Open in Colab

# Laboratorio 1

**Bienvenidos al primer laboratorio de Deep Learning y Sistemas Inteligentes. Espero que este laboratorio les sirva para consolidar sus conocimientos de las primeras dos semanas.**

**Este laboratorio consta de dos partes. En la primera trabajaremos una Regresión Logística con un acercamiento más a una Red Neuronal. En la segunda fase, usaremos PyTorch para crear un modelo similar pero ya usando las herramientas de Deep Learning aunque aún implementando algunos pasos "a mano".**

**Para este laboratorio estaremos usando una herramienta para Jupyter Notebooks que facilitará la calificación, no solo asegurándo que ustedes tengan una nota pronto sino también mostrandoles su nota final al terminar el laboratorio.**

**Por favor noten que es primera vez que uso este acercamiento para laboratorios por ende, pido su compresión y colaboración si algo no funciona como debería. Ayúdenme a mejorarlo para las proximas iteraciones.**

# Laboratorio 1 de Deep Learning Realizado por:


## Nelson García Bravatti 22434

## Joaquín Puentes 22296


### Repositorio:

https://github.com/nel-eleven11/Lab1_DL

### Catedrático:

**Luis Alberto Suriano**

**Julio de 2025**

# Antes de Empezar

**Por favor actualicen o instalen la siguiente librería que sirve para visualizaciones de la calificacion, además de otras herramientas para calificar mejor las diferentes tareas. Pueden correr el comando mostrado abajo (quitando el signo de comentario) y luego reiniciar el kernel (sin antes volver a comentar la linea), o bien, pueden hacerlo sdesde una cmd del ambiente de Anaconda**

***Creditos:***

**Esta herramienta pertence a sus autores, Dr John Williamson et al.**

In [1]:

```
!pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutils/zipball/master
```

```
Collecting https://github.com/johnhw/jhwutils/zipball/master
  Downloading https://github.com/johnhw/jhwutils/zipball/master
     | 119.1 kB 995.6 kB/s 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: jhwutils
  Building wheel for jhwutils (pyproject.toml) ... done
```

```
  Building wheel for jhwutils (pyproject.toml) ... done
  Created wheel for jhwutils: filename=jhwutils-1.3-py3-none-any.whl size=41911 sha256=ab
81d4b211a3fe8660235c3e8eb778238924a73dafeeb78ec210646d594589db
  Stored in directory: /tmp/pip-ephem-wheel-cache-lkkm26nq/wheels/c9/d0/2e/946a586bab0de8
4a4ee2b053e8d50eb28d56d8556f3ebefa84
Successfully built jhwutils
Installing collected packages: jhwutils
  Attempting uninstall: jhwutils
    Found existing installation: jhwutils 1.3
    Uninstalling jhwutils-1.3:
      Successfully uninstalled jhwutils-1.3
Successfully installed jhwutils-1.3
```

**La librería previamente instalada también tiene una dependencia, por lo que necesitarán instalarla.**

In [2]:

```
!pip install scikit-image
```

```
Requirement already satisfied: scikit-image in ./venv/lib/python3.13/site-packages (0.25.
2)
Requirement already satisfied: numpy>=1.24 in ./venv/lib/python3.13/site-packages (from s
cikit-image) (2.3.1)
Requirement already satisfied: scipy>=1.11.4 in ./venv/lib/python3.13/site-packages (from
scikit-image) (1.16.0)
Requirement already satisfied: networkx>=3.0 in ./venv/lib/python3.13/site-packages (from
scikit-image) (3.5)
Requirement already satisfied: pillow>=10.1 in ./venv/lib/python3.13/site-packages (from
scikit-image) (11.3.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in ./venv/lib/python3.13/site-packa
ges (from scikit-image) (2.37.0)
Requirement already satisfied: tifffile>=2022.8.12 in ./venv/lib/python3.13/site-packages
(from scikit-image) (2025.6.11)
Requirement already satisfied: packaging>=21 in ./venv/lib/python3.13/site-packages (from
scikit-image) (25.0)
Requirement already satisfied: lazy-loader>=0.4 in ./venv/lib/python3.13/site-packages (f
rom scikit-image) (0.4)
```

In [3]:

```
import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy
from PIL import Image
import os

# Other imports
from unittest.mock import patch
from uuid import getnode as get_mac

from jhwutils.checkarr import array_hash, check_hash, check_scalar, check_string
import jhwutils.image_audio as ia
import jhwutils.tick as tick

###
tick.reset_marks()

%matplotlib inline
```

In [4]:

```
# Hidden cell for utils needed when grading (you can/should not edit this)
# Celda escondida para utlidades necesarias, por favor NO edite esta celda
```

***Información del estudiante en dos variables***

- **carne : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.**
- **firma_mecanografiada: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)**

In [5]:

```
carne = "22434"
firma_mecanografiada = "Nelson Garcia"
# YOUR CODE HERE
```

In [6]:

```
# Deberia poder ver dos checkmarks verdes [0 marks], que indican que su información básic
a está OK

with tick.marks(0):
    assert(len(carne)>=5)

with tick.marks(0):
    assert(len(firma_mecanografiada)>0)
```

✓ [0 marks]

✓ [0 marks]

**Dataset a Utilizar**

**Para este laboratorio estaremos usando el dataset de Kaggle llamado Cats and Dogs image classification. Por favor, descarguenlo y ponganlo en una carpeta/folder de su computadora local.**

# Parte 1 - Regresión Logística como Red Neuronal

**Créditos: La primera parte de este laboratorio está tomado y basado en uno de los laboratorios dados dentro del curso de "Neural Networks and Deep Learning" de Andrew Ng**

In [7]:

```
# Por favor cambien esta ruta a la que corresponda en sus maquinas
data_dir = '/home/nelson/Documents/Uvg/Deep Learning/Lab1_DL/images'

train_images = []
train_labels = []
test_images = []
test_labels = []

def read_images(folder_path, label, target_size, color_mode='RGB'):
    for filename in os.listdir(folder_path):
        image_path = os.path.join(folder_path, filename)
        # Use PIL to open the image
        image = Image.open(image_path)

        # Convert to a specific color mode (e.g., 'RGB' or 'L' for grayscale)
        image = image.convert(color_mode)

        # Resize the image to the target size
        image = image.resize(target_size)

        # Convert the image to a numpy array and add it to the appropriate list
        if label == "cats":
            if 'train' in folder_path:
                train_images.append(np.array(image))
                train_labels.append(0)  # Assuming 0 represents cats
            else:
                test_images.append(np.array(image))
```

```
                test_labels.append(0)  # Assuming 0 represents cats
        elif label == "dogs":
            if 'train' in folder_path:
                train_images.append(np.array(image))
                train_labels.append(1)  # Assuming 1 represents dogs
            else:
                test_images.append(np.array(image))
                test_labels.append(1)  # Assuming 1 represents dogs
# Call the function for both the 'train' and 'test' folders
train_cats_path = os.path.join(data_dir, 'train', 'cats')
train_dogs_path = os.path.join(data_dir, 'train', 'dogs')
test_cats_path = os.path.join(data_dir, 'test', 'cats')
test_dogs_path = os.path.join(data_dir, 'test', 'dogs')


# Read images
target_size = (64, 64)
read_images(train_cats_path, "cats", target_size)
read_images(train_dogs_path, "dogs", target_size)
read_images(test_cats_path, "cats", target_size)
read_images(test_dogs_path, "dogs", target_size)
```

In [8]:

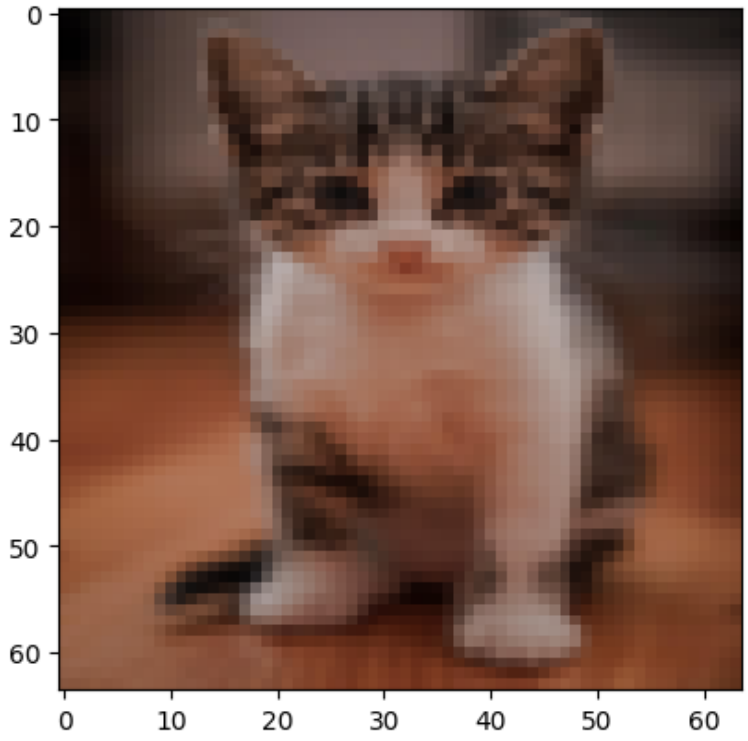
```
# Convert the lists to numpy arrays
train_images = np.array(train_images)
train_labels = np.array(train_labels)
test_images = np.array(test_images)
test_labels = np.array(test_labels)

# Reshape the labels
train_labels = train_labels.reshape((1, len(train_labels)))
test_labels = test_labels.reshape((1, len(test_labels)))
```

In [9]:

```
# Ejemplo de una imagen
index = 25
plt.imshow(train_images[index])
print ("y = " + str(train_labels[0][index]) + ", es una imagen de un " + 'gato' if train
_labels[0][index]==0 else 'perro' +  "'.")
```

```
y = 0, es una imagen de un gato
```




## Ejercicio 1

**Para este primer ejercicio, empezaremos con algo súper sencillo, lo cual será soalmente encontrar los valores de las dimensiones de los vectores con los que estamos trabajando**

- **m_train: número de ejemplos de entrenamiento**
- **m_test: número de ejemplos de testing**
- **num_px: Alto y ancho de las imagenes**

In [10]:

```
print(train_images.shape)
print(test_images.shape)
```

```
(557, 64, 64, 3)
(140, 64, 64, 3)
```

In [11]:

```
m_train = 557
m_test = 140
num_px = 64

print ("Número de datos en entrenamiento: m_train = " + str(m_train))
print ("Número de datos en testing: m_test = " + str(m_test))
print ("Alto y ancho de cada imagen: num_px = " + str(num_px))
print ("Cada imagen tiene un tamañado de: (" + str(num_px) + ", " + str(num_px) + ", 3)"
)
print ("train_images shape: " + str(train_images.shape))
print ("train_labels shape: " + str(train_labels.shape))
print ("test_images shape: " + str(test_images.shape))
print ("test_labels shape: " + str(test_labels.shape))
```

```
Número de datos en entrenamiento: m_train = 557
Número de datos en testing: m_test = 140
Alto y ancho de cada imagen: num_px = 64
Cada imagen tiene un tamañado de: (64, 64, 3)
train_images shape: (557, 64, 64, 3)
train_labels shape: (1, 557)
test_images shape: (140, 64, 64, 3)
test_labels shape: (1, 140)
```

In [12]:

```
with tick.marks(2):
    assert m_train == 557
with tick.marks(2):
    assert m_test == 140
with tick.marks(1):
    assert num_px == 64
```

**✓ [2 marks]**

**✓ [2 marks]**

**✓ [1 marks]**

## Ejercicio 2

**Para conveniencia, deberán cambiar la forma (reshape) de las imagenes (num_px, num_px, 3) en cada numpy-array a una forma de (num_px $*$ num_px $*$ 3, 1). De esta manera, tanto el training como testing dataset sera un**

**numpy-array donde cada columna representa una imagen "aplanada". Deberan haber m_train y m_test columnas**

**Entonces, para este ejercicio deben cambiar la forma (reshape) de tanto el dataset de entrenamiento como el de pruebas (training y testing) de esa forma, obtener un vector de la forma mencionada anteriormente (num_px $*$ num_px $*$ 3, 1)**

**Una forma de poder "aplanar" una matriz de forma (a,b,c,d) a una matriz de de forma (b $*$c$*$d, a), es usar el método "reshape" y luego obtener la transpuesta**

```
    X_flatten = X.reshape(X.shape[0], -1).T      # X.T es la transpuesta de X
```

In [13]:

```
train_images_flatten = train_images.reshape(train_images.shape[0], -1).T
test_images_flatten = test_images.reshape(test_images.shape[0], -1).T

print ("train_images_flatten shape: " + str(train_images_flatten.shape))
print ("train_labels shape: " + str(train_labels.shape))
print ("test_images_flatten shape: " + str(test_images_flatten.shape))
print ("test_labels shape: " + str(test_labels.shape))
```

```
train_images_flatten shape: (12288, 557)
train_labels shape: (1, 557)
test_images_flatten shape: (12288, 140)
test_labels shape: (1, 140)
```

In [14]:

```
# Test escondido para revisar algunos pixeles de las imagenes en el array aplanado
# Tanto en training [3 marks]
# Como en test [2 marks]
```

**Para representar el color de las imagenes (rojo, verde y azul - RGB) los canales deben ser específicados para cada pixel, y cada valor de pixel es de hecho un vector de tres números entre 0 y 255.**

**Una forma muy comun de preprocesar en ML es el centrar y estandarizar el dataset, es decir que se necesita restar la media de todo el array para cada ejemplo, y luego dividir cada observacion por la desviación estándar de todo el numpy array. Pero para dataset de imagenes, es más simple y más conveniente además que funciona tan bien, el solo dividir cada fila del dataset por 255 (el máximo del valor de pixeles posible.**

**Por ello, ahora estandarizaremos el dataset**

In [15]:

```
train_set_x = train_images_flatten / 255.
test_set_x = test_images_flatten / 255.
```

## Arquitectura General

**Ahora empezaremos a construir un algoritmo que nos permita diferenciar perros de gatos.**

**Para esto estaremos construyendo una Regresión Logística, usando un pensamiento de una Red Neuronal. Si se observa la siguiente imagen, se puede apreciar porque hemos dicho que la Regresión Logística es de hecho una Red Neuronal bastante simple.**

**Recordemos la expresión matematica vista en clase.**

**Por ejemplo para una observación $x^{(i)}$:**

$$z^{(i)} = w^T x^{(i)} + b \tag{1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = \tag{3}$$

$$
\begin{aligned}
&-y^{(i)} \log(a^{(i)}) \\
&-(1-y^{(i)}) \log \\
&(1-a^{(i)}) \\
&J = \frac{1}{m} \\
&\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, \\
&y^{(i)})
\end{aligned} \tag{4}
$$

**Recordemos que los pasos más importantes para construir una Red Neuronal son:**

1. **Definir la estructura del modelo (como el número de features de entrada)**
2. **Inicializar los parámetros del modelo**
3. **Iterar de la siguiente forma: a. Calcular la pérdida (forward) b. Calcular el gradiente actual (backward propagation) c. Actualizar los parámetros (gradiente descendiente)**

**Usualmente se crean estos pasos de forma separada para luego ser integrados en una función llamada "model()"**

**Antes de continuiar, necesitamos definir una función de soporte, conocida como sigmoide Recuerden que para hacer predicciones, necesitamos calcular:** $sigmoid(z)$ **para** $z = w^T x + b$

$$= \frac{1}{1+e^{-z}}$$

In [16]:

```
def sigmoid(z):
    """
    Computa el valor sigmoide de z

    Arguments:
    z: Un escalar o un numpy array

    Return:
    s: sigmoide(z)
    """
    s = 1 / (1 + np.exp(-z))

    return s
```

## Ejercicio 3 - Inicializando parámetros con cero

**Implemente la inicialización de parámetros. Tiene que inicializar w como un vector de zeros, considere usar np.zeros()**

In [17]:

```
def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w and initializes b to
0.
    Crea un vector de zeros de dimensión (dim, 1) para w, inicia b como cero

    Argument:
    dim: Tamaño

    Returns:
    w: Vector w (dim, 1)
    b: Escalar, debe ser flotante
    """

    # Aprox 2 línas de código
    w = np.zeros((dim, 1))
```

```
    b = 0.0
    # YOUR CODE HERE

    return w, b
```

In [18]:

```
dim = 3 # No cambiar esta dimensión por favor
w, b = initialize_with_zeros(dim)

print ("w = " + str(w))
print ("b = " + str(b))
```

```
w = [[0.]
 [0.]
 [0.]]
b = 0.0
```

## Ejercicio 4 - Forward and Backward propagation

**Tras inicializar los parámetros, necesitamos hacer el paso de "forward" y "backward propagation" para optimizar los parámetros.**

**Para empezar, implemente la función "propagate()" que calcula la función de costo y su gradiente.**

**Recuerde**

- **Si tiene X**
- **Se puede calcular** $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$
- **Y luego se puede calcular la función de costo:** $J = -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} \log (a^{(i)}) + (1 - y^{(i)}) \log (1 - a^{(i)}))$

**Por ende recuerd estas fórmulas (que probablemente estará usando):**

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T \tag{5}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{6}$$

In [19]:

```
def propagate(w, b, X, Y):
    """
    Implementa la función de costo y su gradiente

    Arguments:
    w: Pesos (num_px * num_px * 3, 1)
    b: bias, un escalar
    X: Data (num_px * num_px * 3, n ejemplos)
    Y: Etiquetas verdaderas (1, n ejemplos)

    Return:
    cost: Log-likelihood negativo
```

```
    dw: Gradiente de la pérdida con respecto de w
    db: Gradiente de la pérdida con respecto de b

    Tips: Recuerde escribir su código paso por paso para la propagación, considere usar n
p.log y np.dot()
    """

    m = X.shape[1]


    # Forward propagation
    A = 1 / (1 + np.exp(-(np.dot(w.T, X) + b)))    # Activación sigmoid
    cost = - (1 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))  # Costo logísti
co

    # Backward propagation
    dw = (1 / m) * np.dot(X, (A - Y).T)
    db = (1 / m) * np.sum(A - Y)

    cost = np.squeeze(np.array(cost))


    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

In [20]:

```
w =  np.array([[1.], [3]])
b = 4.5
X = np.array([[2., -2., -3.], [1., 1.5, -5.2]])
Y = np.array([[1, 1, 0]])
grads, cost = propagate(w, b, X, Y)


print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))


with tick.marks(0):
    assert type(grads["dw"]) == np.ndarray
with tick.marks(0):
    assert grads["dw"].shape == (2, 1)
with tick.marks(0):
    assert type(grads["db"]) == np.float64
```

```
dw = [[ 0.00055672]
 [-0.00048178]]
db = -0.0003283816747260056
cost = 0.0003290226268065l8
```

✓ [0 marks]

✓ [0 marks]

✓ [0 marks]

### Ejercicio 5 - Optimización

**Escriba una función de optimización. El objetivo es aprender $w$ y $b$ al minimizar la función de costo $J$. Para un**

**parametro $\theta$, la regla de actualización es $\theta = \theta - \alpha\ d\theta$, donde $\alpha$ es el learning rate.**

In [21]:

```
def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009, print_cost=False):
    """
    Función que optmiza w y b al ejecutar el algoritmo de gradiente descendiente

    Arguments:
    w: Pesos (num_px * num_px * 3, 1)
    b: bias, un escalar
    X: Data (num_px * num_px * 3, n ejemplos)
    Y: Etiquetas verdaderas (1, n ejemplos)
    num_iterations: Número de iteraciones
    learning_rate: Learning rate
    print_cost: True para mostrar la pérdida cada 100 pasos

    Returns:
    params: Dictionario con w y b
    grads: Dictionario con las gradientes de los pesos y bias con respecto a J
    costs: Lista de todos los costos calculados

    Hints:
    Necesita escribir dos pasos de la iteracion:
        1. Calcular el costo y la gradiente de los parámetros actuales, Use propagate(),
la funcion que definió antes
        2. Actualice los parametros usando la regla de gradiente descendiente para w y b
    """

    w = copy.deepcopy(w)
    b = copy.deepcopy(b)

    costs = []

    for i in range(num_iterations):
        # FORWARD + BACKWARD PROPAGATION
        grads, cost = propagate(w, b, X, Y)

        # Obtener derivadas
        dw = grads["dw"]
        db = grads["db"]

        # ACTUALIZACIÓN DE PARÁMETROS
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Guardar costo cada 100 iteraciones
        if i % 100 == 0:
            costs.append(cost)
            if print_cost:
                print(f"Costo en iteración {i}: {cost:.6f}")

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

In [22]:

```
# Recuerde NO cambiar esto por favor
params, grads, costs = optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009, pri
nt_cost=False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print("Costs = " + str(costs))
```

```
w = [[0.99949949]
 [3.00043321]]
b = 4.50029528361711
dw = [[ 0.00055554]
 [-0.00048091]]
db = -0.0003278045123969942
Costs = [array(0.00032902)]
```

## Ejercicio 6 - Predicción

**Con w y b calculados, ahora podemos hacer predicciones del dataset. Ahora implemente la función "predict()". Considere que hay dos pasos en la función de predicción:**

1. **Calcular** $\hat{Y} = A = \sigma(w^T X + b)$
2. **Convertir la entrada a un 0 (si la activación es <= 0.5) o 1 (si la activación fue > 0.5), y guardar esta predicción en un vector "Y_prediction".**

In [23]:

```
def predict(w, b, X):
    '''
    Predice si la etiqueta es 0 o 1 usando lo aprendido

    Arguments:
    w: Pesos (num_px * num_px * 3, 1)
    b: bias, un escalar
    X: Data (num_px * num_px * 3, n ejemplos)

    Returns:
    Y_prediction: Numpy Array con las predicciones
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[0], 1)

    A = 1 / (1 + np.exp(-(np.dot(w.T, X) + b)))

    for i in range(A.shape[1]):
        if A[0, i] > 0.5:
            Y_prediction[0, i] = 1
        else:
            Y_prediction[0, i] = 0

    return Y_prediction
```

In [24]:

```
w = np.array([[0.112368795], [0.48636775]])
b = -0.7
X = np.array([[1., -1.1, -3.2],[1.2, 2., 0.1]])
predictions_ = predict(w, b, X)
print ("predictions = " + str(predictions_))
```

```
predictions = [[0. 1. 0.]]
```

## Ejercicio 7 - Modelo

**Implemente la función "model()", usando la siguiente notación:**

- **Y_prediction_test para las predicciones del test set**
- **Y_prediction_train para las predicciones del train set**
- **parameters, grads, costs para las salidas de "optimize()"**

In [25]:

```
def model(X_train, Y_train, X_test, Y_test, num_iterations=2000, learning_rate=0.5, print_cost=False):
    """
    Construye la regresión logística llamando las funciones hechas

    Arguments:
    X_train: Training set  (num_px * num_px * 3, m_train)
    Y_train: Training labels (1, m_train)
    X_test: Test set (num_px * num_px * 3, m_test)
    Y_test: Test labels (1, m_test)
    num_iterations: Número de iteraciones
    learning_rate: Learning rate
    print_cost: True para mostrar la pérdida cada 100 pasos

    Returns:
    d: Dictionario conteniendo la info del modelo
    """

    # Inicializar parámetros a cero
    w = np.zeros((X_train.shape[0], 1))
    b = 0

    # Gradiente descendente
    params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)

    # Obtener parámetros finales
    w = params["w"]
    b = params["b"]

    # Predicciones en entrenamiento y prueba
    Y_prediction_train = predict(w, b, X_train)
    Y_prediction_test = predict(w, b, X_test)

    # Print train/test Errors
    if print_cost:
        print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
        print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

In [26]:

```
logistic_regression_model = model(train_set_x, train_labels, test_set_x, test_labels, num_iterations=2000, learning_rate=0.005, print_cost=True)
```

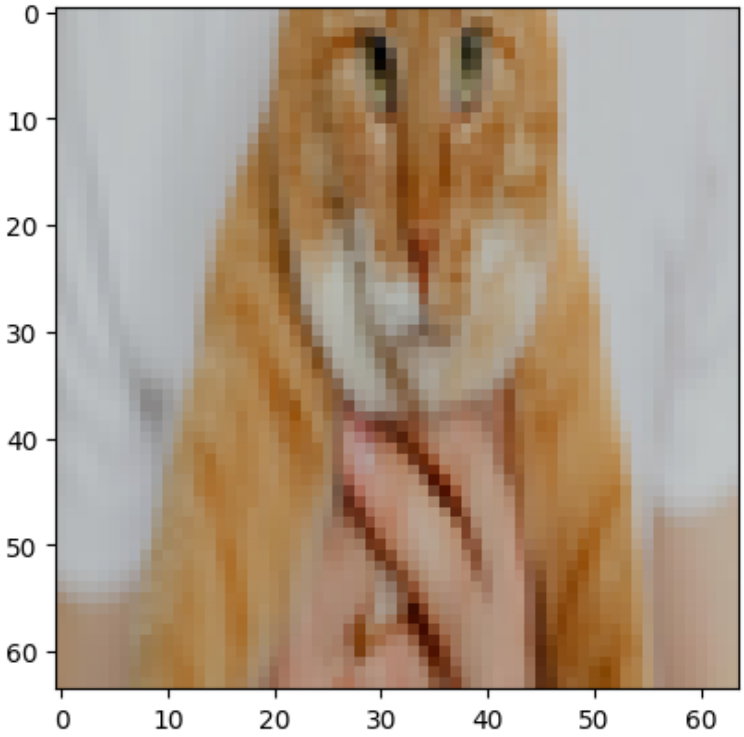
```
Costo en iteración 0: 0.693147
Costo en iteración 100: 2.052139
Costo en iteración 200: 1.878137
Costo en iteración 300: 1.758717
Costo en iteración 400: 1.663785
Costo en iteración 500: 1.582662
Costo en iteración 600: 1.510157
Costo en iteración 700: 1.443258
Costo en iteración 800: 1.380178
Costo en iteración 900: 1.319844
Costo en iteración 1000: 1.261642
Costo en iteración 1100: 1.205307
Costo en iteración 1200: 1.150853
```

```
Costo en iteración 1200: 1.150853
Costo en iteración 1300: 1.098467
Costo en iteración 1400: 1.048387
Costo en iteración 1500: 1.000794
Costo en iteración 1600: 0.955770
Costo en iteración 1700: 0.913290
Costo en iteración 1800: 0.873235
Costo en iteración 1900: 0.835411
train accuracy: 67.14542190305207 %
test accuracy: 50.71428571428571 %
```

In [27]:

```
# Example of a picture that was wrongly classified.
index = 1
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_labels[0,index]) + ", predice que este es un \"" + 'gato' if in
t(logistic_regression_model['Y_prediction_test'][0,index])==0 else 'perro' +  "\" pictur
e.")
```

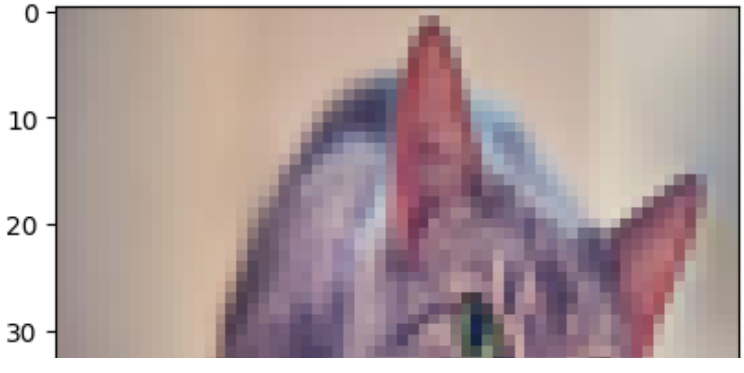
```
y = 0, predice que este es un "gato
```

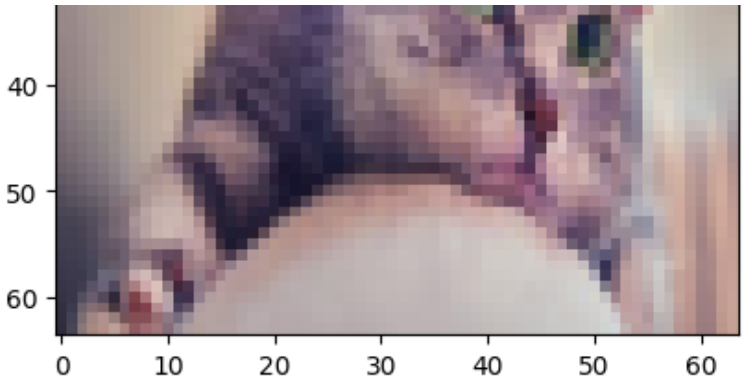



In [28]:

```
# Example of a picture that was wrongly classified.
index = 3
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print("y = " + str(test_labels[0, index]) +
      ", " +
      ("predice que este es un \"gato\" picture." if int(logistic_regression_model['Y_pre
diction_test'][0, index]) == 0
       else "predice que este es un \"perro\" picture."))
```

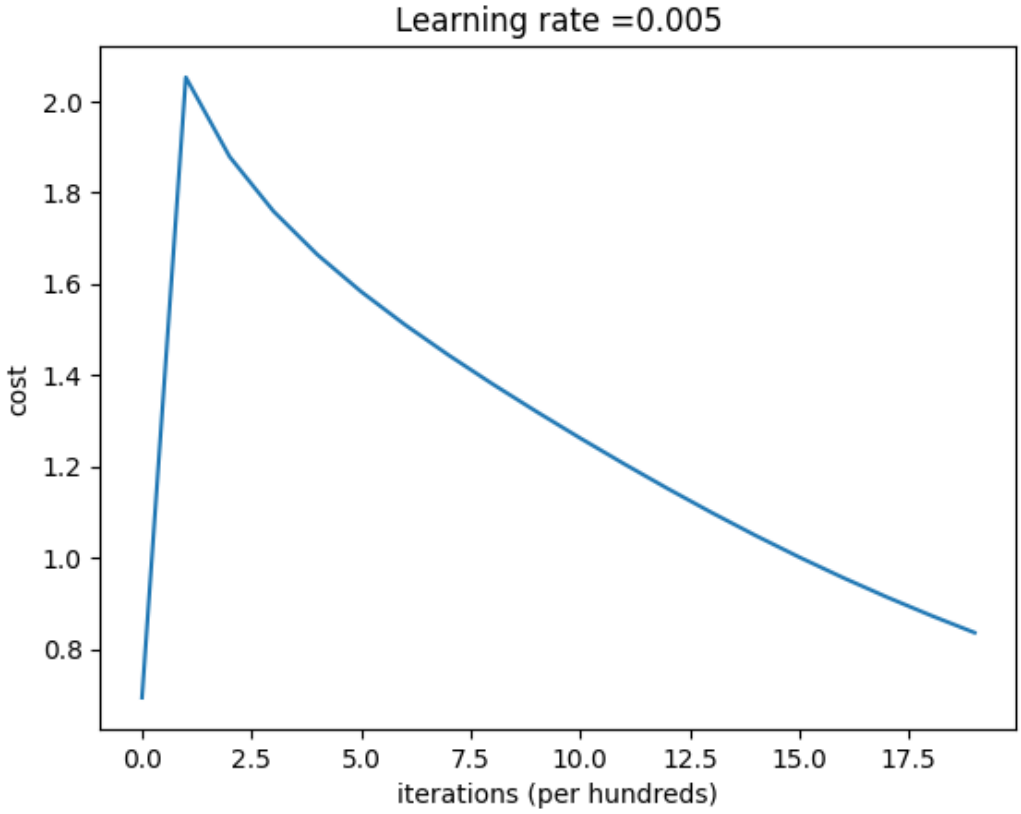
```
y = 0, predice que este es un "gato" picture.
```






In [29]:

```
# Plot learning curve (with costs)
costs = np.squeeze(logistic_regression_model['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(logistic_regression_model["learning_rate"]))
plt.show()
```




**NOTA: Dentro de los comentarios de la entrega (en Canvas) asegurese de contestar**

1. **¿Qué se podría hacer para mejorar el rendimiento de esta red?**
2. **Interprete la gráfica de arriba**

# Parte 2 - Red Neuronal Simple con PyTorch

**Para esta parte seguiremos usando el mismo dataset que anteriormente teníamos.**

**Entonces volvamos a cargar las imagenes por paz mental :)**

In [30]:

```
train_images = []
train_labels = []
test_images = []
test_labels = []
```

```
# Call the function for both the 'train' and 'test' folders
train_cats_path = os.path.join(data_dir, 'train', 'cats')
train_dogs_path = os.path.join(data_dir, 'train', 'dogs')
test_cats_path = os.path.join(data_dir, 'test', 'cats')
test_dogs_path = os.path.join(data_dir, 'test', 'dogs')


# Read images
target_size = (64, 64)
read_images(train_cats_path, "cats", target_size)
read_images(train_dogs_path, "dogs", target_size)
read_images(test_cats_path, "cats", target_size)
read_images(test_dogs_path, "dogs", target_size)


# Convert the lists to numpy arrays
train_images = np.array(train_images)
train_labels = np.array(train_labels)
test_images = np.array(test_images)
test_labels = np.array(test_labels)
```

**Nuevas librerías a usar**

**Asegúrense de instalar las librerías que les hagan falta del siguiente grupo de import.**

**Recuerden usar virtual envs!**

In [31]:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import torch.utils.data as data
import random


# Seed all possible
seed_ = 2023
random.seed(seed_)
np.random.seed(seed_)
torch.manual_seed(seed_)

# If using CUDA, you can set the seed for CUDA devices as well
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed_)
    torch.cuda.manual_seed_all(seed_)

import torch.backends.cudnn as cudnn
cudnn.deterministic = True
cudnn.benchmark = False
```

**Para poder usar PyTorch de una mejor manera con nuestro dataset de imagenes, tendremos que "formalizar" la manera en que cargamos las imagenes. Para ello crearemos una clase que represente el Dataset con el que estaremos trabajando**

In [32]:

```
class CatsAndDogsDataset(data.Dataset):
    def __init__(self, data_dir, target_size=(28, 28), color_mode='RGB', train=True):
        self.data_dir = data_dir
        self.target_size = target_size
        self.color_mode = color_mode
        self.classes = ['cats', 'dogs']
        self.train = train
        self.image_paths, self.labels = self.load_image_paths_and_labels()
```

```
    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = Image.open(image_path)
        image = image.convert(self.color_mode)
        image = image.resize(self.target_size)
        image = np.array(image)
        image = (image / 255.0 - 0.5) / 0.5  # Normalize to range [-1, 1]
        image = torch.tensor(image, dtype=torch.float32)
        image = image.view(-1)

        label = torch.tensor(self.labels[idx], dtype=torch.long)

        return image, label

    def load_image_paths_and_labels(self):
        image_paths = []
        labels = []
        for class_idx, class_name in enumerate(self.classes):
            class_path = os.path.join(self.data_dir, 'train' if self.train else 'test',
class_name)
            for filename in os.listdir(class_path):
                image_path = os.path.join(class_path, filename)
                image_paths.append(image_path)
                labels.append(class_idx)
        return image_paths, labels
```

## Definición de la red neuronal

**Una de las formas de definir una red neuronal con PyTorch es através del uso de clases. En esta el constructor usualmente tiene las capas que se usaran, mientras que la función que se extiende "forward()" hace clara la relación entre las capas.**

**Para poder entenderlo, hay que leer desde la función más interna hacia afuera y de arriba hacia abajo. Por ejemplo, en la línea 8, la capa fc1 (que es una lineal), pasa luego a una función de activación ReLU, despues la información pasa a una segunda lineal (fc2), para finalmente pasar por una función de activación SoftMax**

In [33]:

```
class SimpleClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))    # Feedforward step: Compute hidden layer activatio
ns
        x = self.fc2(x)               # Feedforward step: Compute output layer activation
s
        return F.log_softmax(x, dim=1)
```

## Definición de la función de entrenamiento

**Una forma de entrenar una red neuronal con PyTorch es, tras haber definido el modelo, se pasa a definir una función que se encargará de realizar el entrenamiento. Esto incluye tanto el paso de feedforward como el de back propagation.**

**Deberá terminar de implementar las funciones dadas según se solicita**

In [34]:

```
loss_history = [] # DO NOT DELETE
```

```
def train_model(model, train_loader, optimizer, criterion, epochs):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            inputs = inputs.view(-1, input_size)

            # Forward pass
            outputs = model(inputs)

            # Calcular la pérdida
            loss = criterion(outputs, labels)

            # Backpropagation
            optimizer.zero_grad()    # Limpiar gradientes previos
            loss.backward()          # Calcular gradientes

            # Actualizar parámetros
            optimizer.step()

            running_loss += loss.item()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}")
        loss_history.append(running_loss/len(train_loader))

    print("Training complete!")
```

In [35]:

```
input_size = 64 * 64 * 3
hidden_size = 125
output_size = 2  # 2 classes: cat and dog

model = SimpleClassifier(input_size, hidden_size, output_size)
optimizer = optim.SGD(model.parameters(), lr=0.01)
criterion = nn.NLLLoss()

# Loading datasets
train_dataset = CatsAndDogsDataset(data_dir, target_size=(64, 64), color_mode='RGB', tra
in=True)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
```

In [36]:

```
train_model(model, train_loader, optimizer, criterion, epochs=5)
```

```
Epoch 1/5, Loss: 0.6918629507223765
Epoch 2/5, Loss: 0.6414735416571299
Epoch 3/5, Loss: 0.6138788494798872
Epoch 4/5, Loss: 0.5827290481991239
Epoch 5/5, Loss: 0.5411496924029456
Training complete!
```

In [37]:

```
print("Loss:", loss_history)
```

```
Loss: [0.6918629507223765, 0.6414735416571299, 0.6138788494798872, 0.5827290481991239, 0.
5411496924029456]
```

**También necesitamos una forma de probar nuestro modelo para ello usamos la siguiente**

In [38]:

```
def test_model(model, test_loader):
    """
    Evaluate the performance of a trained neural network model on the test data.

    Arguments:
    model: The trained neural network model to be evaluated.
```

```
    test_loader: The DataLoader containing the test data and labels.
    """

    model.eval()  # Set the model in evaluation mode

    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs = inputs.view(-1, input_size)
            labels = labels.view(-1)  # Reshape the labels to be compatible with NLLLoss
()

            # Forward pass
            outputs = model(inputs)

            # Get predictions
            _, predicted = torch.max(outputs.data, 1)

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy:.2f}%")
    return accuracy
```

In [39]:

```
test_dataset = CatsAndDogsDataset(data_dir, target_size=(64, 64), color_mode='RGB', trai
n=False)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=True)
```

In [40]:

```
# Evaluate the model on the test dataset
asset_accuracy = test_model(model, test_loader)

asset_accuracy
```

```
Test Accuracy: 53.57%
```

Out[40]:

```
53.5714285714286
```

**NOTA: Dentro de los comentarios de la entrega (en Canvas) asegurese de contestar**

1. **¿En qué consiste `optim.SGD`?**
2. **¿En qué consiste `nn.NLLLoss`?**
3. **¿Qué podría hacer para mejorar la red neuronal, y si no hay mejoras, por qué?**

**Al preguntarlse "en qué consiste...", se espera que las expliquelas en sus propias palabras**

# Calificación

**Asegúrese de que su notebook corra sin errores (quite o resuelva los raise NotImplementedError()) y luego reinicie el kernel y vuelva a correr todas las celdas para obtener su calificación correcta**

In [41]:

```
print()
print("La fraccion de abajo muestra su rendimiento basado en las partes visibles de este
laboratorio")
tick.summarise_marks() #
```

```
La fraccion de abajo muestra su rendimiento basado en las partes visibles de este laborat
orio
```

## 5 / 5 marks (100.0%)

In [ ]: