PRÁCTICA DE LABORATORIO 7

RED NEURONAL PROFUNDA, ¡CON TANTAS CAPAS COMO DESEE!

Ha entrenado previamente una red neuronal de 2 capas (con una sola capa oculta). Esta semana, construirá una red neuronal profunda, ¡con tantas capas como desee!

NSTRUCCIONES:

No utilice bucles (for / while) en su código

OBJETIVO: Desarrollar una intuición de la estructura general de una red neuronal.

- Implementar funciones (por ejemplo, propagación hacia adelante, propagación hacia atrás, pérdida logística, etc.) que lo ayudarán a descomponer su código y facilitar el proceso de construcción de una red neuronal.
- Inicializar / actualizar los parámetros de acuerdo a su estructura deseada.

BIBLIOGRAFÍA:

- http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/
- https://stats.stackexchange.com/questions/211436/why-do-we-normalize-images-by-subtracting-the-datasets-image-mean-and-not-the-c

1 - PACKAGES

Primero, importar todos los paquetes que necesitará durante esta práctica.

Numpy: es el paquete fundamental para la computación científica con Python.
matplotlib :es una biblioteca para trazar gráficos en Python.
dnn_utils: proporciona algunas funciones necesarias para esta práctica.
testCases proporciona algunos casos de prueba para evaluar la corrección de sus funciones
np.random.seed (1) se usa para mantener consistentes todas las llamadas de función aleatorias. Ayuda a calificar la práctica para obtener los resultados esperados. Por favor no cambies la semilla.

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
from testCases_v4 import *
from dnn_utils_v2 import sigmoid, sigmoid_backward, relu, relu_backward

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

2 – INICIALIZACIÓN

Escribirá dos funciones de ayuda que inicializarán los parámetros para su modelo. La primera función se utilizará para inicializar parámetros para un modelo de dos capas. El segundo generalizará este proceso de inicialización a las capas L.

2.1 Red neuronal de 2 capas

ACTIVIDAD 1: Cree e inicialice los parámetros de la red neuronal de 2 capas.

Instrucciones:

- La estructura del modelo es: LINEAR -> RELU -> LINEAL -> SIGMOID.

- Utilice inicialización aleatoria para las matrices de peso. Use np.random.randn (shape) * 0.01 con la forma correcta.
- Use cero inicialización para los BIAS. Utilice np.zeros (shape).

```
# FUNCION: initialize_parameters
def initialize_parameters(n_x, n_h, n_y):

    np.random.seed(1)
    W1 =
    b1 =
    W2 =
    b2 =

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
#Probando la funcion
parameters = initialize_parameters(3,2,1)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Salida Esperada

```
W1=[[ 0.01624345 -0.00611756 -0.00528172][-0.01072969  0.00865408 -0.02301539]]
b1=[[ 0.] [ 0.]]
W2=[[ 0.01744812 -0.00761207]]
b2=[[ 0.]]
```

2.2 Red neuronal de L capas

La inicialización de una red neuronal de capa L más profunda es más complicada porque hay muchas más matrices de peso y vectores de sesgo. Al completar el initialize_parameters_deep, debe asegurarse de que sus dimensiones coincidan entre cada capa. Recuerde que $n^{[l]}$ es el número de unidades en la capa l. Así, por ejemplo, si el tamaño de nuestra entrada X es (12288,209) (con m = 209 ejemplos de entrenamiento) entonces:

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
⋮	⋮	⋮	⋮	⋮
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

ACTIVIDAD 2: Implementar la inicialización de una red neuronal de capa L.

Instrucciones:

- La estructura del modelo es [LINEAR -> RELU] X (L-1) -> LINEAR -> SIGMOID. Es decir, tiene L − 1 capas utilizando una función de activación ReLU seguida de una capa de salida con una función de activación sigmoide.
- Utilice inicialización aleatoria para las matrices de peso. Utilice np.random.randn (shape) * 0.01.
- Utilice la inicialización de ceros para los *bias*. Utilice np.zeros (shape).
- Almacenaremos $n^{[l]}$, el número de unidades en diferentes capas, en una variable layer_dims. Por ejemplo, los layer_dims para el "Modelo de clasificación de datos planares" de la semana pasada habrían sido [2, 4,1]: había dos entradas, una capa oculta con 4 unidades ocultas y una capa de salida con 1 unidad de salida. De este modo, la forma de W1 era (4,2), b1 era (4,1), W2 era (1,4) y b2 (1,1). ¡Ahora vas a generalizar esto a las capas L!
- Aquí está la implementación para L = 1 (red neuronal de una capa). Debería inspirarte a implementar el caso general (red neuronal de capa L).

```
if L == 1:
    parameters["W" + str(L)] = np.random.randn(layer_dims[1], layer_dims[0]) * 0.01
    parameters["b" + str(L)] = np.zeros((layer_dims[1], 1))
```

```
# FUNCCION: initialize_parameters_deep
def initialize_parameters_deep(layer_dims):
    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)          # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] =
        parameters['b' + str(l)] =

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))

    return parameters
```

```
#Probando la función
parameters = initialize_parameters_deep([5,4,3])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"])
```

Salida Esperada

```
W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493 -0.00277388]
 [-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]
 [-0.01313865  0.00884622  0.00881318  0.01709573  0.00050034]
 [-0.00404677 -0.0054536  -0.01546477  0.00982367 -0.01101068]]
b1 = [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
W2 = [[-0.01185047 -0.0020565   0.01486148  0.00236716]
 [-0.01023785 -0.00712993  0.00625245 -0.00160513]
 [-0.00768836 -0.00230031  0.00745056  0.01976111]]
b2 = [[ 0.]
 [ 0.]
 [ 0.]]
```

3 – MÓDULO DE FORWARD PROPAGATION

3.1 Forward Lineal

Ahora que ha inicializado los parámetros, implemente el módulo de propagación hacia adelante. Comenzará por implementar algunas funciones básicas que utilizará más adelante al implementar el modelo. Completarás tres funciones en este orden:

- LINEAL
- LINEAR -> ACTIVACIÓN donde ACTIVACIÓN será ReLU o Sigmoid.
- [LINEAR -> RELU] X (L-1) -> LINEAR -> SIGMOID (modelo completo)

El módulo Forward Lineal (vectorizado en todos los ejemplos) calcula las siguientes ecuaciones:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

Donde: $A^{[0]} = X$

ACTIVIDAD 3: Implementar la propagación hacia adelante lineal.

```
# FUNCTION: linear_forward
def linear_forward(A, W, b):

    Z = np.dot(W,A)+b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

```
#Probando la función
A, W, b = linear_forward_test_case()
Z, linear_cache = linear_forward(A, W, b)
print("Z = " + str(Z)
```

```
Z = [[ 3.26295337 -1.23429987]]
```

3.2 Activación Lineal en Forward

Se utilizara dos funciones de activación

- **Sigmoid**: Esta función devuelve dos elementos: el valor de activación "a" y una "caché" que contiene "Z" (es lo que introduciremos en función de backpropagation correspondiente). Para usarlo puedes simplemente llamar:

  A, activation_cache = sigmoide (Z)

- **ReLU:** Esta función devuelve dos elementos: el valor de activación "a" y una "caché" que contiene "Z" (es lo que introduciremos en función de backpropagation correspondiente). Para usarlo puedes simplemente llamar:

  ```
  A, activation_cache = relu(Z)
  ```

Para mayor comodidad, agrupará dos funciones (Lineal y Activación) en una función (LINEAR-> ACTIVACIÓN). Por lo tanto, implementará una función que realiza el paso de avance LINEAR seguido de un paso de avance de ACTIVACIÓN.

El módulo Forward Lineal (vectorizado en todos los ejemplos) calcula las siguientes ecuaciones:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

Donde: $A^{[0]} = X$

ACTIVIDAD 4: Implementar la propagación hacia adelante de la capa LINEAR-> ACTIVACIÓN. La relación matemática es:

$$A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$$

Donde la activación "g" puede ser sigmoid () o relu (). Use linear_forward () y la función de activación correcta.

```
# FUNCTION: linear_activation_forward
def linear_activation_forward(A_prev, W, b, activation):
    if activation == "sigmoid":
        Z, linear_cache =
        A, activation_cache =

    elif activation == "relu":
        Z, linear_cache =
        A, activation_cache =

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

```
#Probando a la función
A_prev, W, b = linear_activation_forward_test_case()
A, linear_activation_cache = linear_activation_forward(A_prev, W, b, })
print("With sigmoid: A = " + str(A))
A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation = "relu")
print("With ReLU: A = " + str(A))
```

Salida Esperada:

```
With sigmoid: A = [[ 0.96890023  0.11013289]]
With ReLU: A = [[ 3.43896131  0.        ]]
```

Para una mayor conveniencia al implementar la red neuronal de L capas, necesitará una función que replique la anterior (linear_activation_forward con RELU) L − 1 veces, luego sigue con una linear_activation_forward con SIGMOID

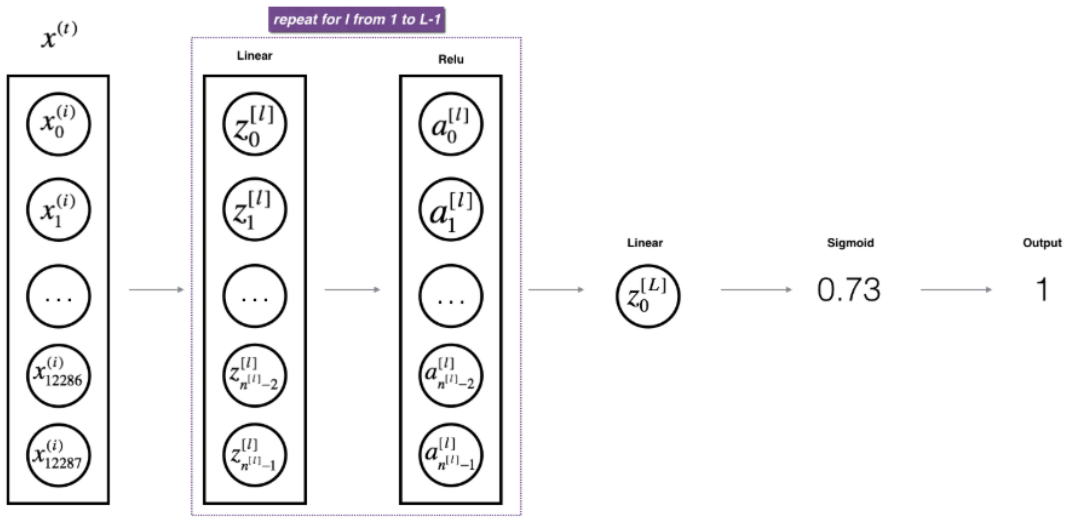

Figure 2 : *[LINEAR -> RELU]* × *(L-1) -> LINEAR -> SIGMOID* model

ACTIVIDAD 5: Implementar la propagación hacia adelante del modelo anterior.

Instrucción: En el código a continuación, la variable AL denota $A^{[l]}$, (Esto a veces también se llama Yhat, es decir, esto es Ŷ.

- Usa las funciones que habías implementado anteriormente.
- Use un bucle for para replicar [LINEAR-> RELU] (L-1) veces
- No olvide realizar un seguimiento de los cachés en la lista de "cachés". Para agregar un nuevo valor c a una lista, puede usar list.append (c)

```
# FUNCTION: L_model_forward

def L_model_forward(X, parameters):

    caches = []
    A = X
    L = len(parameters) // 2                  # número de capas de la red
    caches = list()

    # Implemente [LINEAR -> RELU]*(L-1). Agregue "cache" a la lista "caches"
    for l in range(1, L):
        A_prev = A

        A, cache =

    # Implementa LINEAR -> SIGMOID. Agregue "cache" a la lista  "caches".

    AL, cache =

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches
```

```
#Probando la función
X, parameters = L_model_forward_test_case_2hidden()
print(parameters)
print(len(parameters)//2)
print("W"+str(2))
AL, caches = L_model_forward(X, parameters)
print("AL = " + str(AL))
print("Length of caches list = " + str(len(caches)))
```

Salida Esperada:

```
AL = [[ 0.03921668  0.70498921  0.19734387  0.04728177]]
Length of caches list = 3
```