Entrega 5: entrenamiento de redes neuronales

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 5 y el proceso de entrenamiento de redes neuronales.

Para esta práctica se usarán los siguientes *imports* vistos en la figura 0.1. Parte del código se reutiliza de la práctica anterior.

```
import numpy as np
import os
import scipy.io as sio
import scipy.optimize as opt
import utils
import matplotlib.pyplot as plt
from logistic_reg import sigmoid, plot_folder, csv_folder
```

Figura 0.1: Código de las bibliotecas usadas

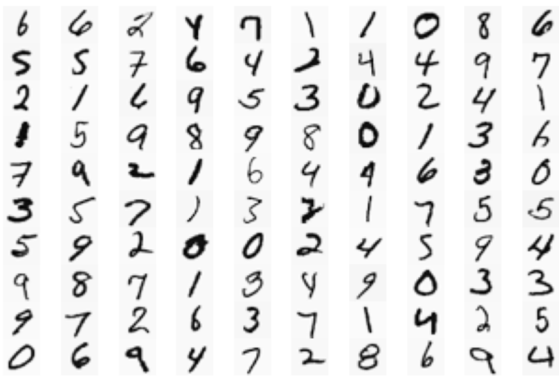

Figura 0.2: Ejemplo de los dígitos del *dataset*

1. Entrenamiento de redes neuronales

Para comprobar la red neuronal utilizaremos la función *neural_network* implementada en la figura 1.1. Esta función se implementa para un número indeterminado de capas.

Reimplementamos la función de coste (figura 1.2) para que se adapte a las dos capas del ejercicio, esta función incluye también la regularización. La función de coste puede dar error para los números 0 y 1 por el uso del logaritmo, usamos la función *fix_data* (figura 1.3) para añadirle un infinitesimal valor para que no sea exactamente 0 o 1. Para la propagación primero haremos uso de la función de red neuronal para hacer la propagación hacia adelante y luego la propagación hacia atrás. Todo esto ocurre en la función *backprop* que se muestra en la figura 1.4.

Hacemos el descenso de gradiente en la función *gradient_descent* implementada en la figura 1.5. En esta función se hace uso de la función *backprop* para calcular los gradientes y actualizar los pesos.

Para terminar usamos *prediction*, *predict_percentage* y *random_init* como utilidades para el proceso de entrenamiento (figuras 1.6, 1.7 y 1.8).

```python

def neural_network(X: np.ndarray, thetas: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """Generate the neural network with a given set of weights

    Args:
        X (np.ndarray): data
        thetas (np.ndarray): array containing the weights for each layer

    Returns:
        tuple[np.ndarray, np.ndarray]: tuple containing the activations and the z values for each layer
    """
    a = []
    z = []
    a.append(X.copy())
    for theta in thetas:
        a[-1] = np.hstack((np.ones((a[-1].shape[0], 1)), a[-1]))
        z.append(np.dot(a[-1], theta.T))
        a.append(sigmoid(z[-1]))
```

Figura 1.1: Función *neural_network*

```python

def cost(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_: float
    = 0.0) -> float:
    """
    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.

    """
    L = 2
    layers = [theta1, theta2]
    k: int = y.shape[1]
    h, z = neural_network(X, [theta1, theta2])

    h = h[-1]

    h = fix_data(h)

    J = y * np.log(h)
    J += (1 - y) * np.log(1 - h)

    J = -1 / X.shape[0] * np.sum(J)

    if lambda_ != 0:
        reg = 0
        for layer in layers:
            reg += np.sum(layer[:, 1:] ** 2)
        J += lambda_ / (2 * X.shape[0]) * reg
```

Figura 1.2: Función de coste

```python

def fix_data(X: np.ndarray) -> np.ndarray:
    """Fixes the data to avoid log(0) errors

    Args:
        X (np.ndarray): train data

    Returns:
        np.ndarray: matrix with no 0 or 1 values
    """
```

Figura 1.3: Función *fix_data*

```python

def backprop(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_:
    float) -> tuple[float, np.ndarray, np.ndarray]:
    """
    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)

    """
    m = X.shape[0]
    L = 2

    delta = np.empty(2, dtype=object)
    delta[0] = np.zeros(theta1.shape)
    delta[1] = np.zeros(theta2.shape)

    a, z = neural_network(X, [theta1, theta2])

    for k in range(m):
        a1k = a[0][k, :]
        a2k = a[1][k, :]
        hk = a[2][k, :]
        yk = y[k, :]

        d3k = hk - yk
        d2k = np.dot(theta2.T, d3k) * a2k * (1 - a2k)

        delta[0] = delta[0] + \
            np.matmul(d2k[1:, np.newaxis], a1k[np.newaxis, :])
        delta[1] = delta[1] + np.matmul(d3k[:, np.newaxis], a2k[np.newaxis, :])

    grad1 = delta[0] / m
    grad2 = delta[1] / m

    if lambda_ != 0:
        grad1[:, 1:] += lambda_ / m * theta1[:, 1:]
        grad2[:, 1:] += lambda_ / m * theta2[:, 1:]

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

Figura 1.4: Función *backprop*

```python

def gradient_descent(X: np.ndarray, y: np.ndarray, theta1: np.ndarray, theta2: np.ndarray,
    alpha: float, lambda_: float, num_iters: int) -> tuple[np.ndarray, np.ndarray, np.ndarray
    ]:
    """Generates the gradient descent for the neural network

    Args:
        X (np.ndarray): Train data
        y (np.ndarray): Expected output in one hot encoding
        theta1 (np.ndarray): initial weights for the first layer
        theta2 (np.ndarray): initial weights for the second layer
        alpha (float): learning rate
        lambda_ (float): regularization parameter
        num_iters (int): number of iterations to run

    Returns:
        tuple[np.ndarray, np.ndarray, np.ndarray]: tuple with the final weights for the first
    and second layer and the cost history
    """
    m = X.shape[0]
    J_history = np.zeros(num_iters)
    for i in range(num_iters):
        print('Iteration: ', i + 1, '/', num_iters, end='\r')
        J, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)
        theta1 = theta1 - alpha * grad1
        theta2 = theta2 - alpha * grad2
        J_history[i] = J
    print('Gradient descent finished.')
```

Figura 1.5: Función *gradient_descent*

```python

def prediction(X: np.ndarray, theta1: np.ndarray, theta2: np.ndarray) -> np.ndarray:
    """Generates the neural network prediction

    Args:
        X (np.ndarray): data
        theta1 (np.ndarray): first layer weight
        theta2 (np.ndarray): second layer weight

    Returns:
        np.ndarray: best prediction for each row in `X`
    """
    m = X.shape[0]
    p = np.zeros(m)
    a, z = neural_network(X, [theta1, theta2])
    h = a[-1]
```

Figura 1.6: Función *prediction*

```python

def predict_percentage(X: np.ndarray, y: np.ndarray, theta1: np.ndarray, theta2: np.ndarray)
    -> float:
    """Gives the accuracy of the neural network

    Args:
        X (ndarray): Train data
        y (ndarray): Expected output
        theta1 (ndarray): First layer weights
        theta2 (ndarray): Second layer weights

    Returns:
        float: Accuracy of the neural network
    """
    m = X.shape[0]
    p = prediction(X, theta1, theta2)
```

Figura 1.7: Función *predict_percentage*

```python

def random_init(size: tuple[int, int]) -> np.ndarray:
    """Generates a random matrix of shape `size` with values between -0.12 and 0.12

    Args:
        size (tuple[int,int]): shape of the generated matrix

    Returns:
        ndarray: random sample of shape `size`
    """
```

Figura 1.8: Función *random_init*

2. Flujo de entrenamiento

El programa llama a la función *test_learning* (figura 2.1) que se encarga de entrenar la red neuronal. Primero codificamos y a la codificación *one hot* mediante el uso de las funciones *oneHotEncoding* y *encoder* (figuras 2.2). Tras esto iniciamos los valores de iteración, número de capas, λ y α e iniciamos aleatoriamente $\theta 1$ y $\theta 2$. Tras esto ejecutamos el descenso de gradiente, guardamos la matriz de confusión (usando la función de la práctica anterior) y guardamos los valores de la red neuronal usando la función *save_nn* (figura 2.3).

```python

def test_learning(X: np.ndarray, y: np.ndarray) -> None:
    """Tests the training of the neural network

    Args:
        X (np.ndarray): train data
        y (np.ndarray): unencoded expected results
    """
    y_encoded = oneHotEncoding(y)
    input_layer_size = X.shape[1]
    hidden_layer_size = 25
    num_labels = 10
    lambda_ = 1
    alpha = 1
    num_iters = 1000

    theta1 = random_init((hidden_layer_size, input_layer_size + 1))
    theta2 = random_init((num_labels, hidden_layer_size + 1))

    theta1, theta2, J_history = gradient_descent(
        X, y_encoded, theta1, theta2, alpha, lambda_, num_iters)

    save_nn(theta1, theta2, model_folder)

    print(y)
    print(prediction(X, theta1, theta2))

    plot_confusion_matrix(y, prediction(
        X, theta1, theta2), f'{plot_folder}confusion_matrix.png')

    print('Expected accuracy: 95%. Got: ',
```

Figura 2.1: Función *test_learning*

```python

def oneHotEncoding(y: np.ndarray) -> np.ndarray:
    """Encodes the expected output to one hot encoding

    Args:
        y (np.ndarray): unencoded data

    Returns:
        np.ndarray: encoded data
    """
    def encoder(number: int) -> np.ndarray:
        aux = np.zeros(10)
        aux[number] = 1
        return aux
    y_encoded = [encoder(y[i] % 10) for i in range(y.shape[0])]
```

Figura 2.2: Función *oneHotEncoding*

```python

def save_nn(theta1: np.ndarray, theta2: np.ndarray, folder: str) -> None:
    """saves the neural network weights to a .mat file

    Args:
        theta1 (np.ndarray): first layer weights
        theta2 (np.ndarray): second layer weights
        folder (str): folder to save the mat
    """
```

Figura 2.3: Función *save_nn*

3. Funciones auxiliares y resultados

Algunas funciones auxiliares del programa son las siguientes:

- *load_nn* (figura 3.2): Carga los valores de la red neuronal.
- *loadData* (figura 3.3): Carga los datos del *dataset*.
- *loadWeights* (figura 3.4): Carga los pesos de la red neuronal de ejemplo.
- *displayData* (figura 3.5): Muestra los dígitos del *dataset*.
- *main* (figura 3.6): Función principal del programa.

Podemos ver que la red neuronal consigue un 95 % de acierto, representado en la figura 3.1.

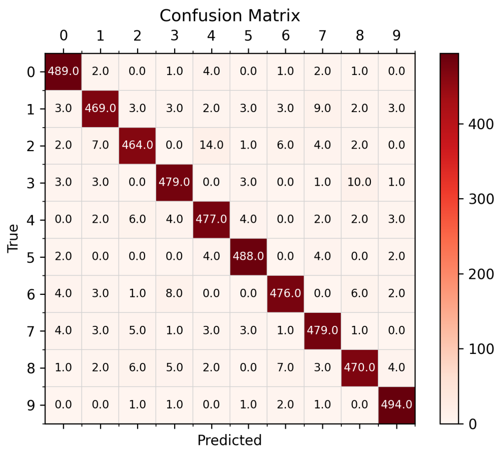

Figura 3.1: Resultado de la red neuronal

```python

def load_nn(folder: str) -> tuple[np.ndarray, np.ndarray]:
    """loads the neural network weights from a .mat file

    Args:
        folder (str): folder where the folder is stored

    Returns:
        tuple[np.ndarray, np.ndarray]: weights for the first and second layer
    """
    data = sio.loadmat(folder + 'nn.mat', squeeze_me=True)
    theta1 = data['theta1']
    theta2 = data['theta2']
```

Figura 3.2: Función *load_nn*

```python

def loadData() -> tuple[np.ndarray, np.ndarray]:
    """Loads the data from the .mat file

    Returns:
        tuple[np.ndarray]: X and y data
    """
    data = sio.loadmat('data/ex3data1.mat', squeeze_me=True)
    X = data['X']
    y = data['y']
```

Figura 3.3: Función *loadData*

```python

def loadWeights() -> tuple[np.ndarray, np.ndarray]:
    """Loads the example weights from the .mat file

    Returns:
        tuple[np.ndarray, np.ndarray]: example weights for the first and second layer
    """
    weights = sio.loadmat('data/ex3weights.mat', squeeze_me=True)
    theta1 = weights['Theta1']
    theta2 = weights['Theta2']
```

Figura 3.4: Función *loadWeights*

```python

def displayData(X: np.ndarray) -> None:
    """Displays the data in a 10x10 grid

    Args:
        X (np.ndarray): data to get a sample
    """
    if os.path.exists(f'{plot_folder}dataset.png'):
        return
    rand_indices = np.random.choice(X.shape[0], 100, replace=False)
    utils.displayData(X[rand_indices, :])
```

Figura 3.5: Función *displayData*

```python

def backprop_adapter(theta: np.ndarray, X: np.ndarray, y: np.ndarray, lambda_: float) -> float
    :
    theta1 = theta[:25 * 401].reshape(25, 401)
    print(theta1.shape)
    theta2 = theta[25 * 401:].reshape(10, 26)
    print(theta2.shape)
    # print(theta)
    cost = backprop(theta1, theta2, X, y, lambda_)[0]
    print(cost)
    return cost


def minimize_test(X, y) -> None:
    y_encoded = oneHotEncoding(y)
    input_layer_size = X.shape[1]
    hidden_layer_size = 25
    num_labels = 10
    lambda_ = 0.1
    alpha = 1
    num_iters = 1

    theta1 = random_init((hidden_layer_size, input_layer_size + 1))
    theta2 = random_init((num_labels, hidden_layer_size + 1))

    pesos = np.random.rand(401 * 25 + 26 * 10) * 0.12 * -0.12
```

Figura 3.6: Función *main*