Week 3 - prog ex: TF for Multi-class Classification (Softmax + ReLU)

笔记本:	DL 2 - Deep NN Hyperparameter Tuning, Regularization & Optimization		
创建时间:	2021/1/9 15:46	**更新时间:**	2021/1/9 16:34

Welcome to the Tensorflow Tutorial! In this notebook you will learn all the basics of Tensorflow. You will implement useful functions and draw the parallel with what you did using Numpy. You will understand what Tensors and operations are, as well as how to execute them in a computation graph.

After completing this assignment you will also be able to implement your own deep learning models using Tensorflow. In fact, using our brand new SIGNS dataset, you will build a deep neural network model to recognize numbers from 0 to 5 in sign language with a pretty impressive accuracy.

y = 0 y = 1 y = 2 y = 3 y = 4 y = 5

TensorFlow Tutorial

Welcome to this week's programming assignment. Until now, you've always used numpy to build neural networks. Now we will step you through a deep learning framework that will allow you to build neural networks more easily. Machine learning frameworks like TensorFlow, PaddlePaddle, Torch, Caffe, Keras, and many others can speed up your machine learning development significantly. All of these frameworks also have a lot of documentation, which you should feel free to read. In this assignment, you will learn to do the following in TensorFlow:

- Initialize variables
- Start your own session
- Train algorithms
- Implement a Neural Network

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

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:

[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.

[4]:
```
sess = tf.Session()
print(sess.run(c))
```

```
20
```

place holder

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 [ ]:
```
# 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()
```

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.

linear function

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

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

    np.random.seed(1)

    """
    Note, to ensure that the "random" numbers generated match the expected results,
    please create the variables in the order given in the starting code below.
    (Do not re-arrange the order).
    """
    ### 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 = 'X')
    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 variable you want to calculate

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

1.2 - Computing the sigmoid

Great! You just implemented a linear function. Tensorflow offers a variety of commonly used neural network functions like `tf.sigmoid` and `tf.softmax`. For this exercise lets compute the sigmoid function of an input.

You will do this exercise using a placeholder variable `x`. When running the session, you should use the feed dictionary to pass in the input `z`. In this exercise, you will have to (i) create a placeholder `x`, (ii) define the operations needed to compute the sigmoid using `tf.sigmoid`, and then (iii) run the session.

Exercise : Implement the sigmoid function below. You should use the following:

- `tf.placeholder(tf.float32, name = "...")`
- `tf.sigmoid(...)`
- `sess.run(..., feed_dict = {x: z})`

Note that there are two typical ways to create and use sessions in tensorflow:

Method 1:

```
sess = tf.Session()
# Run the variables initialization (if needed), run the operations
result = sess.run(..., feed_dict = {...})
sess.close() # Close the session
```

Method 2:

```
with tf.Session() as sess:
    # run the variables initialization (if needed), run the operations
    result = sess.run(..., feed_dict = {...})
    # This takes care of closing the session for you :)
```

```
    ### START CODE HERE ### ( approx. 4 lines of code)
    # Create a placeholder for x. Name it 'x'.
    x = tf.placeholder(tf.float32, name = 'x')

    # compute sigmoid(x)
    sigmoid = tf.sigmoid(x)

    # Create a session, and run it. Please use the method 2 explained above.
    # You should use a feed_dict to pass z's value to x.
    with tf.Session() as sess:
        # Run session and call the output "result"
        result = sess.run(sigmoid, feed_dict={x: z})

    ### END CODE HERE ###

    return result
```

To summarize, you how know how to:

1. Create placeholders
2. Specify the computation graph corresponding to operations you want to compute
3. Create the session
4. Run the session, using a feed dictionary if necessary to specify placeholder variables' values.

---

cross entropy loss

1.3 - Computing the Cost

You can also use a built-in function to compute the cost of your neural network. So instead of needing to write code to compute this as a function of $a^{[2](i)}$ and $y^{(i)}$ for i=1...m:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log a^{[2](i)} + (1-y^{(i)})\log(1-a^{[2](i)})\right) \tag{2}$$

you can do it in one line of code in tensorflow!

Exercise: Implement the cross entropy loss. The function you will use is:

- `tf.nn.sigmoid_cross_entropy_with_logits(logits = ...,  labels = ...)`

Your code should input `z`, compute the sigmoid (to get `a`) and then compute the cross entropy cost J. All this can be done using one call to `tf.nn.sigmoid_cross_entropy_with_logits`, which computes

$$-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\sigma(z^{[2](i)}) + (1-y^{(i)})\log(1-\sigma(z^{[2](i)}))\right) \tag{2}$$

```python
    ### START CODE HERE ###

    # Create the placeholders for "logits" (z) and "labels" (y) (approx. 2 lines)
    z = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)

    # Use the loss function (approx. 1 line)
    cost = tf.nn.sigmoid_cross_entropy_with_logits(logits = z,  labels = y)

    # Create a session (approx. 1 line). See method 1 above.
    sess = tf.Session()

    # Run the session (approx. 1 line).
    cost = sess.run(cost, feed_dict = {z: logits, y: labels})

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

    ### END CODE HERE ###

    return cost
```

In [51]:
```python
logits = np.array([0.2,0.4,0.7,0.9])

cost = cost(logits, np.array([0,0,1,1]))
print ("cost = " + str(cost))
```

```
cost = [ 0.79813886  0.91301525  0.40318605  0.34115386]
```

Expected Output :

```
cost = [ 0.79813886  0.91301525  0.40318605  0.34115386]
```

one hot encodings

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:

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

This is called a "one hot" encoding, because in the converted representation exactly one element of each column is "hot" (meaning set to 1). 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.

```python
    # 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 [59]:

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

```python
    # 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
```

[61]:

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

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

Expected Output:

ones	[1. 1. 1.]

build NN (SIGNS Dataset)

- **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.

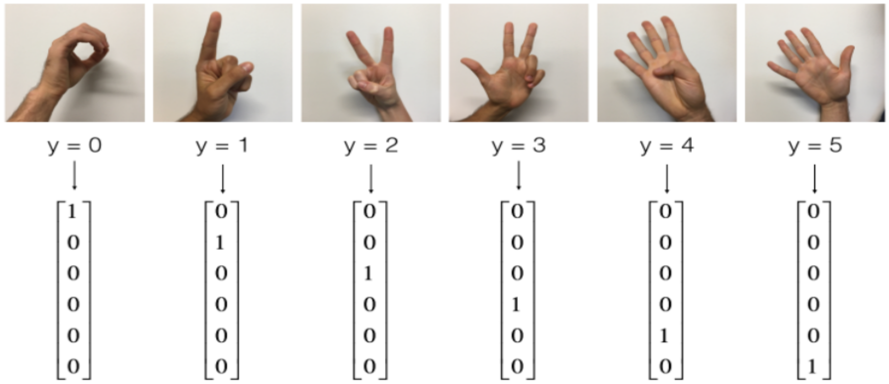

Figure 1: SIGNS dataset

Run the following code to load the dataset.

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.

Your goal is to build an algorithm 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 in numpy for cat recognition (but now using a softmax output). It is a great occasion to compare your numpy implementation to the tensorflow one.

The model is *LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SOFTMAX*. The SIGMOID output layer has been converted to a SOFTMAX. A SOFTMAX layer generalizes SIGMOID to when there are more than two classes.

2.1 - Create placeholders

Your first task is to create placeholders for `X` and `Y`. This will allow you to later pass your training data in when you run your session.

Exercise: Implement the function below to create the placeholders in tensorflow.

In [65]:
```
# 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 "tf.float32"
    Y -- placeholder for the input labels, of shape [n_y, None] and dtype "tf.float32"

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

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

    return X, Y
```

In [66]:
```
X, Y = create_placeholders(12288, 6)
print ("X = " + str(X))
print ("Y = " + str(Y))
```

```
X = Tensor("Placeholder_22:0", shape=(12288, ?), dtype=float32)
Y = Tensor("Placeholder_23:0", 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=(6, ?), dtype=float32) (not necessarily Placeholder_2)

Initialize

2.2 - Initializing the parameters

Your second task is to initialize the parameters in tensorflow.

Exercise: Implement the function below to initialize the parameters in tensorflow. You are going use Xavier Initialization for weights and Zero Initialization for biases. The shapes are given below. As an example, to help you, for W1 and b1 you could use:

```
W1 = tf.get_variable("W1", [25,12288], initializer = tf.contrib.layers.xavier_initializer(seed = 1))
b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
```

Please use `seed = 1` to make sure your results match ours.

```
    tf.set_random_seed(1)                       # so that your "random" numbers match
ours

    ### START CODE HERE ### (approx. 6 lines of code)
    W1 = tf.get_variable("W1", [25,12288], initializer =
tf.contrib.layers.xavier_initializer(seed = 1))
    b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
    W2 = tf.get_variable("W2", [12,25], initializer =
tf.contrib.layers.xavier_initializer(seed = 1))
    b2 = tf.get_variable("b2", [12,1], initializer = tf.zeros_initializer())
    W3 = tf.get_variable("W3", [6,12], initializer =
tf.contrib.layers.xavier_initializer(seed = 1))
    b3 = tf.get_variable("b3", [6,1], initializer = tf.zeros_initializer())
    ### END CODE HERE ###


    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2,
                  "W3": W3,
                  "b3": b3}
```

Forward propagation

2.3 - Forward propagation in tensorflow

You will now implement the forward propagation module in tensorflow. The function will take in a dictionary of parameters and it will complete the forward pass. The functions you will be using are:

- `tf.add(...,...)` to do an addition
- `tf.matmul(...,...)` to do a matrix multiplication
- `tf.nn.relu(...)` to apply the ReLU activation

Question: Implement the forward pass of the neural network. We commented for you the numpy equivalents so that you can compare the tensorflow implementation to numpy. It is important to note that the forward propagation stops at `z3`. The reason is that in tensorflow the last linear layer output is given as input to the function computing the loss. Therefore, you don't need `a3`!

```python
def forward_propagation(X, parameters):
    """
    Implements the forward propagation for the model: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SOFTMAX

    Arguments:
    X -- input dataset placeholder, of shape (input size, number of examples)
    parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3"
                  the shapes are given in initialize_parameters

    Returns:
    Z3 -- the output of the last LINEAR unit
    """

    # Retrieve the parameters from the dictionary "parameters"
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    W3 = parameters['W3']
    b3 = parameters['b3']

    ### START CODE HERE ### (approx. 5 lines)              # Numpy Equivalents:
    Z1 = tf.add(tf.matmul(W1, X), b1)                                    # Z1 = np.dot(W1, X) + b1
    A1 = tf.nn.relu(Z1)                                                  # A1 = relu(Z1)
    Z2 = tf.add(tf.matmul(W2, A1), b2)                                   # Z2 = np.dot(W2, A1) + b2
    A2 = tf.nn.relu(Z2)                                                  # A2 = relu(Z2)
    Z3 = tf.add(tf.matmul(W3, A2), b3)                                   # Z3 = np.dot(W3, A2) + b3
    ### END CODE HERE ###

    return Z3
```

You may have noticed that the forward propagation doesn't output any cache. You will understand why below, when we get to brackpropagation.

Cost

2.4 Compute cost

As seen before, it is very easy to compute the cost using:

```python
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = ..., labels = ...))
```

Question: Implement the cost function below.

- It is important to know that the "`logits`" and "`labels`" inputs of `tf.nn.softmax_cross_entropy_with_logits` are expected to be of shape (number of examples, num_classes). We have thus transposed Z3 and Y for you.
- Besides, `tf.reduce_mean` basically does the summation over the examples.

In [75]:
```python
# GRADED FUNCTION: compute_cost

def compute_cost(Z3, Y):
    """
    Computes the cost

    Arguments:
    Z3 -- output of forward propagation (output of the last LINEAR unit), of shape (6, number of examples)
    Y -- "true" labels vector placeholder, same shape as Z3

    Returns:
    cost - Tensor of the cost function
    """

    # to fit the tensorflow requirement for tf.nn.softmax_cross_entropy_with_logits(...,...)
    logits = tf.transpose(Z3)
    labels = tf.transpose(Y)

    ### START CODE HERE ### (1 line of code)
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = logits, labels = labels))
    ### END CODE HERE ###

    return cost
```

Backward

2.5 - Backward propagation & parameter updates

This is where you become grateful to programming frameworks. All the backpropagation and the parameters update is taken care of in 1 line of code. It is very easy to incorporate this line in the model.

After you compute the cost function. You will create an "`optimizer`" object. You have to call this object along with the cost when running the tf.session. When called, it will perform an optimization on the given cost with the chosen method and learning rate.

For instance, for gradient descent the optimizer would be:

```python
optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(cost)
```

To make the optimization you would do:

```python
_ , c = sess.run([optimizer, cost], feed_dict={X: minibatch_X, Y: minibatch_Y})
```

This computes the backpropagation by passing through the tensorflow graph in the reverse order. From cost to inputs.

Note When coding, we often use _ as a "throwaway" variable to store values that we won't need to use later. Here, _ takes on the evaluated value of `optimizer`, which we don't need (and `c` takes the value of the `cost` variable).

whole model:

```python
def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.0001,
          num_epochs = 1500, minibatch_size = 32, print_cost = True):
    """
    Implements a three-layer tensorflow neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SOFTMAX.

    Arguments:
    X_train -- training set, of shape (input size = 12288, number of training examples = 1080)
    Y_train -- test set, of shape (output size = 6, number of training examples = 1080)
    X_test -- training set, of shape (input size = 12288, number of training examples = 120)
    Y_test -- test set, of shape (output size = 6, number of test examples = 120)
    learning_rate -- learning rate of the optimization
    num_epochs -- number of epochs of the optimization loop
    minibatch_size -- size of a minibatch
    print_cost -- True to print the cost every 100 epochs

    Returns:
    parameters -- parameters learnt by the model. They can then be used to predict.
```

```
    """

    ops.reset_default_graph()                          # to be able to rerun the
model without overwriting tf variables
    tf.set_random_seed(1)                              # to keep consistent
results
    seed = 3                                           # to keep consistent
results
    (n_x, m) = X_train.shape                           # (n_x: input size, m :
number of examples in the train set)
    n_y = Y_train.shape[0]                             # n_y : output size
    costs = []                                         # To keep track of the cost

    # Create Placeholders of shape (n_x, n_y)
    ### START CODE HERE ### (1 line)
    X, Y = create_placeholders(n_x, n_y)
    ### END CODE HERE ###


    # Initialize parameters
    ### START CODE HERE ### (1 line)
    parameters = initialize_parameters()
    ### END CODE HERE ###

    # Forward propagation: Build the forward propagation in the tensorflow graph
    ### START CODE HERE ### (1 line)
    Z3 = forward_propagation(X, parameters)
    ### END CODE HERE ###

    # Cost function: Add cost function to tensorflow graph
    ### START CODE HERE ### (1 line)
    cost = compute_cost(Z3, Y)
    ### END CODE HERE ###

    # Backpropagation: Define the tensorflow optimizer. Use an AdamOptimizer.
    ### START CODE HERE ### (1 line)
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
    ### END CODE HERE ###

    # Initialize all the variables
    init = tf.global_variables_initializer()


    # Start the session to compute the tensorflow graph
    with tf.Session() as sess:

        # Run the initialization
        sess.run(init)

        # Do the training loop
        for epoch in range(num_epochs):


            epoch_cost = 0.                       # Defines a cost related to an
epoch
            num_minibatches = int(m / minibatch_size) # number of minibatches of
size minibatch_size in the train set
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_train, minibatch_size,
seed)


            for minibatch in minibatches:


                # Select a minibatch
                (minibatch_X, minibatch_Y) = minibatch

                # IMPORTANT: The line that runs the graph on a minibatch.
```

```
                # Run the session to execute the "optimizer" and the "cost", the
feedict should contain a minibatch for (X,Y).
                ### START CODE HERE ### (1 line)
                _ , minibatch_cost = sess.run([optimizer, cost], feed_dict = {X:
minibatch_X, Y: minibatch_Y})
                ### END CODE HERE ###

                epoch_cost += minibatch_cost / minibatch_size


            # Print the cost every epoch
            if print_cost == True and epoch % 100 == 0:
                print ("Cost after epoch %i: %f" % (epoch, epoch_cost))
            if print_cost == True and epoch % 5 == 0:
                costs.append(epoch_cost)

        # plot the cost
        plt.plot(np.squeeze(costs))
        plt.ylabel('cost')
        plt.xlabel('iterations (per fives)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()


        # lets save the parameters in a variable
        parameters = sess.run(parameters)
        print ("Parameters have been trained!")


        # Calculate the correct predictions
        correct_prediction = tf.equal(tf.argmax(Z3), tf.argmax(Y))


        # Calculate accuracy on the test set
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))


        print ("Train Accuracy:", accuracy.eval({X: X_train, Y: Y_train}))
        print ("Test Accuracy:", accuracy.eval({X: X_test, Y: Y_test}))

        return parameters
```

Expected Output:

Train Accuracy	0.999074
Test Accuracy	0.716667

Amazing, your algorithm can recognize a sign representing a figure between 0 and 5 with 71.7% accuracy.

Insights:

- Your model seems big enough to fit the training set well. However, given the difference between train and test accuracy, you could try to add L2 or dropout regularization to reduce overfitting.
- Think about the session as a block of code to train the model. Each time you run the session on a minibatch, it trains the parameters. In total you have run the session a large number of times (1500 epochs) until you obtained well trained parameters.

You indeed deserved a "thumbs-up" although as you can see the algorithm seems to classify it incorrectly. The reason is that the training set doesn't contain any "thumbs-up", so the model doesn't know how to deal with it! We call that a "mismatched data distribution" and it is one of the various of the next course on "Structuring Machine Learning Projects".

What you should remember:

- Tensorflow is a programming framework used in deep learning
- The two main object classes in tensorflow are Tensors and Operators.
- When you code in tensorflow you have to take the following steps:
    - Create a graph containing Tensors (Variables, Placeholders ...) and Operations (tf.matmul, tf.add, ...)
    - Create a session
    - Initialize the session
    - Run the session to execute the graph
- You can execute the graph multiple times as you've seen in model()
- The backpropagation and optimization is automatically done when running the session on the "optimizer" object.