Practica 5: Neural Networks Learning

Alberto Muñoz Fernández

Óscar García Castro

```
import numpy as np
import scipy.io as sc
import scipy.optimize as scop
import utils
import logistic_reg as lgr

def feed_forward(theta1, theta2, X):        #Size in this example:
    a1 = np.c_[np.ones(len(X)), X]          #Size(5000 * 401)
    z2 = np.dot(theta1, a1.T)
    a2 = lgr.sigmoid(z2)
    a2 = np.c_[np.ones(len(a2[0])), a2.T]   #Size(5000 * 26)
    z3 = np.dot(theta2, a2.T)
    a3 = lgr.sigmoid(z3)
    a3 = a3.T                               #Size(5000 * 1)
    return a3, a2, a1

def cost(theta1, theta2, X, y, lambda_):
    """
    Compute cost for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of
dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
```

```
    -------
    J : float
        The computed value for the cost function.

    """
    p, p2, p1 = feed_forward(theta1, theta2, X)
    m = len(X)

    cost = np.sum(y * np.log(p) + (1-y) * np.log(1 - p))
    rCost = np.sum(theta1[:, 1:]**2) + np.sum(theta2[:, 1:]**2)

    return -cost/m + ((lambda_/(2*m))*rCost)



def backprop(theta1, theta2, X, y, lambda_):
    """
    Compute cost and gradient for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of
dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    grad1 : array_like
        Gradient of the cost function with respect to weights
        for the first layer in the neural network, theta1.
        It has shape (2nd hidden layer size x input size + 1)
```

```
    grad2 : array_like
        Gradient of the cost function with respect to weights
        for the second layer in the neural network, theta2.
        It has shape (output layer size x 2nd hidden layer size + 1)

    """
    grad1 = np.zeros([len(theta1), len(theta1[0])])
    grad2 = np.zeros([len(theta2), len(theta2[0])])

    m = len(X)
    for i in range(m):
        a3, a2, a1 = feed_forward(theta1, theta2, [X[i]])

        sigma3 = a3 - y[i]
        gPrima = a2 * (1 - a2)
        sigma2 = np.dot(sigma3, theta2) * gPrima

        sigma = sigma2[:, 1:]

        grad1 += np.dot(sigma.T, a1)
        grad2 += np.dot(sigma3.T, a2)

    grad1[:,1:] += lambda_*theta1[:,1:]
    grad2[:,1:] += lambda_*theta2[:,1:]

    return (cost(theta1, theta2, X, y, lambda_), grad1/m, grad2/m)

def gradiant_descend(theta1, theta2, X, y, iter, alpha = 0, lambda_ = 0):
    for i in range (iter):
        c, th1, th2 = backprop(theta1, theta2, X, y, lambda_)
        theta1 -= alpha*th1
        theta2 -= alpha*th2
        print(i)

    return theta1, theta2

def backprop_aux(thetas, X, y, lambda_):
    th1 = np.reshape(thetas[:25 * (len(X[0]) + 1)], (25, len(X[0])+1))
    th2 = np.reshape(thetas[25 * (len(X[0]) + 1):], (len(y[0]), 26))
    c, g1, g2 = backprop(th1, th2, X, y, lambda_)
    return c, np.concatenate([np.ravel(g1), np.ravel(g2)])

def main():
    data = sc.loadmat('data/ex3data1.mat', squeeze_me=True)
    y = data['y']
    y_hot = np.zeros([len(y), 10])
    for i in range(len(y)):
        y_hot[i][y[i]] = 1
```

```
    X = data['X']

    # weights = sc.loadmat('data/ex3weights.mat')
    # theta1, theta2 = weights['Theta1'], weights['Theta2']
    # c = cost(theta1, theta2, X, y_hot, 1)
    # print(c)
    #utils.checkNNGradients(backprop, 1)

    theta1 = np.random.random((25, len(X[0]) + 1)) * (2*0.12)  - 0.12
    theta2 = np.random.random((10, 26))  * (2*0.12)  - 0.12

    theta1, theta2 = gradiant_descend(theta1, theta2, X, y_hot, 1000, 1,
1)

    cont =0
    for i in range(len(y)):
        if(yP[i] == y[i]):
             cont += 1
    print("Gradiant descent cost: ", cont / len(y) * 100, "%")

    theta1 = np.random.random((25, len(X[0]) + 1)) * (2*0.12)  - 0.12
    theta2 = np.random.random((10, 26))  * (2*0.12)  - 0.12

    arr = np.concatenate([theta1.ravel(), theta2.ravel()])
    result = scop.minimize(backprop_aux, arr, args=(X, y_hot, 1),
method="TNC", jac=True, options={'maxiter': 100})

    theta1 = np.reshape(result.x[:25 * (len(X[0]) + 1)], (25,
len(X[0])+1))
    theta2 = np.reshape(result.x[25 * (len(X[0]) + 1):], (len(y_hot[0]),
26))

    yP = np.argmax(feed_forward(theta1, theta2, X)[0], 1)

    cont =0
    for i in range(len(y)):
        if(yP[i] == y[i]):
             cont += 1
    print("Scipy optimize cost: ", cont / len(y) * 100, "%")

main()
```