MO444 – Aprendizado de Máquina

Edgar Rodolfo Quispe Condori - RA 192617

March 24, 2017

Observações:

- O código original esta disponivel no `https://www.dropbox.com/sh/l4rj8e6mw64rsf6/AAAuHBJLba46X533cAxJi8TOa?dl=0`
- A explicação do cálculo matemático do gradiente se encontra ao final do documento.

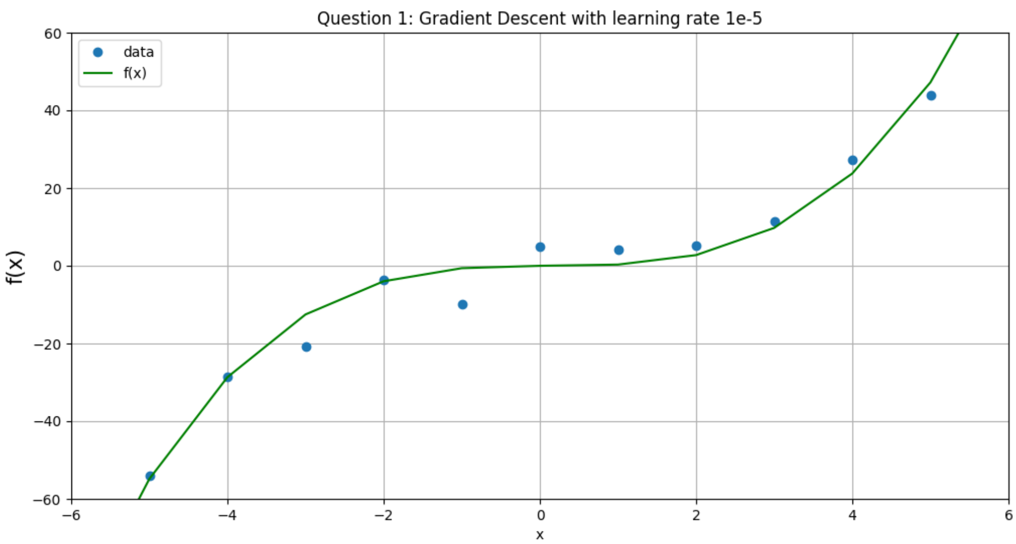
Questão 1. *Implemente uma decida do gradiente (escreva explicitamente o código para a função que computa o gradiente). Use um learning rate the 1.0e-5, inicie do ponto a=0,b=0,c=0,d=0, e rode 50 iterações.*

Qual a solução encontrada (os valores de a,b,c,e d)? Qual o erro na solução? Plote a função com os valores solução em conjunto com os dados.

Os resultados obtidos são:

- $erro = 224.107804051$
- $a = 0.40582178$
- $b = -0.15346867$
- $c = 0.05946159$
- $d = -0.00842238$.

O plote da função:



O código:

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

#define the original function
def f(x, w):
  return w[0]*(x**3) + w[1]*(x**2) + w[2]*(x) + w[3]

#define  squared sum as cost function
def cost(calculated_y, expected_y):
  return ((expected_y - calculated_y)**2).sum()

#given the weights w (a, b, c, d), x
#and y (training data), compute gradient
def gradient(x, y, w):
  fact = np.vstack( (x*x*x, x*x, x , np.ones(len(x))) )
  return 2 * (f(x, w) - y) * fact

# define the update function delta w
def delta_w(w_k, x, t, learning_rate):
  return learning_rate * gradient(x, t, w_k).sum(axis = 1)

def plot_results(x, y, w):
  fig = plt.figure(figsize=(12, 6))
  fig.add_subplot(111)

  # Plot the target versus the input x
  plt.plot(x, y, 'o', label='data')
  # Plot the initial line
  x = list(range(-10, 10))
  plt.plot(x, [f(i,w) for i in x],'g-',label='f(x)')

  #set extra plot parameters
  plt.title('Question 1: Gradient Descent with learning rate 1e-5')
  plt.xlim(-6,6)
  plt.ylim(-60,60)
  plt.xlabel('x')
  plt.ylabel('f(x)', fontsize=15)
  plt.legend(loc=2)
  plt.grid()

  plt.show()
  fig.savefig("t01_1.png")


if __name__ == "__main__":

  x_train = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
  y_train = np.array([-53.9, -28.5, -20.7, -3.6, -9.8, 5.0,
```

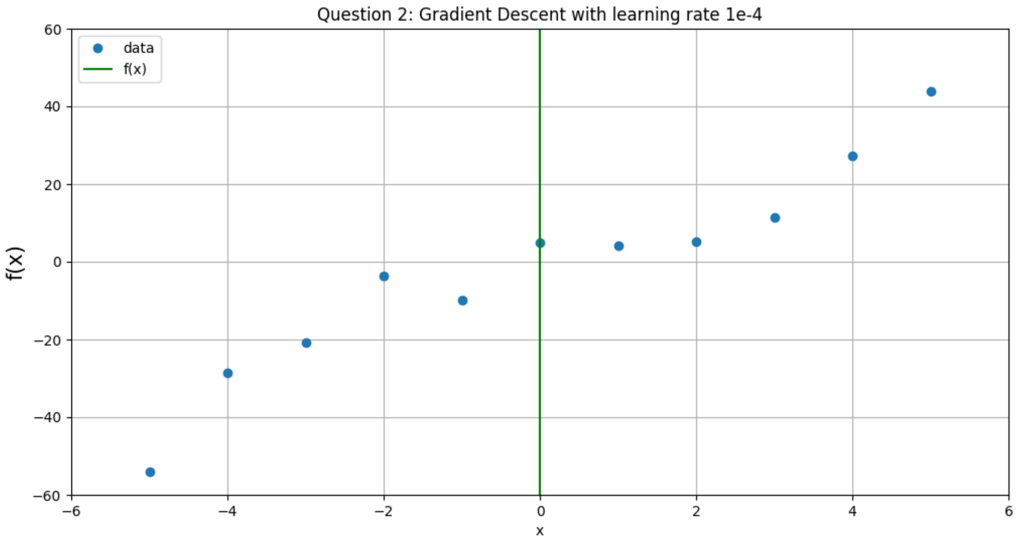
```
                    4.2, 5.1, 11.4, 27.4, 44.0])


# Set the initial weight parameter
w = np.array([0, 0, 0, 0])
# Set the learning rate
learning_rate = 1.0e-5

#number of gradient descent iterations
nb_of_iterations = 50

# Start performing the gradient descent updates,
#and print the weights and cost:

# List to store the weight,costs values
w_cost = [ cost(f(x_train, w), y_train)]
w_status = [w]

for i in range(nb_of_iterations):
    #Get the delta w update
    dw = delta_w(w, x_train, y_train, learning_rate)
    #Update the current weight parameter
    w = w - dw
    #Add weight,cost to lists
    w_cost.append(cost(f(x_train, w), y_train))
    w_status.append(w)

print ('iterations', nb_of_iterations)
print ('cost', w_cost[-1])
print ('best parameters', w_status[-1])
plot_results(x_train, y_train, w_status[-1])
```

Questão 2. *Use a decida do gradiente com learning rate de 1.e-4 (para convergir mais rápido). O que aconteceu?*

Quando o learning rate é 1e-4 la função não converge e o erro só aumenta:

- $erro = 5.22130488436\text{e+}89$
- $a = -3.55918263\text{e+}42$
- $b = 8.03113601\text{e+}24$
- $c = -1.69916825e\text{+}41$
- $d = -2.78549698\text{e+}23$

O plote da função:



Para obter os resultados mostrados so é necessário trocar a siguente linha no código da pergunta 1:

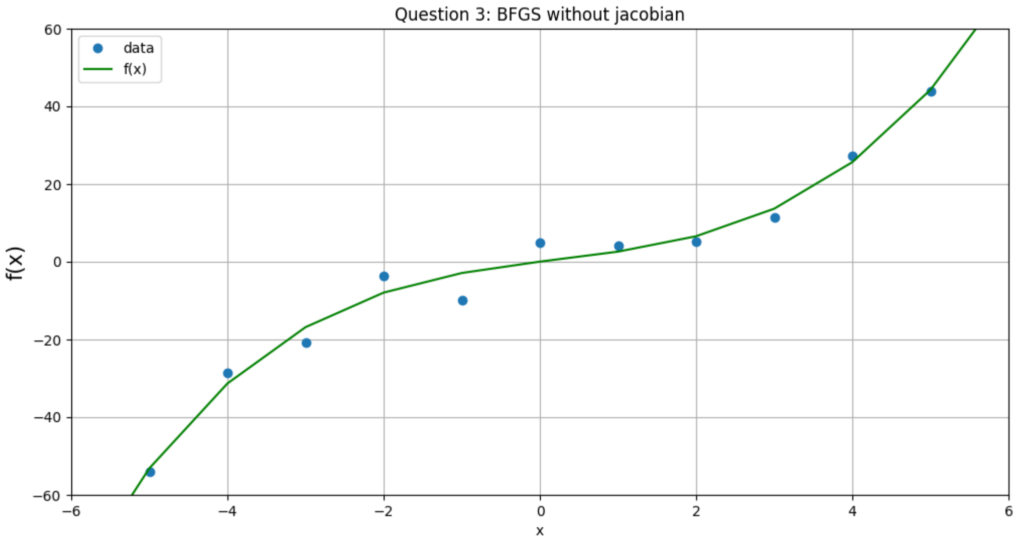
```
learning_rate = 1.0e-4
```

Questão 3. *Use o método de BFGS do scipy.optimize.minimize. Use o BFGS sem jacobiano (o método vai computar o Jacobiano usando diferenças finitas) De novo, imprima a solução, o erro e plote a função encontrada e os dados originais. Quantas interações foram precisas?*

Os resultados obtidos são:

- $erro = 127.565070$
- $a = 0.29123927$
- $b = -0.17995339$
- $c = 2.45957736$
- $d = 0.03589753.$
- **Iterations = 7**
- $Function\ evaluations = 66$
- $Gradient\ evaluations = 11$

O plote da função:



O código:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

#define the original function
def f(x, w):
  return w[0]*(x**3) + w[1]*(x**2) + w[2]*(x) + w[3]

#objetive fuction
def cost(w, x, y):
  return ((f(x, w) - y)**2).sum()

def plot_results(x, y, w):
  fig = plt.figure(figsize=(12, 6))
  fig.add_subplot(111)

  # Plot the target t versus the input x
  plt.plot(x, y, 'o', label='data')
  # Plot the initial line
  x = list(range(-10, 10))
  plt.plot(x, [f(i,w) for i in x],'g-',label='f(x)')

  #set extra plot parameters
  plt.title('Question 3: BFGS without jacobian')
  plt.xlim(-6,6)
  plt.ylim(-60,60)
  plt.xlabel('x')
  plt.ylabel('f(x)', fontsize=15)
  plt.legend(loc=2)
  plt.grid()
```

```
    plt.show()
    fig.savefig("t01_3.png")

if __name__ == "__main__":

    x_train = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
    y_train = np.array([-53.9, -28.5, -20.7, -3.6, -9.8, 5.0,
                         4.2, 5.1, 11.4, 27.4, 44.0])

    # initial guess
    w = np.array([0, 0, 0 , 0])

    # minimize fuction
    res = minimize(cost, w, args=(x_train, y_train, ),
              method='BFGS', jac = False, options={'disp': True})

    # show results
    print (res)
    plot_results(x_train, y_train, res.x)
```

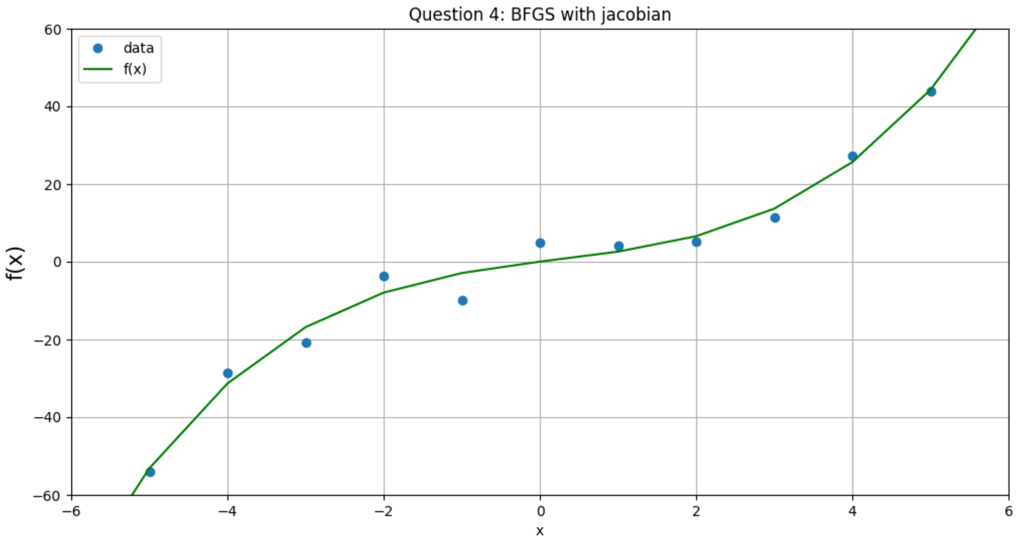
Questão 4. *Use o método de BFGS do scipy.optimize.minimize. Use o Jacobiano (gradiente da solução anterior. Houve diferença entre a solução anterior? Mesmo número de chamadas para a função?*

Os resultados obtidos são:

- $erro = 127.565070$
- $a = 0.29123932$
- $b = -0.17995338$
- $c = 2.45957653$
- $d = 0.03589744$.
- **Iterations = 7**
- $Function\ evaluations = 11$
- $Gradient\ evaluations = 11$

No caso do erro não tem diferencia alguma, exite uma diferencia nos parametros a, b, c e d, mas esta é muito pequena (no máximo 1e-5). No caso do BFGS com jacobiano, tive-se só 11 chamadas e no BFGS sim jacobiano 66 chamadas (55 chamadas de mais).

O plote da função:



Para obter os resultados mostrados solo é necessário criar a função jacobiano (está baseada no gradiente):

```
#return the jacobian (gradient)
def jacobian(w, x, y):
  fact = np.vstack( (x*x*x, x*x, x , np.ones(len(x))) )
  return (2 * (f(x, w) - y) * fact).sum(axis = 1)
```

E trocar a seguinte linha no código da pergunta 3:

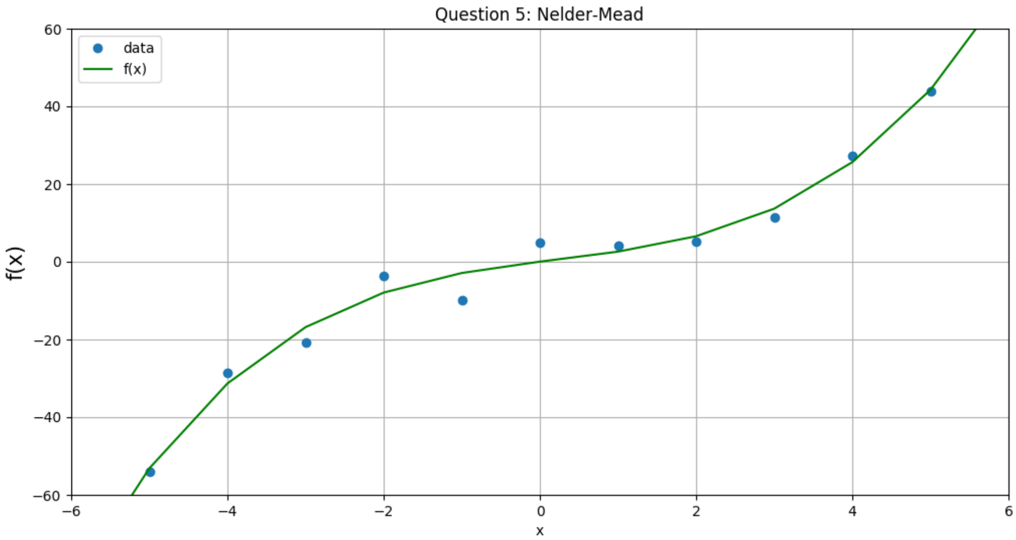
```
res = minimize(cost, w, args=(x_train, y_train, ),
        method='BFGS', jac = jacobian, options={'disp': True})
```

Questão 5. *Use o método Nelder Mead do scipy.optimize.minimize. Imprima e plote. Quantas interações?*

Os resultados obtidos são:

- $erro = 127.565070$
- $a = 0.29124112$
- $b = -0.17995509$
- $c = 2.45954951$
- $d = 0.03587784$.
- **Iterations = 360**
- $Function\ evaluations = 602$

O plote da função:



Para obter os resultados mostrados solo é necessário trocar a siguente linha no codigo da pergunta 3:

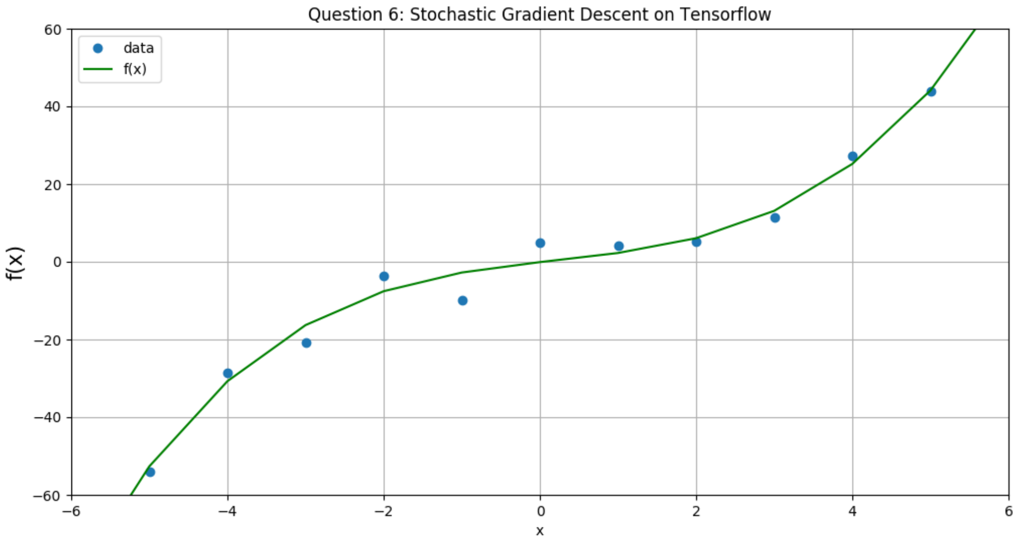
```
res = minimize(cost, w, args=(x_train, y_train, ),
          method='nelder-mead', options={'disp': True})
```

Questão 6. *Implemente uma solução usando decida do gradiente usando o Tensorflow. Use o otimizador AdamOptimizer com learning rate de 0.01. Rode 200 iterações. Plote a solução.Voce pode tanto usar uma otimização SGD (atualização dos pesos um dado por vez como é mais comum em redes neurais) ou uma solução batch (atualização acontece após processar todos os dados). Mas voce precisa dizer na resposta qual das duas voce implementou.*

Os resultados obtidos são:

- $erro = 127.565070$
- $a = 0.29123932$
- $b = -0.17995338$
- $c = 2.45957653$
- $d = 0.03589744.$

O plote da função:



O código implementa o SGD:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

#define the original function
def f(x, w):
  return w[0]*(x**3) + w[1]*(x**2) + w[2]*(x) + w[3]

def plot_results(x, y, w):
  fig = plt.figure(figsize=(12, 6))
  fig.add_subplot(111)

  # Plot the target t versus the input x
  plt.plot(x, y, 'o', label='data')
  # Plot the initial line
  x = list(range(-10, 10))
  plt.plot(x, [f(i,w) for i in x],'g-',label='f(x)')

  #set extra plot parameters
  plt.title('Question 6: Stochastic Gradient Descent on Tensorflow')
  plt.xlim(-6,6)
  plt.ylim(-60,60)
  plt.xlabel('x')
  plt.ylabel('f(x)', fontsize=15)
  plt.legend(loc=2)
  plt.grid()

  plt.show()
  fig.savefig("t01_6.png")
```

```python
if __name__ == "__main__":

  #Model parameters
  a = tf.Variable([np.random.rand()], tf.float32)
  b = tf.Variable([np.random.rand()], tf.float32)
  c = tf.Variable([np.random.rand()], tf.float32)
  d = tf.Variable([np.random.rand()], tf.float32)

  # Our model of y = a*x^3 + b*x^2 + c*x + d
  x = tf.placeholder(tf.float32)
  cubic_model = a*x*x*x + b*x*x + c*x + d
  y = tf.placeholder(tf.float32)

  # Our error is defined as the sum of the squares
  loss = tf.reduce_sum(tf.square(cubic_model - y))

  # Defining AdamOptimizer to calulate gradient
  optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
  train = optimizer.minimize(loss)

  #training data
  x_train = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
  y_train = np.array([-53.9, -28.5, -20.7, -3.6, -9.8, 5.0,
                      4.2, 5.1, 11.4, 27.4, 44.0])

  # number of gradient descent iterations
  nb_of_iterations = 200

  # training loop
  init = tf.global_variables_initializer()
  sess = tf.Session()
  sess.run(init) # reset values to wrong

  errors = []

  #SGD optimization with random order
  order = list(range(len(x_train)))
  for i in range(nb_of_iterations):
    np.random.shuffle(order)
    for j in order:
      x_value = x_train[j]
      y_value = y_train[j]
      sess.run(train, {x:x_value, y:y_value})
    errors.append(sess.run(loss, {x:x_train, y:y_train}))

  #recover solution
  curr_a, curr_b, curr_c, curr_d, curr_loss  =  sess.run(
   [a, b, c, d, loss], {x:x_train, y:y_train})
```

```
# show solution
print ('iterations', nb_of_iterations)
print ('cost', curr_loss)
print ('best parameters', curr_a, curr_b, curr_c, curr_d)
plot_results(x_train, y_train, np.array([curr_a, curr_b,
                                         curr_c, curr_d]))
```

Cálculo do Gradiente

Se ξ é a função de custo, então:

$\frac{\partial\xi}{\partial w} = \frac{\partial y}{\partial w}\frac{\partial\xi}{\partial y}$

$\frac{\partial\xi}{\partial y} = \frac{\partial(y-f(x))^2}{\partial y} = 2(f(x)-y)$

No caso do $\frac{\partial y}{\partial w}$, este debe ser calculado respeito a: a,b,c,d

$\frac{\partial ax^3+bx^2+cx+d}{\partial a} = x^3$

$\frac{\partial ax^3+bx^2+cx+d}{\partial b} = x^2$

$\frac{\partial ax^3+bx^2+cx+d}{\partial c} = x$

$\frac{\partial ax^3+bx^2+cx+d}{\partial d} = 1$

Então $\frac{\partial\xi}{\partial w} = \begin{pmatrix} 2x^3(f(x)-y \\ 2x^2(f(x)-y \\ 2x(f(x)-y \\ 2(f(x)-y) \end{pmatrix}$