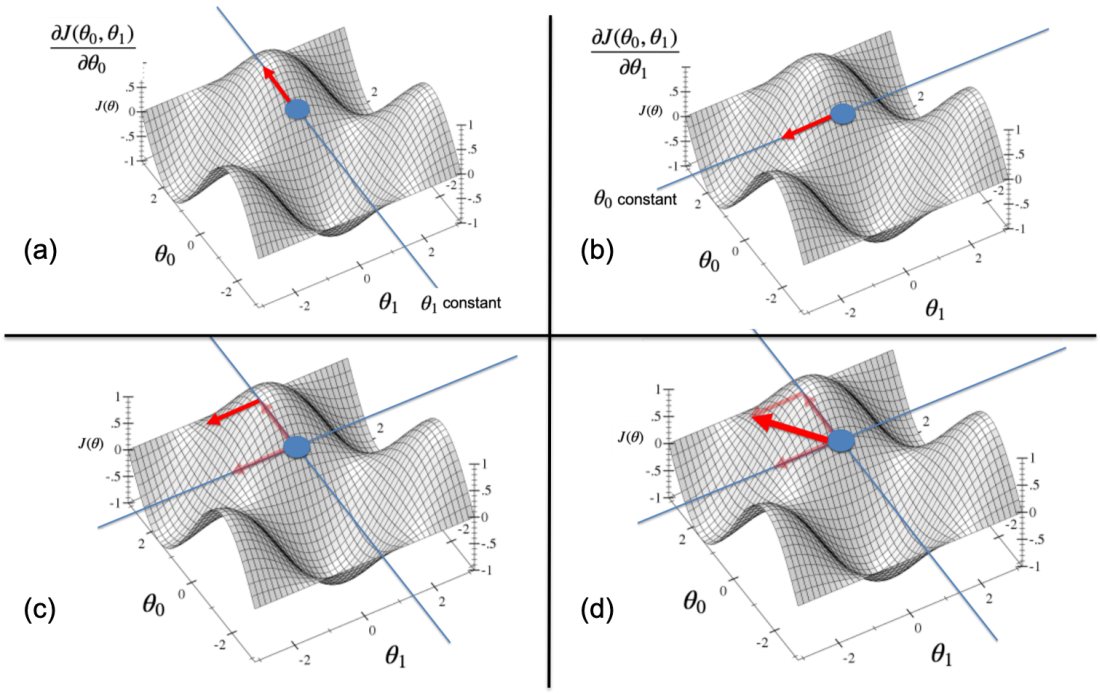
Petit rappel sur les dérivées partielles

Nous savons que notre fonction de perte est :

$$J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}((\theta_0 + \theta_1 x^{(i)}) - y^{(i)})^2$$

Le gradient est le vecteur des dérivées partielles selon chaque dimension. Donc pour calculer la descente de gradient il est nécessaire de calculer les dérivées partielles par rapport à θ_0 et θ_1 pour savoir dans quelle direction aller.

Une dérivée partielle d'une fonction de plusieurs variables, ici $J(\theta)$, est la dérivée de cette fonction selon une variable, les autres étant considérées constantes. Considérons le point positionné sur la figure suivante. Dans (a), nous fixons θ_1 comme constante et nous pouvons donc calculer la dérivée partielle $\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_0}$. Dans notre exemple, le point est positionné en descente le sens du vecteur est donc vers les valeurs négatives. Dans (b), en fixant θ_0 comme constante, nous pouvons calculer la dérivée partielle $\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_1}$ et là aussi comme le point est sur une pente, le sens du vecteur est orienté vers des valeurs négatives. Dans (c), il suffit de faire la somme des deux vecteurs précédents et le vecteur résultat est indiqué dans (d).



Ainsi, à partir de :

$$J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} ((\theta_0 + \theta_1 x^{(i)}) - y^{(i)})^2$$

les dérivées partielles par rapport à θ_0 et θ_1 :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = \frac{-2}{m} \sum_{i=1}^{m} ((\theta_0 + \theta_1 x^{(i)}) - y^{(i)}) x^{(i)}$$

et

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = \frac{-2}{m} \sum_{i=1}^{m} ((\theta_0 + \theta_1 x^{(i)}) - y^{(i)})$$

L'algorithme de descente de gradient consiste donc à déplacer θ_0 et θ_1 jusquà atteindre un minimum pour la fonction de coût :

Initialisation de θ_0 *et* θ_1 *avec des nombres aléatoires*
Répéter jusqu'à convergence{

$$\theta_0 = \theta_0 - \eta \frac{-2}{m} \sum_{i=1}^{m} ((\theta_0 + \theta_1 x^{(i)}) - y^{(i)})$$
$$\theta_1 = \theta_1 - \eta \frac{-2}{m} \sum_{i=1}^{m} ((\theta_0 + \theta_1 x^{(i)}) - y^{(i)}) x^{(i)}$$

}

Cela consiste donc à boucler suffisamment longtemps, i.e. spécification du nombre d'epochs par exemple, sur les données pour mettre à jour θ_0 et θ_1.

Implémentation simple

On redéfinie la fonction predict

In [3]:

```
def predict(X, theta0, theta1):
    return theta0 + theta1*X
```

Définition de la fonction de coût (ici MSE)

In [4]:

```
def cost_function(X, y, theta0, theta1):
    m = len(y)
    total_error = 0.0
    for i in range(m):
        total_error += ((theta0 + theta1*X[i])-y[i] )**2
    return total_error / float(m)
```

In [5]:

```
def update_weights(X, y, theta0, theta1, eta):
    theta0_deriv = 0
    theta1_deriv = 0
    m = len(y)

    for i in range(m):
        # Calcul des dérivées partielles
        # -2((theta0 + theta1x) -y)
        theta0_deriv -= -2*((theta0 + theta1*X[i])-y[i])

        # -2x((theta0 + theta1x) - y)
        theta1_deriv -= -2*X[i] * ((theta0 + theta1*X[i])-y[i])

    # soustraction ds dérivées et multiplication par eta

    theta0 -= (theta0_deriv / float(m)) * eta
    theta1 -= (theta1_deriv / float(m)) * eta

    return theta0, theta1

```

In [6]:

```
def gradient_descent(X, y, theta0, theta1, eta=0.01, epochs=1000):

    #variables pour affichage à la sortie
    cost_history = []
    theta_history = np.zeros((epochs,2),dtype=float)

    for i in range(epochs):

        theta0,theta1 = update_weights(X, y, theta0, theta1, eta)
        # stockage des valeurs pour l'affichage
        theta_history[i][0]=theta0
        theta_history[i][1]=theta1

        cost = cost_function(X, y, theta0, theta1)
        cost_history.append(cost)

    return theta0, theta1, cost_history, theta_history
```

Le code suivant execute l'appel à la fonction

In [7]:

```
eta =0.1
epochs = 1000
X = np.random.rand(100,1)
y = 5 + 4*X+np.random.randn(100,1)
#initialisation aléatoire de theta0 et de theta1
theta0=np.random.randn()
theta1=np.random.randn()

theta0,theta1,cost_history,theta_history=gradient_descent(X, y,theta0, theta1

print ('theta0 = %0.2f'%theta0,' theta1 = %.2f'%theta1,' cost(%0.2f)'%cost_hi
```

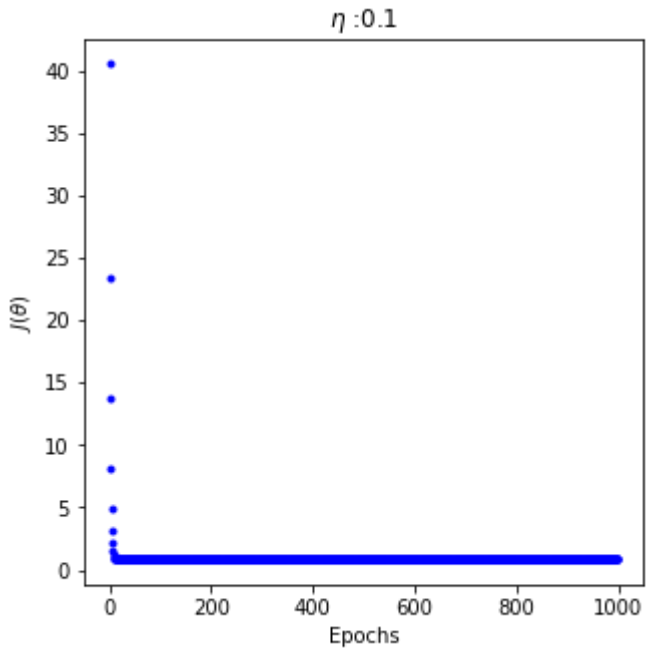
```
theta0 = 5.20  theta1 = 3.63  cost(0.83)
```

In [8]:

```
def plot_history (eta,epochs,cost_history):
    fig,ax = plt.subplots(figsize=(5,5))
    ax.set_ylabel(r'$J(\theta)$')
    ax.set_xlabel('Epochs')
    ax.set_title(r"$\eta$ :{}".format(eta))
    ax.plot(range(epochs),cost_history,'b.')
```

In [9]:

```
plot_history(eta,epochs,cost_history)
```



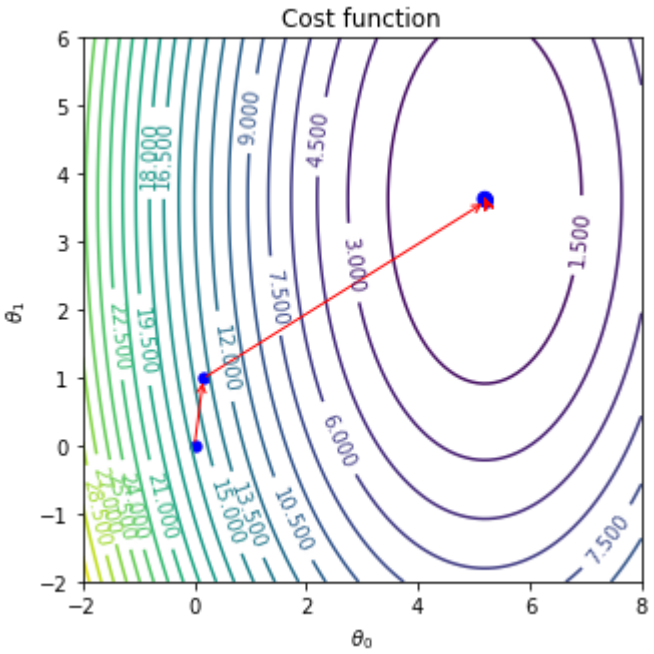
In [10]:

```
def plot_2D(theta,theta_history, epochs, pas=25):
    """
    Affichage en 2D de la cost_function
    Paramètres :
    Entrée :
    ----------------
    theta : array theta0 = theta[0], theta1 = theta[1]
    theta_history array : contient l'historique des thetas lors de l'apprenti
    epochs int : epochs utilisés lors de l'apprentissage
    pas int : indique dans l'historique combien de points doivent être plotés
    """
    theta0_reel = theta[0]
    theta1_reel = theta[1]
    fig, ax = plt.subplots(figsize=(5,5))
    # x et y à tracer
    m=10
    x = np.linspace(-1,1,m)
    y = theta0_reel + theta1_reel * x

    #fonction plus simple qui calcule le coût pour les valeurs de l'affichage
    def cost_func(theta0, theta1):
        return np.average((y-(theta0 + theta1*x))**2, axis=2)/2

    # Construction de la grille
    # y_theta0=int(theta0_true+2) pour centrer le dessin idem pour y_theta1
    # nb_points = nombre de points de la courbe
    y_theta0=int(theta0_reel+3)
    y_theta1=int(theta1_reel+3)
    nb_points=100
    theta0_grid = np.linspace(-2,y_theta0,nb_points)
    theta1_grid = np.linspace(-2,y_theta1,nb_points)

    #recuperation de la cost function pour les differentes valeurs de theta0
    #définie par les theta0_grid et theta1_grid

    J_grid = cost_func(theta0_grid[np.newaxis,:,np.newaxis],
                       theta1_grid[:,np.newaxis,np.newaxis])

    # Mise en place des courbes de niveau et du contour
    X, Y = np.meshgrid(theta0_grid, theta1_grid)
    contours = ax.contour(X, Y, J_grid, 30)
    ax.clabel(contours)
    ax.scatter([theta0_reel]*2,[theta1_reel]*2,s=[50,10], color=['b','r'])

    #selection des points à afficher ici tous les pas dans l'historique
    #on commence par 00 pour le sens des flêches
    thetas = [np.array((0,0))]
    N=1
    for i in range(epochs-1):
        if i % pas ==0:
            N+=1
            this_theta = np.empty((2,))
            this_theta=theta_history[i]
            thetas.append(this_theta)

    # fleches
    for i in range(1,N):
        ax.annotate('', xy=thetas[i], xytext=thetas[i-1],
                    arrowprops={'arrowstyle': '->', 'color': 'r', 'lw': 1},
```

```
                    va='center', ha='center')

    ax.scatter(*zip(*thetas), c='b', s=40, lw=0)

    ax.set_xlabel(r'$\theta_0$')
    ax.set_ylabel(r'$\theta_1$')
    ax.set_title('Cost function')

    plt.show()
```

In [11]:

```
#transformation en tableau pour theta0 et theta1
theta=[theta0,theta1]
plot_2D(theta,theta_history,epochs,pas=100)
```

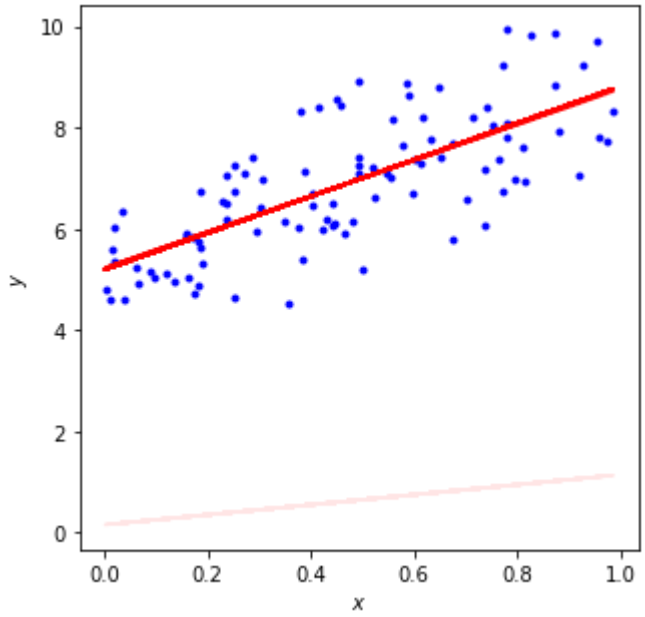


In [12]:

```
def plot_lines (eta,epochs,theta_history):
    tr =0.1
    fig,ax = plt.subplots(figsize=(5,5))
    ax.set_xlabel("$x$", fontsize=10)
    ax.set_ylabel("$y$", fontsize=10)
    ax.plot(X,y,'b.')
    for i in range (epochs):
        pred=predict(X, theta_history[i][0], theta_history[i][1])
        if ((i % 25 == 0) ):
            _ = ax.plot(X,pred,'r-',alpha=tr)
            if tr < 0.8:
                tr = tr+0.2

```

In [13]:

```
1  plot_lines(eta,epochs,theta_history)
```

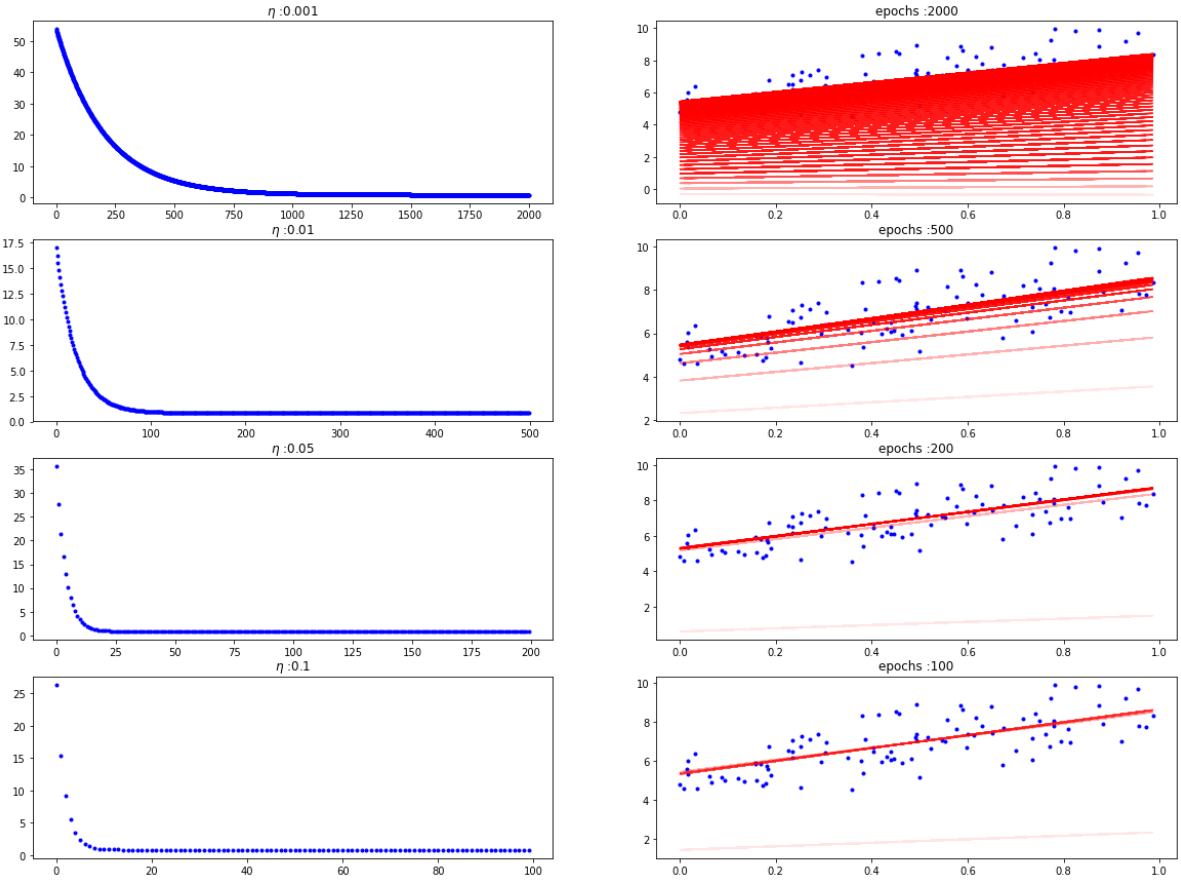


In [14]:

```
1  def plot_all (eta,epochs,theta_history,ax,ax1):
2      tr =0.1
3      #fig,ax = plt.subplots(figsize=(5,5))
4      ax1.plot(X,y,'b.')
5      for i in range (epochs):
6          pred=predict(X, theta_history[i][0], theta_history[i][1])
7          if ((i % 25 == 0) ):
8              ax1.plot(X,pred,'r-',alpha=tr)
9              if tr < 0.8:
10                 tr = tr+0.2
11     ax.plot(range(epochs),cost_history,'b.')
```

In [15]:

```
fig = plt.figure(figsize=(20,15))
fig.subplots_adjust(hspace=0.2, wspace=0.2)
epochs_eta =[(2000,0.001),(500,0.01),(200,0.05),(100,0.1)]

count =0
for epochs, eta in epochs_eta:
    count += 1
    ax = fig.add_subplot(4,2,count)
    ax.set_title(r"$\eta$ :{}".format(eta))
    count += 1

    ax1 = fig.add_subplot(4,2,count)
    ax1.set_title("epochs :{}".format(epochs))


    theta0=np.random.randn()
    theta1=np.random.randn()

    theta0,theta1,cost_history,theta_history=gradient_descent(X, y,theta0, th
    plot_all(eta,epochs,theta_history,ax,ax1)


```



Generalisation et regression linéaire multiple

Dans la section précédente nous avons vu les principes d'une regression linéaire simple. Dans cette section, nous étendons le principe à une régression linéaire multiple (*Multivariate Linear Regression*), i.e. ayant plusieurs paramètres. De plus nous présentons une manière plus efficace d'implémenter la descente de gradient à l'aide de matrice.

Dans une regression linéaire simple on cherche à modéliser la relation entre deux variables quantitatives continues. Comme nous l'avons vu précédemment un modèle de régression linéaire simple est de la forme $h_\theta(x) = \theta_0 + \theta_1 x$ où :

- $h_\theta(x)$ est le modèle/l'hypothèse pour la variable y à prédire;
- x est la variable prédictive;
- θ_0 et θ_1 les paramètres à estimer.

Plus formellement le modèle est plutôt de la forme $h_\theta(x) = \theta_0 + \theta_1 x + \epsilon$ où ϵ correspond au terme d'erreur aléatoire du modèle. Dans notre cas nous le négligeons pour le moment.

Dans une regression linéaire multiple on cherche à modéliser la relation entre plus de deux variables quantitatives continues. Un modèle de régression linéaire multiple est donc de la forme :

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n$$

Par exemple pour connaître la concentration d'ozone dans l'air (y), il faut tenir compte de la température (x_1) et de la vitesse du vent (x_2).

Représentation sous la forme de matrice

Ce modèle peut tout à fait être représenté sous la forme de matrices (cette implémentation est beaucoup plus efficace que la précédente) et d'utiliser des produits scalaires entre matrices.

Pour cela il faut au préalable ajouter une colonne de 1 à la matrice des variable prédictives (le nombre de valeurs prédictives est donc incrémenté de 1). Ce 1 permet de pouvoir traiter en même temps que les autres le θ_0, i.e. $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n$.
Avec m le nombre d'enregistrements et n le nombre de variables prédictives nous avons :

$$\begin{aligned} h_\theta(x) &= \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n \\ &= \theta^T X \end{aligned}$$

où θ^T correspond à la transposée de la matrice θ.

$$X = \begin{bmatrix} 1 & x_1^1 & x_2^1 & \ldots & x_n^1 \\ 1 & x_1^2 & x_2^2 & \ldots & x_n^2 \\ 1 & x_1^3 & x_2^3 & \ldots & x_n^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^m & x_2^m & \ldots & x_n^m \end{bmatrix} (taille : m \times n + 1)$$

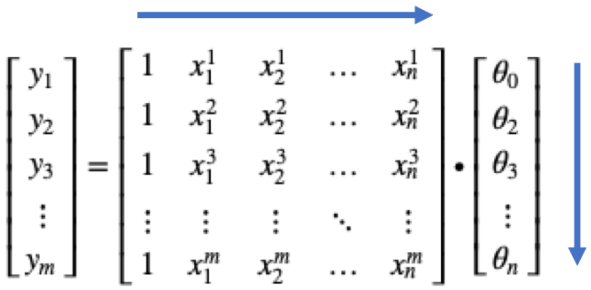
pour les θ :

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix} (taille : n+1 \times 1)$$

Et pour y, nous avons :

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} (taille : m \times 1)$$

La prédiction peut donc, par exemple, facilement se faire par un simple produit scalaire :



$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & x_1^1 & x_2^1 & \dots & x_n^1 \\ 1 & x_1^2 & x_2^2 & \dots & x_n^2 \\ 1 & x_1^3 & x_2^3 & \dots & x_n^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^m & x_2^m & \dots & x_n^m \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & x_1^1 & x_2^1 & \dots & x_n^1 \\ 1 & x_1^2 & x_2^2 & \dots & x_n^2 \\ 1 & x_1^3 & x_2^3 & \dots & x_n^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^m & x_2^m & \dots & x_n^m \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} \theta_0 & \theta_1 x_1^1 & x_2^1 & \dots & \theta_n x_n^1 \\ \theta_0 & \theta_1 x_1^2 & x_2^2 & \dots & \theta_n x_n^2 \\ \theta_0 & \theta_1 x_1^3 & x_2^3 & \dots & \theta_n x_n^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_0 & \theta_1 x_1^m & x_2^m & \dots & \theta_n x_n^m \end{bmatrix}$$

La dérivée partielle par rapport à θ_j est maintenant :

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{-2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

L'algorithme de descente de gradient peut donc se généraliser en :

Initialisation de θ_j avec des nombres aléatoires

Répéter jusqu'à convergence{

$$\theta_j = \theta_j - \eta \frac{-2}{m} \sum_{i=1}^{m} (h_\theta x^{(i)}) - y^{(i)}) x_j^{(i)}$$

en mettant simultanément θ_j à jour pour $j = (1, \dots, n)$

}

ce qui donne :

$$\theta_0 = \theta_0 - \eta \frac{-2}{m} \sum_{i=1}^{m} (h_\theta x^{(i)}) - y^{(i)}) x_0^{(i)} \; avec x_0^{(i)} = 1$$

$$\theta_1 = \theta_1 - \eta \frac{-2}{m} \sum_{i=1}^{m} (h_\theta x^{(i)}) - y^{(i)}) x_1^{(i)}$$

$$\theta_2 = \theta_2 - \eta \frac{-2}{m} \sum_{i=1}^{m} (h_\theta x^{(i)}) - y^{(i)}) x_2^{(i)}$$

$$\vdots$$

ou tout simplement :

Initialisation de θ_j *avec des nombres aléatoires*

Répéter jusqu'à convergence{

$$\theta = \theta - \eta \frac{-2}{m} (X \bullet \theta - y) X^T$$

}

In [16]:

```
import numpy as np
matrice=np.random.randint(10, size=(2, 3))
print ("une matrice\n",matrice)
print ("la transposée de la matrice\n",matrice.T)
```

```
une matrice
 [[0 4 1]
 [2 1 8]]
la transposée de la matrice
 [[0 2]
 [4 1]
 [1 8]]
```

Implémentation

Nous définissons d'abord la fonction predict

In [17]:

```
def predict(X, theta):
    """
    Prediction de la valeur de X
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives)
    theta array : theta[0], theta1 = theta[1]
    ----------------
    Output :
    array correspondant au produit matricielle de X avec les thetas
    """
    return X.dot(theta)
```

La fonction de coût se fait simplement à l'aide d'un produit scalaire (fonction dot en python)

In [18]:

```
def cost_function(X,y,theta):
    """
    Calcul de la fonction de coût
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives)
    y array : données d'apprentissage (valeur à prédire)
    theta array : theta[0], theta1 = theta[1]
    ----------------
    Output :
    MSE
    """

    m=len(y)
    prediction=X.dot(theta)
    square_err=(prediction - y)**2
    return 1/(2*m) * np.sum(square_err)
```

La descente de gradient (ne pas oublier la transposé de X)

In [19]:

```
def gradient_descent(X,y,theta,eta=0.01, epochs=1000):
    """
    Descente de gradient simple (vanilla descent gradient)
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives)
    y array : données d'apprentissage (valeur à prédire)
    theta array : theta[0], theta1 = theta[1]
    eta float : le learning rate
    epochs int : nombre de répétitions
    ----------------
    Output :
    theta array : theta[0], theta1 = theta[1]
    cost_history array : liste du coût de theta à chaque itération
    theta_history array : liste des thetas à chaque itération
    """

    m=len(y) # attention prendre y et non x car il y a un entier en plus
    cost_history=[]
    theta_history = np.zeros((epochs,2),dtype=float)
    #Attention theta_history est pour afficher en 2D donc vecteur à 2 dimensi

    for i in range(epochs):

        #Le code ci-dessous peut s'écrire en une ligne
        #theta = theta -(1/(2*m))*eta*( X.T.dot((prediction - y)))
        prediction = predict(X,theta)
        error = X.T.dot((prediction - y))
        descent=eta * (1/(2*m)) * error
        theta-=descent

        #pour affichage
        theta_history[i][0]=theta[0]
        theta_history[i][1]=theta[1]
        cost_history.append(cost_function(X,y,theta))

    return theta, cost_history, theta_history
```

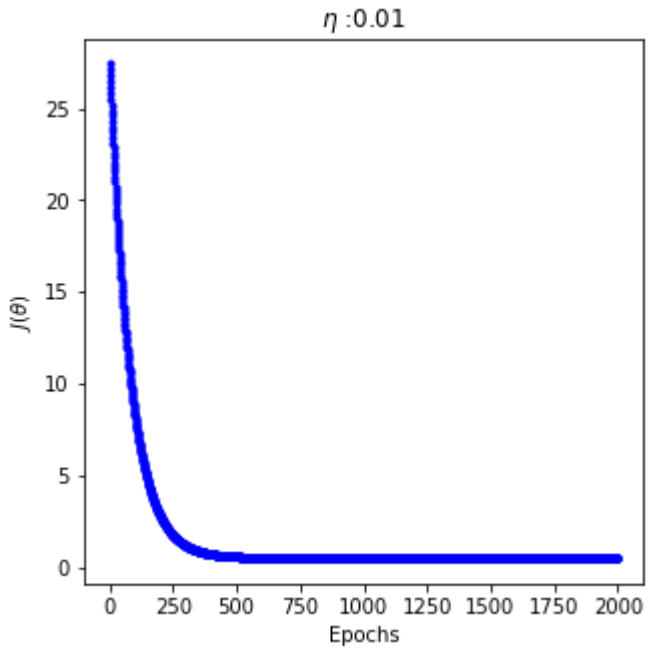
In [20]:

```
eta =0.01
epochs = 2000
#initialisation aléatoire de theta0 et de theta1
# dans cette exemple on considère une fonction du type
# theta0 + theta1x_1 pour l'affichage en 2D elle est bien sûr généralisable
theta = np.random.randn(2,1)
X = np.random.rand(100,1)
y = 5 + 4*X+np.random.randn(100,1)

#X_0 ajoute un 1 au début de X pour avoir : theta0x_0 + theta1x_1 ...
X_0 = np.c_[np.ones((X.shape[0])), X]


#appel de la fonction attention c'est avec X_0
theta,cost_history,theta_history = gradient_descent(X_0,y,theta,eta,epochs)
print ('theta0 = %0.2f'%theta[0],' theta1 = %.2f'%theta[1],' cost(%0.2f)'%cos

```

```
theta0 = 5.31  theta1 = 3.76  cost(0.45)
```

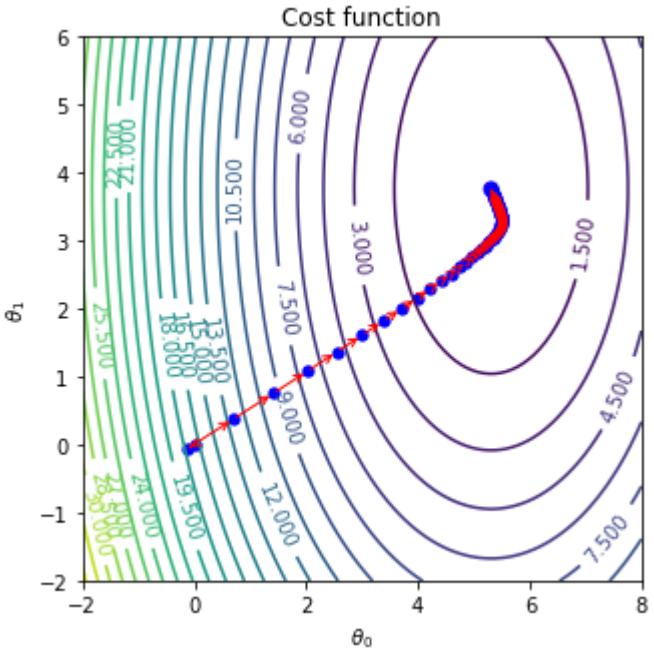
In [21]:

```
plot_history(eta,epochs,cost_history)
```



In [22]:

```
1  plot_2D(theta,theta_history,epochs,pas=25)
```



Optimisation : différentes stratégies

L'algorithme de descente de gradient que nous avons vu précédemment est également appelée descente de gradient par lots (*Batch Gradient Descent*) et est le plus simple : il calcule l'erreur pour chaque exemple dans l'ensemble d'apprentissage. Après avoir évalué tous les enregistrements de l'ensemble d'apprentissage (1 epoch), il met à jour les paramètres du modèle.

Il présente l'avantage d'être assez efficace en termes de calcul et de produire un gradient d'erreur stable et une convergence stable. L'un des inconvénients est que ce gradient d'erreur stable peut parfois conduire à un état de convergence qui n'est pas optimal. L'autre inconvénient (important) est aussi qu'il nécessite que l'ensembe du jeu de données d'apprentissage réside en mémoire et ce n'est pas possible pour des jeux de données volumineux.

Pour cela différentes stratégies ont été proposées.

Stochastic gradient descent

La descente de gradient stochastique (*Stochastic Gradient Descent*) met à jour les paramètres en fonction du gradient de l'erreur par rapport à un seul exemple d'apprentissage (batch_size=1) tiré aléatoirement (d'où le terme stochastique) même s'il est tout à fait possible de considérer un sous ensemble d'exemples. Ceci est différent de Batch Gradient Descent, qui met à jour les paramètres une fois que tous les exemples d'apprentissage ont été évalués.

Il ne consomme pas beaucoup de mémoire et est souvent plus rapide car un seul échantillon est traité à la fois. Ainsi pour des grands ensembles de données, la convergence peut être plus rapide car les mises à jour des paramètres sont plus fréquentes. L'avantage de ces mises à jour fréquentes est que les étapes effectuées vers les minima de la fonction de perte ont des oscillations qui peuvent aider à sortir des minimums locaux.

En fonction des données, elle est généralement plus rapide que la descente de gradient par lots. Cependant il s'avère qu'à cause des mises à jour fréquentes la convergence prenne plus de temps (descente dans des directions différentes). Les mises à jour fréquentes sont coûteuses en termes de calcul car elles utilisent toutes les ressources pour traiter un échantillon d'apprentissage à la fois. Enfin, quand il y a un seul exemple, nous perdons tous les avantages de l'utilisation des matrices.

Implémentation du Stochastique Gradient Descent

Dans la boucle principale (epoch), lors du parcours du jeu d'apprentissage, un nombre aléatoire d'exemples sont choisis et le calcul se fait sur eux.

In [23]:

```
def stocashtic_gradient_descent(X,y,theta,learning_rate=0.01,epochs=1000):
    """
    Descente de gradient stochastique
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives) avec un 1 ajout
    y array : données d'apprentissage (valeur à prédire)
    theta array : theta[0], theta1 = theta[1]
    eta float : le learning rate
    epochs int : nombre de répétitions
    ----------------
    Output :
    theta array : theta[0], theta1 = theta[1]
    cost_history array : liste du coût de theta à chaque itération
    theta_history array : liste des thetas à chaque itération
    """

    m=len(y)
    cost_history=[]
    theta_history = np.zeros((epochs,2),dtype=float)
    for i in range(epochs):
        cost = 0.0
        for j in range(m):
            #tirage d'un nombre aléatoire entre 0 et m
            rand_ind = np.random.randint(0,m)
            #recuperation du X et y correspondants au nombre aléatoire
            X_j = X[rand_ind,:].reshape(1,X.shape[1])
            y_j = y[rand_ind].reshape(1,1)
            prediction = predict(X_j,theta)
            error = X_j.T.dot((prediction - y_j))
            descent=eta * (1/(2*m)) * error
            theta-=descent
        cost+=cost_function(X_j,y_j,theta)

        theta_history[i][0]=theta[0]
        theta_history[i][1]=theta[1]
        cost_history.append(cost)
    return theta, cost_history, theta_history
```

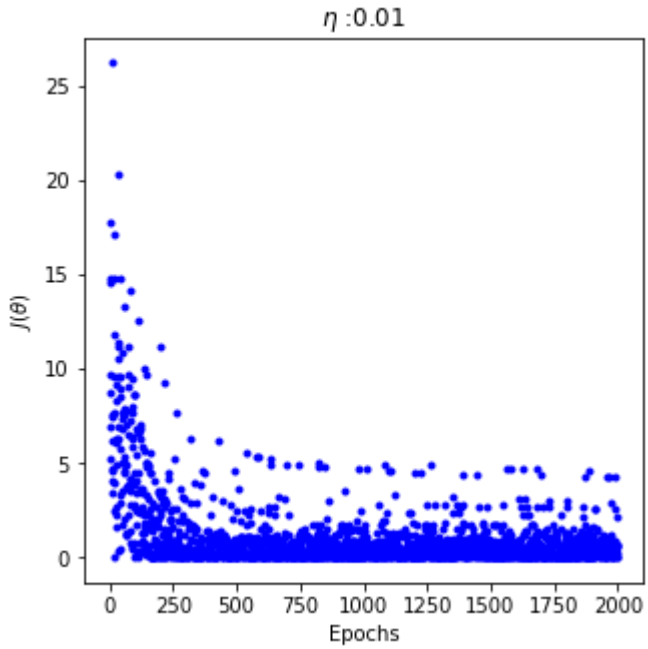
In [24]:

```
eta =0.01
epochs = 2000
#initialisation aléatoire de theta0 et de theta1
theta = np.random.randn(2,1)
X = np.random.rand(100,1)
y = 5 + 4*X+np.random.randn(100,1)

#X_0 ajoute un 1 au début de X pour avoir : theta0x_0 + theta1x_1 ...
X_0 = np.c_[np.ones((X.shape[0])), X]

theta,history_stoc,theta_history = stocashtic_gradient_descent(X_0,y,theta,et
print ('theta0 = %0.2f'%theta[0],' theta1 = %.2f'%theta[1],' cost(%0.2f)'%his
```

```
theta0 = 5.38  theta1 = 2.96  cost(0.63)
```

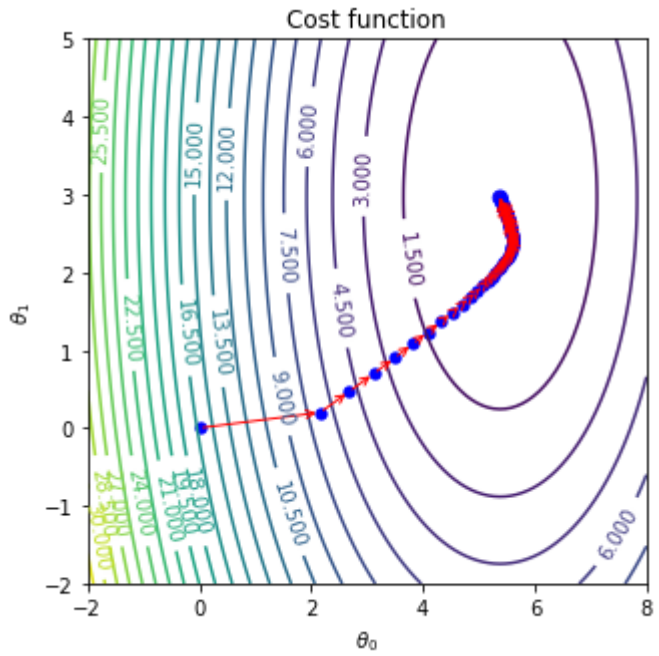
In [25]:

```
1 plot_history(eta,epochs,history_stoc)
```



In [26]:

```
1 plot_2D(theta,theta_history,epochs,pas=25)
```



Mini-batch gradient descent

L'algorithme de descente de gradient par mini lots (*mini-batch gradient descent*) est souvent privilégié, car il associe une descente en gradient stochastique et une descente en gradient par lots. Il sépare simplement le jeu d'apprentissage en petits lots et effectue une mise à jour pour chacun de ces lots. Cela crée donc un équilibre entre l'efficacité de la descente de gradient par lots et la robustesse de la descente de gradient stochastique. Il est très utilisé en apprentissage profond (deep learning).

Implémentation du Mini-batch gradient descent

Le jeu d'apprentissage est découpé en petit batch en tirant des valeurs d'exemple au hasard. Ensuite chaque batch est examiné par la descente de gradient.

In [27]:

```
def next_batch(X, y, batchsize):
    """
    Création des batches
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives) avec un 1 ajout
    y array : données d'apprentissage (valeur à prédire)
    batchsize int : taille du batch
    """
    m=len(y)
    # permutation aléatoire de X et y pour faire des batchs avec des valeurs
    indices = np.random.permutation(m)
    X = X[indices]
    y = y[indices]
    for i in np.arange(0, X.shape[0], batchsize):
        # creation des batchs de taille batchsize
        yield (X[i:i + batchsize], y[i:i + batchsize])
```

In [28]:

```
def minibatch_gradient_descent(X,y,theta,learning_rate=0.01,epochs=10,batchsi
    """
    Descente de gradient par mini batch
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives) avec un 1 ajout
    y array : données d'apprentissage (valeur à prédire)
    theta array : theta[0], theta1 = theta[1]
    eta float : le learning rate
    epochs int : nombre de répétitions
    batchsize int : taille du batch
    ----------------
    Output :
    theta array : theta[0], theta1 = theta[1]
    cost_history array : liste du coût de theta à chaque itération
    theta_history array : liste des thetas à chaque itération
    """
    m=len(y)
    cost_history=[]
    theta_history = np.zeros((epochs,2),dtype=float)
    for i in range(epochs):
        cost_batch = []
        for (batchX, batchY) in next_batch(X, y, batchsize):
            # Extraction et traitement d'un batch à la fois
            prediction = predict(batchX,theta)
            error = batchX.T.dot((prediction - batchY))
            descent=eta * (1/(2*m)) * error
            theta-=descent
            cost_batch.append(cost_function(batchX,batchY,theta))
            theta_history[i][0]=theta[0]
            theta_history[i][1]=theta[1]

        cost_history.append(np.average(cost_batch))

    return theta, cost_history, theta_history

```

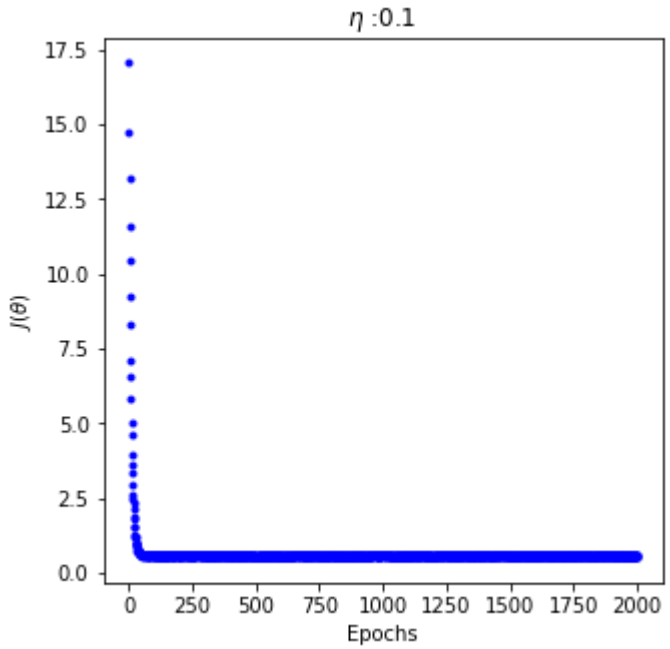
In [29]:

```
eta =0.1
epochs = 2000
#initialisation aléatoire de theta0 et de theta1
theta = np.random.randn(2,1)
X = np.random.rand(100,1)
y = 5 + 4*X+np.random.randn(100,1)

#X_0 ajoute un 1 au début de X pour avoir : theta0x_0 + theta1x_1 ...
X_0 = np.c_[np.ones((X.shape[0])), X]

theta,history_mini,theta_history = minibatch_gradient_descent(X_0,y,theta,eta
print ('theta0 = %0.2f'%theta[0],' theta1 = %.2f'%theta[1],' cost(%0.2f)'%his
```

```
theta0 = 4.89  theta1 = 3.99  cost(0.59)
```

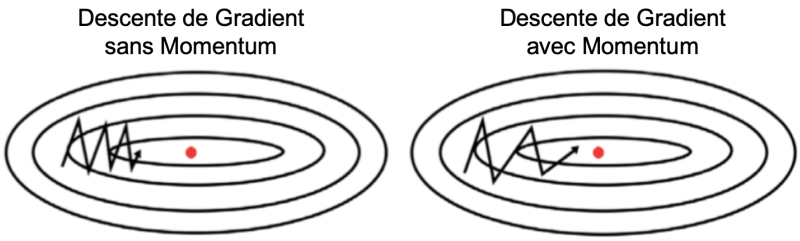
In [30]:

```
1  plot_history(eta,epochs,history_mini)
```



Momentum

Les approches précédentes ont du mal dans les zones où la surface s'incline beaucoup plus fortement dans une dimension que dans une autre. Dans ces scénarios, la descente de gradient oscille sur les pentes et ne progresse que doucement vers le minima local. Momentum est une méthode qui permet d'accélérer la descente de gradient dans la direction voulue et d'atténuer les oscillations (C.f Figure).



Le principe est le suivant. Considérons une balle qui descend le long d'une colline. Elle accumule de l'élan (*Momentum*) dans la descente et devient de plus en plus rapide jusqu'à ce qu'elle atteigne sa vitesse limite s'il existe une résistance de l'air. Dans le cas de la mise à jour des paramètres, la quantité de mouvement augmente pour les dimensions dont les gradients pointent dans la même direction et réduit les mises à jour pour les dimensions dont les gradients changent de direction. Par conséquent la convergence est plus rapide car le nombre d'oscillation diminue.

Momentum permet de modifier la descente de gradient en introduisant deux nouveaux paramètres : la vitesse v que nous essayons d'optimiser et le frottement (β).

Formellement il est défini, pour une fonction de coût J quelconque, de la manière suivante :

$$v = \beta v - \eta \Delta J$$
$$\theta = \theta - v$$

Dans le cas des fonctions linéaires précédentes nous avons :

$$v = \beta v - \eta \times \frac{-2}{m} \times (X \bullet \theta - y)X^T$$
$$\theta = \theta - v$$

L'avantage de momentum est qu'il ne change pas beaucoup l'algorithme de descente de gradient mais qu'il accélère considérablement l'apprentissage. Généralement β est fixé à 0.9.

Implémentation du momentum

L'approche par mini-batch étant la plus efficace, il suffit de remplacer la partie mise à jour des gradients par la formule précédente.

In [31]:

```
def next_batch(X, y, batchsize):
    """
    Création des batches
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives) avec un 1 ajout
    y array : données d'apprentissage (valeur à prédire)
    batchsize int : taille du batch
    """
    m=len(y)
    # permutation aléatoire de X et y pour faire des batchs avec des valeurs
    indices = np.random.permutation(m)
    X = X[indices]
    y = y[indices]
    for i in np.arange(0, X.shape[0], batchsize):
        # creation des batchs de taille batchsize
        yield (X[i:i + batchsize], y[i:i + batchsize])
```

In [32]:

```
def minibatch_gradient_descent_Momentum(X,y,v,beta,theta,learning_rate=0.01,e
    """
    Descente de gradient par mini batch avec momentum
    Paramètres :
    ----------------
    Input :
    X array : données d'apprentissage (variables prédictives) avec un 1 ajout
    y array : données d'apprentissage (valeur à prédire)
    theta array : theta[0], theta1 = theta[1]
    eta float : le learning rate
    epochs int : nombre de répétitions
    batchsize int : taille du batch
    beta : momentum
    ----------------
    Output :
    theta array : theta[0], theta1 = theta[1]
    cost_history array : liste du coût de theta à chaque itération
    theta_history array : liste des thetas à chaque itération
    """
    m=len(y)
    cost_history=[]
    v_history=np.zeros((epochs,2),dtype=float)
    theta_history = np.zeros((epochs,2),dtype=float)

    #beta=0.9
    for i in range(epochs):
        cost_batch = []
        for (batchX, batchY) in next_batch(X, y, batchsize):
            # Extraction et traitement d'un batch à la fois
            prediction = predict(batchX,theta)
            error = batchX.T.dot((prediction - batchY))
            # calcul du momentum
            grad=(1/(2*m)) * error
            v = beta * v + eta*grad
            theta-= v
            cost_batch.append(cost_function(batchX,batchY,theta))
            theta_history[i][0]=theta[0]
            theta_history[i][1]=theta[1]

        cost_history.append(np.average(cost_batch))

    return theta, cost_history, theta_history


```

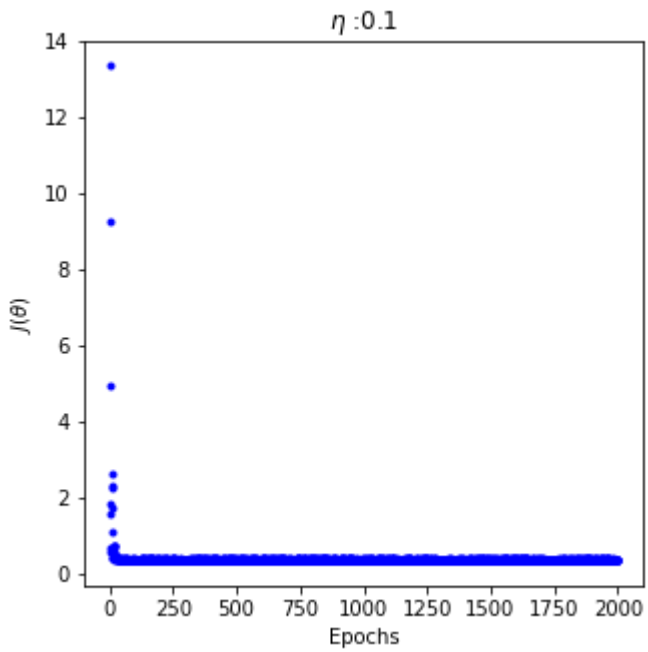
In [33]:

```
eta =0.1
epochs = 2000
beta = 0.9 # frottement
#initialisation aléatoire de theta0 et de theta1
theta = np.random.randn(2,1)
X = np.random.rand(100,1)
y = 5 + 4*X+np.random.randn(100,1)
# ajout du vecteur de vitesse
v = np.zeros(shape=(2,1))
#X_0 ajoute un 1 au début de X pour avoir : theta0x_0 + theta1x_1 ...
X_0 = np.c_[np.ones((X.shape[0])), X]

theta,history_minimomentum,theta_history = minibatch_gradient_descent_Momentu
print ('theta0 = %0.2f'%theta[0],' theta1 = %.2f'%theta[1],' cost(%0.2f)'%his
```

```
theta0 = 4.94  theta1 = 4.08  cost(0.59)
```

In [34]:

```
plot_history(eta,epochs,history_minimomentum)
```



Il existe d'autres types d'optimisation :

- *Nesterov Momentum* assez proche de Momentum mais qui, au lieu de calculer le gradient à la position actuelle, le calcul à une nouvelle position approximé.
- *Adagrad* s'intéresse au learning rate. Précédemment η a toujours été considéré constant. L'objectif d'Adagrad est de s'adapter pour pouvoir accélérer ou ralentir.
- *RMSprop* est une extension d'Adagrad pour corriger le fait que ce dernier considère une fonction d'accumulation du gradient qui est croissante monotone. Ceci peut cependant poser des problèmes car le taux d'apprentissage diminue également de façon monotone et l'apprentissage peut s'arrêter car η devient trop petit.
- *ADAM* cumule les avantages de Momentum et de RMSprop.

Toutes ces optimisations sont disponibles dans la plupart des approches de deep learning et sont utilisées lors des étapes de backpropagation.

Un exemple d'utilisation : la regression logistique

La régression logistique (*Logistic regression*) ou modèle logit constitue un cas particulier de modèle linéaire généralisé et est particulièrement utilisé comme approche supervisée de classification. Par contre, contrairement au modèle linéaire, la variable prédite ne peut prendre qu'un nombre limité de valeurs.

Dans le cas de la régression logistique binaire, étant donné un ensemble d'entrées X, l'objectif est de les classer dans l'une des deux catégories possible : 0 ou 1. En d'autres termes, elle modélise la probabilité que chaque entrée appartienne à une catégorie particulière.

Dans notre cas, nous savons que le modèle de regréssion linéaire est de la forme :

$$h_\theta(x) = \theta^T x$$

Pour pouvoir effectuer la classication, nous devons appliquer une fonction :

$$h_\theta(x) = \sigma(\theta^T x)$$

Dans le cas binaire, l'une des fonction d'activation couramment utilisée est la fonction sigmoide

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

L'hypothèse pour la regression logistique devient donc : '

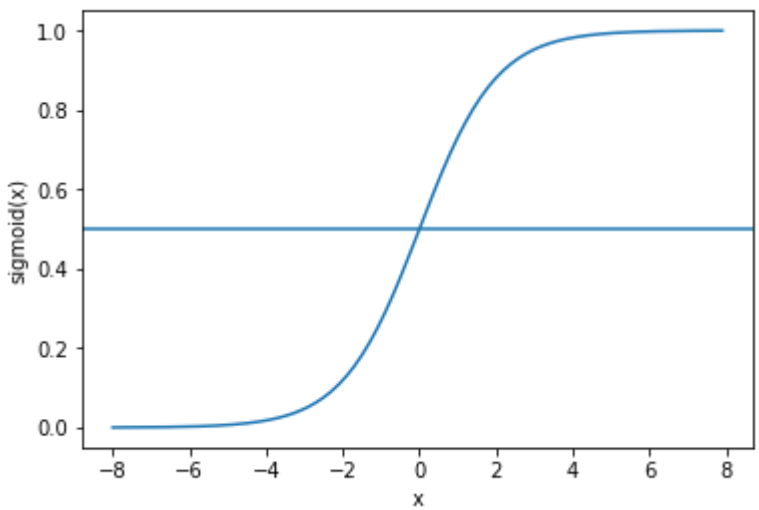
$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

In [35]:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

In [36]:

```
x = np.arange(-8, 8, 0.1)
f = sigmoid(x)
plt.plot(x, f)
plt.xlabel('x')
plt.axhline(.5)
plt.ylabel('sigmoid(x)')
plt.show()
```



Nous remarquons sur la courbe que sigmoid(x) > 0.5 quand $x > 0$ et sigmoid (x) < 0.5 quand $x < 0$:

$$h_\theta(x) = \begin{cases} > 0.5, \ si\ \theta^T x > 0 \\ < 0.5, \ si\ \theta^T x < 0 \end{cases}$$

En d'autres termes, si la somme pondérée des entrées est supérieure à zéro, la classe prédite est 1 et inversement si elle est inférieure à 0, la classe prédite est 0.

La fonction de coût généralement associée est la suivante :

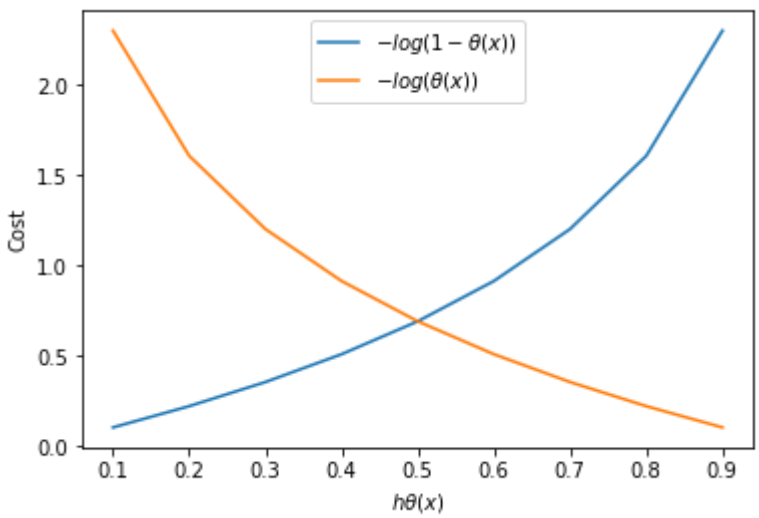
$$Cost = \begin{cases} -log(h_\theta(x)), \ si\ y = 1 \\ -log(1 - h_\theta(x)), \ si\ y = 0 \end{cases}$$

L'intuition derrière la fonction de coût est la suivante :

Si la classe réelle est 1 et que le modèle prédit 0, il faut fortement le pénaliser et vice-versa. Comme vous pouvez le constater sur la figure ci-dessous, pour le graphique $-log(h_\theta(x))$, lorsque $h_\theta(x)$ approche de 1, le coût est égal à 0 et lorsque $h_\theta(x)$ est proche de 0, le coût est égal à l'infini (le modèle est fortement pénalisé). De même pour le graphique $-log(1 - h_\theta(x))$ lorsque la valeur réelle est 0 et que le modèle prédit 0, le coût est 0 et le coût devient l'infini lorsque $h_\theta(x)$ se rapproche de 1.

In [37]:

```
1  x = np.arange(0.1, 1, 0.1)
2  f = -np.log(x)
3  g = -np.log(1-x)
4  plt.plot(x, g,label=r'$-log(1-\theta(x))$')
5  plt.plot(x, f,label=r'$-log(\theta(x))$')
6  plt.xlabel(r'$h\theta(x)$')
7  #plt.xlabel(r'$h_\theta(x)$')
8  plt.ylabel('Cost')
9  plt.legend()
10 plt.show()
```



Nous pouvons donc combiner les deux équations précédentes :

$$Cost(h_\theta(x), y) = -ylog(h_\theta(x) - (1 - y)log(1 - h_\theta(x))$$

La perte pour tous les exemples d'apprentissage, m, est comme précédemment calculé en prenant la moyenne de tous les coûts de tous les échantillons d'apprentissage :

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)}) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))]$$

ou

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)}) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))]$$

La dérivée partielle par rapport à θ_j est :

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y)x_j^{(i)}$$

La descente de gradient consiste donc à répéter, comme précédemment, jusqu'à convergence :

$$\theta_j = \theta_j - \eta\frac{1}{m}X^T(\sigma(X.\theta) - y)$$

In [38]:

```
def cost_function(h, y):
    return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()
```

Sélection du jeu de données Iris et récupération des 100 premiers enregistrements (*Serosa* et *Versicolor*). Nous nous intéressons uniquement aux deux premiers attributs (*longueur des sépales* et *largeur des sépales*).

In [39]:

```
from sklearn import datasets
data = datasets.load_iris()

#recupération des 100 premiers (ils sont ordonnés)
X = data.data[:100, :2]
y = data.target[:100]

```

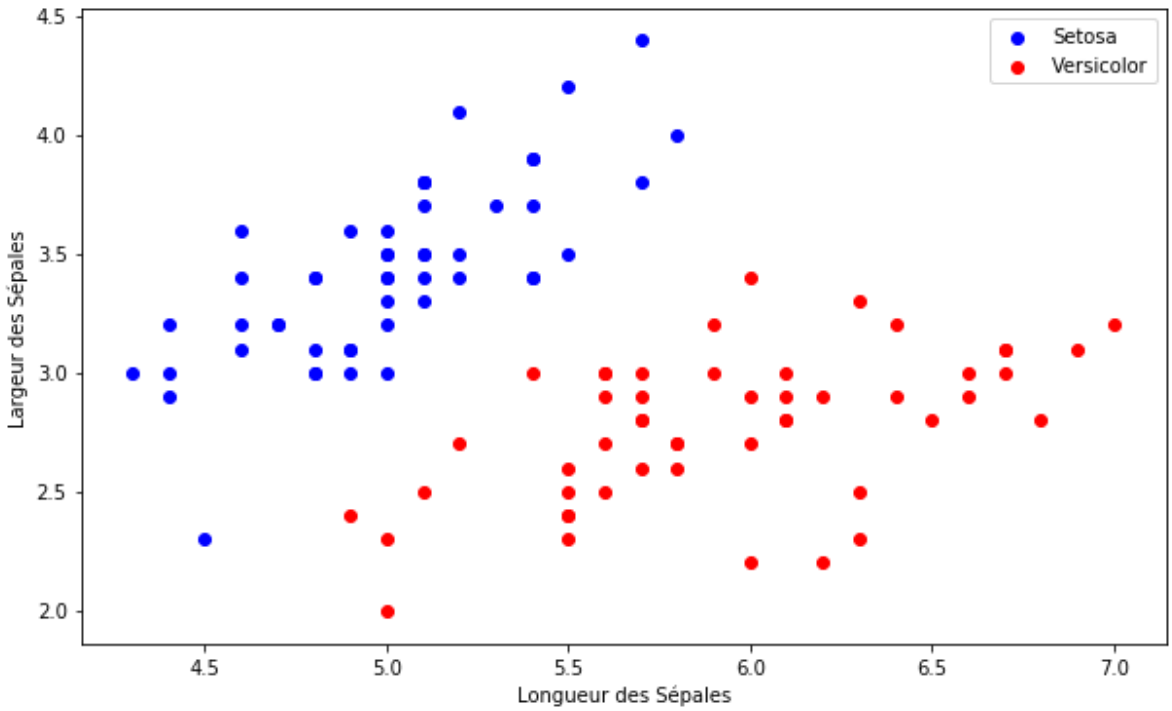
Affichage du jeu de données

In [40]:

```
1  plt.figure(figsize=(10, 6))
2  setosa = plt.scatter(X[:50,0], X[:50,1], c='b')
3  versicolor = plt.scatter(X[50:,0], X[50:,1], c='r')
4  plt.xlabel("Longueur des Sépales")
5  plt.ylabel("Largeur des Sépales")
6  plt.legend((setosa, versicolor), ("Setosa", "Versicolor"))
```

Out[40]:

```
<matplotlib.legend.Legend at 0x11bc069b0>
```



In [41]:

```
def gradient_descent(X, y,theta, eta=0.01, epochs=1000):
    m=len(y)
    cost_history=[]
    for i in range(epochs):
        z = np.dot(X,theta)
        h = sigmoid(z)
        gradient = np.dot(X.T, (h - y)) / m
        theta -= eta * gradient

        cost_history.append(cost_function(h,y))

    return theta, cost_history
```

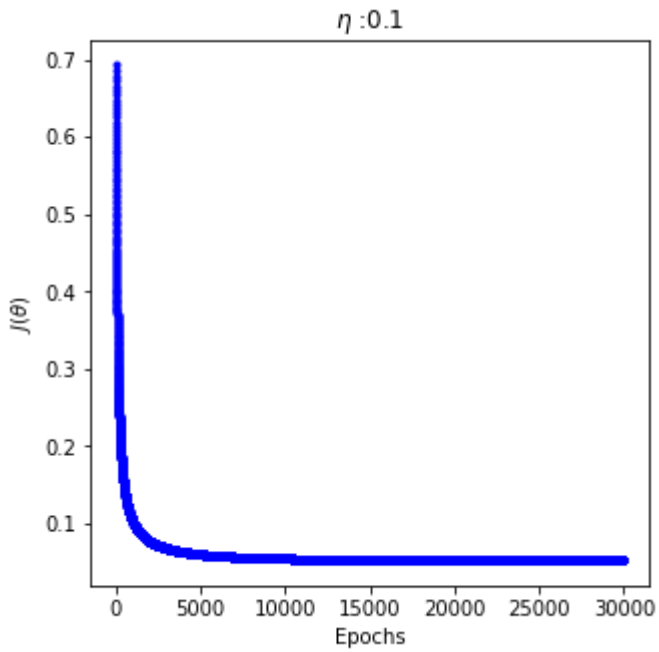
In [42]:

```
eta =0.1
epochs = 30000
#Initialisation de theta (ici à zéro)
theta = np.zeros(X.shape[1])

theta,cost_history=gradient_descent(X, y,theta, eta, epochs)

print ("Les paramètres sont : ",theta)
```

```
Les paramètres sont :  [  6.66602804 -11.67484425]
```

In [43]:

```
plot_history(eta,epochs,cost_history)
```



Définissons, à présent, la fonction de prediction. Par rapport ce que nous avons vu précédemment, la valeur de seuil pour sigmoide est de 0.5.

In [44]:

```
def predicts(X, theta, threshold=0.5):
    return sigmoid(np.dot(X, theta)) >= threshold
```

In [45]:

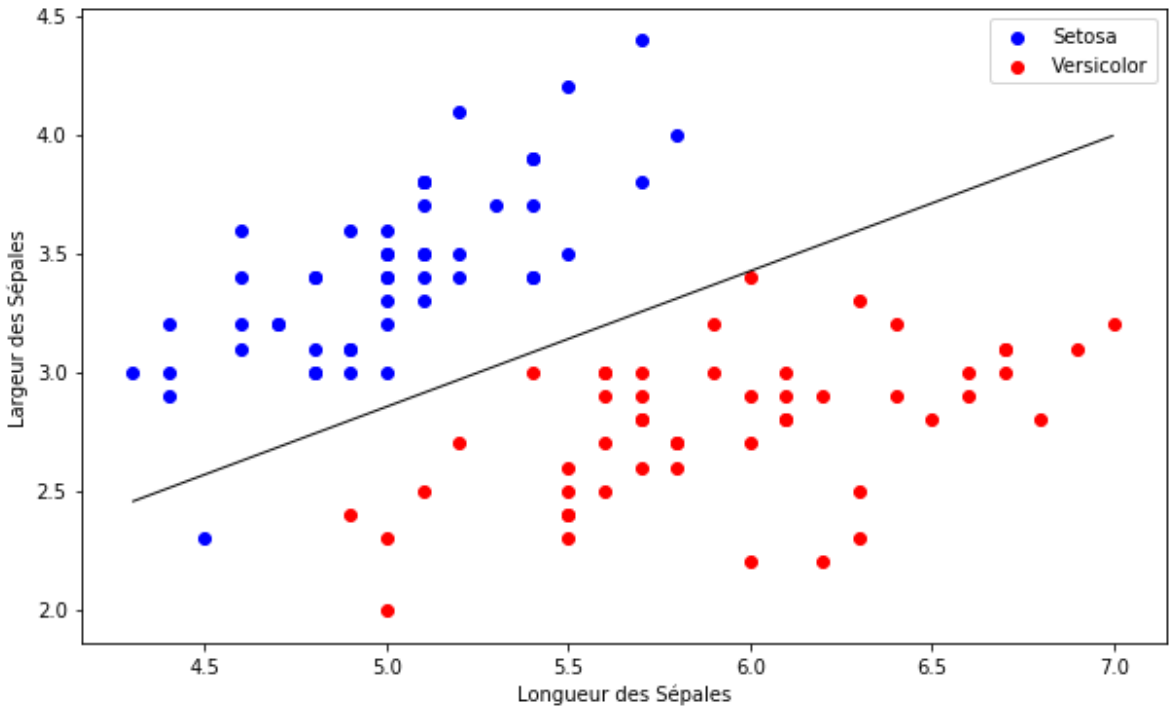
```
y_predits = predicts(X,theta)
print (theta)
preds = predicts(X,theta)
(preds == y).mean()
print ("Nombre de bons prédits",np.sum(y == y_predits),' sur ',y.size)
```

```
[  6.66602804 -11.67484425]
Nombre de bons prédits 99  sur  100
```

Affichage de la zone de décision

In [46]:

```
plt.figure(figsize=(10, 6))
setosa = plt.scatter(X[:50,0], X[:50,1], c='b')
versicolor = plt.scatter(X[50:,0], X[50:,1], c='r')
plt.xlabel("Longueur des Sépales")
plt.ylabel("Largeur des Sépales")
plt.legend((setosa, versicolor), ("Setosa", "Versicolor"))


x1_min, x1_max = X[:,0].min(), X[:,0].max(),
x2_min, x2_max = X[:,1].min(), X[:,1].max(),
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_ma
grid = np.c_[xx1.ravel(), xx2.ravel()]
probs = sigmoid(np.dot(grid, theta)).reshape(xx1.shape)
plt.contour(xx1, xx2, probs, [0.5], linewidths=1, colors='black');
```



Essayons à présent le modèle de Regression Logistique de Sickit Learn.

In [47]:

```
from sklearn import linear_model
clf = linear_model.LogisticRegression(solver='lbfgs')
clf.fit(X, y)
print ("Nombre de bons prédits avec Sickit Learn : ",np.sum(y == clf.predict(
```

```
Nombre de bons prédits avec Sickit Learn :  100  sur  100
```

Pas mal du tout !

Conclusion

Attention dans les exemples précédents nous n'avons pas normalisé les données. Avant toute opération il est indispensable de le faire !

Outre le fait qu'il puisse y avoir des données avec des valeurs très différentes, le fait de normaliser permet surtout de converger plus vite. Par exemple en utilisant la fonction suivante :

In [48]:

```
from sklearn import preprocessing

normalized_X = preprocessing.normalize(X)
```

Considérez la figure suivante. Précédemment nous avons vu qu'il fallait ajouter un 1 dans la première colonne de la matrice. Dans la figure ce x_O s'appelle le biais. Si nous appelons σ, la fonction de sortie, fonction d'activation, nous obtenons un perceptron qui est la base des algorithmes de réseaux de neurones ... et donc de l'apprentissage profond. La regression logistique ne vise pas le même objectif que les réseaux de neurones car il s'agit de déterminer dans le premier cas une probabilité sur la valeur de sortie et dans le second cas il constitue une étape vers d'autres couches éventuelles mais comme vous pouvez le constater la construction est très similaire, les θ qui correspondent à nos paramètres sont généralement appelés *poids* dans les réseaux de neurones mais il s'agit de la même chose.