TensorFlow Tutorial

Welcome to this week's programming assignment. In this assignment, you will learn to do the following in TensorFlow:

- Initialize variables
- Start your own session

Programing frameworks can not only shorten your coding time, but sometimes also perform optimizations that speed up your code.

1 - Exploring the Tensorflow Library

To start, you will import the library:

In [1]:

```
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(1)
```

Now that you have imported the library, we will walk you through its different applications. You will start with an example, where we compute for you the loss of one training example.

$$loss = \mathcal{L}(\hat{y}, y) = (\hat{y}^{(i)} - y^{(i)})^2 \tag{1}$$

In [2]:

```
y_hat = tf.constant(36, name='y_hat')            # Define y_hat constant. Set to 36.
y = tf.constant(39, name='y')                    # Define y. Set to 39

loss = tf.Variable((y - y_hat)**2, name='loss')  # Create a variable for the loss

init = tf.global_variables_initializer()         # When init is run later (session.r
                                                 # the loss variable will be initia
with tf.Session() as session:                    # Create a session and print the ou
    session.run(init)                            # Initializes the variables
    print(session.run(loss))                     # Prints the loss
```

9

Writing and running programs in TensorFlow has the following steps:

1. Create Tensors (variables) that are not yet executed/evaluated.
2. Write operations between those Tensors.
3. Initialize your Tensors.
4. Create a Session.
5. Run the Session. This will run the operations you'd written above.

Therefore, when we created a variable for the loss, we simply defined the loss as a function of other quantities, but did not evaluate its value. To evaluate it, we had to run `init=tf.global_variables_initializer()`. That initialized the loss variable, and in the last line we were finally able to evaluate the value of `loss` and print its value.

Now let us look at an easy example. Run the cell below:

In [3]:

```
a = tf.constant(2)
b = tf.constant(10)
c = tf.multiply(a,b)
print(c)
```

```
Tensor("Mul:0", shape=(), dtype=int32)
```

As expected, you will not see 20! You got a tensor saying that the result is a tensor that does not have the shape attribute, and is of type "int32". All you did was put in the 'computation graph', but you have not run this computation yet. In order to actually multiply the two numbers, you will have to create a session and run it.

In [4]:

```
sess = tf.Session()
print(sess.run(c))
```

```
20
```

Great! To summarize, **remember to initialize your variables, create a session and run the operations inside the session**.

Next, you'll also have to know about placeholders. A placeholder is an object whose value you can specify only later. To specify values for a placeholder, you can pass in values by using a "feed dictionary" (`feed_dict` variable). Below, we created a placeholder for x. This allows us to pass in a number later when we run the session.

In [5]:

```
# Change the value of x in the feed_dict

x = tf.placeholder(tf.int64, name = 'x')
print(sess.run(2 * x, feed_dict = {x: 3}))
sess.close()
```

```
6
```

When you first defined `x` you did not have to specify a value for it. A placeholder is simply a variable that you will assign data to only later, when running the session. We say that you **feed data** to these placeholders when running the session.

Here's what's happening: When you specify the operations needed for a computation, you are telling TensorFlow how to construct a computation graph. The computation graph can have some placeholders whose values you will specify only later. Finally, when you run the session, you are telling TensorFlow to execute the computation graph.

1.1 - Linear function

Lets start this programming exercise by computing the following equation: $Y = WX + b$, where W and X are random matrices and b is a random vector.

Exercise: Compute $WX + b$ where W, X, and b are drawn from a random normal distribution. W is of shape (4, 3), X is (3,1) and b is (4,1). As an example, here is how you would define a constant X that has shape (3,1):

```
X = tf.constant(np.random.randn(3,1), name = "X")
```

You might find the following functions helpful:

- tf.matmul(..., ...) to do a matrix multiplication
- tf.add(..., ...) to do an addition
- np.random.randn(...) to initialize randomly

In [6]:

```
# GRADED FUNCTION: linear_function

def linear_function():
    """
    Implements a linear function:
            Initializes W to be a random tensor of shape (4,3)
            Initializes X to be a random tensor of shape (3,1)
            Initializes b to be a random tensor of shape (4,1)
    Returns:
    result -- runs the session for Y = WX + b
    """

    np.random.seed(1)

    ### START CODE HERE ### (4 lines of code)
    X = tf.constant(np.random.randn(3,1), name = "X")
    W = tf.constant(np.random.randn(4,3), name = "W")
    b = tf.constant(np.random.randn(4,1), name = "X")
    Y = tf.add(tf.matmul(W, X), b)
    ### END CODE HERE ###

    # Create the session using tf.Session() and run it with sess.run(...) on the va

    ### START CODE HERE ###
    sess = tf.Session()
    result = sess.run(Y)
    ### END CODE HERE ###

    # close the session
    sess.close()

    return result
```

In [7]:

```
print( "result = " + str(linear_function()))
```

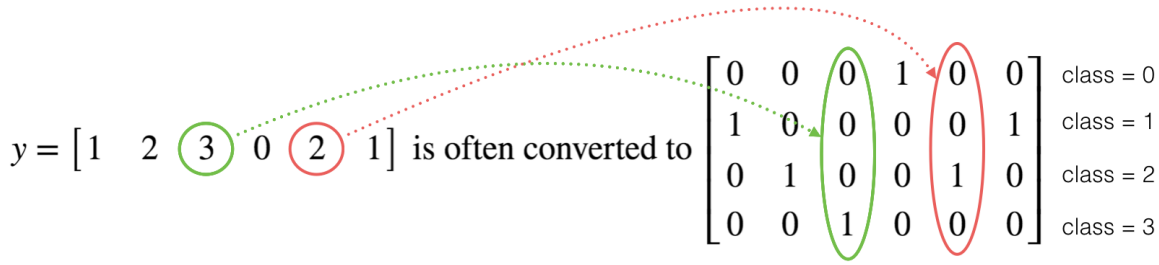
```
result = [[-2.15657382]
 [ 2.95891446]
 [-1.08926781]
 [-0.84538042]]
```

*** Expected Output ***: 아래와 같이 출력이 되어야 함

result [[-2.15657382] [2.95891446] [-1.08926781] [-0.84538042]]

1.4 - 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:



This is called a "one hot" encoding, because in the converted representation exactly one element of each column is "hot" (meaning set to 1). To do this conversion in numpy, you might have to write a few lines of code. In tensorflow, you can use one line of code:

- tf.one_hot(labels, depth, axis)

Exercise: Implement the function below to take one vector of labels and the total number of classes C, and return the one hot encoding. Use `tf.one_hot()` to do this.

In [8]:

```
# GRADED FUNCTION: one_hot_matrix

def one_hot_matrix(labels, C):
    """
    Creates a matrix where the i-th row corresponds to the ith class number and the
                    corresponds to the jth training example. So if example j had a
                    will be 1.

    Arguments:
    labels -- vector containing the labels
    C -- number of classes, the depth of the one hot dimension

    Returns:
    one_hot -- one hot matrix
    """

    ### START CODE HERE ###

    # Create a tf.constant equal to C (depth), name it 'C'. (approx. 1 line)
    C = tf.constant(C, name="C")

    # Use tf.one_hot, be careful with the axis (approx. 1 line)
    one_hot_matrix = tf.one_hot(labels, C, axis = 0)

    # Create the session (approx. 1 line)
    sess = tf.Session()

    # Run the session (approx. 1 line)
    one_hot = sess.run(one_hot_matrix)

    # Close the session (approx. 1 line). See method 1 above.
    sess.close()

    ### END CODE HERE ###

    return one_hot
```

In [9]:

```
labels = np.array([1,2,3,0,2,1])
one_hot = one_hot_matrix(labels, C = 4)
print ("one_hot = " + str(one_hot))
```

```
one_hot = [[0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0.]]
```

Expected Output:

one_hot [[0. 0. 0. 1. 0. 0.] [1. 0. 0. 0. 0. 1.] [0. 1. 0. 0. 1. 0.] [0. 0. 1. 0. 0. 0.]]

1.5 - Initialize with zeros and ones

Now you will learn how to initialize a vector of zeros and ones. The function you will be calling is `tf.ones()`. To initialize with zeros you could use tf.zeros() instead. These functions take in a shape and return an array of dimension shape full of zeros and ones respectively.

Exercise: Implement the function below to take in a shape and to return an array (of the shape's dimension of ones).

- tf.ones(shape)

In [10]:

```
# GRADED FUNCTION: ones

def ones(shape):
    """
    Creates an array of ones of dimension shape

    Arguments:
    shape -- shape of the array you want to create

    Returns:
    ones -- array containing only ones
    """

    ### START CODE HERE ###

    # Create "ones" tensor using tf.ones(...). (approx. 1 line)
    ones = tf.ones(shape)

    # Create the session (approx. 1 line)
    sess = tf.Session()

    # Run the session to compute 'ones' (approx. 1 line)
    ones = sess.run(ones)

    # Close the session (approx. 1 line). See method 1 above.
    sess.close()

    ### END CODE HERE ###
    return ones
```

In [11]:

```
print ("ones = " + str(ones([3])))
```

```
ones = [1. 1. 1.]
```

Expected Output:

ones	[1. 1. 1.]

In [12]:

```
# GRADED FUNCTION: create_placeholders

def create_placeholders(n_x, n_y):
    """
    Creates the placeholders for the tensorflow session.

    Arguments:
    n_x -- scalar, size of an image vector (num_px * num_px = 64 * 64 * 3 = 12288)
    n_y -- scalar, number of classes (from 0 to 5, so -> 6)

    Returns:
    X -- placeholder for the data input, of shape [n_x, None] and dtype "float"
    Y -- placeholder for the input labels, of shape [n_y, None] and dtype "float"

    Tips:
    - You will use None because it let's us be flexible on the number of examples yo
      In fact, the number of examples during test/train is different.
    """

    ### START CODE HERE ### (approx. 2 lines)
    X = tf.placeholder(tf.float32, shape=[n_x, None], name = 'X')
    Y = tf.placeholder(tf.float32, shape=[n_y, None], name = 'Y')
    ### END CODE HERE ###

    return X, Y
```

In [13]:

```
X, Y = create_placeholders(12288, 6)
print ("X = " + str(X))
print ("Y = " + str(Y))
```

```
X = Tensor("X_2:0", shape=(12288, ?), dtype=float32)
Y = Tensor("Y:0", shape=(6, ?), dtype=float32)
```

Expected Output:

X	Tensor("Placeholder_1:0", shape=(12288, ?), dtype=float32) (not necessarily Placeholder_1)
Y	Tensor("Placeholder_2:0", shape=(10, ?), dtype=float32) (not necessarily Placeholder_2)