Entrenamiento de redes neuronales

Código

```
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

data = loadmat('p4/ex4data1.mat')
y = data['y']
X = data['X']

weights = loadmat('p4/ex4weights.mat')
theta1, theta2 = weights['Theta1'], weights['Theta2']

X_new = np.ones((5000, 401))
X_new[:, 1:] = X
size = len(y)

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

def initializeWeights(LIn, LOut):
    weights = np.random.uniform(low=-0.12, high=0.12, size=(LOut, 1 + LIn))
    return weights

def displayData(X):
    num_plots = int(np.size(X, 0)**.5)
    fig, ax = plt.subplots(num_plots, num_plots, sharex=True, sharey=True)
    plt.subplots_adjust(left=0, wspace=0, hspace=0)
    img_num = 0
    for i in range(num_plots):
        for j in range(num_plots):
            # Convert column vector into 20x20 pixel matrix
            # transpose
            img = X[img_num, :].reshape(20, 20).T
            ax[i][j].imshow(img, cmap='Greys')
            ax[i][j].set_axis_off()
            img_num += 1

    return (fig, ax)

def displayImage(im):
    fig2, ax2 = plt.subplots()
    image = im.reshape(20, 20).T
    ax2.imshow(image, cmap='gray')
    return (fig2, ax2)

sample = np.random.choice(X.shape[0], 100)
displayData(X[sample, :])

def encodeLabels(num_labels, labels):
    labels = np.array(labels)
    oneHot = np.zeros((labels.shape[0], num_labels))
```

```
    for i in range(labels.shape[0]):
        labels[i] = (labels[i] - 1) % 10
        if labels[i] == 10:
            oneHot[i][0] = 1
        else:
            oneHot[i][labels[i]] = 1
    return oneHot

def encodeLabelsForGradientChecking(num_labels, labels):
    labels = np.array(labels)
    oneHot = np.zeros((labels.shape[0], num_labels))
    for i in range(labels.shape[0]):
        if labels[i] == 10:
            oneHot[i][0] = 1
        else:
            oneHot[i][labels[i]] = 1
    return oneHot

params_rn = np.concatenate((theta1, theta2), axis=None)

def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y,
lambda_reg):
    theta1 = np.reshape(params_rn[:num_ocultas*(num_entradas + 1)], (num_ocultas,
(num_entradas + 1)))
    theta2 = np.reshape(params_rn[num_ocultas*(num_entradas + 1):], (num_etiquetas,
(num_ocultas + 1)))
    m = X.shape[0]
    y = encodeLabels(num_etiquetas, y)
    a2 = sigmoid(X.dot(theta1.T))
    a2_new = np.ones((a2.shape[0], a2.shape[1]+1))
    a2_new[:, 1:] = a2
    a3 = sigmoid(a2_new.dot(theta2.T))
    J = (1.0/m) * np.sum(np.sum((-y * np.log(a3)) - ((1 - y) * np.log(1 - a3))))
    regularization = (np.sum(np.sum(np.square(theta1[:,1:]))) +
np.sum(np.sum(np.square(theta2[:,1:])))) * (float(lambda_reg)/(2*m));
    J = J + regularization

    theta1Gradient = np.zeros(theta1.shape)
    theta2Gradient = np.zeros(theta2.shape)

    z2 = X.dot(theta1.T)
    a2 = np.hstack((np.ones((z2.shape[0], 1)), sigmoid(z2)))

    z3 = a2.dot(theta2.T)
    a3 = sigmoid(z3)

    delta3 = a3 - y

    delta2 = (theta2.T.dot(delta3.T)).T * np.hstack((np.ones((z2.shape[0], 1)),
derivative(z2)))
    delta2 = delta2[:, 1:]

    theta1Gradient = theta1Gradient + delta2.T.dot(X)
    theta2Gradient = theta2Gradient + delta3.T.dot(a2)

    theta1Gradient = (1/float(m)) * theta1Gradient
    theta2Gradient = (1/float(m)) * theta2Gradient
```

```
    theta1Gradient[:, 1:] = theta1Gradient[:, 1:] + (float(lambda_reg)/m)*theta1[:,
1:]
    theta2Gradient[:, 1:] = theta2Gradient[:, 1:] + (float(lambda_reg)/m)*theta2[:,
1:]

    gradients = np.concatenate((theta1Gradient, theta2Gradient), axis=None)

    return J, gradients

def backpropforGradientChecking(params_rn, num_entradas, num_ocultas,
num_etiquetas, X, y, lambda_reg):
    theta1 = np.reshape(params_rn[:num_ocultas*(num_entradas + 1)], (num_ocultas,
(num_entradas + 1)))
    theta2 = np.reshape(params_rn[num_ocultas*(num_entradas + 1):], (num_etiquetas,
(num_ocultas + 1)))
    m = X.shape[0]
    y = encodeLabelsForGradientChecking(num_etiquetas, y)
    X_new = np.ones((X.shape[0], X.shape[1]+1))
    X_new[:, 1:] = X
    a2 = sigmoid(X_new.dot(theta1.T))
    a2_new = np.ones((a2.shape[0], a2.shape[1]+1))
    a2_new[:, 1:] = a2
    a3 = sigmoid(a2_new.dot(theta2.T))
    J = (1.0/m) * np.sum(np.sum((-y * np.log(a3)) - ((1 - y) * np.log(1 - a3))))
    regularization = (np.sum(np.sum(np.square(theta1[:,1:]))) +
np.sum(np.sum(np.square(theta2[:,1:])))) * (float(lambda_reg)/(2*m));
    J = J + regularization

    theta1Gradient = np.zeros(theta1.shape)
    theta2Gradient = np.zeros(theta2.shape)

    z2 = X_new.dot(theta1.T)
    a2 = np.hstack((np.ones((z2.shape[0], 1)), sigmoid(z2)))

    z3 = a2.dot(theta2.T)
    a3 = sigmoid(z3)

    delta3 = a3 - y

    delta2 = (theta2.T.dot(delta3.T)).T * np.hstack((np.ones((z2.shape[0], 1)),
derivative(z2)))
    delta2 = delta2[:, 1:]

    theta1Gradient = theta1Gradient + delta2.T.dot(X_new)
    theta2Gradient = theta2Gradient + delta3.T.dot(a2)

    theta1Gradient = (1/float(m)) * theta1Gradient
    theta2Gradient = (1/float(m)) * theta2Gradient

    theta1Gradient[:, 1:] = theta1Gradient[:, 1:] + (float(lambda_reg)/m)*theta1[:,
1:]
    theta2Gradient[:, 1:] = theta2Gradient[:, 1:] + (float(lambda_reg)/m)*theta2[:,
1:]

    gradients = np.concatenate((theta1Gradient, theta2Gradient), axis=None)

    return J, gradients

result1 = backprop(params_rn, 400, 25, 10, X_new, y, 0)
```

```
print("Cost without regularization = " + str(result1[0]))
result2 = backprop(params_rn, 400, 25, 10, X_new, y, 1)
print("Cost with regularization = " + str(result2[0]))

def debugInitializeWeights(fan_in, fan_out):
    """
    Initializes the weights of a layer with fan_in incoming connections and
    fan_out outgoing connections using a fixed set of values.
    """

    # Set W to zero matrix
    W = np.zeros((fan_out, fan_in + 1))

    # Initialize W using "sin". This ensures that W is always of the same
    # values and will be useful in debugging.
    W = np.array([np.sin(w) for w in
                  range(np.size(W))]).reshape((np.size(W, 0), np.size(W, 1)))

    return W

def computeNumericalGradient(J, theta):
    """
    Computes the gradient of J around theta using finite differences and
    yields a numerical estimate of the gradient.
    """

    numgrad = np.zeros_like(theta)
    perturb = np.zeros_like(theta)
    tol = 1e-4

    for p in range(len(theta)):
        # Set perturbation vector
        perturb[p] = tol
        loss1 = J(theta - perturb)
        loss2 = J(theta + perturb)

        # Compute numerical gradient
        numgrad[p] = (loss2 - loss1) / (2 * tol)
        perturb[p] = 0

    return numgrad

def checkNNGradients(costNN, reg_param):
    """
    Creates a small neural network to check the back propogation gradients.
    Outputs the analytical gradients produced by the back prop code and the
    numerical gradients computed using the computeNumericalGradient function.
    These should result in very similar values.
    """
    # Set up small NN
    input_layer_size = 3
    hidden_layer_size = 5
    num_labels = 3
    m = 5

    # Generate some random test data
    Theta1 = debugInitializeWeights(hidden_layer_size, input_layer_size)
    Theta2 = debugInitializeWeights(num_labels, hidden_layer_size)
```

```
    # Reusing debugInitializeWeights to get random X
    X = debugInitializeWeights(input_layer_size - 1, m)

    # Set each element of y to be in [0,num_labels]
    y = [(i % num_labels) for i in range(m)]

    # Unroll parameters
    nn_params = np.append(Theta1, Theta2).reshape(-1)

    # Compute Cost
    cost, grad = costNN(nn_params,
                        input_layer_size,
                        hidden_layer_size,
                        num_labels,
                        X, y, reg_param)

    def reduced_cost_func(p):
        """ Cheaply decorated nnCostFunction """
        return costNN(p, input_layer_size, hidden_layer_size, num_labels,
                      X, y, reg_param)[0]

    numgrad = computeNumericalGradient(reduced_cost_func, nn_params)

    # Check two gradients
    np.testing.assert_almost_equal(grad, numgrad)
    return (grad - numgrad)

gradientChecking1 = checkNNGradients(backpropforGradientChecking, 0)
print("Gradient check without regularization = " + str(gradientChecking1))
gradientChecking2 = checkNNGradients(backpropforGradientChecking, 1)
print("Gradient check with regularization = " + str(gradientChecking2))

def neuralNetwork(X, theta1, theta2):
    a2 = sigmoid(X.dot(theta1.T))
    a2_new = np.ones((a2.shape[0], a2.shape[1]+1))
    a2_new[:, 1:] = a2
    a3 = sigmoid(a2_new.dot(theta2.T))
    predictions = np.zeros(len(a3))
    predictions = predictions.reshape(X.shape[0], 1)
    for i in range(len(a3)):
        idx = np.argmax(a3[i])
        idx = (idx + 1) % 10
        if(idx == 0):
            predictions[i] = 10
        else:
            predictions[i] = idx
    return predictions

thetaTrain1 = initializeWeights(400, 25)
thetaTrain2 = initializeWeights(25, 10)
thetasTrain = np.concatenate((thetaTrain1, thetaTrain2), axis=None)
resultTheta = opt.fmin_tnc(func=backprop, x0=thetasTrain, fprime=None, args=(400,
25, 10, X_new, y, 1), maxfun=70, disp=5)
resultTheta = resultTheta[0]
theta1 = np.reshape(resultTheta[:25*401], ((25, 401)))
theta2 = np.reshape(resultTheta[25*401:], ((10, 26)))

predictions = neuralNetwork(X_new, theta1, theta2)
number = np.sum(predictions == y)
```

```
accuracy = (float(number)/X.shape[0])*100
print("Accuracy (iterations = 70, lambda = 1) = " + str(accuracy) + "%")

for maxfun in np.arange(40, 100, 20):
    for lambda_reg in np.arange(0.5, 1.7, 0.3):
        lambda_reg = round(lambda_reg, 1)
        thetaTrain1 = initializeWeights(400, 25)
        thetaTrain2 = initializeWeights(25, 10)
        thetasTrain = np.concatenate((thetaTrain1, thetaTrain2), axis=None)
        resultTheta = opt.fmin_tnc(func=backprop, x0=thetasTrain, fprime=None,
args=(400, 25, 10, X_new, y, lambda_reg), maxfun=maxfun, disp=5)
        resultTheta = resultTheta[0]
        theta1 = np.reshape(resultTheta[:25*401], ((25, 401)))
        theta2 = np.reshape(resultTheta[25*401:], ((10, 26)))

        predictions = neuralNetwork(X_new, theta1, theta2)
        number = np.sum(predictions == y)
        accuracy = (float(number)/X.shape[0])*100
        print("Accuracy (iterations = " + str(maxfun) + ", lambda = " +
str(lambda_reg) + ") = " + str(accuracy) + "%")
```

Resultados

```
Cost without regularization = 0.2876291651613189
Cost with regularization = 0.38376985909092365
Gradient check without regularization = [ 5.27761168e-11 -3.29852682e-13
  7.89324855e-12  9.17629861e-12
 -6.08260803e-11  2.08457210e-12 -1.07556533e-11 -4.82478224e-11
 -9.29990251e-11  9.26888774e-12 -4.20321417e-11 -1.26826272e-10
 -2.17855178e-11  2.76548229e-12 -1.04682443e-11 -2.49761531e-11
  2.15736456e-11 -4.96176017e-13  9.77740458e-12  2.73879461e-11
  6.03760375e-11  1.32927558e-11  6.81166235e-12  3.04718750e-12
  1.67883762e-11  1.66236747e-11  6.93309576e-11  1.56080704e-11
  4.89146224e-12  1.15286947e-11  1.93192129e-11  1.79336823e-11
  7.55120411e-11  1.60134961e-11  8.61835603e-12  1.78092541e-11
  1.66117953e-11  2.04546380e-11]
Gradient check with regularization = [ 5.27761168e-11 -1.48769885e-12  8.82988127e-
12  9.75092229e-12
 -6.08260803e-11  2.10972906e-12 -1.16537890e-11 -4.70333217e-11
 -9.29990251e-11  7.81531784e-12 -4.12793688e-11 -1.26643918e-10
 -2.17855178e-11  2.13601359e-12 -9.22964483e-12 -2.43030873e-11
  2.15736456e-11  2.27595720e-13  9.77740458e-12  2.84505475e-11
  6.03760375e-11  1.38673795e-11  6.28552765e-12  3.07233405e-12
  1.58902475e-11  1.56177293e-11  6.93309576e-11  1.41545109e-11
  3.42378903e-12  1.17110488e-11  1.87037608e-11  1.95246597e-11
  7.55120411e-11  1.66865410e-11  8.55093774e-12  1.85330362e-11
  1.56829827e-11  2.22044327e-11]

Accuracy (iterations = 70, lambda = 1) = 94.26%

Accuracy (iterations = 40, lambda = 0.5) = 83.82%
Accuracy (iterations = 40, lambda = 0.8) = 84.98%
Accuracy (iterations = 40, lambda = 1.1) = 80.78%
Accuracy (iterations = 40, lambda = 1.4) = 84.42%
Accuracy (iterations = 60, lambda = 0.5) = 91.3%
Accuracy (iterations = 60, lambda = 0.8) = 86.22%
Accuracy (iterations = 60, lambda = 1.1) = 91.52%
Accuracy (iterations = 60, lambda = 1.4) = 92.5%
```

```
Accuracy (iterations = 80, lambda = 0.5) = 94.52%
Accuracy (iterations = 80, lambda = 0.8) = 94.42%
Accuracy (iterations = 80, lambda = 1.1) = 95.34%
Accuracy (iterations = 80, lambda = 1.4) = 94.98%
```

Comentarios

Cuando el número de iteraciones se aumenta, el valor de accuracy es mejor. Cuando lambda se aumenta, el valor de accuracy es mejor pero normalmente para valores de lambda demasiado grandes es peor. Normalmente para valores de lambda demasiado pequeños se ocurre overfitting y para valores demasiado grandes – underfitting. Esto no es bien visible en este ejemplo de entrenamiento.