1 - Import 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.mount("/content/drive", force_remo
```

```
import os
# start your code here
os.chdir("/content/drive/MyDrive/DL/Homework6") # change your working folder here

# end your code here
```

```
import math
import numpy as np
import h5py
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.python.framework import ops
from tf_utils import load_dataset, random_mini_batches, convert_to_one_hot, predict

%matplotlib inline
np.random.seed(123)
tf.random.set_seed(123)
```

2 - Problem statement: SIGNS Dataset

One afternoon, with some friends we decided to teach our computers to decipher sign language. We spent a few hours taking pictures in front of a white wall and came up with the following dataset. It's now your job to build an algorithm that would facilitate communications from a speech-impaired person to someone who doesn't understand sign language.

- **Training set**: 1080 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (180 pictures per number).
- **Test set**: 120 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (20 pictures per number).

Note that this is a subset of the SIGNS dataset. The complete dataset contains many more signs.

Here are examples for each number, and how an explanation of how we represent the labels. These are the original pictures, before we lowered the image resolutoion to 64 by 64 pixels.

Image Caption

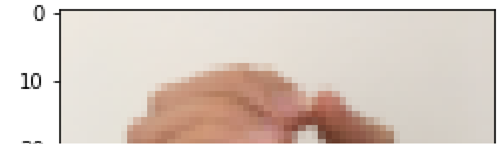
Run the following code to load the dataset.

```
# Loading the dataset
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset()
```

Change the index below and run the cell to visualize some examples in the dataset.

```
# Example of a picture
index = 1
plt.imshow(X_train_orig[index])
print ("y = " + str(np.squeeze(Y_train_orig[:, index])))
```

```
y = 0
```



3. Using One Hot encodings

Many times in deep learning you will have a y vector with numbers ranging from 0 to C-1, where C is the number of classes. If C is for example 4, then you might have the following y vector which you will need to convert as follows:

$$y = \begin{bmatrix} 1 & 2 & 3 & 0 & 2 & 1 \end{bmatrix} \text{ is often converted to } \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{array}{l} \text{class} = 0 \\ \text{class} = 1 \\ \text{class} = 2 \\ \text{class} = 3 \end{array}$$

This is called a "one hot" encoding, because in the converted representation exactly one element of each column is "hot" (meaning set to 1). In tensorflow, we can tranform the labels of samples to one hot vectors by calling

```
tf.keras.utils.to_categorical(y)
```

Please transform the lables (or the output of the dataset) to one hot vectors.

As usual you flatten the image dataset, then normalize it by dividing by 255. On top of that, you will convert each label to a one-hot vector . Run the cell below to do so.

```
# Flatten the training and test images
X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1)
X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1)
# Normalize image vectors
X_train = X_train_flatten/255.
X_test = X_test_flatten/255.
# Flatten the training and test labels
Y_train_flatten=Y_train_orig.flatten()
Y_test_flatten=Y_test_orig.flatten()

# Convert training labels (Y_train_flatten) and test labels (Y_test_flatten) to one hot matrices\
### START CODE HERE ###'
Y_train = tf.keras.utils.to_categorical(Y_train_flatten)
Y_test = tf.keras.utils.to_categorical(Y_test_flatten)
### END CODE HERE ###

print ("number of training examples = " + str(X_train.shape[1]))
print ("number of test examples = " + str(X_test.shape[1]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))
```

```
number of training examples = 12288
number of test examples = 12288
X_train shape: (1080, 12288)
Y_train shape: (1080, 6)
X_test shape: (120, 12288)
Y_test shape: (120, 6)
```

Note that 12288 comes from $64 \times 64 \times 3$. Each image is square, 64 by 64 pixels, and 3 is for the RGB colors. Please make sure all these shapes make sense to you before continuing.

4.Building your model

Your goal is to build an model capable of recognizing a sign with high accuracy. To do so, you are going to build a tensorflow model that is almost the same as one you have previously built for cat recognition and multi-class classification problems. If you already forget previous homeworks, please go back to review them to get some clues.

Previous related homeworks

"Planar_data_tensorflow.ipynb" in Homework 2

"deepNN_tensorflow.ipynb" in Homework 3

"regularized_deepNN_tensorflow.ipynb" in Homework 4

"batch-size-test-student-version.ipynb" In Homework 5

```
### START CODE HERE ###
model=tf.keras.Sequential([
    #Input layer
    tf.keras.layers.Input(shape=(12288)), # Set how many elements are there in every training image?

    #Start adding hidden layers
    tf.keras.layers.Dense(25,activation='relu', # how many elements are there in every training output
                            kernel_initializer='glorot_uniform',
                            bias_initializer='zeros'), # Please set your neuron numbers, activation function and so on.

    #please feel free to add more hidden layers here.
     tf.keras.layers.Dense(50, activation='relu',
                           kernel_initializer= 'glorot_uniform',
                           bias_initializer= 'zeros'),
    tf.keras.layers.Dense(75, activation='relu',
                          kernel_initializer= 'glorot_uniform',
                          bias_initializer= 'zeros'),
    # End adding hidden layers

    #output layer
    tf.keras.layers.Dense(6, activation='softmax') # Set how many elements are there in every training output

])
### END CODE HERE ###'
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 25)                307225

 dense_1 (Dense)             (None, 50)                1300

 dense_2 (Dense)             (None, 75)                3825

 dense_3 (Dense)             (None, 6)                 456

=================================================================
Total params: 312,806
Trainable params: 312,806
Non-trainable params: 0
_________________________________________________________________
```

```
### START CODE HERE ###'
model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate=0.005), # Optimizer. Please change 'None' to some meaningful
              loss = tf.keras.losses.BinaryCrossentropy(),  # loss. Please change 'None' to some meaningful loss
              metrics=['accuracy'], # List of metrics to monitor
                 )
### END CODE HERE ###'


### START CODE HERE ###'
history = model.fit(
    X_train, #  training input data
    Y_train, # training output data
    batch_size=10, # how many samples (points) do you want to use to update weights once?
    epochs=100 # how many iterations do you want to train your model? Your data will be reused to train the model for the nu
)
### END CODE HERE ###'
```

```
108/108 [==============================] - 0s 4ms/step - loss: 0.0248 - accuracy: 0.9898
Epoch 71/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0234 - accuracy: 0.9907
Epoch 72/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0241 - accuracy: 0.9852
Epoch 73/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0251 - accuracy: 0.9880
```

```
108/108 [==============================] - 0s 4ms/step - loss: 0.0231 - accuracy: 0.9880
Epoch 74/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0235 - accuracy: 0.9926
Epoch 75/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0244 - accuracy: 0.9824
Epoch 76/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0216 - accuracy: 0.9889
Epoch 77/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0257 - accuracy: 0.9852
Epoch 78/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0198 - accuracy: 0.9954
Epoch 79/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0617 - accuracy: 0.9639
Epoch 80/100
108/108 [==============================] - 1s 7ms/step - loss: 0.0285 - accuracy: 0.9806
Epoch 81/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0206 - accuracy: 0.9926
Epoch 82/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0450 - accuracy: 0.9630
Epoch 83/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0213 - accuracy: 0.9898
Epoch 84/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0171 - accuracy: 0.9944
Epoch 85/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0292 - accuracy: 0.9870
Epoch 86/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0268 - accuracy: 0.9889
Epoch 87/100

108/108 [==============================] - 0s 4ms/step - loss: 0.0184 - accuracy: 0.9935
Epoch 88/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0428 - accuracy: 0.9731
Epoch 89/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0170 - accuracy: 0.9972
Epoch 90/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0189 - accuracy: 0.9926
Epoch 91/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0218 - accuracy: 0.9907
Epoch 92/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0195 - accuracy: 0.9935
Epoch 93/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0145 - accuracy: 0.9963
Epoch 94/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0157 - accuracy: 0.9963
```

```
108/108 [==============================] - 0s 4ms/step - loss: 0.0157 - accuracy: 0.9963
Epoch 95/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0164 - accuracy: 0.9963
Epoch 96/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0176 - accuracy: 0.9954
Epoch 97/100
108/108 [==============================] - 0s 4ms/step - loss: 0.1111 - accuracy: 0.9241
Epoch 98/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0324 - accuracy: 0.9796
Epoch 99/100
108/108 [==============================] - 0s 4ms/step - loss: 0.0585 - accuracy: 0.9500
```

If the training works, congratulations! You have built a model for recognizing SIGN language.

```
model.evaluate(X_test,Y_test)
```

```
4/4 [==============================] - 0s 6ms/step - loss: 0.1127 - accuracy: 0.9083
[0.11268094927072525, 0.9083333611488342]
```

5. Grading rule

***Try your best to achieve high accuarcy. The grade for this homework will based on your model final training accurracy and test accuracy. ***

For example, if your final training accurracy is 87.8 (consider one decimal) and test accuracy is 70.9, the total grade is 8.78 + 7.09 =15.87. The full credit is 10+10 =20. It is hard to get full credit in this homework, please try your best.

Tips

You can try to improve its accuracy by:

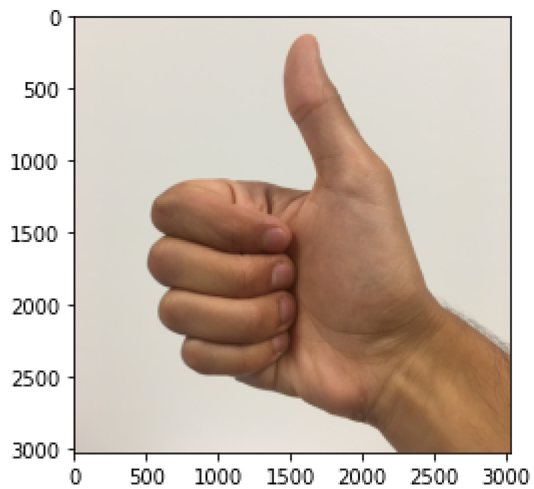
1. tuning the hyperparameters (learning rate, batch size, epoches and so on)
2. adjusting your layers numbers or nuron numbers.
3. changing optimizer or loss function
4. using dropout layer

5. add regularizers.L2
6. .....

Once again, here's a thumbs up for your work!

```
%pylab inline
import matplotlib.image as mpimg
img = mpimg.imread('images/thumbs_up.jpg')
imgplot = plt.imshow(img)
plt.show()
```

```
Populating the interactive namespace from numpy and matplotlib
```



0s completed at 9:33 PM