UC Berkeley Extension

Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Machine Learning With TensorFlow®**
- Introduction, Python - pros and cons
- Python modules, DL packages and scientific blocks
- Working with the shell, IPyton and the editor
- Installing the environment with core packages
- Writing “Hello World” HW1 (10pts)

- **Tensorflow and TensorBoard basics (cont.)**
- Ecosystem, Competition, Users
- Linear algebra recap
- Data types in Numpy and Tensorflow
- Basic operations in Tensorflow
- Graph models and structures with Tensorboard

- **TensorFlow operations**
- Sessions, graphs, variables, placeholders
- Overloaded operators
- Using Aliases

- **Data Mining and Machine Learning concepts**
- TF 2.0 vs. 1.X comparison
- Name scopes
- Basic Deep Learning Models, k-Means
- Linear and Logistic Regression
- Softmax classification HW2 (10pts)

- **Neural Networks 1/2**
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation

TensorFlow

- Example:

```python
## Detecting Spikes:
import tensorflow as tf
sess = tf.InteractiveSession()

# Create a boolean variable called `spike` to detect a sudden increase in
# a series of numbers. Since all variables must be initialized, initialize the
# variable by calling `run()` on its `initializer`:
vector = [3., -2., 8., -4., 0.2, 2.3, 7.5, 14.8]
spike = tf.Variable(False)

# Initiallizing our variable in one of two ways (choose one):
# sess.run(spike.initializer)
spike.initializer.run()

# Loop through the data and update the spike variable when there is a
# significant increase:
for i in range(1, len(vector)):
    if vector[i] - vector[i-1] > 5:
        updater = tf.assign(spike, tf.constant(True))
        updater.eval()
    else:
        tf.assign(spike, False).eval()
    print("Spike", spike.eval())

# Check to see if there some uninitialized variables:
print(sess.run(tf.report_uninitialized_variables()))

sess.close()
```

Python

```
('Spike', False)
('Spike', True)
('Spike', False)
('Spike', False)
('Spike', False)
('Spike', True)
('Spike', True)
```

Shows the difference between

… try it in class

TensorFlow

- Example:

```python
## Saving Variables in TensorFlow
import tensorflow as tf
sess = tf.InteractiveSession()

# Create a boolean vector called `spike` to locate a sudden spike in data.
# Since all variables must be initialized, initialize the variable by calling
# `run()` on its `initializer`.
vector = [3., -2., 8., -4., 0.2, 2.3, 7.5, 14.8]
spikes = tf.Variable([False] * len(vector), name='spikes')

# Initiallizing our variable in one of two ways (choose one):
# sess.run(spikes.initializer)
spikes.initializer.run()

# The saver op will enable saving and restoring
saver = tf.train.Saver()

# Loop through the data and update the spike variable when there is a significant increase

for i in range(1, len(vector)):
    if vector[i] - vector[i-1] > 5:
        spikes_val = spikes.eval()
        spikes_val[i] = True
        updater = tf.assign(spikes, spikes_val)
        updater.eval()

save_path = saver.save(sess, "./spikes.ckpt")
print("spikes data saved in file: %s" % save_path)

sess.close()
```

```
spikes data saved in file: ./spikes.ckpt
```

```
checkpoint
spikes.ckpt.data-00000-of-00001
spikes.ckpt.index
spikes.ckpt.meta
```

... try it in class

TensorFlow

- Example:

```python
## Loading Variables in TensorFlow
import tensorflow as tf
sess = tf.InteractiveSession()

# Create a boolean vector called `spike` to locate a sudden spike in data.
# Since all variables must be initialized, initialize the variable by calling
# `run()` on its `initializer`.
spikes = tf.Variable([False]*8, name='spikes')
saver = tf.train.Saver()

saver.restore(sess, "./spikes.ckpt")
print(spikes.eval())

sess.close()
```

```
INFO:tensorflow:Restoring parameters from ./spikes.ckpt
[False False  True False False False  True  True]
```


... try it in class

TensorFlow

- Example:

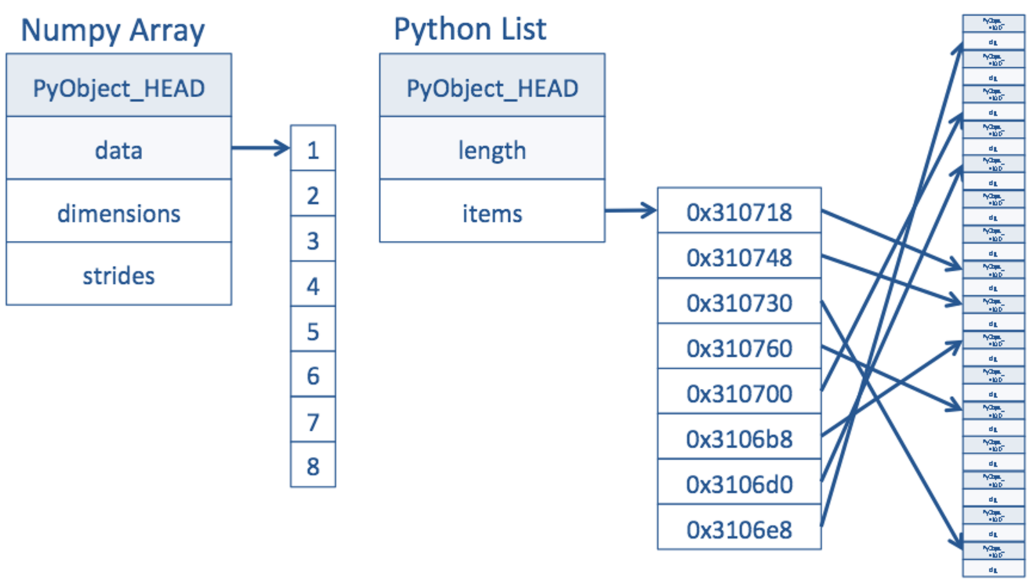
```python
## Log example:
import tensorflow as tf

# Let's create a simple matrix:
matrix = tf.constant([[3., 4.]])

# Now negate it:
negMatrix = tf.negative(matrix)

# Let's see where each operation is mapped to:
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    result = sess.run(negMatrix)

# Print results on screen:
print(result)
print(matrix.shape)

# To access each member inside a tensor do:
print(matrix.shape[0])
```

```
2018-05-28 14:51:29.346491: I tensorflow/core/common_runtime/direct_session.cc:297] Device mapping:

2018-05-28 14:51:29.347240: I tensorflow/core/common_runtime/placer.cc:874] Neg: (Neg)/job:localhost/
replica:0/task:0/device:CPU:0
[[-3. -4.]]2018-05-28 14:51:29.347253: I tensorflow/core/common_runtime/placer.cc:874] Const: (Const)
/job:localhost/replica:0/task:0/device:CPU:0

(1, 2)
1
```


... try it in class

NumPy arrays recap

- Difference between NumPy arrays vs Python Lists
  - NumPy array:
    - A NumPy array is a Python object build around a C array
    - This means that it has a pointer to a contiguous data buffer of values
  - Python Lists:
    - A Python list has a pointer to a contiguous buffer of pointers
    - All of them point to different Python objects, which in turn has references to its data (in this case, integers)
  - Conclusion:
    - NumPy is much more efficient than Python, in the cost of storage and in speed of access



TensorFlow

let's recall

- Let's discuss the basics of computation graphs without the context of TensorFlow

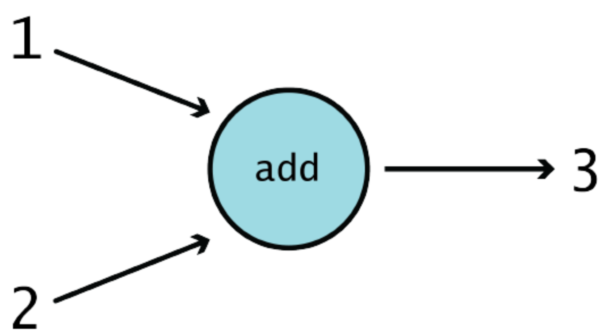
- This includes:

  - defining nodes
  - defining edges
  - dependencies
  - examples to illustrate key principles

TensorFlow

let's recall

- Graph basics:
  - At the core of every TensorFlow program is the computation graph
  - It is a is a specific type of directed graph that is used for defining computational structure
  - In TensorFlow it is, a series of functions chained together, each passing its output to zero, one, or more functions further along in the chain

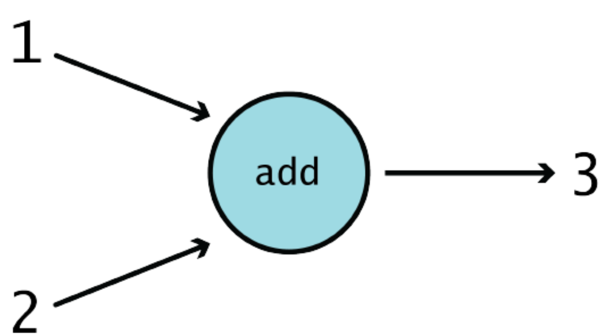


$$f(1,2) = 1 + 2 = 3$$

TensorFlow

let's recall

- Graph basics:
  - Nodes: typically drawn as circles, ovals, or boxes, represent some sort of computation or action being done on or with data in the graph's context. In the example below, the operation "add" is the sole node.

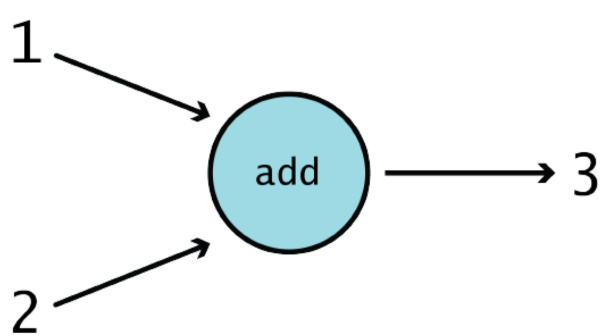


$$f(1,2) = 1 + 2 = 3$$

TensorFlow

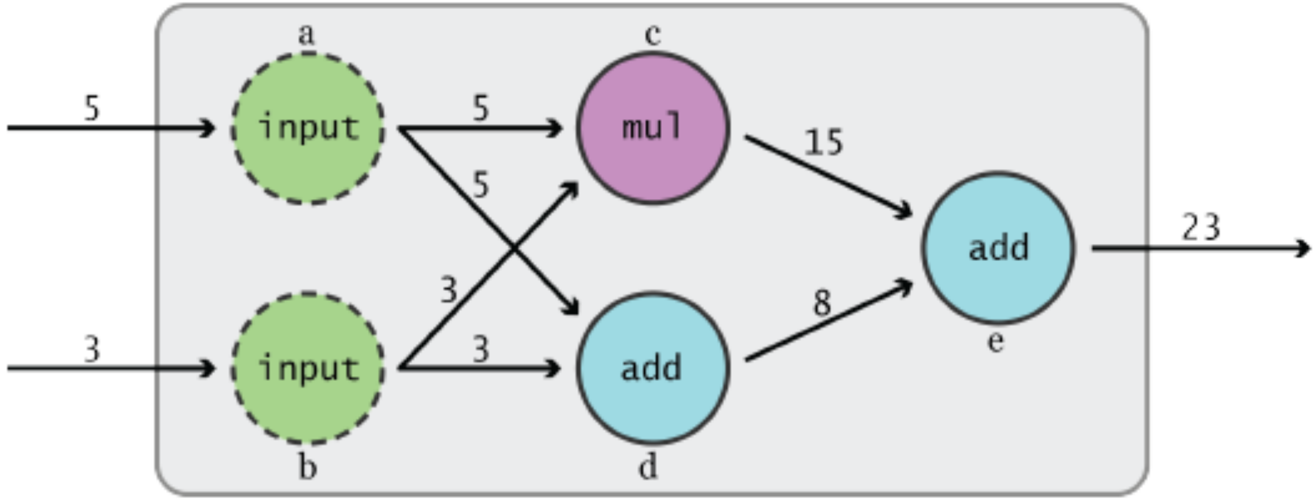
let's recall

- Graph basics:
  - Edges: are the actual values that get passed to and from *Operations*, and are typically drawn as arrows
  - In the "add" example, the inputs 1 and 2 are both edges leading into the node, while the output 3 is an edge leading out of the node
  - We can think of edges as the link between different *Operations* as they carry information from one node to the next



$$f(1,2) = 1 + 2 = 3$$

TensorFlow



TensorFlow

We can decompose this graphical representation as a series of equations like this:

$a = input_1;\ b = input_2$

$c = a \cdot b;\ d = a + b$

$e = c + d$

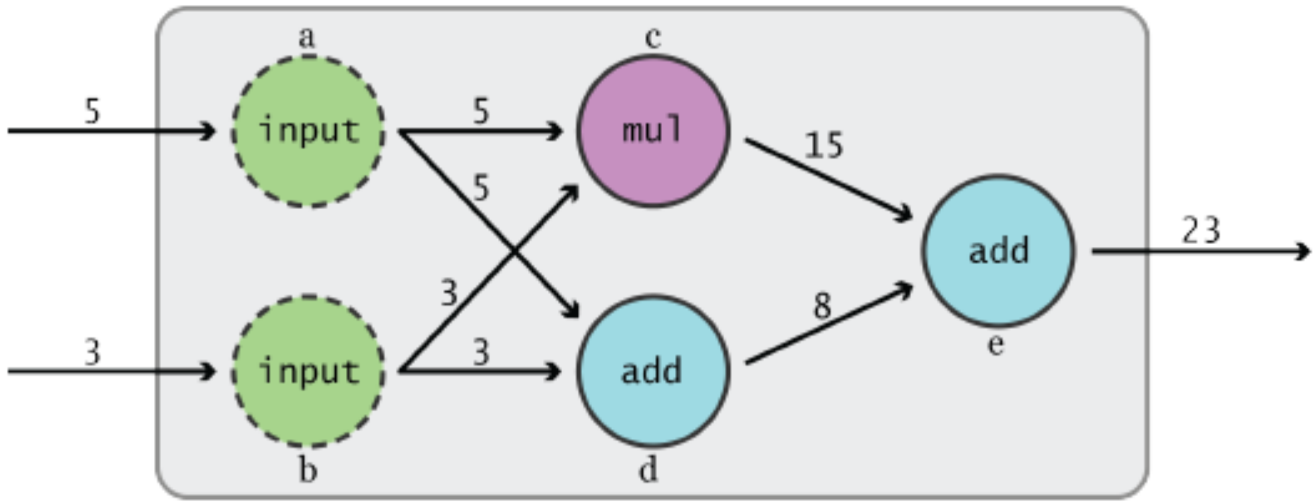
to solve e for $a = 5$ and $b = 3$,

$a = 5;\ b = 3$

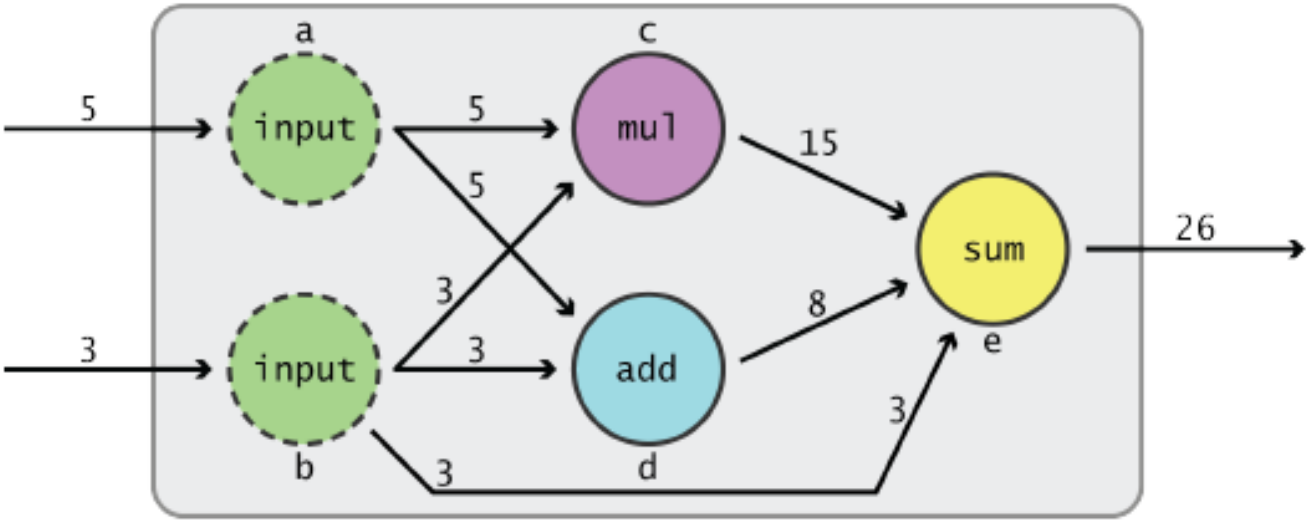
$e = (a \cdot b) + (a + b)$

$e = (5 \cdot 3) + (5 + 3)$

$e = 15 + 8 = 23$

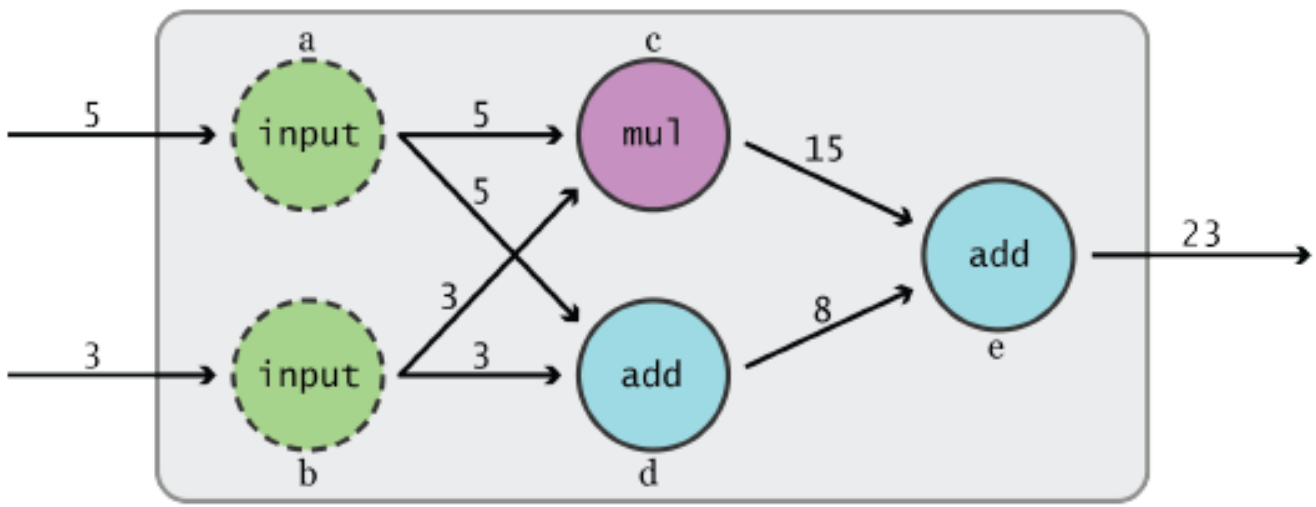


TensorFlow



TensorFlow

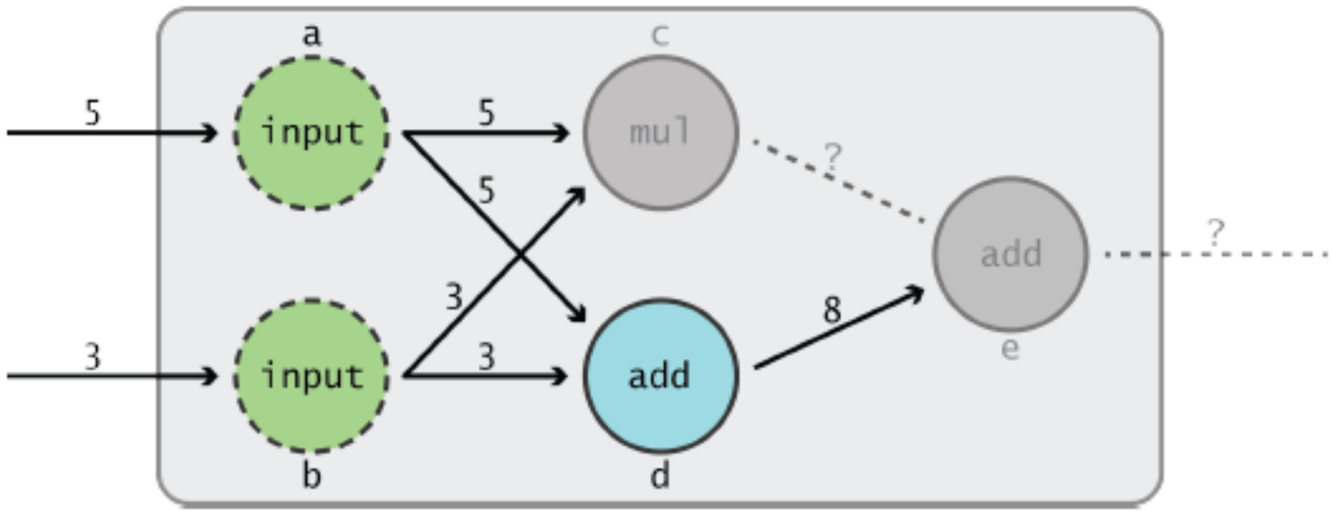
- Graph basics:
  - Dependencies: there are certain types of connections between nodes that aren't allowed, the most common of which is one that creates an unresolved *circular dependency*



TensorFlow

- Graph basics:
    - Dependencies: …

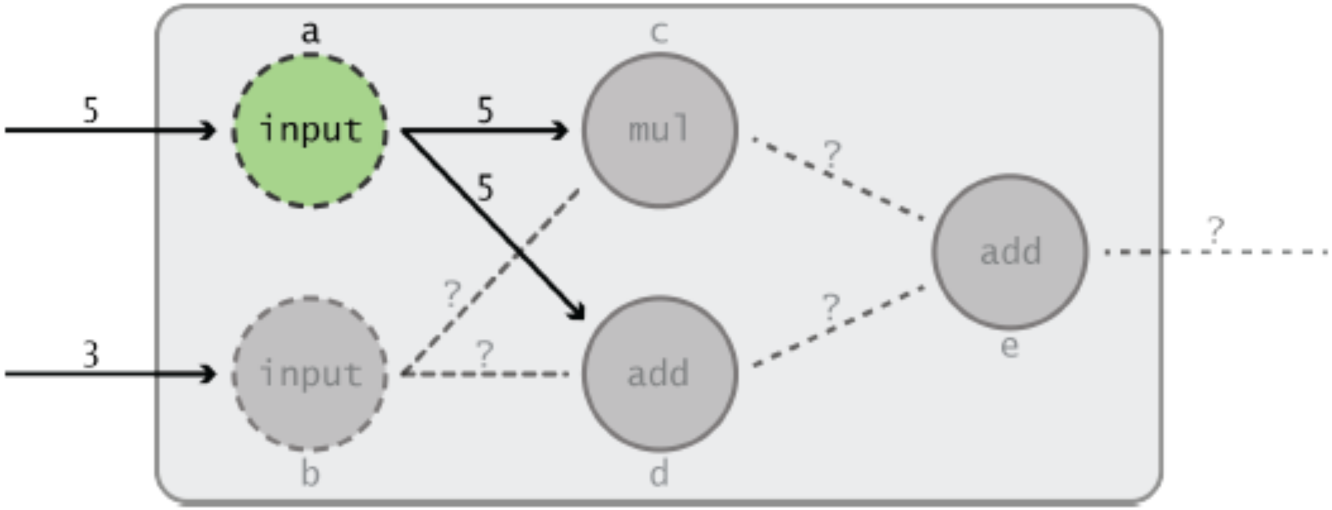
      let's look at what happens if the multiplication node c is unable to finish its computation (for whatever reason):



TensorFlow

- Graph basics:
  - Dependencies: …

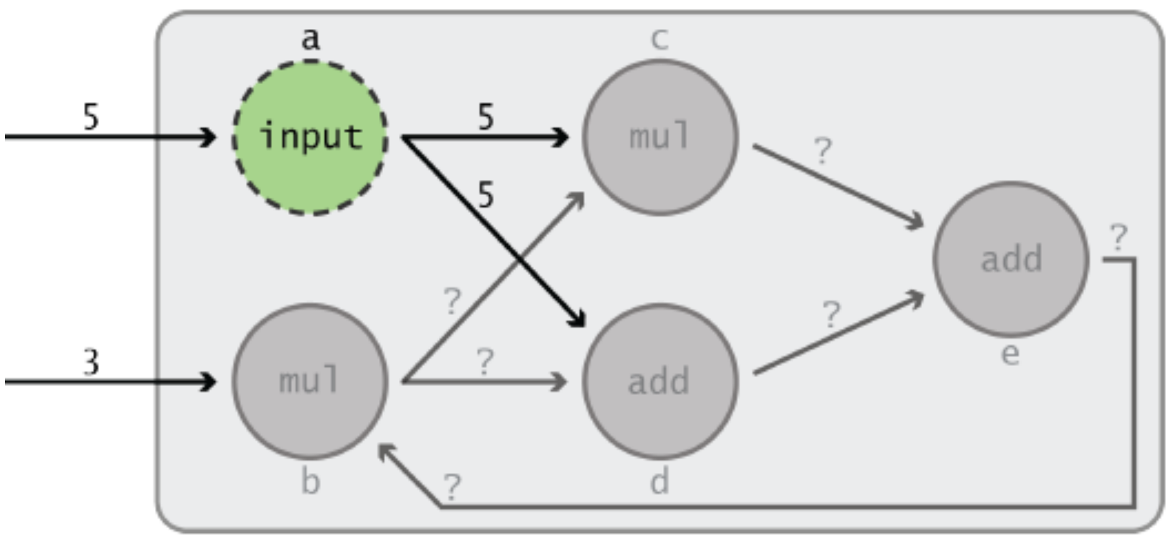
    What happens if one of the inputs fails to pass its data on to the next functions in the graph?



TensorFlow

- Graph basics:
  - Dependencies: …

    Let's see what happens if we redirect the output of a graph back into an earlier portion of it:

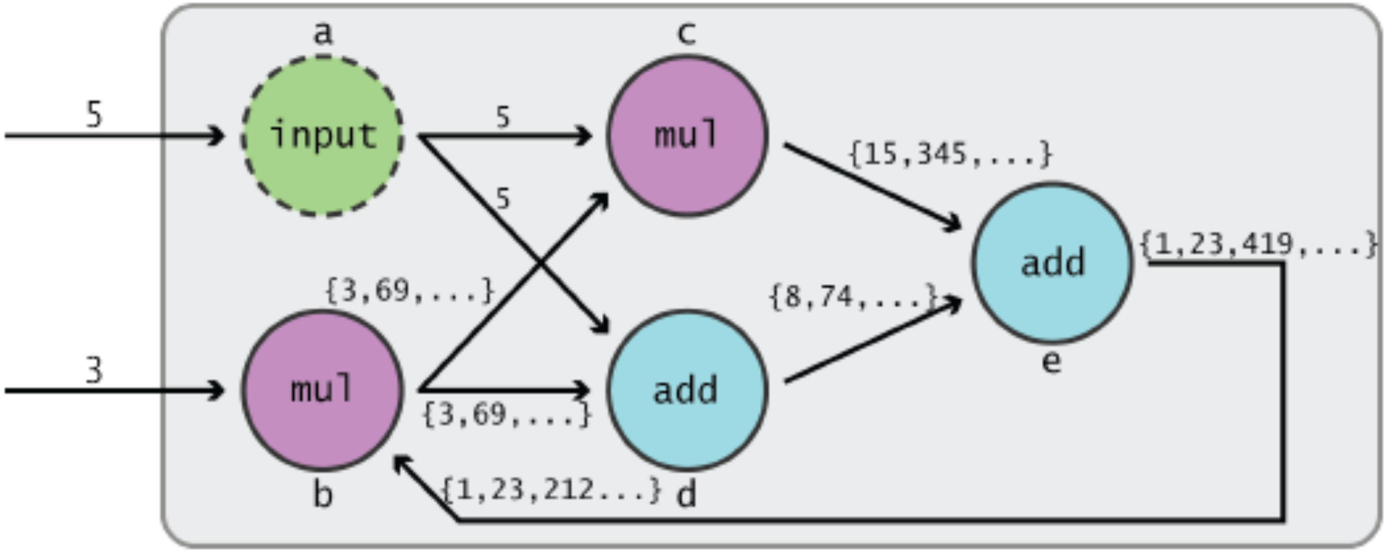


TensorFlow

- Graph basics:
  - Dependencies: …

Because of this, truly circular dependencies can't be expressed in TensorFlow, which is not a bad thing.

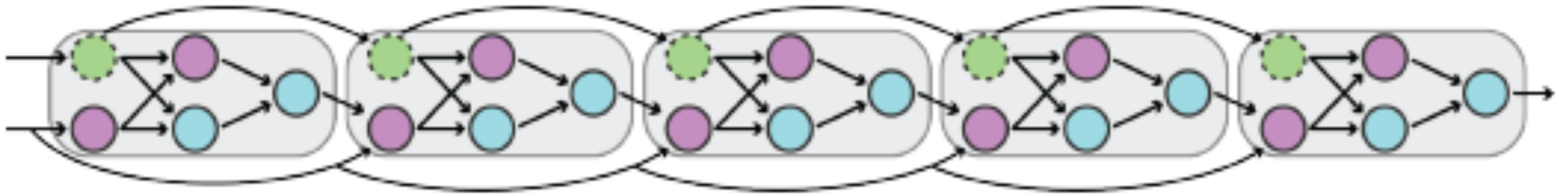
Let's provide an initial state to the value feeding into either *b* or *e* . Let's give the graph a kick-start by giving the output of *e* an initial value of 1:



TensorFlow

- Graph basics:
  - Dependencies: …

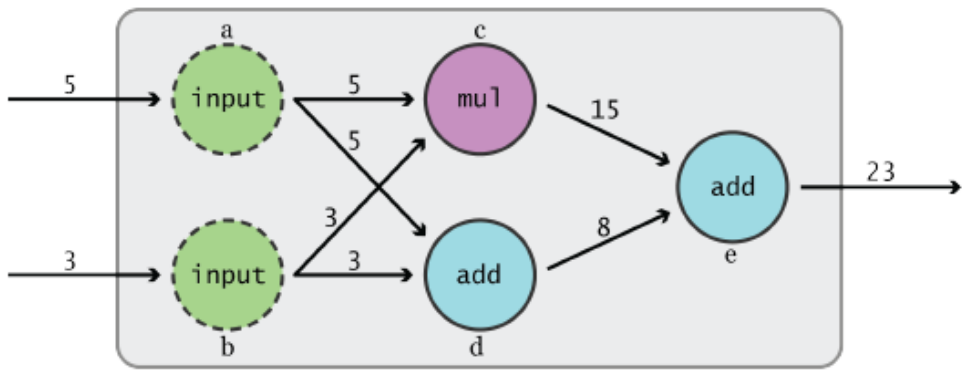
    By unrolling our graph like this, we can simulate useful cyclical dependencies while maintaining a deterministic computation.



TensorFlow

let's recall

- Building our first graph in TensorFlow:



simple_graph.py

```python
## Building our first TensorFlow graph:

# First we need to import TensorFlow:
import tensorflow as tf

# Let's define our input nodes:
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")

# Defining the next two nodes in our graph:
c = tf.multiply(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")

# This last line defines the final node in our graph:
e = tf.add(c,d, name="add_e")
```

TensorFlow

let's recall

- Building our first graph in TensorFlow:

```
Python                                                        /Users/alex
In [6]: whos
Variable   Type      Data/Info
------------------------------
a          Tensor    Tensor("input_a_1:0", shape=(), dtype=int32)
b          Tensor    Tensor("input_b_1:0", shape=(), dtype=int32)
c          Tensor    Tensor("mul_c:0", shape=(), dtype=int32)
d          Tensor    Tensor("add_d:0", shape=(), dtype=int32)
e          Tensor    Tensor("add_e:0", shape=(), dtype=int32)
tf         module    <module 'tensorflow' from<...>tensorflow/__init__.pyc'>
```

simple_graph.py

```python
## Building our first TensorFlow graph:

# First we need to import TensorFlow:
import tensorflow as tf

# Let's define our input nodes:
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")

# Defining the next two nodes in our graph:
c = tf.multiply(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")

# This last line defines the final node in our graph:
e = tf.add(c,d, name="add_e")
```

To run we must add the two extra lines and run them in the shell:

```
In [7]: sess = tf.Session()

In [8]: sess.run(e)
Out[8]: 23
```

TensorFlow

- Let's construct the actual graph using TensorBoard:
  - First, we need to make sure we have generated summary data in a log directory by creating a summary writer:

```
tf.Graph().as_default()
```

sess.graph.as_graph_def()
contains the graph definition that enables the Graph Visualizer

```
# To create the graph:
sess.graph.as_graph_def()
file_writer = tf.summary.FileWriter('./', sess.graph)
```

More on TF 2.x later!

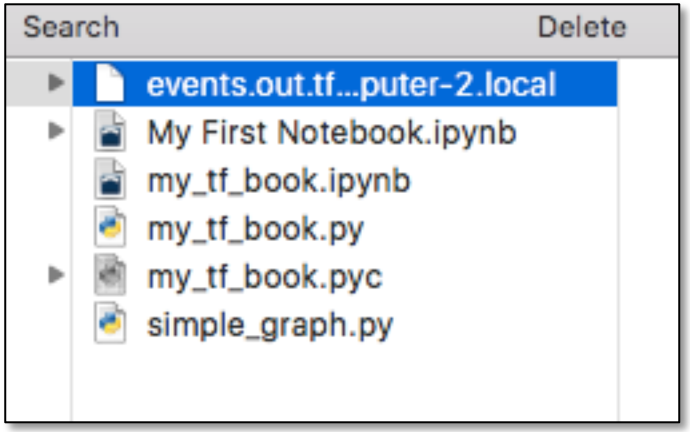
Docstring:
Writes `Summary` protocol buffers to event files.

'/path/to/logs'

```
tf.summary.FileWriter is deprecated. Please use tf.compat.v1.summary.FileWriter instead.
```

TensorFlow

- Let's construct the actual graph using TensorBoard:
  - Once we run the previous code, a file with the session is generated in our current folder:

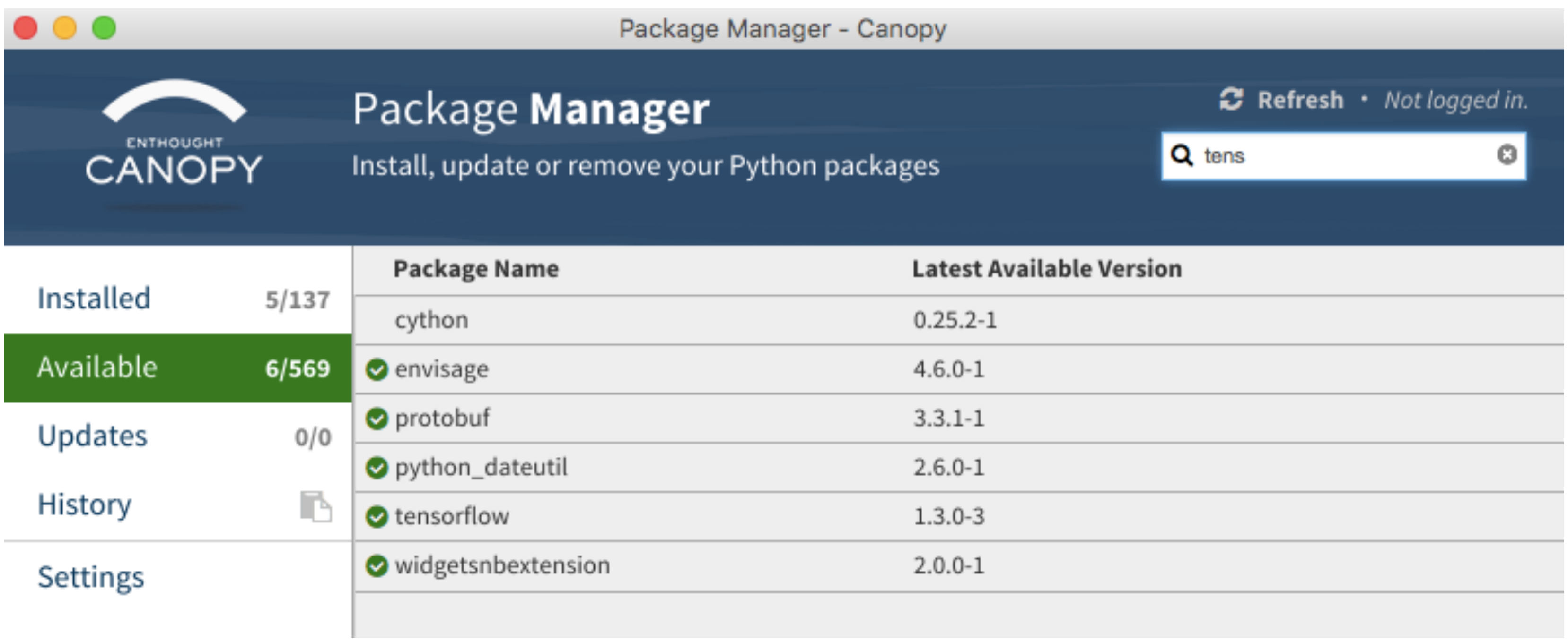


TensorFlow

legacy

- Let's construct the actual graph using TensorBoard:

  - Before we continue, we need to check if we have TensorBoard installed in our system:

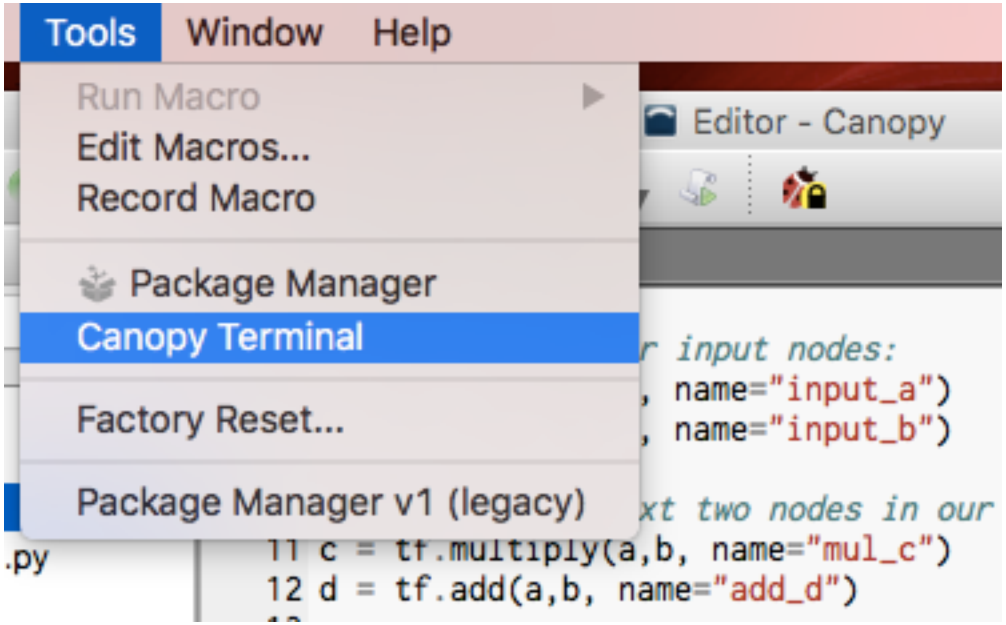


TensorFlow

legacy

- Let's construct the actual graph using TensorBoard:

  - Canopy does not provide TensorBoard in its repository, therefore we need to install it via the Canopy Terminal:



TensorFlow

legacy

- Let's construct the actual graph using TensorBoard:

  - WARNING: We check for compatibility: tensorflow + tensorboard

```
tensorboard                           1.8.0
tensorflow                            1.6.0
```

When installing tensorboard you might get this message:

```
tensorflow 1.6.0 has requirement tensorboard<1.7.0,>=1.6.0, but you'll have tensorboard 1.8.0 which is incompatible.
```

When running tensorboard you might get this message:

```
(Canopy 64bit) Alexandomputer2:alex_examples alex$ tensorboard --logdir ./ --eve
nt_file events.out.tfevents.1527382420.Alexandomputer2
2018-05-26 18:00:37.024647: I tensorflow/core/platform/cpu_feature_guard.cc:140]
 Your CPU supports instructions that this TensorFlow binary was not compiled to
use: SSE4.1
```

So you have to install a compatible version:

```
alex$ pip install tensorboard==1.6.0
```

Make sure they match:

```
tensorboard                           1.6.0
tensorflow                            1.6.0
```

When running tensorboard it runs without warnings:

```
(Canopy 64bit) Alexandomputer2:alex_examples alex$ tensorboard --logdir ./ --event_file events.
TensorBoard 1.6.0 at http://Alexandomputer2:6006 (Press CTRL+C to quit)
```

TensorFlow

legacy

- Let's construct the actual graph using TensorBoard:

  - Once we have the event file(s), we run TensorBoard while providing the log directory:

```
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ tensorboard --logdir ./
--event_file events.out.tfevents.1522372465.Alexander-Ilievs-Computer-2.local
```

  - and specifically request the file to be executed

  - or if you are in the same directory where the file resides simply run:

```
tensorboard --logdir ./
```

TensorFlow

legacy

- Let’s construct the actual graph using TensorBoard:
  - And the graph is already running in the background:

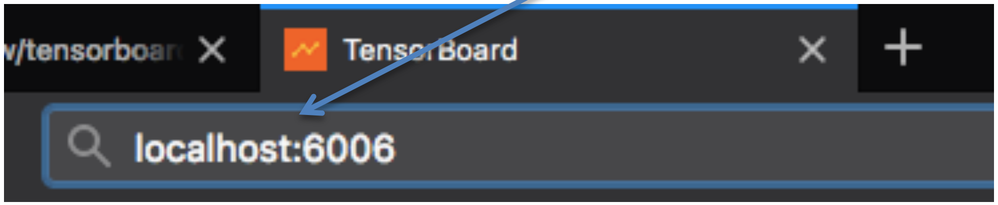
```
Lecture 1 — Canopy Terminal — tensorboard --logdir ./ --event_file==events.out.tfevents.1550186842.Alexandom...

[(Canopy 64bit) (base) airbears2-10-142-167-9:/ alex$ ls
1.HD                    Downloads               RPI_bkp                 include
AndroidStudioProjects   Dropbox                 Sites                   lib
Applications            Library                 bin                     nltk_data
Books                   Movies                  bm_test                 numpy
Creative Cloud Files    Music                   datasets                scikit_learn_data
Desktop                 Pictures                eclipse-workspace       scipy
Documents               Public                  env                     wekafiles
[(Canopy 64bit) (base) airbears2-10-142-167-9:/ alex$ cd '/Users/alex/1.HD/Alex/Work/3.Berkeley Extension/3. fi
nal course material/3. Machine Learning With TensorFlow/4. code/alex_examples/Lecture 1/'
[(Canopy 64bit) (base) airbears2-10-142-167-9:Lecture 1 alex$ ls
build_a_graph.py                                my_tf_book_normal.py
events.out.tfevents.1550186842.Alexandomputer2  my_tf_book_poison.py
my_tf_book.ipynb                                saving_a_graph.py
my_tf_book.pyc
[(Canopy 64bit) (base) airbears2-10-142-167-9:Lecture 1 alex$ tensorboard --logdir ./ --event_file=='events.out
.tfevents.1550186842.Alexandomputer2'
/Users/alex/Library/Enthought/Canopy_64bit/User/lib/python2.7/site-packages/h5py/__init__.py:34: FutureWarning
: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it w
ill be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
TensorBoard 1.6.0 at http://airbears2-10-142-167-9.airbears2.1918.berkeley.edu:6006 (Press CTRL+C to quit)
```

TensorFlow

legacy

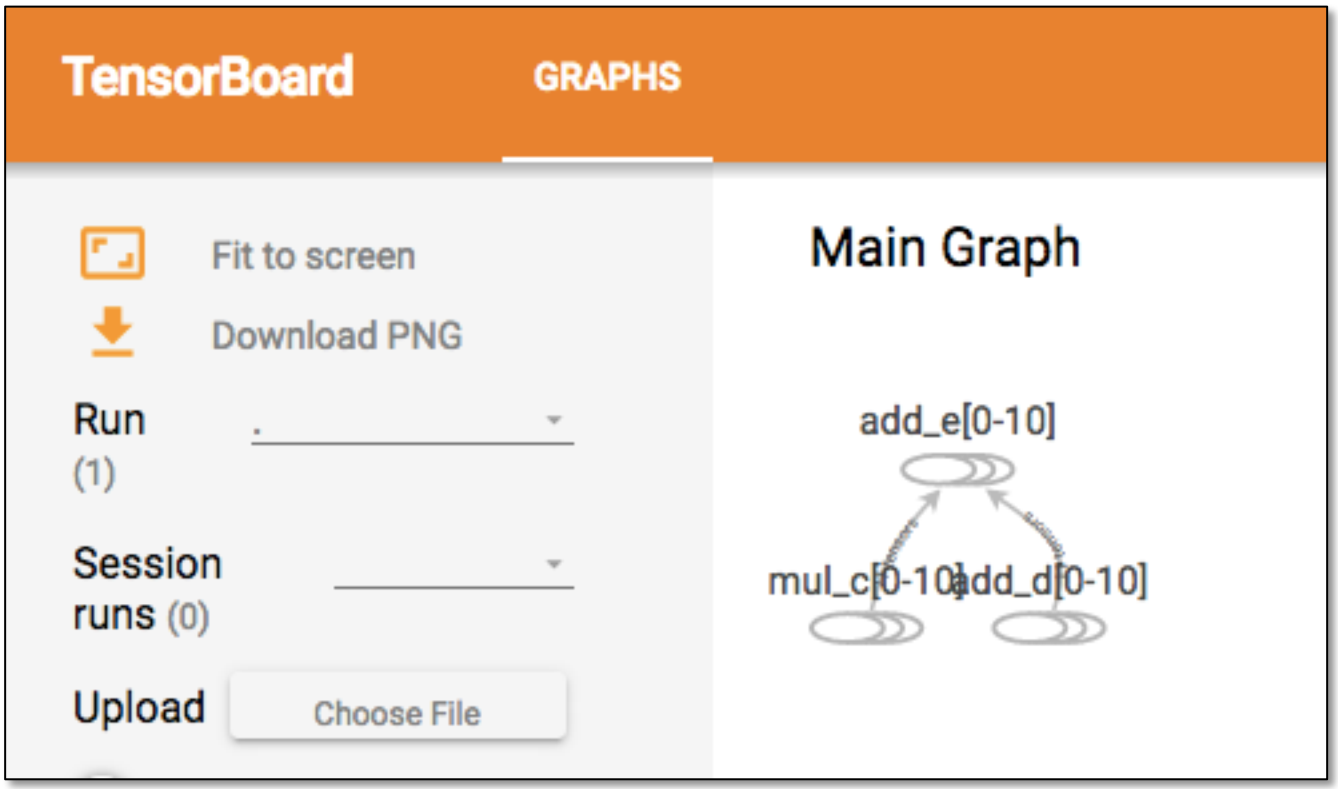
- Let's construct the actual graph using TensorBoard:

  - Next, we open our browser and look at the graph by typing:



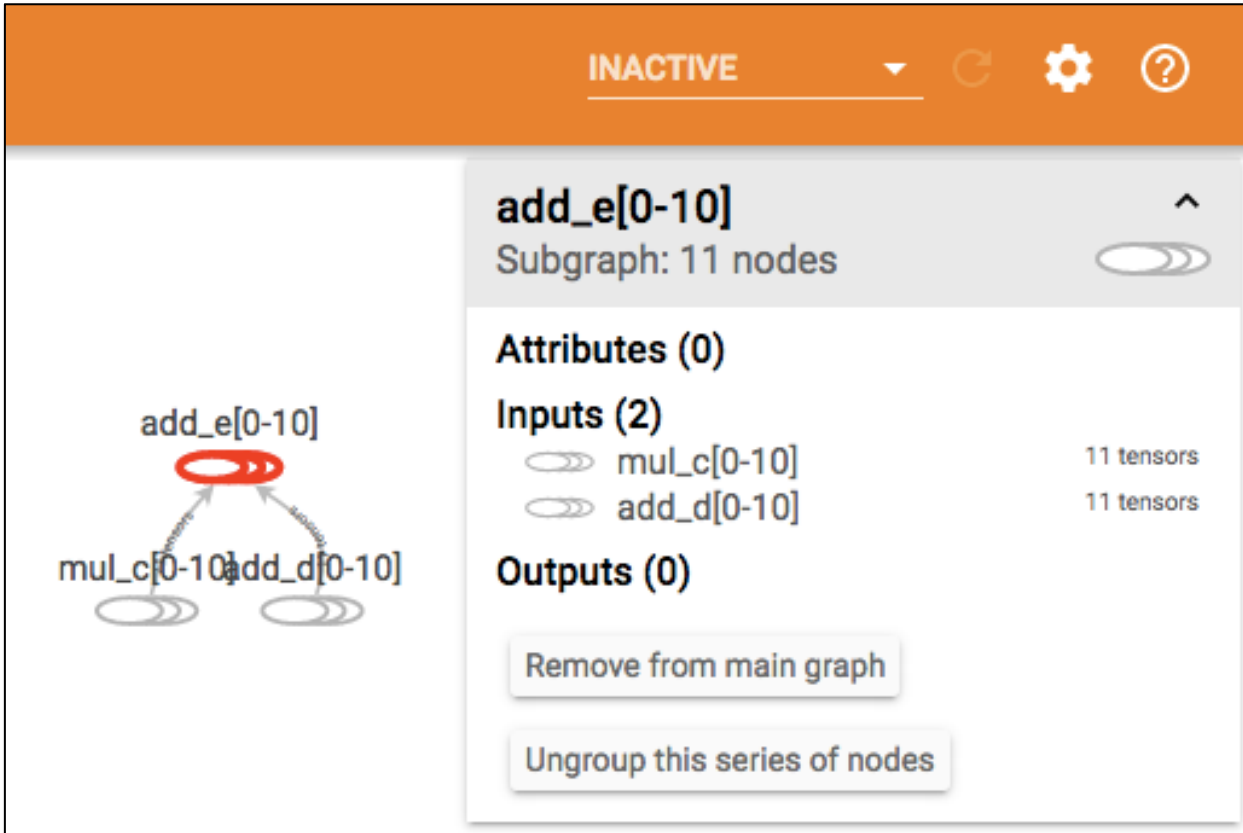
TensorFlow

- Let’s construct the actual graph using TensorBoard:
  - Below is a graphical representation of our first graph:



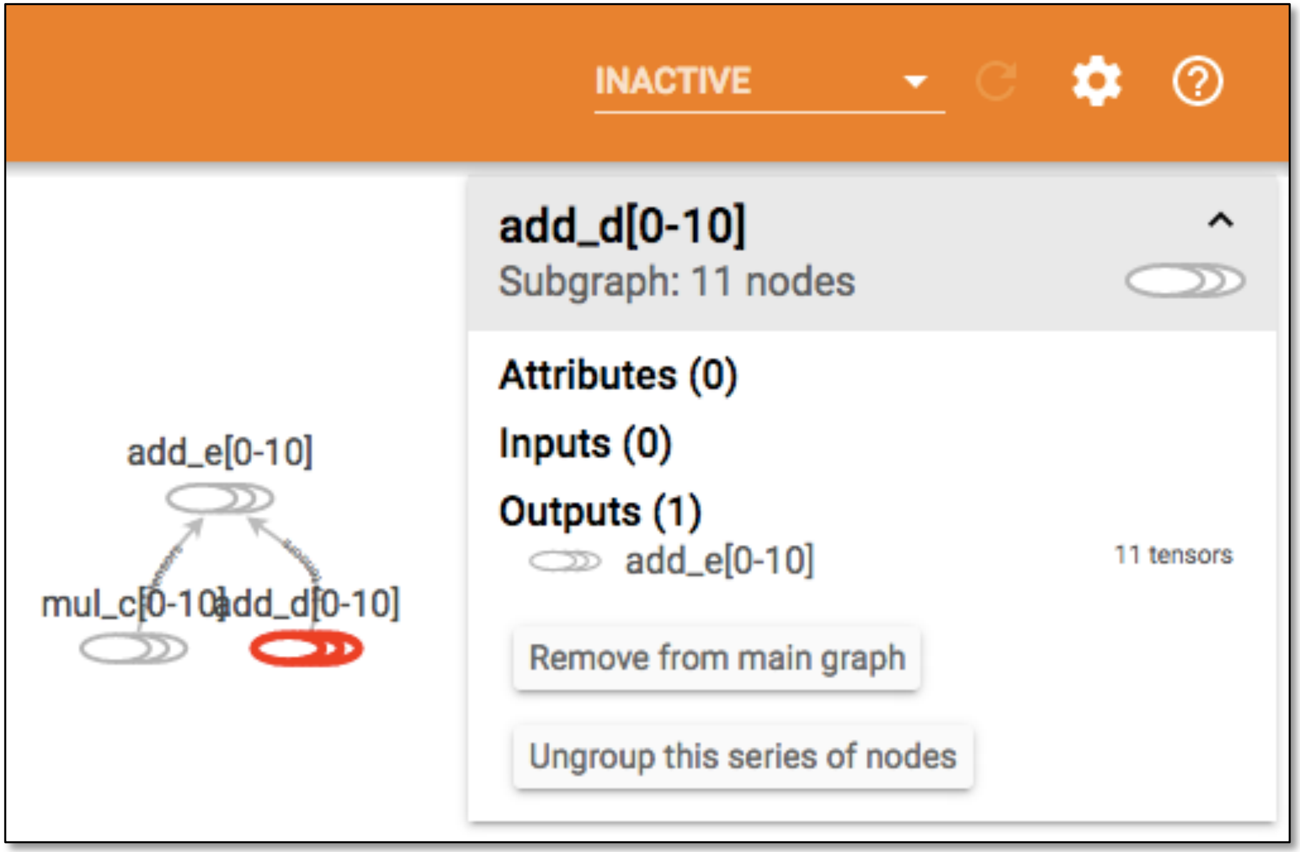
TensorFlow

- Let's construct the actual graph using TensorBoard:
  - Simply click on any of the nodes to inspect them more closely:



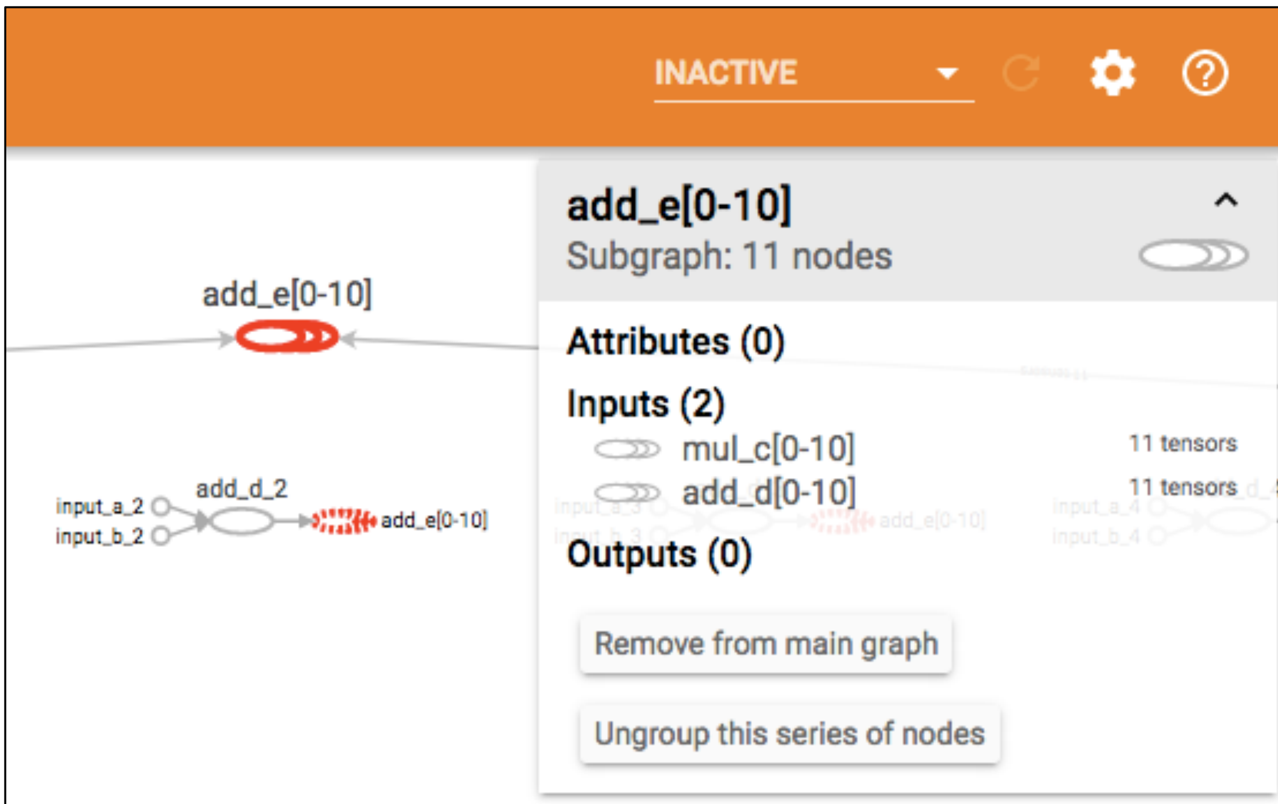
TensorFlow

- Let's construct the actual graph using TensorBoard:
  - Simply click on any of the nodes to inspect them more closely:



TensorFlow

- Let’s construct the actual graph using TensorBoard:
  - Simply click on any of the nodes to inspect them more closely:



TensorFlow

Symbol	Meaning
	High-level node representing a name scope. Double-click to expand a high-level node.
	Sequence of numbered nodes that are not connected to each other.
	Sequence of numbered nodes that are connected to each other.
	An individual operation node.
	A constant.
	A summary node.
	Edge showing the data flow between operations.
	Edge showing the control dependency between operations.
	A reference edge showing that the outgoing operation node can mutate the incoming tensor.

TensorFlow

legacy

- Let's construct the actual graph using TensorBoard:

  - Once we are done constructing our graph, we need to clean up and close the *file_writer* and *sess*:

```
# We clean up before we exit:
file_writer.close()
sess.close()
```

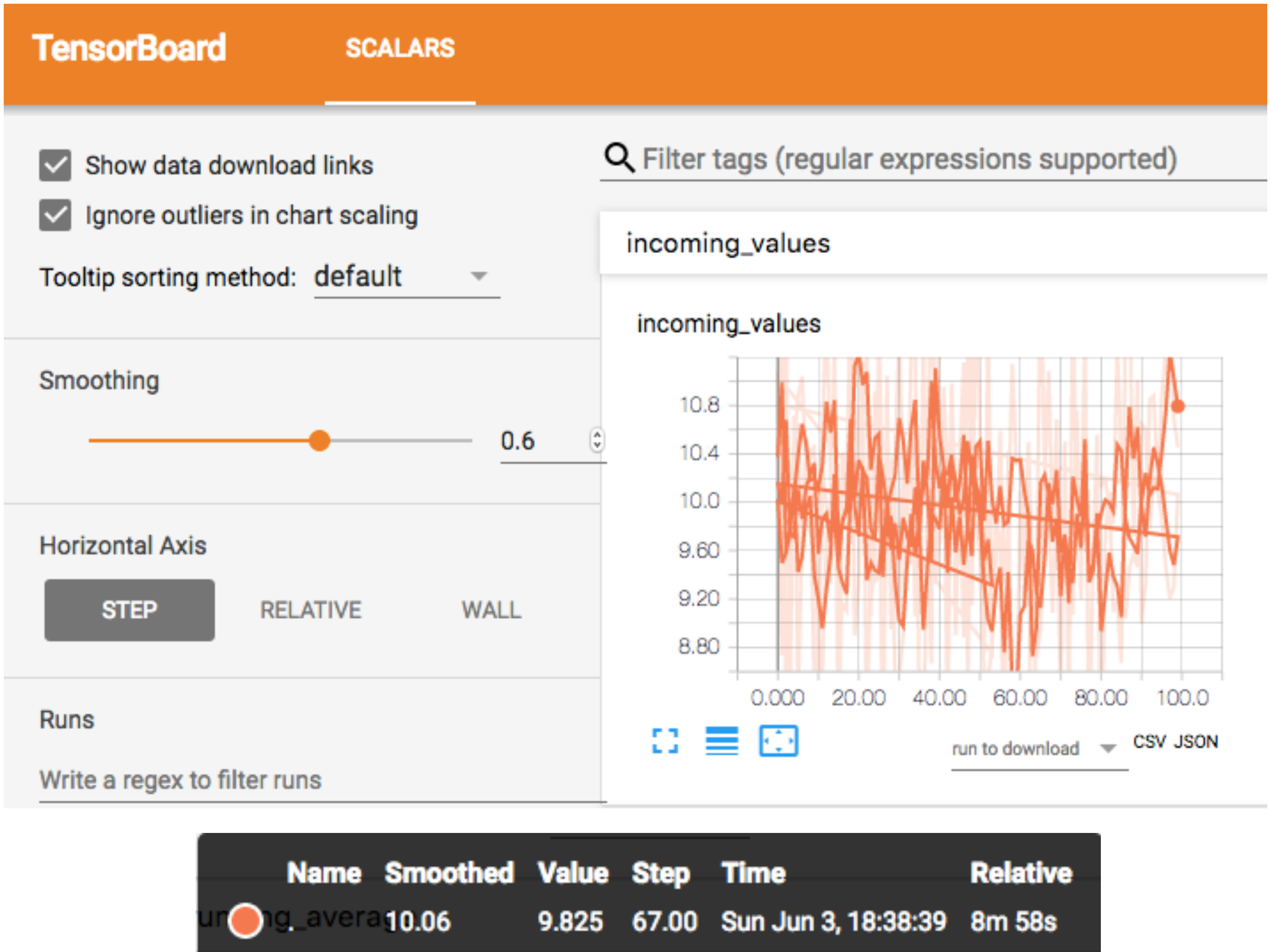
```
tf.summary.FileWriter is deprecated. Please use tf.compat.v1.summary.FileWriter instead.
```

  - In general, *Session* objects close automatically when the program terminates (or, in the interactive case, when you close/restart the Python kernel)

  - However, it's best to explicitly close out of the *Session* to avoid any sort of weird edge case scenarios.

TensorFlow

- Let’s construct the actual graph using TensorBoard:



TensorFlow operations

recall

- Remember: Tensors are just a superset of matrices!
- TensorFlow Operations, aka Ops, are nodes that perform computations on or with Tensor objects
- After computation, they return zero or more tensors, which can be used by other Ops later in the graph
- To create an Operation, you call its constructor in Python
- The Python constructor returns a handle to the Operation's output, then it is passed on to other Ops or Session.run

TensorFlow operations

recall

- Example:

```
import tensorflow as tf
import numpy as np

# Initialize some tensors to use in computation
a = np.array([2, 3], dtype=np.int32)
b = np.array([4, 5], dtype=np.int32)

# Use `tf.add()` to initialize an "add" Operation
# The variable `c` will be a handle to the Tensor output of
this Op
c = tf.add(a, b)
```

TensorFlow operations

- Overloaded operators:
  - TensorFlow also overloads common mathematical operators to make multiplication, addition, subtraction, and other operations more concise
  - If one or more arguments to the operator is a Tensor object, a TensorFlow Operation will be called and added to the graph
  - For example, you can easily add two tensors together like this:

```
# Assume that `a` and `b` are `Tensor` objects with
matching shapes
c = a + b
```

TensorFlow operations

- Creating aliases:
  - TensorFlow allows us to create aliases for all common mathematical operators such as negation, subtraction, multiplication, or other operations to be more concise
  - For example, you can easily create aliases like this:

```
In [18]: tf.neg = tf.negative

In [19]: tf.sub = tf.subtract

In [20]: tf.mul = tf.multiply
```

TensorFlow operations

- Overloaded operators using aliases (from previous slide):
  - A complete list of overloaded operators is given for tensors:

Unary operators

Operator	Related TensorFlow Operation	Description
-x	`tf.neg()` `tf.neg = tf.negative`	Returns the negative value of each element in x
~x	`tf.logical_not()`	Returns the logical NOT of each element in x. Only compatible with `Tensor` objects with `dtype` of `tf.bool`
abs(x)	`tf.abs()`	Returns the absolute value of each element in x

TensorFlow operations

- Overloaded operators using aliases:

Binary operators	Related TensorFlow Operation	Description
x + y	`tf.add()`	Add x and y, element-wise
x - y	`tf.sub()` `tf.sub = tf.subtract`	Subtract y from x, element-wise
x * y	`tf.mul()` `tf.mul = tf.multiply`	Multiply x and y, element-wise
x / y (Python 2)	`tf.div()`	Will perform element-wise integer division when given an integer type tensor, and floating point ("true") division on floating point tensors
x / y (Python 3)	`tf.truediv()`	Element-wise floating point division (including on integers)
x // y (Python 3)	`tf.floordiv()`	Element-wise floor division, not returning any remainder from the computation

TensorFlow operations

- Overloaded operators:

Binary operators	Related TensorFlow Operation	Description
x % y	`tf.mod()`	Element-wise modulo
x ** y	`tf.pow()`	The result of raising each element in x to its corresponding element y, element-wise
x < y	`tf.less()`	Returns the truth table of x < y, element-wise
x `<=` y	`tf.less_equal()`	Returns the truth table of x `<= y`, element-wise
x > y	`tf.greater()`	Returns the truth table of x > y, element-wise
x >= y	`tf.greater_equal()`	Returns the truth table of x >= y, element-wise

TensorFlow operations

- Overloaded operators:
  - A complete list of overloaded operators is given for tensors:

Binary operators	Related TensorFlow Operation	Description
x & y	`tf.logical_and()`	Returns the truth table of x & y, element-wise. `dtype` must be `tf.bool`
x | y	`tf.logical_or()`	Returns the truth table of x | y, element-wise. `dtype` must be `tf.bool`
x ^ y	`tf.logical_xor()`	Returns the truth table of x ^ y, element-wise. `dtype` must be `tf.bool`

A quick reminder:

A	B	AND	OR	XOR	NOT B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

TensorFlow operations

- Overloaded operators:
  - Using these overloaded operators can be great when quickly putting together code
  - Technically, the == operator is overloaded as well, but it will not return a Tensor of boolean values. It will return **True** if the two tensors being compared are the same object, and **False** otherwise.
  - To check for equality or inequality, try:

    tf.equal() and tf.not_equal, respectively.

TensorFlow graphs

- Creating a Graph is simple:
  - its constructor doesn't take any variables:

```
import tensorflow as tf

# Create a new graph:
g = tf.Graph()
```

  - Once we have our Graph initialized, we can add Operations to it:

```
with g.as_default():
    # Create Operations as usual; they will be added to graph
`g`
    a = tf.mul(2, 3)
    ...
```

TensorFlow graphs

- TensorFlow automatically creates a Graph when the library is loaded and assigns it to be the default.
- Thus, any Operations, tensors, etc. defined outside of a Graph.as_default() context manager will automatically be placed in the default graph:

```
# Placed in the default graph
in_default_graph = tf.add(1,2)

# Placed in graph `g`
with g.as_default():
    in_graph_g = tf.mul(2,3)

# We are no longer in the `with` block, so this is placed in
the default graph
also_in_default_graph = tf.sub(5,1)
```

TensorFlow graphs

- If you'd like to get a handle to the default graph, use the tf.get_default_graph() function:

```
default_graph = tf.get_default_graph()
```

- In most TensorFlow programs, you will only ever deal with the default graph

```
tf.compat.v1.get_default_graph()
```

- When defining multiple graphs in one file, it's better to either not use the default graph or immediately assign a handle to it
- This ensures that nodes are added to each graph in a uniform manner

TensorFlow graphs

- This ensures that nodes are added to each graph in a uniform manner:

Correct - Create new graphs, ignore default graph:

```
import tensorflow as tf


g1 = tf.Graph()
g2 = tf.Graph()

with g1.as_default():
    # Define g1 Operations, tensors, etc.
    ...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

TensorFlow graphs

- This ensures that nodes are added to each graph in a uniform manner:

Correct - Get handle to default graph:

```
import tensorflow as tf

g1 = tf.get_default_graph()
g2 = tf.Graph()

with g1.as_default():
    # Define g1 Operations, tensors, etc.
    ...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

```
tf.compat.v1.get_default_graph()
```

TensorFlow graphs

- This ensures that nodes are added to each graph in a uniform manner:

Incorrect - Mix default graph and user-created graph styles:

```
import tensorflow as tf

g2 = tf.Graph()

# Define default graph Operations, tensors, etc.
...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

TensorFlow graphs

- Additionally, it is possible to load in previously defined models from other TensorFlow scripts and assign them to Graph objects
- This can be done by using a combination of the graph.as_graph_def() and tf.import_graph_def functions
- Thus, a user can compute and use the output of several separate models in the same Python file.

TensorFlow sessions

- Sessions, are responsible for graph execution
- The constructor takes in three optional parameters:

target specifies the execution engine to use:

- When using sessions in a distributed setting, this parameter is used to connect to tf.train.Server instances

graph specifies the Graph object that will be launched in the Session.

- When using multiple graphs, it's best to explicitly pass in the Graph you'd like to run (instead of creating the Session inside of a with block).

config allows users to specify options to configure the session, such as limiting the number of CPUs or GPUs to use, setting optimization parameters for graphs, and logging options.

TensorFlow sessions

- In a typical TensorFlow program, Session objects will be created without changing any of the default construction parameters:

```
import tensorflow as tf

# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

# Start up a `Session` using the default graph
sess = tf.Session()
```

- Note that these two calls are identical:

```
sess = tf.Session()
sess = tf.Session(graph=tf.get_default_graph())
```

```
The name tf.Session is deprecated. Please use tf.compat.v1.Session instead.
```

TensorFlow sessions

- Once a Session is opened, you can use its primary method, run(), to calculate the value of a desired Tensor output:

```
sess.run(b)  # Returns 21
```

- Session.run() takes in one required parameter, fetches,

  (as well as three optional parameters: feed_dict , options , and run_metadata)

- In the previous example, we set fetches to the tensor **b** (the output of the tf.mul Operation)

TensorFlow sessions

- We can also pass in a list of graph elements:

```
sess.run([a, b])  # returns [7, 21]
```

- When fetches is a list, the output of run() will be a list with values corresponding to the output of the requested elements.
- In this example, we ask for the values of **a** and **b**, in that order
- Since both **a** and **b** are tensors, we receive their values as output

TensorFlow sessions

- We can give fetches a direct handle to an Operation
- An example of this is tf.global_variables_initializer(), which prepares all TensorFlow Variable objects to be used
- We still pass the Op as the fetches parameter, but the result of Session.run() will be None:

```
# Performs the computations needed to initialize Variables,
but returns `None`
sess.run(tf.initialize_all_variables())
```

```
tf.global_variables_initializer()
```

notice the change in latest 1.x TF versions

notice the change in TF 2.1

```
tf.compat.v1.initialize_all_variables()
```

TensorFlow sessions

- The parameter feed_dict is used to override Tensor values in the graph, and it expects a Python dictionary object as input.

- The keys in the dictionary are handles to Tensor objects that should be overridden, while the values can be numbers, strings, lists, or NumPy arrays (as described previously)

- The values must be of the same type (or able to be converted to the same type) as the Tensor key.

TensorFlow sessions

- Here is an example of how feed_dict is used to override Tensor value **a**:

```python
import tensorflow as tf

# Create Operations, Tenso
a = tf.add(2, 5)
b = tf.mul(a, 3)

# Start up a `Session` usi
sess = tf.Session()
```

```
tf.compat.v1.Session()
```

```python
# Define a dictionary that
with 15
replace_dict = {a: 15}

# Run the session, passing in `replace_dict` as the value to
`feed_dict`
sess.run(b, feed_dict=replace_dict)  # returns 45
```

result:

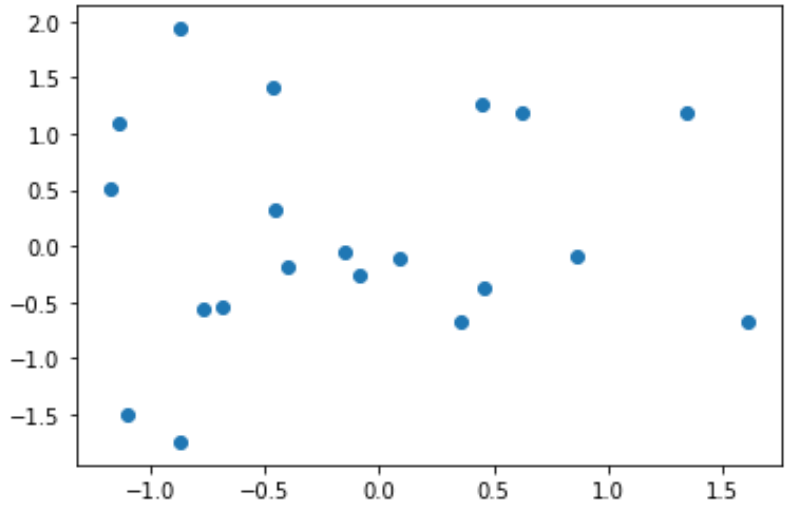
```
Python

In [3]: a = tf.add(2, 5)

In [4]: b = tf.multiply(a, 3)

In [5]: sess = tf.Session()

In [6]: replace_dict = {a: 15}

In [7]: sess.run(b, feed_dict = replace_dict)
Out[7]: 45
```

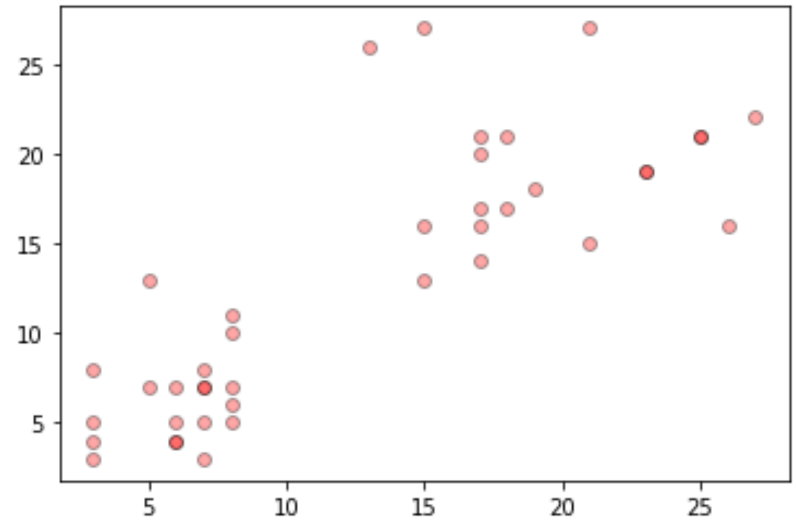
Machine Learning With TensorFlow

Class exercise 1/2:

- Given what we discussed in class today create a short TF program that:

- Creates and plots a *normally distributed* cluster of random numbers, say [2, 20] *[var,#]
- Creates and plots two clusters using *poison distribution*, say [6,20],[2,20] *in red*
- Comment on your results





UC Berkeley Extension

Machine Learning With TensorFlow

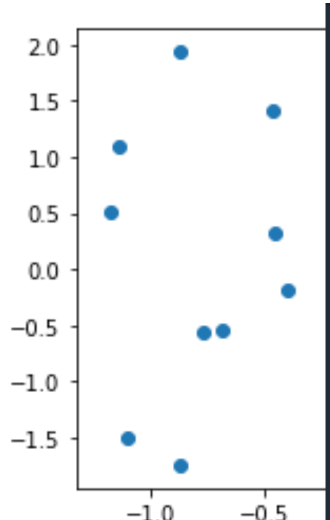
```python
# Normal distribution:
import matplotlib.pyplot as plt
import tensorflow as tf

# This is only for Jupyter plotting in-line:
# % matplotlib inline

a = tf.random_normal([2,20])

sess = tf.Session()
out = sess.run(a)
x, y = out
plt.scatter(x, y)

plt.show()
```

/2:

ate a short TF program that:

f random numbers, say [2, 20] *[var,#]

bution, say [6,20],[2,20] *in red*




```python
# Poison distribution:
import matplotlib.pyplot as plt
import tensorflow as tf

# This is only for Jupyter plotting in-line:
# % matplotlib inline

a = tf.random_poisson([6,20],[2,20])

sess = tf.Session()
out = sess.run(a)
x, y = out

color = 'red'
plt.scatter(x, y, c=color, label=color, alpha=0.35, edgecolors='black')

plt.show()
```

TensorFlow placeholder

- Adding Inputs with placeholder nodes
- To take values from the client and plug them into our graph we use what is called a “placeholder”
- Placeholders, act as if they are Tensor objects, but they do not have their values specified when created
- Instead, they hold the place for a Tensor that will be fed at runtime, hence become input nodes

TensorFlow placeholder

- Adding Inputs with placeholder nodes
- Example:

```
import tensorflow as tf
import numpy as np

# Creates a placeholder vector of length 2 with data type int32
a = tf.placeholder(tf.int32, shape=[2], name="my_input")

# Use the placeholder as if it were any other Tensor object
b = tf.reduce_prod(a, name="prod_b")
c = tf.reduce_sum(a, name="sum_c")

# Finish off the graph
d = tf.add(b, c, name="add_d")
```

```
tf.math.reduce_prod()
tf.math.reduce_sum()
tf.math.add()
```

TensorFlow placeholder

- Adding Inputs with placeholder nodes

- tf.placeholder takes in a required parameter dtype, as well as the optional parameter shape:

    - **dtype** specifies the data type of values that will be passed into the placeholder. This is required, in order to ensure that there will be no type mismatch errors

    - **shape** specifies what shape the fed Tensor will be. The default value of shape is None, which means a Tensor of any shape will be accepted

TensorFlow placeholder

- You can also specify a name identifier to tf.placeholder

```
# Open a TensorFlow Session
sess = tf.Session()

# Create a dicti
# Key: `a`, the
# Value: A vecto
input_dict = {a:

# Fetch the valu
into `a`
sess.run(d, feed
```

result:

```
In [13]: import tensorflow as tf

In [14]: a = tf.placeholder(tf.int32, shape=[2], name="my_input")

In [15]: b = tf.reduce_prod(a, name="prod_b")

In [16]: c = tf.reduce_sum(a, name="sum_c")

In [17]: d = tf.add(b, c, name="add_d")

In [18]: sess = tf.Session()

In [19]: import numpy as np

In [20]: input_dict = {a: np.array([5, 3], dtype=np.int32)}

In [21]: sess.run(d, feed_dict = input_dict)
Out[21]: 23
```

TensorFlow variables

- Tensor and Operation objects are immutable
- We need a way to save changing values over time
- This is accomplished in TensorFlow with Variable objects
- You can create a Variable by using its constructor, tf.Variable():

```
import tensorflow as tf

# Pass in a starting value of three for the variable
my_var = tf.Variable(3, name="my_variable")
```

TensorFlow variables

- Variables can be used in TensorFlow functions/Operations anywhere you might use a Tensor
- Its present value will be passed on to the Operation using it:

```
add = tf.add(5, my_var)
mul = tf.mul(8, my_var)
```

- The initial value of Variables will often be large tensors of zeros, ones, or random values

- TensorFlow has a number of helper Ops, such as:
  tf.zeros(), tf.ones(), tf.random_normal(), and tf.random_uniform()

TensorFlow variables

- Each of these takes in a shape parameter which specifies the dimension of the desired Tensor:

```
# 2x2 matrix of  zeros
zeros = tf.zeros([2, 2])

# vector of length 6 of ones
ones = tf.ones([6])

# 3x3x3 Tensor of random uniform  values between 0 and 10
uniform = tf.random_uniform([3, 3, 3], minval=0, maxval=10)

# 3x3x3 Tensor of normally distributed numbers; mean 0 and
standard deviation 2
normal = tf.random_normal([3, 3, 3], mean=0.0, stddev=2.0)
```

```
tf.random.normal()
tf.random.uniform()
```

TensorFlow variables

- Instead of using tf.random_normal(), you'll often see use of tf.truncated_normal() instead

```
# No values below 3.0 or above 7.0 will be returned in this
Tensor
trunc = tf.truncated_normal([2, 2], mean=5.0, stddev=1.0)
```

```
tf.random.truncated_normal()
```

- You can pass in these Operations as the initial values of Variables as you would a handwritten Tensor:

```
# Default value of mean=0.0
# Default value of stddev=1.0
random_var = tf.Variable(tf.truncated_normal([2, 2]))
```

TensorFlow variables

- Variable objects live in the Graph like most other TensorFlow objects, but their state is actually managed by a Session.
- Because of this, Variables have an extra step involved in order to use them:
    - you must initialize the Variable within a Session

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

```
Use `tf.global_variables_initializer` instead.
```

```
tf.compat.v1.Session()
```

```
tf.compat.v1.initialize_all_variables()
```

TensorFlow changing variables

- If you'd only like to initialize a subset of Variables defined in the graph, you can use tf.variables_initializer()
- This takes in a list of Variables to be initialized:

```
var1 = tf.Variable(0, name="initialize_me")
var2 = tf.Variable(1, name="no_initialization")
init = tf.initialize_variables([var1], name="init_var1")
sess = tf.Session()
sess.run(init)
```

```
tf.compat.v1.variables_initializer()
```

```
tf.compat.v1.Session()
```

- In order to change the value of the Variable, you can use the Variable.assign() method, which gives the Variable the new value

* Note that Variable.assign() is an Operation, and must be run in a Session to take effect

TensorFlow changing variables

- Example:

```
# Create variable with starting value of 1
my_var = tf.Variable(1)

# Create an operation that multiplies the variable by 2 each
time it is run
my_var_times_two = my_var.assign(my_var * 2)

# Initialization operation
init = tf.initialize_all_variables()

# Start a session
sess = tf.Session()

# Initialize variable
sess.run(init)

# Multiply variable by two and return it
sess.run(my_var_times_two)
## OUT: 2

# Multiply again
sess.run(my_var_times_two)
## OUT: 4

# Multiply again
sess.run(my_var_times_two)
## OUT: 8
```

```
tf.compat.v1.initialize_all_variables()
```

```
tf.compat.v1.Session()
```

TensorFlow changing variables

- For simple incrementing and decrementing of Variables, TensorFlow includes the Variable.assign_add() Variable.assign_sub() methods:

```
# Increment by 1
sess.run(my_var.assign_add(1))

# Decrement by 1
sess.run(my_var.assign_sub(1))
```

```
tf.compat.v1.assign_add()
tf.compat.v1.assign_sub()
```

- Why is this good? … Because Sessions maintain Variable values separately, each Session can have its own current value for a Variable defined in a graph (… on next slide)

TensorFlow changing variables

```python
# Create Ops
my_var = tf.Variable(0)
init = tf.initialize_all_variables()

# Start Sessions
sess1 = tf.Session()
sess2 = tf.Session()

# Initialize Variable in sess1, and increment value of my_var
in that Session
sess1.run(init)
sess1.run(my_var.assign_add(5))
## OUT: 5

# Do the same with sess2, but use a different increment value
sess2.run(init)
sess2.run(my_var.assign_add(2))
## OUT: 2

# Can increment the Variable values in each Session independ-
ently

sess1.run(my_var.assign_add(5))
## OUT: 10

sess2.run(my_var.assign_add(2))
## OUT: 4
```

```
tf.compat.v1.initialize_all_variables()
tf.compat.v1.Session()
tf.compat.v1.assign_add()
tf.compat.v1.assign_add()
tf.compat.v1.assign_add()
tf.compat.v1.assign_add()
```

TensorFlow changing variables

- If you'd like to reset your Variables to their starting value, simply call tf.global_variables_initializer() again
- Or you can use tf.variables_initializer() if you only want to reset a subset of them (see previous slides):

```
# Create Ops
my_var = tf.Variable(0)
init = tf.initialize_all_variables()

# Start Session
sess = tf.Session()

# Initialize Variables
sess.run(init)

# Change the Variable
sess.run(my_var.assign(10))

# Reset the Variable to 0, its initial value
sess.run(init)
```

```
tf.compat.v1.initialize_all_variables()
```

```
tf.compat.v1.Session()
```

TensorFlow trainable variables

- Optimizer classes automatically train machine learning models
- If there are Variables in your graph that should only be changed manually and not with an Optimizer, you need to set their trainable parameter to False when creating them:

```
not_trainable = tf.Variable(0, trainable=False)
```