1 PRACTICA 4

1.1 FUNCIÓN DE COSTE

```
def cost_function(theta1, theta2, x, y, a, numLabel, lam=1):
    X = np.ones(shape=(x.shape[0], x.shape[1] + 1))
    X[:, 1:] = x
    m = X.shape[0]
    y_aux = np.zeros((m, numLabel))
    cost = 0

    for i in range(1, numLabel + 1):
        y_aux[:, i - 1][:, np.newaxis] = np.where(y == i, 1, 0)

    for i in range(0, numLabel):
        label = y_aux[:, i]
        aux = a[:, i]
        error = -label * np.log(aux) - ((1 - label) * np.log(1 - aux))
        cost = cost + sum(error)

    J = 1 / m * cost
    reg_cost = J + lam / (2 * m) * (np.sum(theta1[:, 1:] ** 2) + np.sum(theta2[:, 1:] ** 2))
    return J, reg_cost
```

Los resultados obtenidos han sido, para diferentes valores de lambda:

Resultado sin Regularizar	Lambda
0.2876291651613188	1
0.2876291651613188	10
0.2876291651613188	100

Resultado con Regularización	Lambda
0.38376985909092354	1
0.38376985909092354	10
0.38376985909092354	100

Estos valores coinciden con los ejemplificados en el documento de la práctica.

1.2 CÁLCULO DEL GRADIENTE

Se derivó sigma:

```
def sig_dev_function(x):
    s = 1 / (1 + np.exp(-x))
    return s * (1 - s)
```

```
def sig_function(x):
    s = 1 / (1 + np.exp(-x))
    return s
```

1.2.1 Retro Propagación y regularización del gradiente

```
def gradient(params_ns, inputSize, hiddenSize, numLabel, x, y, lam=1):
    theta1 = params_ns[:((inputSize + 1) * hiddenSize)].reshape(hiddenSize, inputSize + 1)
    theta2 = params_ns[((inputSize + 1) * hiddenSize):].reshape(numLabel, hiddenSize + 1)

    delta1 = np.zeros(theta1.shape)
    delta2 = np.zeros(theta2.shape)

    X = np.ones(shape=(x.shape[0], x.shape[1] + 1))
    X[:, 1:] = x

    m = x.shape[0]

    a1, z2, a2, a3, h = forwardPropagation(x, theta1, theta2)
    for t in range(m):
        a1t = a1[t, :]  # (1, 401)
        a2t = a2[t, :]  # (1, 26)
        ht = h[t, :]  # (1, 10)
        yt = y[t]  # (1, 10)
        d3t = ht - yt  # (1, 10)
        d2t = np.dot(theta2.T, d3t) * (a2t * (1 - a2t))  # (1, 26)
        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])
        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    delta1 = 1 / m * delta1
    delta2 = 1 / m * delta2

    delta1Reg = delta1 + (lam / m) * np.hstack((np.zeros((theta1.shape[0], 1)), theta1[:, 1:]))
    delta2Reg = delta2 + (lam / m) * np.hstack((np.zeros((theta2.shape[0], 1)), theta2[:, 1:]))

    # Calculate Cost
    jVal, jValGrad = cf.cost_function(theta1=theta1, theta2=theta2, x=x, y=y, a=h,
numLabel=numLabel)

    deltaVec = np.concatenate((delta1Reg.ravel(), delta2Reg.ravel()))
    return jVal, deltaVec
```

1.2.2 Check Gradiente

En este apartado no conseguí aplicar la fórmula debido a como esta implementado la función de coste.

1.3 APRENDIZAJE DE PARÁMETROS

```
def backpropagationLearning(x, y, theta1, theta2, hiddenSize=25, numLabel=10, inputSize=400):
    # Initialize params
    params_ns = (np.random.random(size=hiddenSize * (inputSize + 1) +
                                    numLabel * (hiddenSize + 1)) - 0.5) * 0.25
    params_ns = np.concatenate((theta1.ravel(), theta2.ravel()))
```

```
fmin = minimize(fun=gradient,
                x0=params_ns,
                args=(inputSize, hiddenSize, numLabel, x, y, 1),
                method='TNC',
                jac=True,
                options={'maxiter': 70})

return fmin
```

Los valores por defecto se han mantenido, y los resultados obtenidos para la función sin regularizar son:

Coste	0.2876291651613188
Gradiente[0]	2.07430506
Fmin.x[0]	-2.25623899e-02
Accuracy Backward Propagation	97.52%

Los valores regularizados:

Coste	0.2876291651613188
Gradiente[0]	2.07430506
Fmin.x[0]	-2.25623899e-02
Accuracy Backward Propagation	97.52%