Ex 02 Implementación de una red neuronal poco profunda

En esta segunda actividad construiremos un clasificador binario para el reconocimiento de gatos. En esta actividad partiremos de la noción de regresión lineal y la implementaremos como una red neuronal artificial.

Propósitos de aprendizaje

- Exploración del conjuntos de datos
- Implementar cada uno de los elementos de un algoritmo de aprendizaje
    - Inicializar parámetros
    - Calcular la función de costo y gradientes
    - Utilizar un algoritmo de optimización (gradiente descendiente)
- Integrar el modelo
- Entrenar el modelo

Evaluación de la actividad:

- Fecha de entrega: 27 de agosto de 2019
- Cada ejercicio resuelto de manera correcta tiene un valor de 12.5
- Si la actividad se entrega retrasada, se aplicará una penalización de 20% por cada día de retraso.
- Entrega: suba a CANVAS su implementación en Python.

Uso de Librerías

Ejecutemos la siguiente celda para importar las librerías que necesitaras durante la actividad.

Librería	Descripción
numpy	numpy es una librería para cómputo cinetífico en Python.
h5py	h5py es una librería para interactuar con conjuntos de datos almacenados en el formato H5.
matplotlib	es una librería para dibujar gráficos en Python.
PIL	es una librería para el procesamiento de imágenes (Python Imaging Library).
scipy	es una librería de cómputo científico que contiene módulos para optimización, álgebra lineal, procesamiento de imágenes n-dimensionales, etc.

Descripción del conjunto de datos

Para esta actividad utilizaremos dos archivos de datos:

- `train_data.h5` es un conjunto de imagenes de entrenamiento (etiquetadas como: `cat` = 1 o `non-cat` = 0). Este archivo contiene los siguientes datasets: "train_set_x" y "train_set_y".

- `test_data.h5` es un conjunto de imagenes de prueba (etiquetadas como 'es gato' = 1 o 'no es gato' = 0). Este archivo contiene los siguientes datasets: "test_set_x", "test_set_y". Adicionalmente contiene "list_classes" que almacena la lista de clases.

Cada imagen tiene las siguientes dimensiones: alto = 64, ancho = 64, y canales = 3 (RGB). Así cada imagen tiene la siguiente dimensión: (64, 64, 3).

1. Exploración del conjunto de datos

Antes de comenzar a construir el modelo, vamos a familiarizarnos con los datos de entrenamiento.

Ejercicio 1. Implementemos la función `load_dataset()` para que cargue los datasets `train_set_x`, `train_set_y`, `test_set_x`, `test_set_y` y `classes` desde los archivos `train_data.h5` y `test_data.h5`.

Salida: La función debe retornar los datasets `train_set_x_original`, `train_set_y`, `test_set_x_original`, `test_set_y` y `classes`. Adicionalmente imprima las dimensiones de los datasets.

Ejercicio 2. Muchos errores de implementación de software en Deep Learnin provienen de tener matrices y vectores con dimensiones que no son apropiadas (que no encajan acorde a las operaciones que se quieren realizar). Si nos aseguramos de mantener correctas las dimensiones de las matrices y vectores será un gran paso para evitar errores de implementación.

Encuentra los valores correspondientes a:

- Número de ejemplos de entrenamiento (`train_m`)
- Número de ejemplos de prueba (`test_m`)
- Alto y ancho de las imágenes en los datasets (`img_dim`)

Salida: imprima el valor de `train_m`, `test_m, y img_dim.`

Nota: recuerde que `train_set_x_original` es un arreglo numpy de dimensiones: `(train_m, img_dim, img_dim, 3)`. Por ejemplo, podemos acceder al valor de **train_m** utilizando: `train_set_x_original.shape[0]`.

2. Preprocesamiento de los datasets

Cuando se tiene un nuevo dataset, es común realizar un preprocesamiento de los datos antes de aplicar las técnicas de aprendizaje automático. Un preprocesamiento común consiste en:

- Redimensionar la forma de los ejemplos
- Estandarizar los datos

Para esta actividad, por conveniencia, vamos a redimensionar las imágenes de la forma `(img_dim, img_dim, 3)` en un arreglo con dimensión `(img_dim * img_dim * 3, 1)`.

Despues del cambio, en nuestros datasets `train_set_x_original` y `test_set_x_original` cada columna representará una imagen. Así, cada dataset deberá tener `train_m` columnas.

Ejercicio 3: Modifica la forma de los datasets `train_set_x_original` y `test_set_x_original` de manera que las imágenes de dimensiones `(img_dim, img_dim, 3)` sean representadas por arreglos individuales con dimensiones: `(img_dim * img_dim * 3, 1)`

Salida: imprima las dimensiones de train_set_x y test_set_x despues del aplanado de las imágenes.

Nota:cuando se quiere aplanar una matriz IMG de dimensiones (w, x, y, z) para que tenga la forma (x * y * z, w) se puede utilizar:

```
IMG_flatten = IMG.reshape(IMG.shape[0], -1).T

# IMG.T es la transpuesta de IMG
```

3. Algoritmo de aprendizaje

En esta actividad diseñaremos e implementaremos un algoritmo simple para el reconocimiento de imagenes. El algoritmo debe clasificar de manera correcta si en una imagen se encuentra un gato o no.

Con el objetivo de reducir la complejidad del algoritmo, de momento utilizaremos la regresión logística pero desde el punto de vista de redes neuronales. Veamos porque la regresión logística es realmente una red neuronal muy simple (Figura 1).

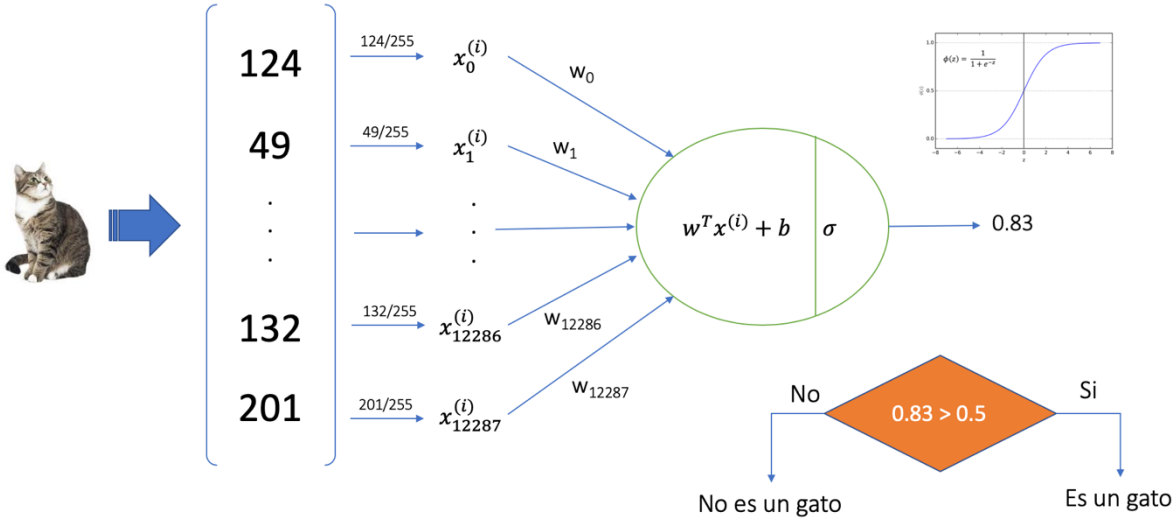

Figura 1Regresión logística vista como una red neuronal

Para un ejemplo $x^{(i)}$ del dataset de entrenamiento, el algoritmo se puede representar como:

$$z^{(i)} = w^T x^{(i)} + b$$
$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$
$$\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

El costo se obtiene mediante el promedio de los errores de clasificación en todos los ejemplos de entrenamiento:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Implementación de los elementos del algoritmo

Para la construcción del modelo, realizaremos los siguiente pasos:

Paso 1: definir la estructura del modelo.

Paso 2: inicializar los parámetros del modelo.

Paso 3: Aprender parámetros que minimizan el costo

a) Calcular el error actual (forward propagation)
b) Calcular el gradiente actual (backward propagation)
c) Actualizar los parámetros (gradiente descendiente)

Implementaremos los pasos 1, 2 y 3 de manera independiente y posteriormente los integraremos en una función llamada `model()`.

3.1 Función sigmoidal

Ejercicio 4: Acorde al diagrama de nuestra red neuronal, necesitamos calcular $\sigma(z)$ para realizar predicciones. Dado que

$$z^{(i)} = w^T x^{(i)} + b,$$

implementemos la función llamada `sigmoid(z)`.

$$sigmoid(w^T x^{(i)} + b) = \frac{1}{1+e^{-(w^T x^{(i)}+b)}}.$$

la función debe retornar la sigmoidal de z, en donde z puede ser un escalar o un arreglo numpy de cualquier tamaño.

Pruebe su implementación con:

```
print(f"sigmoid([-1, 0, 1]) = {sigmoid(np.array([-1, 0, 1]))}")
```

Nota: utilicemos la función `np.exp()`.

3.2 Inicialización de parámetros

Ejercicio 5: Ahora implementemos la función `init_parameters(dim)` para realizar la inicialización de los parámetros. Para esta actividad, inicialicemos w como un vector de tamaño `dim` con valor cero en todos los elementos del arreglo y b igual a cero.

La función debe retornar w y b con sus valores inicializados en cero.

Pruebe su implementación con el siguiente código:

```
dim = 4
w, b = init_parameters(dim)
print(f"w.shape = {w.shape}")
print(f"w = {w}")
print(f"b = {b}")
```

3.3 Forward y backward propagation

Es momento de implementar el forward y backward propagation para el aprendizaje de los parámetros.

Ejercicio 5: Implementemos la función `fb_propagation()` que debe calcular la función de costo y su gradiente.

Pasos del forward propagation:

- Obtener X
- Calcular las activaciones:

$$A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$$

- Calcular la función de costo:

$$J(w, b) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} log(a^{(i)}) + (1 - y^{(i)}) log(1 - a^{(i)})$$

Para el cálculo del gradiente:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T$$
$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

La función `fb_propagation()` debe calcular la función de costo y su gradiente

Los parámetros que debe recibir son:

- w (pesos) es un arreglo numpy de tamaño (img_dim * img_dim * 3, 1)
- b (bias) es un escalar
- X son los ejemplos de tamaño: (img_dim * img_dim * 3, número de ejemplos)
- Y es un vector con las etiquetas de los ejemplos de entrenamiento, su tamaño es (1, número de ejemplos)

La función debe retornar: cost, dw y db

Tip: utilice para su código np.log(), np.dot(), y np.sum()

Pruebe su implementación con el siguiente código:

```
X = np.array([[1,2],[3,4],[5,6]])
Y = np.array([[1,0]])
w = np.array([[1],[2],[3]])
b = 1
gradients, cost = fb_propagation(X, Y, w, b)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

3.4 Optimización

Hasta este momento hemos implementado funciones que nos permiten:

- Inicializar parámetros
- Calcular la sigmoidal
- Calcular la función de costo y sus gradientes

Ahora, necesitamos implementar una función que permita actualizar los parámetros. En esta actividad utilizaremos el gradiente descendiente.

Ejercicio 6: Implementemos la función `optimization()`. Esta función debe aprender los parámetros w y b que minimizan la función de costo $J(w,b)$. Las reglas para actualizar los parámetros w y b son las siguientes:

$$w=w-\alpha\ dw$$
$$b=b-\alpha\ db$$

donde α es la tasa de aprendizaje.

En resumen necesitas implementar dos pasos e iterar sobre ellos:

- Calcular el costo y el gradiente para los parámetros actuales.
  Utilizemos $propagation()$
- Actualizar los parámetros utilizando el gradiente descendiente para w y b.

La función debe aprender los parámetros w y b utilizando el gradiente descendiente.

Los parámetros que debe recibir son:

- w (pesos) es un arreglo numpy de tamaño (img_dim * img_dim * 3, 1)

- b (bias) es un escalar
- X son los ejemplos de entrenamiento de tamaño: (img_dim * img_dim * 3, número de ejemplos)
- Y es un vector con las etiquetas de los ejemplos de entrenamiento, su tamaño es (1, número de ejemplos)

La función debe retornar:

- parameters es un diccionario que contiene los pesos y bias
- gradients es un diccionario que contiene los dw y db con respecto a la función de costo.
- costs es una lista de costos calculados durante la optimización (la utilizaremos para dibujar la curva de aprendizaje)

Tip: utilice para su código np.log(), np.dot(), y np.sum()

Pruebe la implementación de la función con el siguiente código:

```
parameters, gradients, costs = optimization(X, Y, w, b, iterations=100,
learning_rate=0.0009)

print(f"w = {parameters['w']}")
print(f"b = {parameters['b']}")
print(f"dw = {gradients['dw']}")
print(f"db = {gradients['db']}")
```

3.5 Clasificador

Ejercicio 7: Implementemos la función `prediction()` que tendrá como objetivo utilizar `w` y `b` (con los valores aprendidos por la función `optimization()`) para predecir las etiquetas para un dataset `X`. Para realizar la predicción se requieren dos pasos:

- Calcular:

$$\hat{Y} = A = \sigma(w^T X + b)$$

- Convertir los valores de `activación` en 0 (si activación <= 0.5) o 1 (si activación > 0.5), y almacenar las predicciones en un vector `Yp`.

La función debe realizar la predicción de la etiqueta (1 o 0) utilizando los parámetros aprendidos w y b.

Los parámetros de la función son:

- w (pesos) es un arreglo numpy de tamaño (img_dim * img_dim * 3, 1).
- b (bias) es un escalar.
- X son los ejemplos de entrenamiento de tamaño: (img_dim * img_dim * 3, número de ejemplos).

La función debe retornar:

- Yp un vector numpy que contiene las predicciones (1/0) de cada ejemplo de entrenamiento en X.

Pruebe la implementación de su función con:

```
print(f"Predicción = {prediction(X, w, b)}")
```

3.6 Integración de funciones en un modelo

Ejercicio 8: Ahora estructuremos el modelo completo. Implemente una función llamada `model()` que integre todas las funciones que hemos implementado anteriormente.

Pasos a implementar utilizando las funciones anteriormente descritas:

- Obtener valor de X_n
- Inicializar de los parámetros w y b
- Aprender parámetros w y b
- Realizar la predicción con los datasets de entrenamiento y prueba
- Retornar un diccionario con la información del modelo

La función `model()` debe recibir como parámetros:

- X_train son los ejemplos de entrenamiento de tamaño: (img_dim * img_dim * 3, número de ejemplos)
- Y_train son las etiquetas de los ejemplos de entrenamiento de tamaño: (1, número de ejemplos)
- X_test son los ejemplos de prueba de tamaño: (img_dim * img_dim * 3, número de ejemplos)
- Y_test son las etiquetas de los ejemplos de prueba de tamaño: (1, número de ejemplos)
- iterations representa el número de iteraciones para la optimización de los parámetros (hiperparámetro)
- learning_rate representa la tasa de aprendizaje utilizada en la regla de actualización (hiperparámetro)

La función debe retornar:

- description es el diccionario que contiene información sobre el modelo. Estructura propuesta para el diccionario:

```
{"Costs": costs,
 "Yp_test": Yp_test,
 "Yp_train": Yp_train,
 "w": w,
```

```
    "b": b,
    "iterations": iterations,
    "learning_rate": learning_rate}
```

Para entrenar su modelo utilice:

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, iterations=1000, learning_rate=0.005)
```

Finalmente, realice lo siguiente y compare sus resultados con los compañeros de clase:

- Encuentre un ejemplo que fue mal clasificado por su modelo
- Muestre la imagen que fue mal clasificada
- Grafique la curva de aprendizaje (costo vs iteraciones)