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.

A picture containing text

Description automatically generated

**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.

**Updates**

**If you were working on the notebook before this update...**

* The current notebook is version "v3b".
* You can find your original work saved in the notebook with the previous version name (it may be either TensorFlow Tutorial version 3" or "TensorFlow Tutorial version 3a.)
* To view the file directory, click on the "Coursera" icon in the top left of this notebook.

**List of updates**

* forward\_propagation instruction now says 'A1' instead of 'a1' in the formula for Z2;  
  and are updated to say 'A2' instead of 'Z2' in the formula for Z3.
* create\_placeholders instruction refer to the data type "tf.float32" instead of float.
* in the model function, the x axis of the plot now says "iterations (per fives)" instead of iterations(per tens)
* In the linear\_function, comments remind students to create the variables in the order suggested by the starter code. The comments are updated to reflect this order.
* The test of the cost function now creates the logits without passing them through a sigmoid function (since the cost function will include the sigmoid in the built-in tensorflow function).
* In the 'model' function, the minibatch\_cost is now divided by minibatch\_size (instead of num\_minibatches).
* Updated print statements and 'expected output that are used to check functions, for easier visual comparison.

**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.



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.run(init)),*

*# the loss variable will be initialized and ready to be computed*

**with** tf.Session() **as** session: *# Create a session and print the output*

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+bY=WX+b, where WW and XX are random matrices and b is a random vector.

**Exercise**: Compute WX+bWX+b where W,XW,X, and bb 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 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 = "W")

b = tf.constant(np.random.randn(4, 1), name = "b")

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 ###*

*# close the session*

sess.close()

​

**return** result

In [7]:

print( "result = \n" **+** str(linear\_function()))

result =

[[-2.15657382]

[ 2.95891446]

[-1.08926781]

[-0.84538042]]

***Expected Output***:

result =

[[-2.15657382]

[ 2.95891446]

[-1.08926781]

[-0.84538042]]

**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 :)*

In [8]:

*# GRADED FUNCTION: sigmoid*

​

**def** sigmoid(z):

"""

Computes the sigmoid of z

Arguments:

z -- input value, scalar or vector

Returns:

results -- the sigmoid of z

"""

*### 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

In [9]:

print ("sigmoid(0) = " **+** str(sigmoid(0)))

print ("sigmoid(12) = " **+** str(sigmoid(12)))

sigmoid(0) = 0.5

sigmoid(12) = 0.999994

***Expected Output***:

|  |  |
| --- | --- |
| **sigmoid(0)** | 0.5 |
| **sigmoid(12)** | 0.999994 |

**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.

**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)a[2](i) and y(i)y(i) for i=1...m:



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 JJ. All this can be done using one call to tf.nn.sigmoid\_cross\_entropy\_with\_logits, which computes

Logo, company name

Description automatically generated

In [10]:

*# GRADED FUNCTION: cost*

​

**def** cost(logits, labels):

"""

    Computes the cost using the sigmoid cross entropy

    Arguments:

    logits -- vector containing z, output of the last linear unit (before the final sigmoid activation)

    labels -- vector of labels y (1 or 0)

Note: What we've been calling "z" and "y" in this class are respectively called "logits" and "labels"

in the TensorFlow documentation. So logits will feed into z, and labels into y.

    Returns:

    cost -- runs the session of the cost (formula (2))

"""

*### START CODE HERE ###*

*# Create the placeholders for "logits" (z) and "labels" (y) (approx. 2 lines)*

z = tf.placeholder(tf.float32, name = "z")

y = tf.placeholder(tf.float32, name = "y")

*# 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 [11]:

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]

**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:

A picture containing text, device

Description automatically generated

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 CC, and return the one hot encoding. Use tf.one\_hot() to do this.

In [12]:

*# 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 jth column

corresponds to the jth training example. So if example j had a label i. Then entry (i,j)

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 [13]:

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.]]

**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 [14]:

*# 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 [15]:

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

ones = [ 1. 1. 1.]

**Expected Output:**

|  |  |
| --- | --- |
| **ones** | [ 1. 1. 1.] |

**2 - Building your first neural network in tensorflow**

In this part of the assignment you will build a neural network using tensorflow. Remember that there are two parts to implement a tensorflow model:

* Create the computation graph
* Run the graph

Let's delve into the problem you'd like to solve!

**2.0 - 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.

Table

Description automatically generated

**Figure 1**: SIGNS dataset

Run the following code to load the dataset.

In [16]:

*# 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.

In [17]:

*# Example of a picture*

index = 0

plt.imshow(X\_train\_orig[index])

print ("y = " **+** str(np.squeeze(Y\_train\_orig[:, index])))

y = 5

A picture containing text

Description automatically generated

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 as shown in Figure 1. Run the cell below to do so.

In [18]:

*# Flatten the training and test images*

X\_train\_flatten = X\_train\_orig.reshape(X\_train\_orig.shape[0], **-**1).T

X\_test\_flatten = X\_test\_orig.reshape(X\_test\_orig.shape[0], **-**1).T

*# Normalize image vectors*

X\_train = X\_train\_flatten**/**255.

X\_test = X\_test\_flatten**/**255.

*# Convert training and test labels to one hot matrices*

Y\_train = convert\_to\_one\_hot(Y\_train\_orig, 6)

Y\_test = convert\_to\_one\_hot(Y\_test\_orig, 6)

​

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 = 1080

number of test examples = 120

X\_train shape: (12288, 1080)

Y\_train shape: (6, 1080)

X\_test shape: (12288, 120)

Y\_test shape: (6, 120)

**Note** that 12288 comes from 64×64×364×64×3. 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 [19]:

*# 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**], name="X")

Y = tf.placeholder(tf.float32, [n\_y, **None**], name="Y")

*### END CODE HERE ###*

**return** X, Y

In [20]:

X, Y = create\_placeholders(12288, 6)

print ("X = " **+** str(X))

print ("Y = " **+** str(Y))

X = Tensor("X\_1:0", 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=(6, ?), dtype=float32) (not necessarily Placeholder\_2) |

**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.

In [21]:

*# GRADED FUNCTION: initialize\_parameters*

​

**def** initialize\_parameters():

"""

Initializes parameters to build a neural network with tensorflow. The shapes are:

W1 : [25, 12288]

b1 : [25, 1]

W2 : [12, 25]

b2 : [12, 1]

W3 : [6, 12]

b3 : [6, 1]

Returns:

parameters -- a dictionary of tensors containing W1, b1, W2, b2, W3, b3

"""

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}

**return** parameters

In [22]:

tf.reset\_default\_graph()

**with** tf.Session() **as** sess:

parameters = initialize\_parameters()

print("W1 = " **+** str(parameters["W1"]))

print("b1 = " **+** str(parameters["b1"]))

print("W2 = " **+** str(parameters["W2"]))

print("b2 = " **+** str(parameters["b2"]))

W1 = <tf.Variable 'W1:0' shape=(25, 12288) dtype=float32\_ref>

b1 = <tf.Variable 'b1:0' shape=(25, 1) dtype=float32\_ref>

W2 = <tf.Variable 'W2:0' shape=(12, 25) dtype=float32\_ref>

b2 = <tf.Variable 'b2:0' shape=(12, 1) dtype=float32\_ref>

**Expected Output**:

|  |  |
| --- | --- |
| **W1** | < tf.Variable 'W1:0' shape=(25, 12288) dtype=float32\_ref > |
| **b1** | < tf.Variable 'b1:0' shape=(25, 1) dtype=float32\_ref > |
| **W2** | < tf.Variable 'W2:0' shape=(12, 25) dtype=float32\_ref > |
| **b2** | < tf.Variable 'b2:0' shape=(12, 1) dtype=float32\_ref > |

As expected, the parameters haven't been evaluated yet.

**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!

In [23]:

*# GRADED FUNCTION: forward\_propagation*

​

**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,Z2) + b3*

*### END CODE HERE ###*

**return** Z3

In [24]:

tf.reset\_default\_graph()

​

**with** tf.Session() **as** sess:

X, Y = create\_placeholders(12288, 6)

parameters = initialize\_parameters()

Z3 = forward\_propagation(X, parameters)

print("Z3 = " **+** str(Z3))

Z3 = Tensor("Add\_2:0", shape=(6, ?), dtype=float32)

**Expected Output**:

|  |  |
| --- | --- |
| **Z3** | Tensor("Add\_2:0", shape=(6, ?), dtype=float32) |

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

**2.4 Compute cost**

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

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 [25]:

*# 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

In [26]:

tf.reset\_default\_graph()

​

**with** tf.Session() **as** sess:

X, Y = create\_placeholders(12288, 6)

parameters = initialize\_parameters()

Z3 = forward\_propagation(X, parameters)

cost = compute\_cost(Z3, Y)

print("cost = " **+** str(cost))

cost = Tensor("Mean:0", shape=(), dtype=float32)

**Expected Output**:

|  |  |
| --- | --- |
| **cost** | Tensor("Mean:0", shape=(), dtype=float32) |

**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:

optimizer = tf.train.GradientDescentOptimizer(learning\_rate = learning\_rate).minimize(cost)

To make the optimization you would do:

\_ , 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).

**2.6 - Building the model**

Now, you will bring it all together!

**Exercise:** Implement the model. You will be calling the functions you had previously implemented.

In [31]:

**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 = 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

Run the following cell to train your model! On our machine it takes about 5 minutes. Your "Cost after epoch 100" should be 1.048222. If it's not, don't waste time; interrupt the training by clicking on the square (⬛) in the upper bar of the notebook, and try to correct your code. If it is the correct cost, take a break and come back in 5 minutes!

In [32]:

parameters = model(X\_train, Y\_train, X\_test, Y\_test)

Cost after epoch 0: 1.913693

Cost after epoch 100: 1.048222

Cost after epoch 200: 0.756012

Cost after epoch 300: 0.590844

Cost after epoch 400: 0.483423

Cost after epoch 500: 0.392928

Cost after epoch 600: 0.323629

Cost after epoch 700: 0.262100

Cost after epoch 800: 0.210199

Cost after epoch 900: 0.171622

Cost after epoch 1000: 0.145907

Cost after epoch 1100: 0.110942

Cost after epoch 1200: 0.088966

Cost after epoch 1300: 0.061226

Cost after epoch 1400: 0.053860

A picture containing shape

Description automatically generated

Parameters have been trained!

Train Accuracy: 0.999074

Test Accuracy: 0.716667

**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.

**2.7 - Test with your own image (optional / ungraded exercise)**

Congratulations on finishing this assignment. You can now take a picture of your hand and see the output of your model. To do that:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.

2. Add your image to this Jupyter Notebook's directory, in the "images" folder

3. Write your image's name in the following code

4. Run the code and check if the algorithm is right!

In [ ]:

**import** scipy

**from** PIL **import** Image

**from** scipy **import** ndimage

​

*## START CODE HERE ## (PUT YOUR IMAGE NAME)*

*#my\_image = "fingers04.png" # Wrong*

*#my\_image = "fingers05.jpg" # OK*

my\_image = "fingers06.jpg" *# Wrong*

*#my\_image = "fingers07.jpg" # Wrong*

*## END CODE HERE ##*

​

*# We preprocess your image to fit your algorithm.*

fname = "images/" **+** my\_image

image = np.array(ndimage.imread(fname, flatten=**False**))

image = image**/**255.

my\_image = scipy.misc.imresize(image, size=(64,64)).reshape((1, 64**\***64**\***3)).T

my\_image\_prediction = predict(my\_image, parameters)

​

plt.imshow(image)

print("Your algorithm predicts: y = " **+** str(np.squeeze(my\_image\_prediction)))

A picture containing chart

Description automatically generated

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.