Práctica 1

En esta práctica, necesitamos completar tres funciones: Compute cost, Compute gradient y Gradient descent.

De la pregunta aprendimos que la fórmula de Compute cost es:

$$J(w,b) = \frac{1}{2m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

por lo tanto la implementación es de esta función es:

```
def compute_cost(x, y, w, b):
    m = x.shape[0]
    total_cost = sum((w * x + b - y) ** 2) / (2 * m)
    return total_cost
```

Luego, la fórmula de Compute gradient es:

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$$

y la implementación es de esta función es:

```
def compute_gradient(x, y, w, b):
    m = x.shape[0]
    dj_dw = sum((w * x + b - y) * x) / m
    dj_db = sum(w * x + b - y) / m

    return dj_dw, dj_db
```

En el último, la fórmula de Gradient descent es:

$b = b - \alpha\frac{\partial}{\partial b}J(w,b)$ como $\frac{\partial}{\partial b}J(w,b)$ se calcula por la función compute_gradient.

$w = w - \alpha\frac{\partial}{\partial w}J(w,b)$ como $\frac{\partial}{\partial w}J(w,b)$ se calcula por la función compute_gradient.

En cada iteración los valores de w y b están un paso más cerca de los valores correctos, así que su implementación es:

```python
def gradient_descent(x, y, w_in, b_in, cost_function, gradient_function, alpha, 
num_iters):
    w = w_in
    b = b_in
    J_history = np.zeros(num_iters)

    for i in range(num_iters):
        dj_dw, dj_db = gradient_function(x, y, w, b)
        w = w - alpha * dj_dw
        b = b - alpha * dj_db
        J_history[i] = cost_function(x, y, w, b)

    return w, b, J_history
```

Código Completo

```python
import numpy as np
import copy
import math

import matplotlib.pyplot as plt
import linear_reg as lr
import public_tests as pt
import utils as ut

########################################################################
# Cost function
#
def compute_cost(x, y, w, b):
    """
    Computes the cost function for linear regression.

    Args:
        x (ndarray): Shape (m,) Input to the model (Population of cities)
        y (ndarray): Shape (m,) Label (Actual profits for the cities)
        w, b (scalar): Parameters of the model

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear 
regression
               to fit the data points in x and y
    """
    m = x.shape[0]
    total_cost = sum((w * x + b - y) ** 2) / (2 * m)
    return total_cost


########################################################################
# Gradient function
#
def compute_gradient(x, y, w, b):
    """
```

```
    Computes the gradient for linear regression
    Args:
      x (ndarray): Shape (m,) Input to the model (Population of cities)
      y (ndarray): Shape (m,) Label (Actual profits for the cities)
      w, b (scalar): Parameters of the model
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameter b
     """
    m = x.shape[0]
    dj_dw = sum((w * x + b - y) * x) / m
    dj_db = sum(w * x + b - y) / m

    return dj_dw, dj_db


#########################################################################
# gradient descent
#
def gradient_descent(x, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      x :    (ndarray): Shape (m,)
      y :    (ndarray): Shape (m,)
      w_in, b_in : (scalar) Initial values of parameters of the model
      cost_function: function to compute cost
      gradient_function: function to compute the gradient
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run gradient descent
    Returns
      w : (ndarray): Shape (1,) Updated values of parameters of the model after
          running gradient descent
      b : (scalar) Updated value of parameter of the model after
          running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each iteration,
          primarily for graphing later
    """
    w = w_in
    b = b_in
    J_history = np.zeros(num_iters)

    for i in range(num_iters):
        dj_dw, dj_db = gradient_function(x, y, w, b)
        w = w - alpha * dj_dw
        b = b - alpha * dj_db
        J_history[i] = cost_function(x, y, w, b)

    return w, b, J_history


# Generate the image
def generate_img(x, y, w, b):
```

```
    plt.scatter(x, y, c='r', marker='x', label='Data')
    plt.plot(x, w * x + b, label='Linear regression')
    plt.xlabel('Population of City in 10,000s')
    plt.ylabel('Profit in $10,000')
    plt.title('Profits vs. Population per city')
    plt.legend()
    plt.show()


# Test function
def test():
    pt.compute_cost_test(compute_cost)
    pt.compute_gradient_test(compute_gradient)


# Main function
def main():
    x,y = ut.load_data()
    alpha = 0.01
    num_iters = 1500
    w_in = 0
    b_in = 0
    w_out, b_out, J_history = lr.gradient_descent(x, y, w_in, b_in,
lr.compute_cost, lr.compute_gradient, alpha, num_iters)
    print('w_out:', w_out, 'b_out:', b_out)

    # Predictions
    print('profit would be in areas of 35,000 people: $', w_out * 35000 + b_out)
    print('profit would be in areas of 70,000 people: $', w_out * 70000 + b_out)
    generate_img(x, y, w_out, b_out)


if __name__ == "__main__":
    test()
    main()
```

Salida

```
tests passed!
Using X with shape (4, 1)
All tests passed!
w_out: 1.166362350335582 b_out: -3.63029143940436
profit would be in areas of 35,000 people: $ 40819.05197030597
profit would be in areas of 70,000 people: $ 81641.73423205134
```