Memoria P5. Entrenamiento Redes Neuronales

Propagación prealimentada

La función **feed_forward** implementa el algoritmo de propagación prealimentada que utilizamos en el método **predict** de la **Práctica 4**.

```
def feed_forward(theta1, theta2, X):
    m = X.shape[0]

    a1 = np.column_stack([np.ones((m, 1)), X])  # 5000 x 401
    z2 = a1 @ theta1.T                          # 5000 x 401 @ 401 x 25

    a2 = ut.sigmoid(z2)
    a2 = np.column_stack([np.ones((m, 1)), a2]) # 5000 x 26
    z3 = a2 @ theta2.T                          # 5000 x 26 @ 26 x 10

    a3 = ut.sigmoid(z3)                         # 5000 x 10

    return a3, a2, a1
```

Cómputo del costo

La función **cost** se utiliza para calcular el coste de una red neuronal de dos capas, y recibe los siguientes parámetros:

- **theta1** : conjunto de pesos de la primera capa de la red.
- **theta2**: conjunto de pesos de la segunda capa de la red.
- **X** : conjunto de ejemplos.
- **y** : conjunto de etiquetas.
- **lambda_**: constante de regularización.

La función comienza definiendo una variable **m** del tamaño del conjunto **X**. Después, se calculan los valores de salida de la red neuronal utilizando la función **feedforward** y tomando la última capa, guardándose en la variable **ho**.

Utilizamos estos valores para calcular el coste total de la red **cost** como la suma de los costes de cada uno de los elementos. A continuación calculamos el factor de regularización **reg_factor** utilizando los conjuntos de pesos **theta1** y **theta2**.

Por último calculamos el coste promedio dividiendo el coste total **cost** entre el número de elementos **m**, y añadiendo la regularización. Devolvemos este valor.

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

    h0, _, _ = feed_forward(theta1, theta2, X)

    cost = np.sum((y * np.log(h0)) + ((1 - y) * np.log(1 - h0)))
    reg_factor = np.sum(pow(theta1[:, 1:], 2)) + np.sum(pow(theta2[:, 1:], 2))
    J = (-cost / m) + (lambda_ * reg_factor / (2 * m))

    return J
```

Propagación retroalimentada

La función **backprop** calcula el coste y el gradiente de coste de una red neuronal de dos capas. Recibe los siguientes parámetros:

- **theta1** : conjunto de pesos de la primera capa de la red.
- **theta2**: conjunto de pesos de la segunda capa de la red.
- **X** : conjunto de ejemplos.
- **y** : conjunto de etiquetas.
- **lambda_**: constante de regularización.

La función comienza definiendo una variable **m** del tamaño del conjunto **X**. A continuación se calcula el coste de la red neuronal usando la función **cost** y se inicializan a 0 las matrices de gradiente **grad1** y **grad2.**

A continuación iteramos a través de los elementos del conjunto **X**, utilizando la función **feed_forward** para implementar propagación prealimentada, y después implementamos el algoritmo de propagación retroalimentada. Calculamos el error **d3** de la capa de salida **a3**, y lo utilizamos para calcular el error **d2** de la capa intermedia **a2**. Tras esto actualizamos las matrices de gradiente **grad1** y **grad2.**

Por último, la función divide cada elemento de **grad1** y **grad2** por el número de ejemplos de entrenamiento **m**, y aplica la regularización a cada uno de estos elementos, excepto al primero de cada fila. La función devuelve el valor de la función de costo **J**, y las matrices de gradiente **grad1** y **grad2**.

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

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

    grad1 = np.zeros((theta1.shape[0], theta1.shape[1]))
    grad2 = np.zeros((theta2.shape[0], theta2.shape[1]))
    for i in range(m):
        a3, a2, a1 = feed_forward(theta1, theta2, X[i, np.newaxis])

        d3 = a3 - y[i]                          # 1 x 10

        gZ = (a2 * (1 - a2))                    # 1 x 26
        d2 = d3 @ theta2 * gZ                   # 1 x 10 @ 10 x 26 * 1 x 26
        d2 = d2[:, 1:]                          # 1 x 25

        grad1 += d2.T @ a1                      # 25 x 1 @ 1 x 401
        grad2 += d3.T @ a2                      # 10 x 1 @ 1 x 26

    grad1[:, 0] /= m
    grad2[:, 0] /= m

    grad1[:, 1:] = (grad1[:, 1:] + lambda_ * theta1[:, 1:]) / m
    grad2[:, 1:] = (grad2[:, 1:] + lambda_ * theta2[:, 1:]) / m

    return J, grad1, grad2                      # 25 x 401; 10 x 26
```

Gradiente sin optimizar

La función **gradient** se utiliza para ajustar los pesos de la red neuronal. Recibe los siguientes parámetros:

- **theta1**: conjunto de pesos de la primera capa de la red.
- **theta2**: conjunto de pesos de la segunda capa de la red.
- **X** : conjunto de ejemplos.
- **y** : conjunto de etiquetas.
- **num_iters**: número de iteraciones a realizar.
- **alpha**: factor de aprendizaje.

La función itera **num_iters** veces. En cada iteración se obtiene el gradiente de coste de cada capa utilizando la función **backprop** y se utiliza para ajustar los pesos **theta1** y **theta2** multiplicando las matrices de gradiente **grad1** y **grad2** por el factor de aprendizaje **alpha**.

```
# unoptimized way to obtain gradient
def gradient(theta1, theta2, X, y, num_iters, alpha, lambda_=0):
    for _ in range(num_iters):
        J, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)

        theta1 -= alpha * grad1
        theta2 -= alpha * grad2

    return J, theta1, theta2
```

Gradiente optimizado

La función **backprop_min** se utiliza para obtener el gradiente de coste de la red neuronal y devolver esta información, Es aplicada por la función **sciopt.minimize**. Recibe los siguientes parámetros:

- **theta**: conjunto de pesos de todas las capas de la red.
- **X** : conjunto de ejemplos.
- **y** : conjunto de etiquetas.
- **t1_s**: tamaño de la primera capa de la red neuronal.
- **t2_s**: tamaño de la segunda capa de la red neuronal.
- **lambda_**: constante de regularización.

La función empieza descomprimiendo el conjunto de pesos **theta** en los pesos de la primera capa (**theta1**) y los de la segunda (**theta2**). A continuación, utiliza la función **backprop** para obtener los gradientes de coste de la red neuronal. Por último esta función devuelve el valor del coste **J**, y las matrices de gradientes **grad1** y **grad2** concatenadas como una sola matriz.

```
def backprop_min(theta, X, y, t1_s, t2_s, lambda_):
    # reshape to adapt to the minimize function
    theta1 = np.reshape(theta[:t1_s[0] * t1_s[1]], (t1_s[0], t1_s[1]))
    theta2 = np.reshape(theta[t1_s[0] * t1_s[1]:], (t2_s[0], t2_s[1]))

    # train neural network
    J, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)

    # return results (adapted again to the sciopt.minimize function)
    return J, np.concatenate([np.ravel(grad1), np.ravel(grad2)])
```

Cómputo de la red neuronal

La función **apply_nn** se utiliza para obtener el resultado del entrenamiento de la red neuronal. Recibe los siguientes parámetros:

- **X** : conjunto de ejemplos.
- **y_onehot** : conjunto de etiquetas.
- **num_iters**: número de iteraciones a realizar.
- **alpha**: factor de aprendizaje.
- **lambda_**: constante de regularización.
- **opt**: selector de modo optimizado.

Primero, obtenemos **theta1** y **theta2** [ambos de tamaño $\mathbf{s_{i+1}}$*$\mathbf{s_i+1}$], rellenandolas de valores aleatorios entre los valores (**-epsilon, epsilon**).

En caso de elegir la opción sin optimizar, aplicaremos la función **gradient** previamente explicada; mientras que, en caso de elegir la opción optimizada, concatenamos los valores de las **thetas** obtenidas, llamamos a la función **sciopt.minimize** con los valores pertinentes, obtenemos el resultado, y lo descomprimimos en las **thetas** a devolver.

```
def apply_nn(X, y_onehot, num_iters, alpha, lambda_, opt=True):
    # initialize thetas with a random factor
    epsilon = 0.12           # s[curr_l + 1]    s[curr_l] + 1
    theta1 = np.random.random((theta1.shape[0], theta1.shape[1])) * (2 *
epsilon) - epsilon
    theta2 = np.random.random((theta2.shape[0], theta2.shape[1])) * (2 *
epsilon) - epsilon

    if opt == False:    # unoptimized
        _, tetha1, tetha2 = nn.gradient(theta1, theta2, X, y_onehot,
num_iters, alpha, lambda_)
    else:               # optimized
        theta = np.concatenate([np.ravel(theta1), np.ravel(theta2)])
        res = sciopt.minimize(fun=nn.backprop_min, x0=theta, args=(X,
y_onehot, theta1.shape, theta2.shape, lambda_), method='TNC', jac=True,
options={'maxiter': num_iters})
        theta1 = np.reshape(res.x[:theta1.shape[0] * theta1.shape[1]],
(theta1.shape[0], theta1.shape[1]))
        theta2 = np.reshape(res.x[theta1.shape[0] * theta1.shape[1]:],
(theta2.shape[0], theta2.shape[1]))

    return theta1, theta2
```

Resultado

Tras cargar los datos, hacer el cómputo de la red neuronal, y predecir los resultados con la red neuronal entrenada, obtenemos estos resultados:

```
Neural network accuracy: 97.8
```