Image Application: cat or not?

After this assignment you will be able to:

- Build and apply a deep neural network to supervised learning.
- Learn how to do data normalization
- learn how to do weight initialization
- Learn how to use regularization

Let's get started!

1 - Packages

Exercise: Please mount your Google drive, and set up your working folder here.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mou
```

```
import os
# start your code here
os.chdir("/content/drive/MyDrive/DL/Homework3") # change your working folder here
# end your code here
```

Let's first import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- matplotlib is a library to plot graphs in Python.
- h5py is a common package to interact with a dataset that is stored on an H5 file.
- PIL and scipy are used here to test your model with your own picture at the end.
- dnn_app_utils provides the customized functions that will be used in this notebook.
- np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

```
import time
import numpy as np
import tensorflow as tf
import h5py
```

```
import matplotlib.pyplot as plt
import dnn_app_utils_v3 as du
from skimage.transform import rescale, resize, downscale_local_mean

# refresh 'planar_utils' module
import imp
imp.reload(du)

%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)
tf.random.set_seed(1)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2 - Dataset

Problem Statement: You are given a dataset ("data.h5") containing: - a training set of m_train images labelled as cat (1) or non-cat (0) - a test set of m_test images labelled as cat and non-cat - each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by running the cell below.

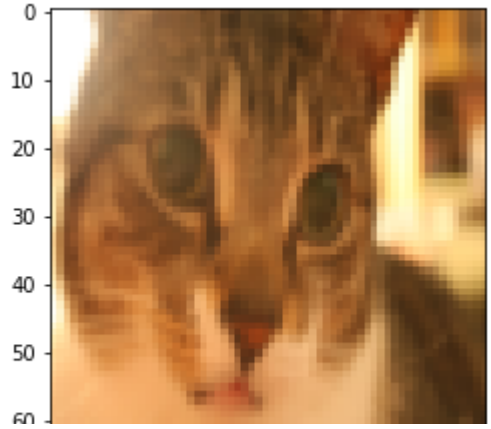
The training input and output are stored in 'train_x_orig' and 'train_y'. The test input and output are stored in 'test_x_orig' and 'test_y'.

```
train_x_orig, train_y, test_x_orig, test_y, classes = du.load_data()
#print(classes)
```

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to see other images.

```
# Example of a picture
index = 200
plt.imshow(train_x_orig[index])
print("y = " + str(train_y[index,0]) + ". It's a " + classes[train_y[index,0]].decode("utf-8'
#print(train_x_orig[index].shape,train_x_orig[index])
```

```
y = 1. It's a cat picture.
```



Let's explore more about the training data. We are curious about the shape of each image and can use 'train_x_orig[index].shape' to expore it. **Exercise**: Can you tell what the pixel number of each image is?

```
# Explore your dataset
image_shape=train_x_orig[index].shape
print("Each image is of size: ", image_shape)
#start your code here
num_px = m_train   # the number of pixels in each column of the image. Hint: The 1st element
#end your code here
print('Number of pixels in each column of the image:', num_px)
```

```
Each image is of size:  (64, 64, 3)
Number of pixels in each column of the image: 209
```

Expected result:

Each image is of size: (64, 64, 3)

Number of pixels in each column of the image: 209

We just take a close look at an image of the image set. Now, let's take a look at the whole training set.

Exercise: The first 4 lines in the following cell print out the shape of the trining images. There are 209 images, and each image has 64 by 64 pixcels and GRB color.

Please print out the shape of the test input (stored in `test_x_orig`) and output (stored in `test_y`) data, and tell how many images we have in the test data.

```
train_x_shape=train_x_orig.shape
m_train = train_x_shape[0]
print ("train_x_orig shape: ",train_x_shape)
print ("Number of training examples: " ,m_train)
```

```
# start your code here
test_x_shape = test_x_orig.shape
test_y_shape = test_y.shape
m_test = test_y_shape[0]
# end your code here

print ("test_x_orig shape: " ,test_x_shape)
print ("test_y shape: " , test_y_shape)
print ("Number of test examples: " ,m_test)
```

```
train_x_orig shape:  (209, 64, 64, 3)
Number of training examples:  209
test_x_orig shape:  (50, 64, 64, 3)
test_y shape:  (50, 1)
Number of test examples:  50
```

Expected result:

train_x_orig shape: (209, 64, 64, 3)

Number of training examples: 209

test_x_orig shape: (50, 64, 64, 3)

test_y shape: (50, 1)

Number of test examples: 50

As mentioned in our lecture, you reshape and standardize the images before feeding them to the network. The code is given in the cell below.

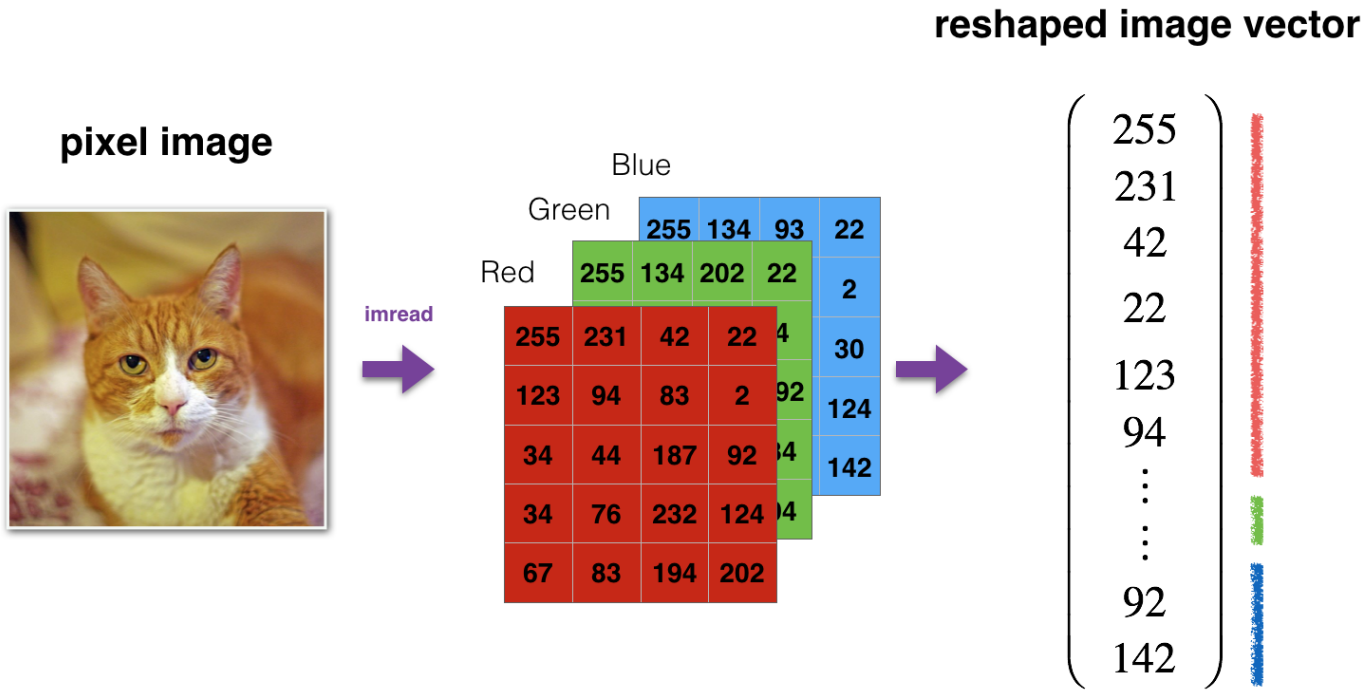


Figure 1: Image to vector conversion.

```
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_shape[0], -1)   # The "-1" makes reshape flatt
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1)
```

Data normalization can help us accelerate the training process. The RGB values of an image are integers in the range of [0,255]. We can normalize the training images to have the feature values between 0 and 1 by using `train_x = train_x_flatten/255.0`.

Exercise: We normalize the training images using `train_x = train_x_flatten/255.0` in the 1st line. Please normalize the test images to have feature values between 0 and 1.

```
# Standardize data to have feature values between 0 and 1.
train_x = train_x_flatten/255.0

#start your code here
test_x = test_x_flatten/255.0
#end your code here

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))
print ("train_y's shape: " + str(train_y.shape))
print ("test_y's shape: " + str(test_y.shape))
```

```
train_x's shape: (209, 12288)
test_x's shape: (50, 12288)
train_y's shape: (209, 1)
test_y's shape: (50, 1)
```

Expected result:

train_x's shape: (209, 12288)

test_x's shape: (50, 12288)

train_y's shape: (209, 1)

test_y's shape: (50, 1)

3- Build your neural network

Now, we are going to use the skills in the first module to build our neural network.

Excep the input layer, you can use

`tf.keras.layers.Dense(20,activation='relu', kernel_initializer='glorot_uniform', bias_initializer='zeros')`

to build the layers. In this example, we have 20 neurons in this layer, use 'relu' as the activation function, initialize our weight randomly using 'glorot_uniform', and our bias is initialized to be 'zeros'.

First, we will need an input layer. How many neurons will you need in the input layer? Hint: how many elements does every training input have?

Second, we will build several hidden layers. You are free to choose the number of neurons in every hidden layer.

Last, we will build the output layer. How many neurons will you need in the output layer? Which activation function is more suitale in this problem? Hint: how many elements does every training output have? What is the range of the training output?

```
tf.random.set_seed(1) # we fixed the seed of the random number generator, so every time runni

# start your code here
cat_model=tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(12288,)), # how many elements are there in every
    tf.keras.layers.Dense(20,activation='relu', # in layer 1, we will have 20 neurons and use
                          kernel_initializer='glorot_uniform', # we will randomly choose th
                          bias_initializer='zeros'), # we will initialize all bias to be ze
    #please follow the example of layer 1 to add more hidden layers here.

    # End adding hidden layers
    tf.keras.layers.Dense(1,activation='sigmoid', # how many elements are there in every trai
                          kernel_initializer='glorot_uniform',
                          bias_initializer='zeros'),
])
# end your code here
cat_model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 20)                245780

 dense_9 (Dense)             (None, 1)                 21

=================================================================
Total params: 245,801
Trainable params: 245,801
Non-trainable params: 0
_________________________________________________________________
```

Now, let's compile the model. When comiling your model, you will need to specify the optimizer, the loss function, and the metrics.

For optimizers, the most popular ones are 'Adam' and its derivatives. You can also try 'RMSprop' and 'SGD' to see which one works better for you. You might need adjust the learning rate to acceperate the training process.

For loss functions, our problem is a two-class classification problem. The most popular loss function would be 'BinaryCrossentropy'. If this is a multiple-class classification problem, you probably want to try 'CategoricalCrossentropy' or 'SparseCategoricalCrossentropy'.

For the metrics, we want to know the accuracy of our model. Therefore, I would recommend 'accuracy'.

```
# start your code here
cat_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005),  # Optimizer. Pl
    # Loss function to minimize
    loss=tf.keras.losses.BinaryCrossentropy(),
    # List of metrics to monitor
    metrics=['accuracy'],
                )
# end your code here
```

It's time to train.

We can plot training history. It can help us to adjust the parameters.

If the training loss increases, we may want to decrease the `learning_rate`.

If the training loss has large vibration, we may want to decrease the `learning_rate`.

If the training loss decreases too slowly, we need to increase `learning_rate`.

If the training loss decreases at a good speed, but ends up large, we may want to increase the epoch number.

```
# start your code here
history = cat_model.fit(
    train_x, # input data
    train_y, # real output data
    epochs=100, # how many iterations do you want to train your model? Your data will be reus
)
# end your code here.
plt.figure()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['loss','val_loss'])
```

```
7/7 [==============================] - 0s 6ms/step - loss: 0.0520 - accuracy: 1.0000
Epoch 86/100
7/7 [==============================] - 0s 7ms/step - loss: 0.0518 - accuracy: 0.9952
Epoch 87/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0515 - accuracy: 1.0000
Epoch 88/100
7/7 [==============================] - 0s 8ms/step - loss: 0.0513 - accuracy: 0.9952
Epoch 89/100
7/7 [==============================] - 0s 7ms/step - loss: 0.0505 - accuracy: 1.0000
Epoch 90/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0501 - accuracy: 1.0000
Epoch 91/100
7/7 [==============================] - 0s 7ms/step - loss: 0.0506 - accuracy: 0.9952
Epoch 92/100
7/7 [==============================] - 0s 7ms/step - loss: 0.0496 - accuracy: 1.0000
Epoch 93/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0498 - accuracy: 0.9952
Epoch 94/100
7/7 [==============================] - 0s 7ms/step - loss: 0.0494 - accuracy: 0.9952
Epoch 95/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0493 - accuracy: 0.9952
Epoch 96/100
7/7 [==============================] - 0s 7ms/step - loss: 0.0517 - accuracy: 0.9952
Epoch 97/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0487 - accuracy: 0.9952
Epoch 98/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0486 - accuracy: 0.9952
Epoch 99/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0489 - accuracy: 1.0000
Epoch 100/100
7/7 [==============================] - 0s 6ms/step - loss: 0.0472 - accuracy: 1.0000
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-64-62100fe6d100> in <module>()
      8 plt.figure()
      9 plt.plot(history.history['loss'])
---> 10 plt.plot(history.history['val_loss'])
     11 plt.legend(['loss','val_loss'])

KeyError: 'val_loss'
```

SEARCH STACK OVERFLOW