Tensorflow:

<http://web.stanford.edu/class/cs20si/syllabus.html>

* Tf provides libraries to build architecture in less lines of code
* “Open source software library for numerical computation using data flow graphs”
* Flexibility + Scalability
* Originally developed by Google as a single infrastructure for machine learning in both production and research
* Companies using: google, open AI, nvidia, Airbnb, airbus
* Projects: Classify skin cancer, Wavenet(text to speech), neural style translation
* Import tensorflow as tf
* Data flow graph: tf seperates definition of computation from their execution
* Tf is a computational graph approach, Any tf program has 2 phases
  + Phase1: Assemble a graph

Phase2:use a session to execute operations in a graph

* Tensor: N dimensional array
* a= tf.ad(5,4)

sess= tf.Session()

sess.run(a)

sess.close()—to get value of a create a session, assign var to sess. Within sess evaluate graph to fetch val of a

* with tf.Session() as sess:

sess.run(a)

* tf.Session():A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated. Session will allocate memory to store current values of variable
* tf.Session.run(fetches,feed\_dict= None, options=None,run\_metadata=None)
* fetches is a list of tensor whose value we want
* Subgraphs: Possible to break graphs into several chunks and run them parallelly across multiple CPUs, GPUs, TPUs, or other devices. Eg: Alexnet
* To put part of a graph on a specific cpu or gpu : with tf.device(‘/gpu:1’)
* Create a session with tf.sess(config= tf.configproto(Log\_device\_placement = true)
* g = tf.graph() to create a graph
* With g.as\_default()
* Tf.get\_default\_graph() to use default graph
* Why graphs:?
  + Save computation. Only run subgraphs that lead to the values you want to fetch.
  + Break computation into small, differential pieces to facilitate auto-differentiation
  + Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices
  + Many common machine learning models are taught and visualized as directed graphs
* In summary, we chose TensorFlow because:
  + Python API
  + Portability: deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API
  + Flexibility: from Raspberry Pi, Android, Windows, iOS, Linux to server farms
  + Visualization (TensorBoard is da bomb)
  + Save and restore models, graphs
  + Auto-differentiation autodiff (no more taking derivatives by hand. Yay)
  + Large community (~300k commits, ~85k repositories)
  + Awesome projects already using TensorFlow
* High level APIs on top of TensorFlow: There are many high level APIs built on top of TensorFlow. Some of the most popular APIs included [Keras](https://keras.io), [TFLearn](http://tflearn.org/), and [Sonnet](https://deepmind.github.io/sonnet/). These high-level APIs allow for faster experimentation -- you can call a complex neural network models in a few lines of code.

Tenorflow ops:

import os  
  
os.environ['TF\_CPP\_MIN\_LOG\_LEVEL']='2'

* Use this to avoid warning: Tensorflow library was no complied to used XXXXX instructions
* Visualize with tensorboard: create summary writer after graph defn and before running session
* writer = tf.summary.FileWriter(‘./location\_filename’, tf.get\_default\_graph())

with tf.Session() as sess:

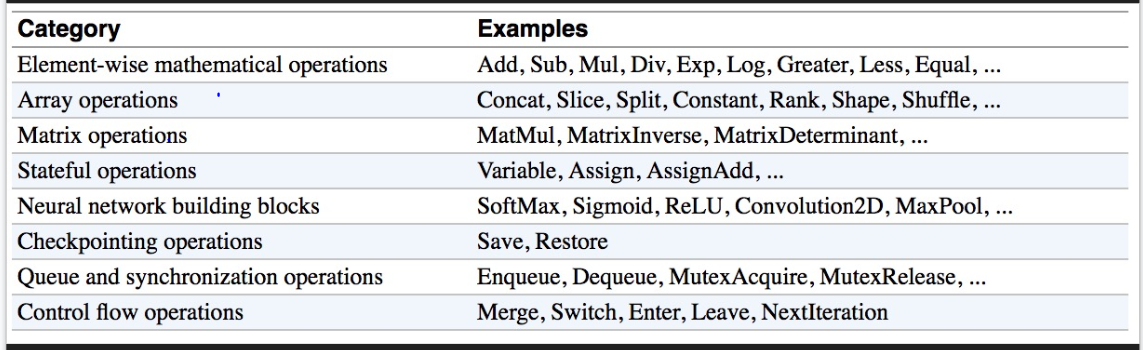
# writer = tf.summary.FileWriter(‘./location\_filename’, sess.graph)

Sess.run(x)

Writer.close() # close writer after done using

* tensorboard --logdir=”./location\_graph” --port 6006

open browser and go <http://localhost:6006/>

* CONSTANTS: a = tf.constant(value,dtype=None,shape=None,name=’const’,verify\_shape= False)
* Tf.zeros(shape, dtype=tf.float32,name= None)
* Tf.zeros\_like(input-tensor,dtype= None, name= None, optimizer=True)-----creates a tensor of shape an dtype as input-tensor but all elements are zero
* Input-tensor=[[0,1][1,1]]
* Tf.zeros\_like(input-tensor) 🡺 [[0,0][0,0]]
* Tf.ones()
* Tf.ones\_like
* Tf.fill(dimensions,value,name=None)
* Tf.fill([2,2],3)🡺 [[3,3],[3,3]]
* Tf.lin\_space(start,stop,num,n9ame=None)
* Tf.range(start,stop,difference)
* 
* Datatypes:
* Constants: are stored in graph definition. This makes loading graphs expensive when constants are big.
* Use variables for data that require more memory
* VARIABLE: tf.Variable(value,name=None)
* Tf.get\_variable(name,shape,initializer) …we can use initializer= tf.constant(2)
* Tf.constant is op, tf.Variable is a clas with many ops
* Initialize variables:
  + We need to **initialize all variables** at once, so use “sess.run(tf.global\_variable\_initializer())” 🡺 initializer is an op. We need to execute it within the context of a session
  + Initialize only **subset** of variables: sess.run(tf.variables\_initializer([a,b])
  + Only one variable: a= tf.Variable(tf.zeros([2,3]))

With tf.Session() as sess:

Sess.run(a.initializer)

Print(a.eval()) #display values

* Tf.Variable.assign():
  + A.assign(100) needs to be executed in a session to take effect
  + Used to assign value to variable
  + Assign\_add(value) to add a value to the variable
  + Assign\_sub(value)
  + Var= tf.variable(10)

With tf.session() as sess:

Sess.run(var.initializer) >>10

Sess.run(var.assign\_add(10))>>20

Sess.run(var.assign\_sub(2))>>18

* Each session maintains its own copy of variables: like if 2 session is created the modification of variable in one session wont affect the value of other.
* Control dependencies: defines which op should run first
  + g= tf.get\_default\_graph() #graph g has 4 ops:a,b,c,d
  + with g.control\_dependencies([a,b]):
  + #c,d will run only after a,b execution
* PLACEHOLDERS
  + A TF program often has 2 phases:
    - Assemble a graph
    - Use a session to execute operations in the graph.
  + Assemble the graph first without knowing the values needed for computation
  + Analogy:
    - Define the function f(x, y) = 2 \* x + y without knowing value of x or y.
    - x, y are placeholders for the actual values.
* Why placeholders: We, or our clients, can later supply their own data when they need to execute the computation.
* Tf.placeholder(dtype,shape=None, name=None)
* Supplement the values to placeholders using a dictionary
* If a=b+c ,,then sess.run(a, feed\_dict ={b:[2,3],c:[4,5]})
* Shape = None means that tensor of any shape will be accepted as value for placeholder.
* shape=None is easy to construct graphs, but nightmarish for debugging
* You can feed\_dict any feedable tensor.
* Placeholder is just a way to indicate that something must be fed
* Tf.graph.is\_feedable(tensor) #true if and only if tensor is feedable
* Tf.data: Instead of doing inference with placeholders and feeding in data later, do inference directly with data :
  + Tf.data.dataset
* dataset = tf.data.Dataset.from\_tensor\_slices((data[:,0],[:,1]))
  + Tf.data.iterator
* iterator = dataset.make\_one\_shot\_iterator()  
  Iterates through the dataset exactly once. No need to initialization.
* iterator = dataset.make\_initializable\_iterator()  
  Iterates through the dataset as many times as we want. Need to initialize with each epoch.
* ------------------------------------------------------------------------------------------------------------------------

EAGER EXECUTION: "A NumPy-like library for numerical computation with support for GPU acceleration and automatic differentiation, and a flexible platform for machine learning research and experimentation."……..It's available as tf.contrib.eager, starting with version 1.50 of TensorFlow.

* Graphs are: **Optimizable**
  + automatic buffer reuse
  + constant folding
  + Inter-op parallelism
  + automatic trade-off between compute and memory
* **Deployable**
  + the Graph is an intermediate representation for models
* **Rewritable**
  + experiment with automatic device placement or quantization

TensorFlow = Operation Kernels + Execution

* Graph construction: Execute compositions of operations with Sessions
* Eager execution: Execute compositions with Python
* Motivation:
  + TensorFlow today: Construct a graph and execute it.
    - This is *declarative* programming. Its benefits include performance and easy translation to other platforms; drawbacks include that declarative programming is non-Pythonic and difficult to debug.
  + What if you could execute operations directly?
    - Eager execution offers just that: it is an *imperative* front-end to TensorFlow.
* Key advantages: Eager execution …
  + is compatible with Python debugging tools
    - pdb.set\_trace() to your heart's content!
  + provides immediate error reporting
  + permits use of Python data structures
    - e.g., for structured input
  + enables you to use and differentiate through Python control flow
* Enabling eager execution requires two lines of code

import tensorflow as tf

import tensorflow.contrib.eager as tfe

tfe.enable\_eager\_execution() # Call this at program start-up

    and lets you write code that you can easily execute in a REPL, like this

x = [[2.]]  # No need for placeholders!

m = tf.matmul(x, x)

print(m)  # No sessions!

# tf.Tensor([[4.]], shape=(1, 1), dtype=float32)

--------------------------------------------------------------------------------------------------------------------------------------

MANAGING.

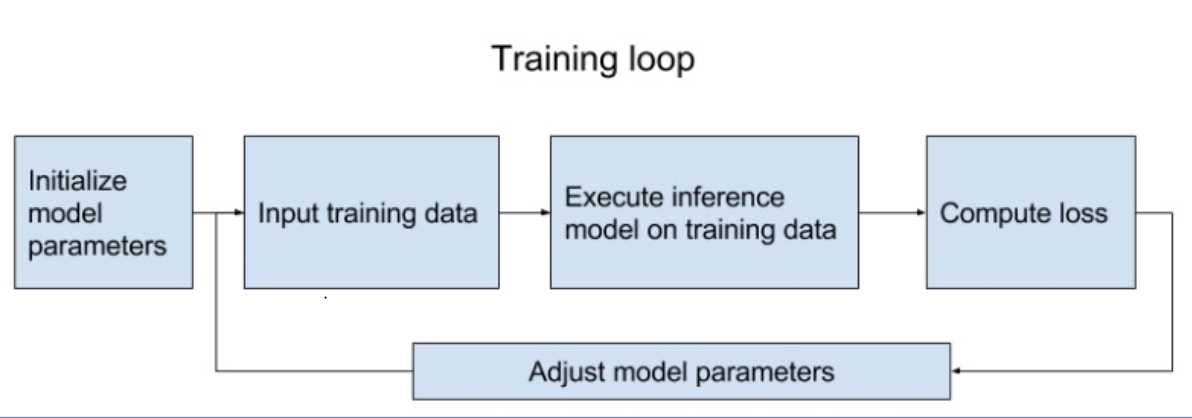
* One hot representation :
  + each word is represented by one vector with single 1 and rest is 0
  + A =[0 0 1], B=[0 1 0],C =[1 0 0]
  + Problems: Vocabulary can be large

=>  massive dimension, inefficient computation

* + Can’t represent relationship between words

=> “anxious” and “nervous” are similar but would have completely different representations

* Word embeddings:
  + Distributed representation
  + Continuous values
  + Low dimension
  + Capture the semantic relationships between words
* Representing a word by means of its neighbours. Know word by company it keeps
* PHASE1: Assemble graph
  + 1.Import data (with tf.data or placeholders)
  + 2. Define the weights
  + 3. Define the inference model
  + 4. Define loss function
  + 5. Define optimizer
* PHASE2: Compute



* NAMESCOPE: tf doesn’t know which nodes to group together unless we tell it
* Group nodes together with tf.name\_scope(name)

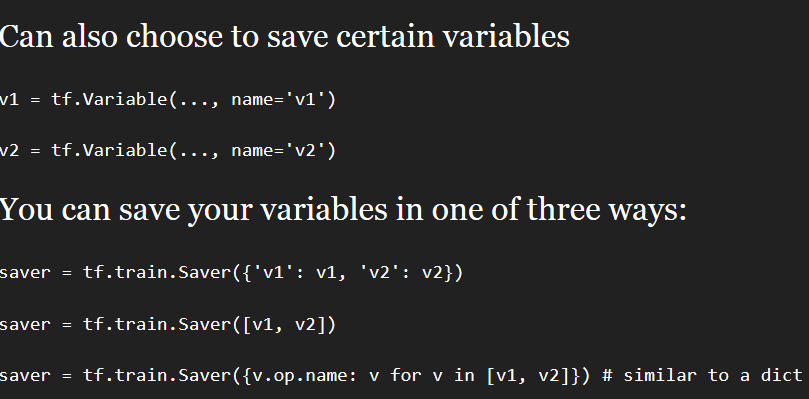
with tf.name\_scope(name\_of\_that\_scope):

# declare op\_1

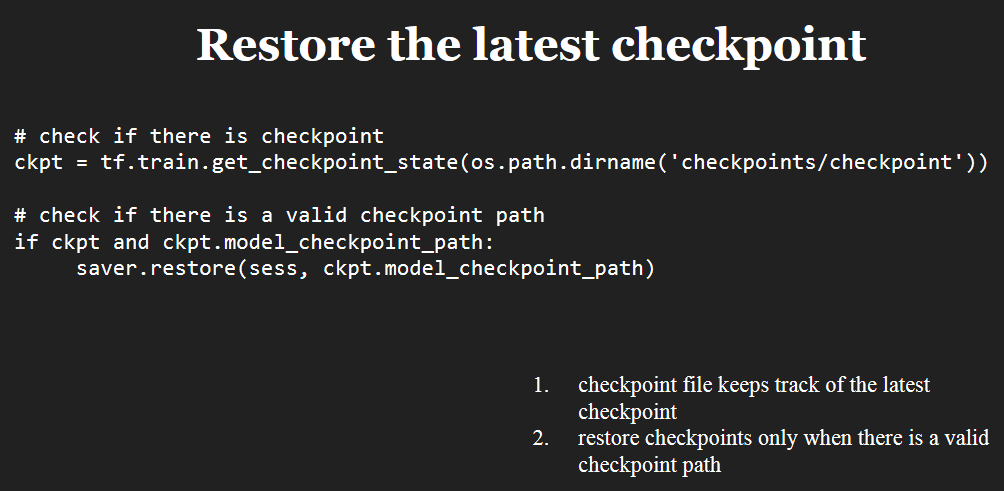
# declare op\_2

* Difference btwn namescope and variablescope: vs facilitates variable sharing
* tf.get\_variable(<name>, <shape>, <initializer>)
  + If a variable with <name> already exists, reuse it
  + If not, initialize it with <shape> using <initializer>
* Put your variables within a scope and reuse all variables within that scope

**Tf.train.Saver**

* To save sessions[**checkpoints**]: (suppose if you want to save parameters after some certain number of steps)—global\_step is step at which model is saved
  + tf.train.Saver.save(sess, save\_path, global\_step=None...)
  + Only save variables, not graph
  + Checkpoints map variable names to tensors
  + 

**tf.train.Saver.restore(sess, save\_path)**

* + saver.restore(sess, 'checkpoints/name\_of\_the\_checkpoint')
  + still need to first build the graph
* 

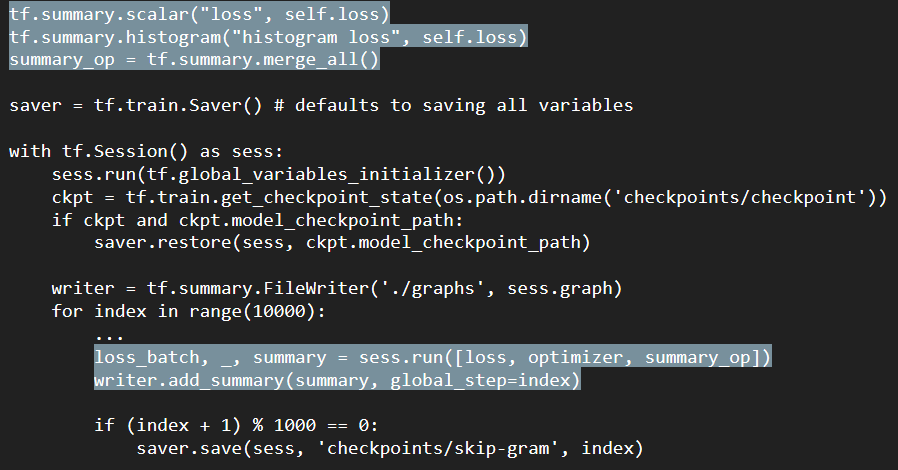
**Tf.Summary:**

* Visualize our summary statistics during our training

tf.summary.scalar

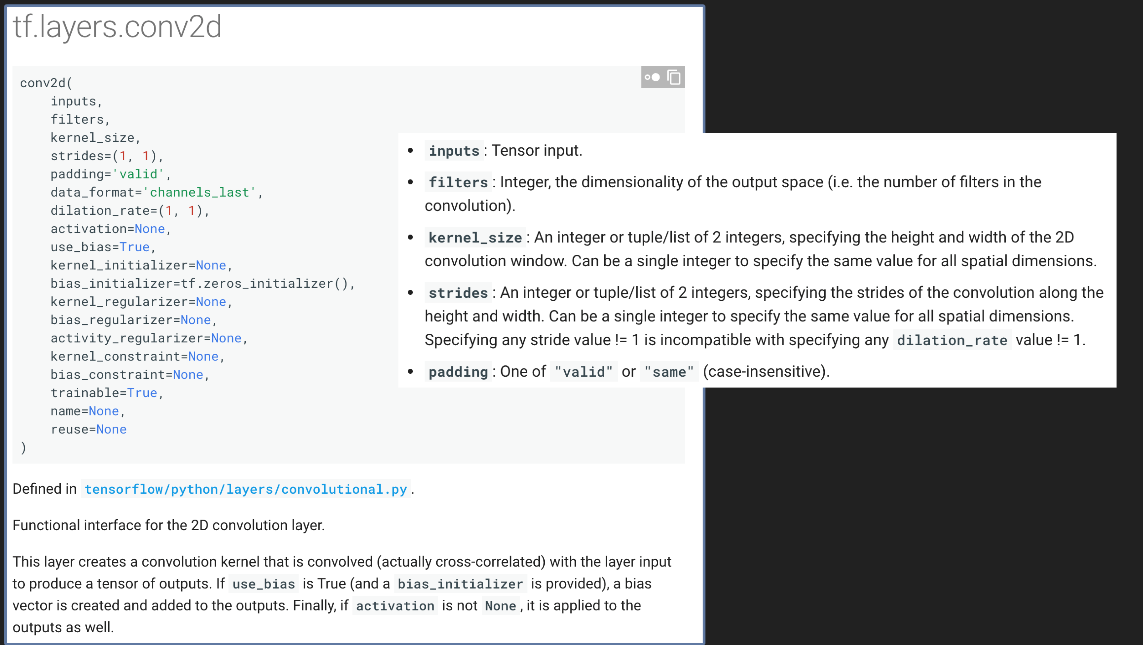
tf.summary.histogram

tf.summary.image

* Steps: create summary, run them, write summaries to file
* 
* tf.gradients(y, [xs]): Take derivative of y with respect to each tensor in the list [xs]
* ----------------------------------------------------------------------------------------------------------------------------

* **CNN**
* Features are hierarchical
  + **Composing high-complexity features out of low-complexity features** is more efficient than learning high-complexity features directly.

e.g.: having an “circle” detector is useful for detecting faces… and basketballs

* + Features are **translationally invariant:** If a feature is useful to compute at (x, y) it is useful to compute that feature at (x’, y’) as well
* 
* Padding: Valid(no padding), zero (with 0 outer)
* Convolutional networks are tailor-made for computer vision tasks.They exploit:
  + Hierarchical nature of features
  + Translation invariance of features
* We can use one single convolutional layer to modify a certain image

tf.nn.conv2d(

   input,            Batch size (N) x Height (H) x Width (W) x Channels (C)

   filter,        Height x Width x Input Channels x Output Channels

   strides,        4 element 1-D tensor, strides in each direction

   padding,        'SAME' or 'VALID'

   use\_cudnn\_on\_gpu=True,

   data\_format='NHWC',

   dilations=[1, 1, 1, 1],

   name=None

)

* **Callback** can be used to control training. Well, the good news is that, the training loop does support callbacks. So in every epoch, you can callback to a code function, having checked the metrics. If they're what you want to say, then you can cancel the training at that point.
* Can be used to stop training when reached certain accuracy.
* class myCallback(tf.keras.callbacks.Callback):

def on\_epoch\_end(self, epoch, logs={}):

if(logs.get('acc')>0.6):

print("\nReached 60% accuracy so cancelling training!")

self.model.stop\_training = True

* callbacks should be passed to run in tf/fit in keras