# Devoir 2 pour IFT6390 - Fondements de l'apprentissage machine

**Par Olivier Malenfant-Thuot**

Matricule: 1012818

## Question 1: Linear and non-linear regularized regression

### 1.1 Linear Regression

1.1.1 L'ensemble des paramètres $\theta$ du modèle de régression linéaire sont $\vec{\mathrm{w}} \in \mathbb{R}^{\mathrm{d}}$ et $b \in \mathbb{R}$.

1.1.2 Le risque empirique pour un training set $D$, évalué à partir de la fonction de perte $L((x,t),f)$ prend la forme

$$\hat{R}(f,D) = \sum_{(x,t)\in D} (f(x)-t)^2 = \sum_{(x,t)\in D} (\vec{\mathrm{w}}^T\vec{x}+b-t)^2$$

1.1.3 Nous pouvons utiliser le ERM pour minimiser le risque, avec:

$$\frac{\partial}{\partial\vec{\mathrm{w}}}\frac{\partial}{\partial b}\sum_{(x,t)\in D} (\vec{\mathrm{w}}^T\vec{x}+b-t)^2 = 0.$$

1.1.4 Le gradient du risque empirique peu être exprimé par:

$$\nabla\hat{R} = \frac{1}{n}\sum_{(x,t)\in D}\begin{bmatrix} \frac{\partial}{\partial w_1}(\vec{\mathrm{w}}^T\vec{x}+b-t)^2 \\ \frac{\partial}{\partial w_2}(\vec{\mathrm{w}}^T\vec{x}+b-t)^2 \\ \vdots \\ \frac{\partial}{\partial w_d}(\vec{\mathrm{w}}^T\vec{x}+b-t)^2 \\ \frac{\partial}{\partial b}(\vec{\mathrm{w}}^T\vec{x}+b-t)^2 \end{bmatrix} = \frac{1}{n}\sum_{(x,t)\in D}\begin{bmatrix} 2x_1(\vec{\mathrm{w}}^T\vec{x}+b-t) \\ 2x_2(\vec{\mathrm{w}}^T\vec{x}+b-t) \\ \vdots \\ 2x_d(\vec{\mathrm{w}}^T\vec{x}+b-t) \\ 2(\vec{\mathrm{w}}^T\vec{x}+b-t) \end{bmatrix}$$

1.1.5 Le gradient du risque empirique indique dans quelle direction de l'espace des paramètres $\theta$ il faut se déplacer afin de diminuer la somme des erreurs de tous les points du training set.

## 1.2 Ridge Regression

1.2.1 Le gradient du risque empirique régularisé devient

$$\nabla \tilde{R} = \frac{1}{n} \sum_{(x,t) \in D} \begin{bmatrix} 2x_1(\vec{\mathrm{w}}^T \vec{x} + b - t) + 2\lambda w_1 \\ 2x_2(\vec{\mathrm{w}}^T \vec{x} + b - t) + 2\lambda w_2 \\ \vdots \\ 2x_d(\vec{\mathrm{w}}^T \vec{x} + b - t) + 2\lambda w_d \\ 2(\vec{\mathrm{w}}^T \vec{x} + b - t) \end{bmatrix}$$

.

Ce nouveau gradient diffère du précédent par un terme $2\lambda w_i$ pour chaque dimension $i$, mais est le même pour le bias.

1.2.2 Pseudocode pour le training:

Déterminer aléatoirement des valeurs de départ pour w et b.

Définir les valeurs des hyperparamètres $\lambda$ et $\eta$.

Définir un nombre d'itérations maximal pour l'arrêt du calcul

Définir une valeur d'arrêt pour le gradient

```
iteration = 0
Début d'une boucle sur itermax:
    Calcul du gradient par la formule du 1.2.1 pour chaque point du dataset
    Calcul de la moyenne des gradients

    Update des valeurs de w et b par:
    Ajout du produit entre les d premières dimensions du gradient et -eta à
 w
    Ajout du produit entre la dernière dimension du gradient et -eta à b

    Calcul de la norme du gradient
    si la norme du gradient est plus petite que la valeur d'arrêt, break la
 boucle
```

Les dernières valeurs de w et b sont les valeurs minimisant le risque empirique régularisé.

1.2.3 Le risque empirique et son gradient peuvent être exprimé de façon matricielle de la façon suivante.

Le risque empirique est $\hat{R} = ||w^T X^T - t^T||^2$, avec $w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$.

Le gradient du risque empirique est $2X^T((w^T X^T)^T - t) + 2\lambda w$.

1.2.4 Le gradient est le même qu'au 1.2.3 plus le terme pour de ridge. La solution au problème de minimisation est

$$\nabla \hat{R} = \vec{\nabla} \cdot (w^T X + \lambda ||w||^2) = 0$$

Quand $\lambda = 0$, nous retrouvons l'expression pour le gradient sans le ridge term. Quand $N < d$,

## 1.3 Regression with a fixed non-linear pre-processing

1.3.1 L'expression complète de $\tilde{f}(x)$ est

$$\tilde{f}(x) = \begin{bmatrix} f(x) \\ f(x^2) \\ \vdots \\ f(x^k) \end{bmatrix}$$

1.3.2 Les paramètres de l'entraînement entrainer sont:

w, de dimension k qui représente le poids à donner à chaque valeur x^i dans la régression,

b, le bias de dimension 1, qui joue le même rôle que précédemment,

k, de dimension 1, un scalaire qui détermine le nombre de dimensions de la transformation pôlynomiale. C'est un hyperparamètre, car le résultat de l'entraînement dépend de sa valeur.

1.3.3 Avec $d = 2$ et $x = (x_1, x_2)$, les transformation prennent la forme:

$$\phi_{poly^1}(x) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \phi_{poly^2}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}, \phi_{poly^3}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \\ x_1^3 \\ x_2^3 \\ x_1^2 x_2 \\ x_1 x_2^2 \end{bmatrix}$$

1.3.4 En analyse combinatoire, le nombre de façon de répartir $k$ objets identique dans $d$ contenant discernables est donné par $\binom{k+d-1}{d-1}$. Par conséquent, la dimensionalité de la transformation $\phi_{poly^k}(x)$, pour un $x$ de dimension $d$, est donnée par:

$$\sum_{i=1}^{k} \binom{i+d-1}{d-1}$$

avec $\binom{m}{n} = \frac{m!}{n!(m-n)!}$.

# Question 2: Practical Part

In [76]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
def h(x):
    return np.sin(x) + 0.3*x - 1
```

In [45]:
```
data_x = 10 * (np.random.rand(15) - 0.5).reshape(15,1)
data_h = h(data_x)
```

In [4]:
```
# Hyperparameters
Lambda = 0.0
step = 0.1
maxiter = 50
```

In [130]:
```
class Regression_gradient:

    def __init__(self, Lambda = 0.1, step = 0.1, maxiter = 50):
        self.Lambda = Lambda
        self.step = step
        self.maxiter = maxiter
        self.maxgrad = 0.001

    def train(self, data, targets):
        self.bias = 0
        self.ndims = data.shape[1]
        self.weights = np.random.rand(self.ndims).reshape(1,self.ndim
s)/100

        iteration = 0
        delta_w = self.maxgrad * 3
        while (iteration < self.maxiter) and (np.linalg.norm(delta_w)
 > self.maxgrad):
            #objectives = self.objective_function(data, targets)
            #print(objectives)
            delta_w = self.dJdW(data,targets)
            delta_b = self.dJdb(data,targets)
            self.weights = self.weights - self.step * delta_w
            self.bias = self.bias - self.step * delta_b
            iteration += 1

    def objective_function(self, data, targets):
        #Seulement utile pour débugger
        return np.sum(np.sum(self.weights*data, axis = 1) + self.bias
 - targets)**2 + self.Lambda * np.sum(self.weights**2)

    def dJdW(self, data, targets):
        derivative = np.zeros(self.ndims).reshape(self.ndims,1)
        intermediate = np.sum(self.weights * data, axis = 1).reshape(
data.shape[0],1) + self.bias - targets
        derivative = np.mean(2 * data * intermediate + 2 * self.Lambd
a * self.weights, axis = 0)
        return derivative

    def dJdb(self, data, targets):
        intermediate = np.sum(self.weights * data, axis = 1).reshape(
data.shape[0],1) + self.bias - targets
        return np.mean(2 * intermediate)

    def predict(self, data):
        predictions = np.sum(self.weights * data, axis = 1) + self.bi
as
        return predictions.reshape((data.shape[0],1))
```

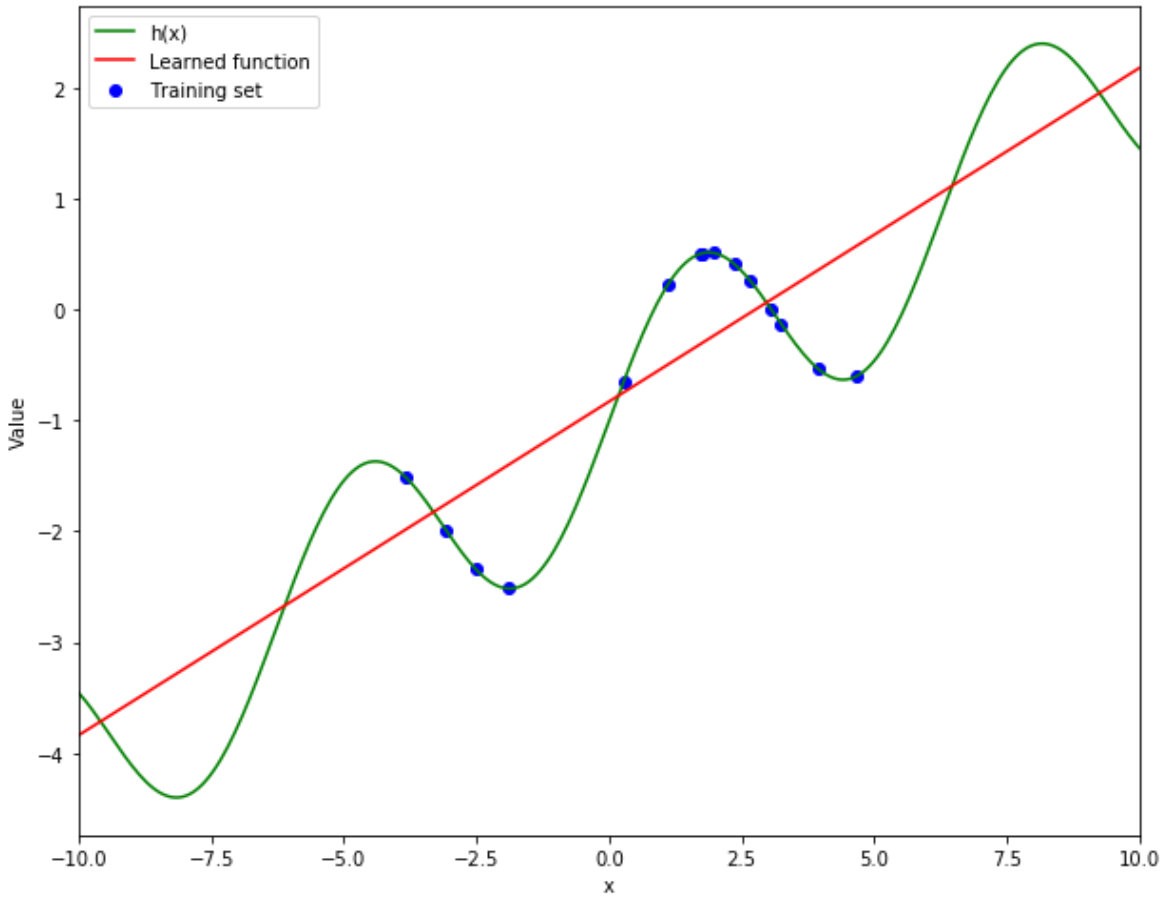
### 2.3 Résultats pour $\lambda = 0$

In [46]:
```
Reg = Regression_gradient(Lambda = 0, maxiter=100)
Reg.train(data_x,data_h)
```

In [47]:
```
fig, ax = plt.subplots(figsize = (10, 8))

x_points = np.linspace(-10, 10, 300).reshape(300,1)

ax.scatter(data_x, data_h, c = 'b', label = 'Training set')
ax.plot(x_points, h(x_points), c = 'g', label = 'h(x)')
ax.plot(x_points, Reg.predict(x_points), c = 'r', label = 'Learned fu
nction')
ax.set_xlim(-10, 10)
ax.set_xlabel('x')
ax.set_ylabel('Value')
ax.legend()
plt.show()
plt.close()
```



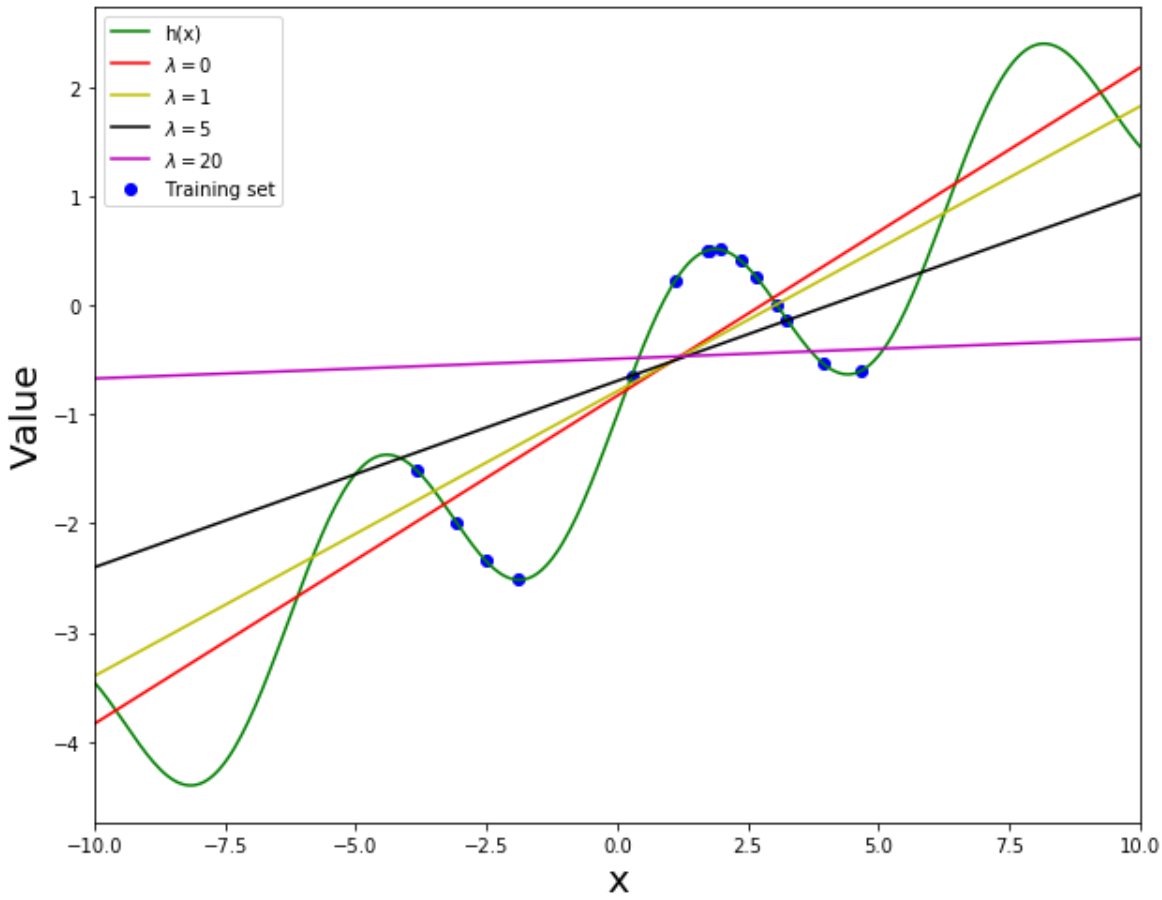

## 2.4 Comparaison de l'effet de $\lambda$

In [77]:
```
Reg_null_lambda = Regression_gradient(Lambda = 0, step = 0.01, maxite
r=5000)
Reg_null_lambda.train(data_x,data_h)
Reg_small_lambda = Regression_gradient(Lambda = 1, step = 0.01, maxit
er=5000)
Reg_small_lambda.train(data_x,data_h)
Reg_medium_lambda = Regression_gradient(Lambda = 5, step = 0.01, maxi
ter=5000)
Reg_medium_lambda.train(data_x,data_h)
Reg_large_lambda = Regression_gradient(Lambda = 100, step = 0.001, ma
xiter=5000)
Reg_large_lambda.train(data_x,data_h)
```

In [78]:
```
fig, ax = plt.subplots(figsize = (10, 8))

x_points = np.linspace(-10, 10, 300).reshape(300,1)

ax.scatter(data_x, data_h, c = 'b', label = 'Training set')
ax.plot(x_points, h(x_points), c = 'g', label = 'h(x)')
ax.plot(x_points, Reg_null_lambda.predict(x_points), c = 'r', label =
 '$\lambda = 0$')
ax.plot(x_points, Reg_small_lambda.predict(x_points), c = 'y', label
= '$\lambda = 1$')
ax.plot(x_points, Reg_medium_lambda.predict(x_points), c = 'k', label
 = '$\lambda = 5$')
ax.plot(x_points, Reg_large_lambda.predict(x_points), c = 'm', label
= '$\lambda = 20$')
ax.set_xlim(-10, 10)
ax.set_xlabel('x')
ax.set_ylabel('Value')
ax.xaxis.label.set_fontsize(20)
ax.yaxis.label.set_fontsize(20)
ax.legend()
plt.show()
plt.close()
```




2.5 Évolution logarithmique de $\lambda$

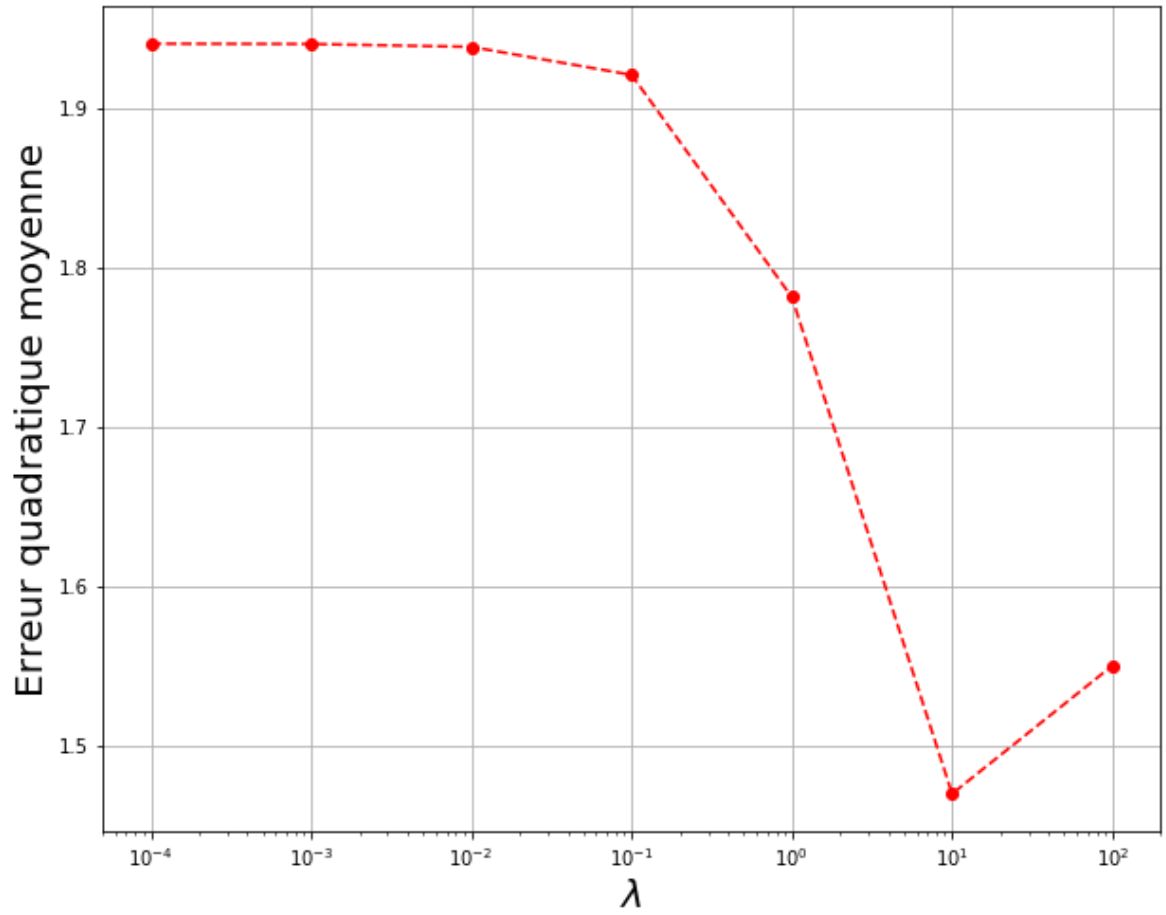
In [79]:
```
data_test = 10 * (np.random.rand(100) - 0.5).reshape(100,1)
targets = h(data_test)
lambda_values = [0.0001, 0.001, 0.01, 0.1, 1., 10., 100.]
step_values = [0.01 , 0.01, 0.01, 0.01, 0.01, 0.005, 0.001]
```

In [80]:
```
quad_loss = []

for Lambda, step in zip(lambda_values, step_values):
    Reg = Regression_gradient(Lambda = Lambda, step = step, maxiter=1
0000)
    Reg.train(data_x,data_h)
    predictions = Reg.predict(data_test)
    quad_loss.append(np.mean((predictions - targets)**2))
```

In [105]:
```
fig, ax = plt.subplots(figsize = (10, 8))

ax.semilogx(lambda_values, quad_loss, 'o--', c = 'r')
ax.set_xlabel('$\lambda$')
ax.set_ylabel('Erreur quadratique moyenne')
ax.xaxis.label.set_fontsize(20)
ax.yaxis.label.set_fontsize(20)
ax.grid(True)
plt.show()
plt.close()
```



L'erreur est au plus haut avec un $\lambda$ très petit et diminue si on augmente lambda. Elle est au minimum lorsque $\lambda$ est égal à 10 et remonte lorsque lambda est plus grand.

2.6

In [82]:
```
Lambda = 0.01
```

In [90]:
```
x_points = np.linspace(-10, 10, 300).reshape(300,1)
```

In [83]:
```
def phi_transform(x,l):
    transformed_points = np.zeros((x.shape[0],l)).reshape(x.shape[0],
l)
    for j in range(x.shape[0]):
        for i in range(l):
            transformed_points[j,i] = x[j]**(i+1)
    return transformed_points
```

In [84]:
```
Reg1 = Regression_gradient(Lambda=Lambda, maxiter = 100)
Reg1.train(phi_transform(data_x,l = 1), data_h)
Reg2 = Regression_gradient(Lambda=Lambda, maxiter = 100000, step=0.01
)
Reg2.train(phi_transform(data_x,l = 2), data_h)
Reg3 = Regression_gradient(Lambda=Lambda, maxiter =100000, step=0.000
5)
Reg3.train(phi_transform(data_x,l = 3), data_h)
```

In [95]:
```
Reg4 = Regression_gradient(Lambda=Lambda, maxiter =600000, step=0.000
02)
Reg4.train(phi_transform(data_x,l = 4), data_h)
```

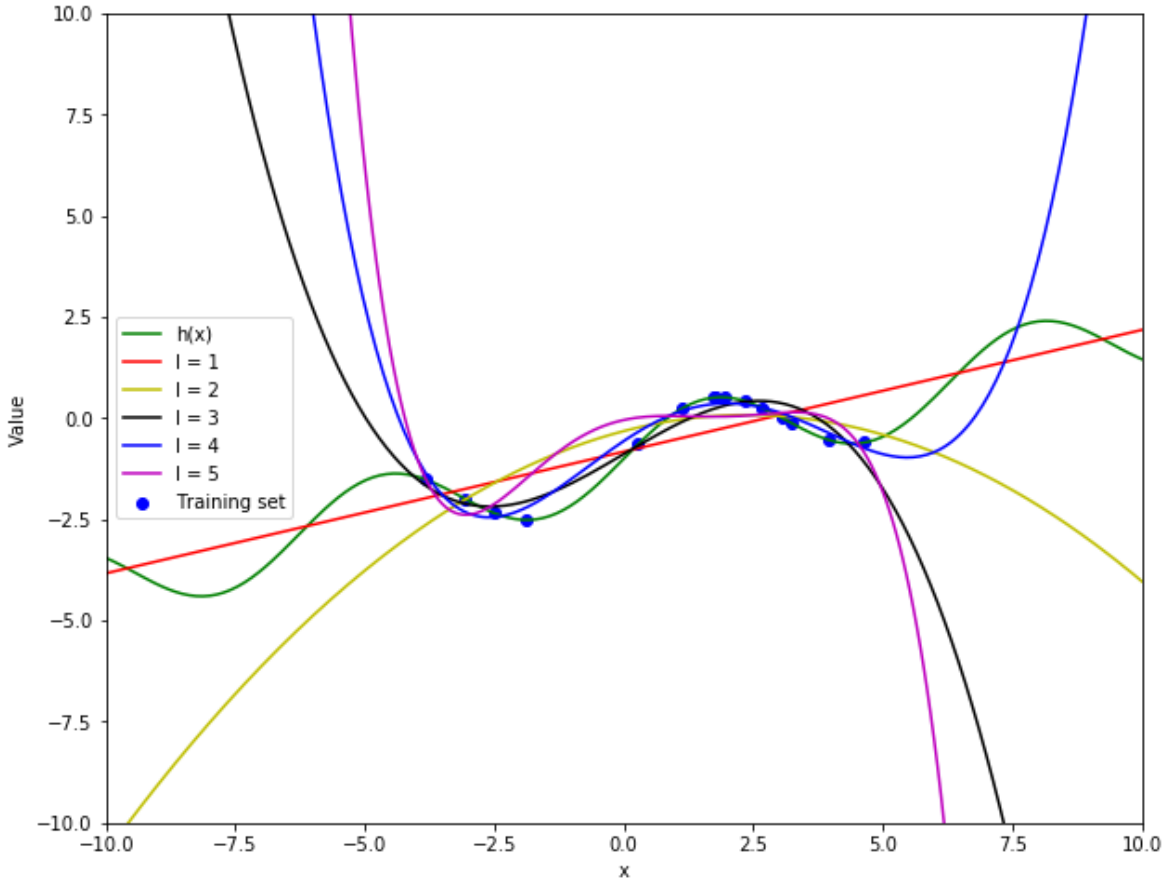
In [98]:
```
Reg5 = Regression_gradient(Lambda=Lambda, maxiter =1000000, step=0.00
00005)
Reg5.train(phi_transform(data_x,l = 5), data_h)
```

In [99]:
```
print(Reg1.weights)
print(Reg2.weights)
print(Reg3.weights)
print(Reg4.weights)
print(Reg5.weights)
```

```
[[0.30068641]]
[[ 0.33536403 -0.07066595]]
[[ 0.75418596  0.00111588 -0.03728825]]
[[ 0.85550511 -0.12845443 -0.04886678  0.00754218]]
[[ 0.27837169 -0.23816802  0.04871561  0.01310047 -0.00367793]]
```

In [150]:
```
fig,ax = plt.subplots(figsize = (10,8))

ax.scatter(data_x, data_h, c = 'b', label = 'Training set')
ax.plot(x_points, h(x_points), c = 'g', label = 'h(x)')
ax.plot(x_points, Reg1.predict(x_points), c = 'r', label = 'l = 1')
ax.plot(x_points, Reg2.predict(phi_transform(x_points, l=2)), c = 'y'
, label = 'l = 2')
ax.plot(x_points, Reg3.predict(phi_transform(x_points, l=3)), c = 'k'
, label = 'l = 3')
ax.plot(x_points, Reg4.predict(phi_transform(x_points, l=4)), c = 'b'
, label = 'l = 4')
ax.plot(x_points, Reg5.predict(phi_transform(x_points, l=5)), c = 'm'
, label = 'l = 5')
ax.set_xlim(-10, 10)
ax.set_ylim(-10, 10)
ax.set_xlabel('x')
ax.set_ylabel('Value')
ax.legend()
plt.show()
plt.close()
```



2.7 Lorsque nous augmentons $l$ et la dimensionalité de l'entraînement, le nombre d'itérations nécessaire pour atteindre un faible gradient augmente énormément, et le step size doit être réduite énormément, sinon nous passons par dessus le petit espace des paramètres où se situe le minimum recherché. En effet, en raison de la transformation polynomiale, les hautes dimensions du vecteur varient très vites avec une petite variation des paramètres. Dans le cas présent, le nombre de dimensions détermine le degré du polynôme qui est utilisé pour fitter les données. La courbe obtenue aura donc $d - 1$ inflexions. Il y a beaucoup de courbes sur le graphique précédent, mais nous pouvons étudier les risques empiriques et réels séparément.

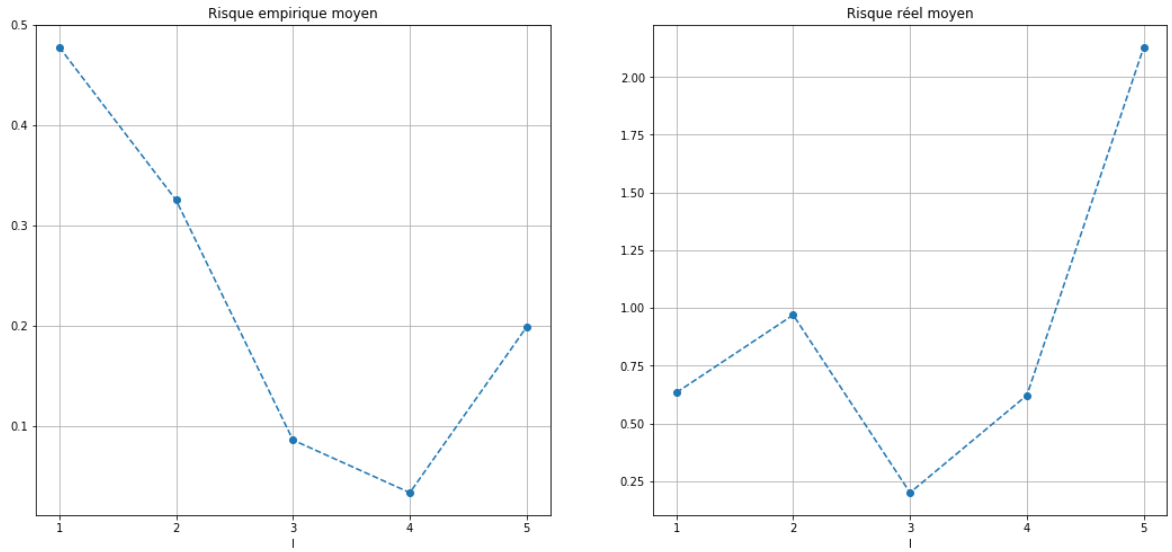
In [149]:
```
fig,ax = plt.subplots(ncols = 2, figsize = (18,8))
ax1,ax2 = ax.flatten()

quad_loss_emp = []
quad_loss_test = []
l_list = [1,2,3,4,5]
reg_list = [Reg1, Reg2, Reg3, Reg4, Reg5]

for reg, l in zip(reg_list, l_list):
    predictions = reg.predict(phi_transform(data_x, l = l))
    quad_loss_emp.append(np.mean((predictions - data_h.T)**2))

for reg, l in zip(reg_list, l_list):
    predictions = reg.predict(phi_transform(data_test, l = l))
    quad_loss_test.append(np.mean((predictions - h(data_test).T)**2))

ax1.plot(l_list,quad_loss_emp,'o--')
ax2.plot(l_list,quad_loss_test,'o--')
ax1.set_title('Risque empirique moyen')
ax2.set_title('Risque réel moyen')
ax1.set_xlabel('l')
ax2.set_xlabel('l')
ax1.xaxis.set_ticks(l_list)
ax2.xaxis.set_ticks(l_list)
ax1.grid(True)
ax2.grid(True)
```



Le risque empirique diminue avec l'introduction de nouvelles dimensions jusqu'à $l = 4$, puis à $l = 5$ il augmente légèrement. Ce comportement s'explique par le fait que un polynôme de haut degré peut plus facilement fitter une distribution de points. L'augmentation entre 4 et 5 dimensions est dû à la nature de la distribution qui est très semblable localement à un polynôme de degré 4. Le comportement du risque réel est différent. Le minimum est situé à $l = 3$, car en allant plus loin, nous faisons un overfit de notre dataset limité et nous éloignons de la fonction $h(x)$.