Jose Daniel Rave Robayo
Ángel López Benítez

Practica 5

Redes Neuronales| Aprendizaje
Entrenamiento de pesos (ϴ)

Y descenso de gradiente

Para esta práctica hemos implementado principalmente el Backpropagation, la base del descenso de gradiente. Además del método cost para computar el costo en cada iteración del descenso.

Para empezar, el Compute Cost utiliza el feedForward, un método de predicción que nos será útil para calcular el error de con respecto a la salida esperada. Además de que usamos el termino de regularización aplicado a los pesos de las capas. En este caso dos.

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

$$+ \frac{\lambda}{2m}\left[\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} \left(\Theta_{j,i}^{(l)}\right)^2\right]$$

Implementación:

```
def cost(theta1, theta2, X, y, lambda_):
    J = 0
    m = y.shape[0]

    #Predecimos
    h0 = mC.feedForward(theta1, theta2, X)[0]
    fun_val = h0

    log_cost = np.log(fun_val)
    log_cost2 = np.log(1 - fun_val)
    #Computamos el coste por cada capa de la red neuronal.
    J += np.sum((-y * log_cost) -  (1 - y) * log_cost2)

    #Dividimos por el numero de datos
    J = J /m

    #Añadimos el término de regularizacion
    reg_tem = lambda_/(2*m) * (np.sum(theta1[:, 1:]** 2) +
np.sum(theta2[:, 1:]** 2) )

    return J + reg_tem
```

Jose Daniel Rave Robayo
Ángel López Benítez

Ahora, para el BackPropagation, el objetivo es entrenar los pesos sobre una base. Haremos una pequeña predicción inicial para empezar a entrenar desde ahí. Usando el FeedPropagation obtenemos los datos necesarios para calcular los errores iterando sobre todos los datos.

Por último, dividiremos todos los valores de cada peso por el número de datos. No obstante, aplicaremos lambda sobre todos aquellos valores que no son la primera columna, ya que ésta representa el termino independiente B (se añade al peso Θ como un pequeño truco para tener todo condensado y facilitar los cálculos).

Implementación:

```python
def backprop(theta1, theta2, X, y, lambda_):
    grad1 = grad2 = 0

    m = X.shape[0]

    J = cost(theta1, theta2,X, y, lambda_)

    #Predicción inicial
    a3, a2, a1, p = mC.feedForward(theta1, theta2, X)

    for i in range(m):
        error_3 = a3[i] - y[i]
        g_primeZ = a2[i] * (np.ones(a2[i].shape)- a2[i])

        error_2 = np.dot(theta2.T,error_3) * g_primeZ.T
        error_2 = error_2[1:] #eliminar primera columna

        #Entrenamos los pesos
        grad1 += np.matmul(error_2[:, np.newaxis], a1[i][np.newaxis, :])
        grad2 += np.matmul(error_3[:, np.newaxis], a2[i][np.newaxis, :])

    #Dividimos por el número de datos y aplicamos lambda sobre cada peso
    #omitimos la primera columna ya que dicha es el termino independiente B.
    grad1[:, 1:] = (1/m) * grad1[:,1:] + (1/m)*lambda_*theta1[:, 1:]
    grad1[:, 0] = (1/m) * grad1[:, 0]

    grad2[:, 1:] = (1/m) * grad2[:,1:] + (1/m)*lambda_*theta2[:, 1:]
    grad2[:, 0] = (1/m) * grad2[:, 0]

    return (J, grad1, grad2)
```

Para comprobar nuestra implementación, usamos utils.checkNNGradients para calcular la diferencia de resultado con respecto a un BackPropagation correcto de la librería **NumPy.**

```
PS C:\Users\joseda\Desktop\Uni\Cuarto\APA\AprendizajeAutomaticoP0\P5> python .\main.py
If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).
Relative Difference: 2.57286e-11
```

Ahora sí, crearemos el Descenso de Gradiente usando BackPropagation. El nivel de acierto debería estar aproximadamente por el 95% con cualquier valor de lambda [0, 1].

Básicamente inicializamos los pesos a valores aleatorios entre [-0’12 , 0’12], ejecutamos el BackPropagation un número de iteración fijo y con los pesos entrenados al finalizar hacemos una predicción sobre los datos de entrada X.

```
def learningParameters(X, Y, Y_encoded, lambda_, alpha, iterations):
    theta1, theta2 = initTheta(X, Y_encoded, 0.12)
    LearnedTheta1, LearnedTheta2_, cost = neuralNet.gradient_descent(X,
Y_encoded, theta1, theta2, neuralNet.backprop, alpha, iterations, lambda_)

    result = mC.feedForward(LearnedTheta1, LearnedTheta2_, X)[3]

    percentage = compareEquals(Y, result)

    print(f"Precision: {percentage}%")
```

```
def gradient_descent(X, y, theta1_in, theta2_in, gradient_function, alpha,
num_iters, lambda_=None):
    J_history = []
    theta1 = copy.deepcopy(theta1_in)
    theta2 = theta2_in

    for i in range(num_iters):
      cost, dj_dj1, dj_dj2 = gradient_function(theta1, theta2, X, y,lambda_)
      theta1 -= alpha * dj_dj1
      theta2 -= alpha * dj_dj2

      if i < 100000:
        J_history.append(cost)

    return theta1, theta2, J_history
```

El resultado es el siguiente:

```
PS C:\Users\joseda\Desktop\Uni\Cuarto\APA\AprendizajeAutomaticoP0\P5> python .\main.py
Precision: 95.36%
```

Ahora, para contrastar, realizaremos un entrenamiento, pero esta vez usando una implementación de la clase SciPy. En concreto usaremos la función *minimize* que, dada una función, encuentra su valor mínimo. Para usar esta función es necesario que los pesos iniciales sean pasados como un array unidimensional. Por este motivo es que enrollamos los datos para introducirlos en la función, y una vez dentro los desenrollamos. El entrenamiento se hará sobre 100 iteraciones y tendrá que tener un valor de acierto similar a nuestra implementación, variando en 1% arriba/abajo debido a la aleatoriedad de los pesos iniciales.

```
PS C:\Users\joseda\Desktop\Uni\Cuarto\APA\AprendizajeAutomaticoP0\P5> python .\main.py
Precision: 94.02000000000001%
```

Jose Daniel Rave Robayo
Ángel López Benítez

Implementación:

```
def learnParametersSciPy(X, Y, Y_encoded, lambda_, num_iters):
    theta1, theta2 = initTheta(X, Y_encoded, 0.12)

    layers = 25
    m = Y_encoded.shape[1]
    n = X.shape[1]

    #Enrrollamos los pesos en un array unidimensional
    thetas = np.concatenate([theta1.ravel(), theta2.ravel()])

    #Pasamos la función a minimizar (BackPropagation) y sus datos
    result = minimize(fun=backpropAuxForMinimize, x0=thetas, args=(layers, X,
Y_encoded, lambda_), method='TNC', jac=True, options={'maxiter': num_iters})

    #Desenrrollamos para poder realizar una prediccion sobre los pesos resultado.
    theta1 = np.reshape(result.x[:layers * (n+1)], (layers, n+1))
    theta2 = np.reshape(result.x[layers * (n+1):], (m, layers+1))

    result = mC.feedForward(theta1, theta2, X)[3]
    percentage = compareEquals(Y, result)
    print(f"Precision: {percentage}%")
```

```
def backpropAuxForMinimize(thetas, layers, X, Y, lambda_):
    m = Y.shape[1]
    n = X.shape[1]

    #Desenrrollamos para poder utilizar nuestra funcion
    theta1 = np.reshape(thetas[:layers * (n+1)], (layers, n+1))
    theta2 = np.reshape(thetas[layers * (n+1):], (m, layers+1))

    J, grad1, grad2 = neuralNet.backprop(theta1, theta2, X, Y, lambda_)

    return J , np.concatenate([np.ravel(grad1), np.ravel(grad2)])
```