## Tensor flow basics:

tf.constant() ==> will generate a tensor with type of constant.

If we want to know what is the type of a variables we need to say:

type(<variable\_name>)

for any operation in tf we need to warp the code inside run:

with tf.Session() as sess:

sess.run()

- we can make constants like this: a = tf.constant(10)

- and we say b = tf.constant(20)

when we say a+b ==> it doesn’t work if its no inside run().

This works:

with tf.Session() as sess:

result = sess.run(a+b)

result

-tf.fill() can make a matrix and fill it with a number.

fill\_mat = tf.fill((4,4),10) ==> makes a 4x4 matrix and fills it with 10.

- tf.ones((4,4)) ==> makes a matrix and fills with ones.

- tf.zeros() ==> similar

tf.random\_normal() => fills the matrix with random normal values.

There are many different random variables we can make.

-sess = tf.InteractiveSession()

makes an interactive session that is used only with notebooks.

We can use it in jupyter instead of always saying with tf.sessions() as sess:

Then we can loop through all ops in my\_op:

and say:

for op in my\_ops:

print(sess.run(op))

the will print all the ops.

- we can use tf.matmul() to do matrix multiplication.

## Tensorflow graphs:

Graphs are sets of connected nodes. The connections are referred as edges. Each node is like a processor that gets an input and returns an output.

n1: constant



n2: constant

n3: Add

1

2

3

Now we want to make this graph in tf.

-tf session has a default graph: print(tf.get\_default\_graph())

- in order to make a new graph we simply say : tf.Graph()

- and we can change the default graph to the graph we made instead of the tf default one:

graph\_one = tf.get\_default\_graph()

graph\_two = tf.Graph()

if we want to change the default to graph\_two:

with graph\_two.as\_default():

print(graph\_two is tf.get\_default\_graph())

in tf we have 2 different data types:

**1) variables:** since tf tunes parameters to improve the model, variables can hold values of weights and biases throughout the session.

But they need to be initialized.

**2) placeholders:** they are initially empty and are used to feed in the actual training example.

We need to declare their datatype with an optional shape argument.

e.g. tf.float32

- we can make variables ==> tf.Variable()

- after that we need to initialize all variables like this:

init = tf.global\_variables\_initializer()

sess.run(init)

then we can run sess and it show variables. sess.run(<var\_name>)

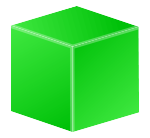
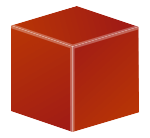
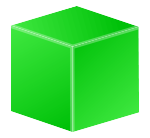
- and here is how we make placeholders:

ph = tf.placeholder(tf.float32, shape=(4,4))

## building the first Neural Network:

we are going to make this: Wx + b = z

the following shape shows that:

****

Activation

Function

B: Variable

Operation:

tf.add()

Operation:

tf.matmul()

X: placeholder

W: Variable

- when we make this graph and add variables and placeholders, we should add the cost function in order for the network to be trained and optimize the parameters.

- since placeholders don’t have data and need data, when we want to run session we have to have a dictionary for feeding data to them.

We need to define each variables: x, w, b

W, b are variables we can make tf.Variable() and inside it we can pass tf.ramdom\_normal() to pass normal distributed numbers and we should pass the shape of it. Again, the rows depend on the sample size and the columns or features have to be defined. ( in this example we said n\_features=10)

n\_dense\_neurons means we need to have a layer of 3 neurons. I think it means the rows of the model because we pass it as the second parameter of the tensorflow shape.

- then we define z as the result of Wx+b

- after it we need to use an activation function like tanh, nn.relu, sigmoid.

In this example we use sigmoid which is the most basic one and pass z.

- then we need to run the session and see the results. As we used sigmoid function the results should be between 0-1

>>> ok this was an example for learning the syntax. Now we take a look at a simple regression case using tf.

- first we define x\_data which is a np.linspace() function to have 10 random data between 0-10 np.linspace(0,10,10)

also we add some noise to it by np.random.uniform() between -1.5, 1.5 to make it messy.

- we do the same thing for y\_label

and since we pick from the same distance there would be a kind of linear regression relationship between x,y

- we define our linear model as m\*x + b

m,b will be variables.

And we use zip() function to make a list of tuples for x,y values and then we will have to have an error variable. Our goal is to minimize the error. We can do it by squaring the y-y^ which is the distance between the real y and predicted y. and we punish higher errors more.

- then we define optimizer:

**optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.001)**

**train = optimizer.minimize(error)**

and the learning rate can be anything depending on how fast or slow we want the learning.

- then we run the session and training steps can be anything. First we made it with only one training. And now we can add 100

and we see it has a better fit.

**6-31 : more realistic regression:**

Now its time to make a more realistic regression model using tf.

We make 2 different datasets for x,y again we have the formula of y = mx + b and we put m=0.5 , b=5 then we use numpy to gernerate 1 million values and apply the formula to calculate y values and then we use pandas to concatenate them in one dataset.

- when we have big datasets, in order to plot them its better to use a sample of all data because if we try to plot the whole data set Jupyter may crash.

We can say: df.sample(n=250) ==> this returns 250 samples of the whole dataset.

- after plotting the data, we want tf to guess the values of m,b and make the regression line for us.

- when we want to feed big data to the model for training, we need to do it in batches not all in once.

- now we define random numbers for m,b again to initialize the model. (variables)

- also for x,y we need to make placeholders.

- we pass the number of batch size that we want inside the tf.placeholder() => tf.placeholder(tf.float32, <batch\_size>)