UC Berkeley Extension

# Machine Learning With TensorFlow

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

Instructor Alexander I. Iliev, Ph.D.

# Machine Learning With TensorFlow

Class 2 …

*TensorFlow and TensorBoard, Data types, Linear algebra fundamentals …*

# 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 (5pts)

- **Tensorflow and TensorBoard basics**
- Linear algebra recap
- Data types in Numpy and Tensorflow
- Basic operations in Tensorflow
- Graph models and structures with Tensorboard

- **TensorFlow operations**
- Overloaded operators
- Using Aliases
- Sessions, graphs, variables, placeholders
- Name scopes HW2 (5pts)

- **Data Mining and Machine Learning concepts**
- Basic Deep Learning Models
- Linear and Logistic Regression
- Softmax classification

- **Neural Networks**
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions HW3 (5pts)

# Linear algebra

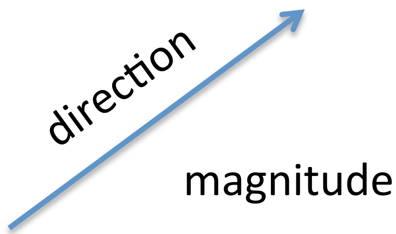
- Vector: 1) is a mathematical object that has a direction and a magnitude, used to find the position of one point in space relative to another point. 2) is a computer object, an array of data with individual items located with a single index
- Matrix is a 2-dimensional (rectangular) array of elements represented by: symbols, numbers, or expressions, all arranged in rows and columns. Matrix consist of vectors
- Array is an arrangement or a series of elements such as symbols, numbers, or expressions. Arrays can be n-dimensional, so matrix is an array with 2 dimensions
- Tensor is an objects describing the linear relationship among scalars, vectors and other tensors
- Rank of a matrix is the maximum number of linearly independent column (or row) vectors in the matrix

# Linear algebra

- Vector
  - 1) is a mathematical object that has a direction and a magnitude, used to find the position of one point in space relative to another point
  - 2) is a computer object, an array of data with individual items located with a single index

Example:

mathematical meaning




computer meaning

[ 3 2 4 5 -6 9 2 10 ]

# Linear algebra

- Matrix is a 2-dimensional (rectangular) array of elements represented by: symbols, numbers, or expressions, all arranged in rows and columns. Matrix consist of vectors

Example:

```
Python

In [1]: a = matrix([[2,-4,6],[-12,5,1],[-3,8,4]])

In [2]: a
Out[2]:
matrix([[  2,  -4,   6],
        [-12,   5,   1],
        [ -3,   8,   4]])

In [3]: b = matrix([[2],[8],[-4]])

In [4]: b
Out[4]:
matrix([[ 2],
        [ 8],
        [-4]])

In [5]: a*b
Out[5]:
matrix([[-52],
        [ 12],
        [ 42]])
```

# Linear algebra

- Array can be n-dimensional, but matrix is an array with 2 dimensions

  Example: numpy.array is a function that returns a numpy.ndarray and there for convenience

```
In [37]: c = np.zeros(shape=(4,5))   # the array contains zeroes for all elelements

In [38]: c
Out[38]:
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])

In [39]: d = np.empty(shape=(2,2))   # the array contains meaningless data

In [40]: d
Out[40]:
array([[ 0.,  0.],
       [ 0.,  0.]])

In [41]: e = np.ndarray(shape=(2,3), dtype=complex, offset=np.float_().itemsize, order='C')

In [42]: e
Out[42]:
array([[  0.00000000e+000 +1.72723382e-077j,
          2.12316144e-314 +2.14479474e-314j,
          2.12375379e-314 +2.24090241e-314j],
       [  2.12530167e-314 +2.12303539e-314j,
          2.24504872e-314 +3.27074300e+015j,
          3.28995843e-318 +8.34402697e-309j]])
```

# Linear algebra

- NumPy array

  Example:

```
In [23]: a = np.array([[12, 34, 41], [54, 62, 18], [72, 84, 96]], np.int16)

In [24]: a
Out[24]:
array([[12, 34, 41],
       [54, 62, 18],
       [72, 84, 96]], dtype=int16)

In [25]: a.size
Out[25]: 9

In [26]: a.shape
Out[26]: (3, 3)

In [27]: type(a)
Out[27]: numpy.ndarray

In [28]: a.dtype
Out[28]: dtype('int16')

In [29]: a[2,2]   # this is how we index a particular elemnt in the array (#9)
Out[29]: 96

In [30]: b = a[0,:]

In [31]: b
Out[31]: array([12, 34, 41], dtype=int16)

In [32]: b.shape
Out[32]: (3,)

In [33]: b[2] = 88   # this is how we reassign another value to a member in the array

In [34]: a[2,2] = 99 # the change above also affects the original array

In [35]: a
Out[35]:
array([[12, 34, 88],
       [54, 62, 18],
       [72, 84, 99]], dtype=int16)

In [36]: b
Out[36]: array([12, 34, 88], dtype=int16)
```

# Linear algebra

- Difference between a numpy.matrix and 2D numpy.ndarray

  - basic operations such as multiplications and transpose are included in NumPy for both matrix and ndarray types
  - numpy.matrix has a proper interface for matrix operations than numpy.ndarray
  - however, the numpy.matrix class does not add anything that cannot be achieved by using 2D numpy.ndarray objects
  - the implementation of this class resembles the one in Matlab
  - scipy.linalg operations can be used just as good to 2D numpy.ndarray objects as well as to numpy.matrix

# Linear algebra

- Difference between a numpy.matrix and 2D numpy.ndarray

the I and T class members serve as shortcuts for inverse and transpose respectively

the numpy.matrix class does not add anything that cannot be achieved by using 2D numpy.ndarray objects

```
In [45]: # Matrix vs Array:

In [46]: # 1. Matrix:

In [47]: import numpy as npy

In [48]: a = npy.mat('[12 34 41;52 64 72]') # create matrix 'a'

In [49]: a
Out[49]:
matrix([[12, 34, 41],
        [52, 64, 72]])

In [50]: type(a)
Out[50]: numpy.matrixlib.defmatrix.matrix

In [51]: a.I # inverse of matrix 'a'
Out[51]:
matrix([[-0.05885099,  0.03258602],
        [ 0.01490374, -0.00181295],
        [ 0.02925572, -0.00803395]])

In [52]: b = npy.mat('[84 92]') # create matrix 'b'

In [53]: b
Out[53]: matrix([[84, 92]])

In [54]: type(b)
Out[54]: numpy.matrixlib.defmatrix.matrix

In [55]: b.T # transpose of matrix 'b'
Out[55]:
matrix([[84],
        [92]])

In [56]: a.T*b.T # multiplication of two matrices
Out[56]:
matrix([[ 5792],
        [ 8744],
        [10068]])
```

# Linear algebra

- Difference between a numpy.matrix and 2D numpy.ndarray

scipy.linalg operations can be used just as good to 2D numpy.ndarray objects as well as to numpy.matrix

Note:
npy.mat and npy.matrix are the same. Try, using 'id'

```
In [57]: # Matrix vs Array:

In [58]: # 2. Array:

In [59]: import numpy as npy

In [60]: from scipy import linalg

In [61]: c = npy.array([[12,34,41],[52,64,72],[84,91,98]]) # create array 'c'

In [62]: c
Out[62]:
array([[12, 34, 41],
       [52, 64, 72],
       [84, 91, 98]])

In [63]: type(c)
Out[63]: numpy.ndarray

In [64]: linalg.inv(c)   # calculate the inverse of a matrix
Out[64]:
array([[-0.10752688,  0.15322581, -0.06758833],
       [ 0.3655914 , -0.87096774,  0.48694316],
       [-0.24731183,  0.67741935, -0.38402458]])

In [65]: d2 = npy.array([[2,12,28]]) # create 2D array

In [66]: d2
Out[66]: array([[ 2, 12, 28]])

In [67]: type(d2)
Out[67]: numpy.ndarray
```

# Linear algebra

- Difference between a numpy.matrix and 2D numpy.ndarray

scipy.linalg operations can be used just as good to 2D numpy.ndarray objects as well as to numpy.matrix

```
In [68]: c*d2 # this is not matrix multiplication
Out[68]:
array([[  24,  408, 1148],
       [ 104,  768, 2016],
       [ 168, 1092, 2744]])

In [69]: c.dot(d2.T) # this is matrix multiplication using dot product function
Out[69]:
array([[1580],
       [2888],
       [4004]])

In [70]: d1 = npy.array([2,12,28]) # this is 1D array

In [71]: d1
Out[71]: array([ 2, 12, 28])

In [72]: type(d1)
Out[72]: numpy.ndarray

In [73]: d1.T  # this is not a matrix transpose
Out[73]: array([ 2, 12, 28])

In [74]: c.dot(d1)  # dot product of two arrays
Out[74]: array([1580, 2888, 4004])
```

# Linear algebra

- Finding the inverse of an array (matrix)

matrix ‘e’ has its inverse matrix ‘f’ such that e*f=I -> I is the identity matrix that has main diagonal with ones

we can then say that: f=e−1

the matrix inverse of the NumPy array ‘e’ is obtained in two ways:

a) using the Scipy linalg.inv, or

b) using e.I when ‘e’ is a matrix so cast it like this: npy.mat(e).I

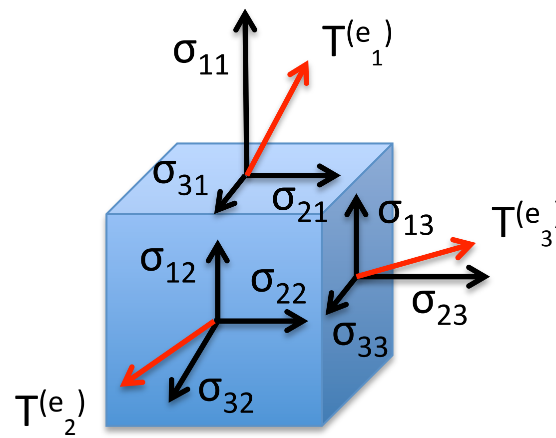
```
In [75]: # Finding the inverse of an array:

In [76]: import numpy as npy

In [77]: from scipy import linalg

In [78]: # Find the inverse of an array:

In [79]: # | 7  5 -4 |

In [80]: # | 4 -9  6 |

In [81]: # | 3  8  2 |

In [82]: e = npy.array([[7,5,-4],[4,-9,6],[3,8,2]]) # left hand-side matrix 'e':

In [83]: e
Out[83]:
array([[ 7,  5, -4],
       [ 4, -9,  6],
       [ 3,  8,  2]])

In [84]: linalg.inv(e) # finding the inverse of an array/matrix
Out[84]:
array([[ 0.10185185,  0.06481481,  0.00925926],
       [-0.0154321 , -0.04012346,  0.08950617],
       [-0.09104938,  0.0632716 ,  0.12808642]])
```

# Linear algebra

- Tensor is an objects describing the linear relationship among scalars, vectors and other tensors

Example:

a 2nd order tensor of a 3-dimensional space represent the matrix:




$$\sigma = [T^{(e_1)} T^{(e_2)} T^{(e_3)}] \qquad \text{or} \qquad \sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix}$$

where, the columns are the forces $e_n$ depicted on the 3 faces of the cube ($e_1$, $e_2$, $e_3$)

# Linear algebra

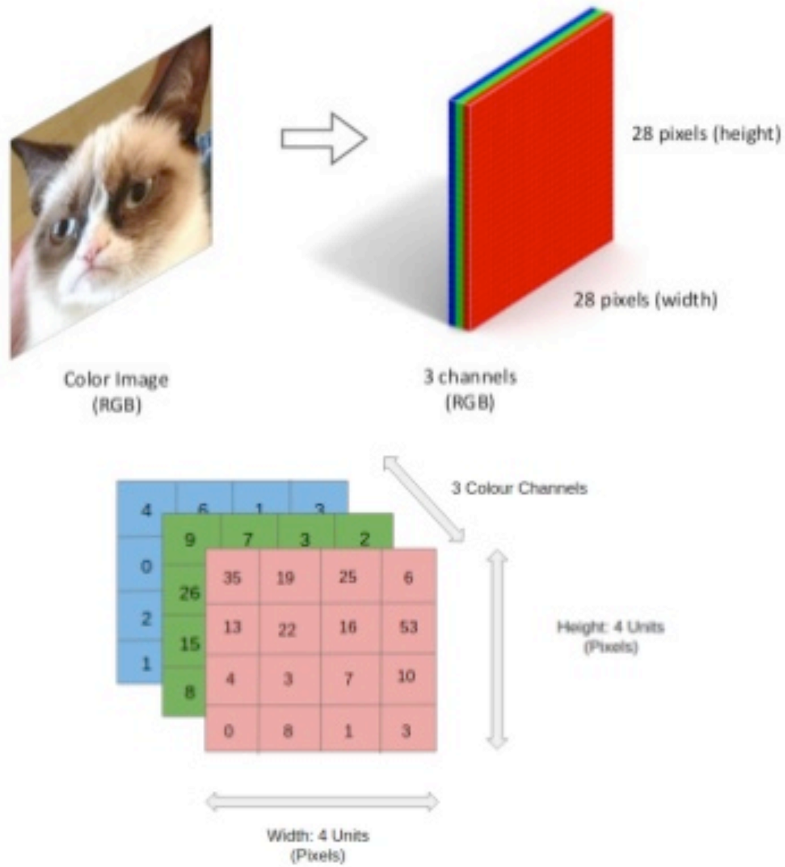
- Tensor is an objects describing the linear relationship among scalars, vectors and other tensors
  - A $0^{th}$ order tensor can be represented by a scalar
  - A $1^{st}$ order tensor can be represented by an array (vector)
  - A $2^{nd}$ order tensor can be represented by a matrix
  - A $3^{rd}$ order tensor can be represented as a 3-dimensional array of numbers
  - However tensor represents more than just an arrangement of components:
    - tensor shows how the array transforms upon a change of its basis
    - tensor is an ndarray satisfying a particular transformation law

# Linear algebra

- Rank of a matrix is the maximum number of linearly independent column (or row) vectors in the matrix, denoted as: *rk(A)* or *rank(A)*
  - The rank *R* of a tensor is independent of the number of dimensions *N* of the underlying space
  - Rank-0 is a scalar – $N^0 = 1$
  - Rank-1 is an array (vector) – $N^1 = N$
  - Rank-2 is a matrix – $N^2 = N \times N$ *aka dyad, dyadic*
  - Rank-3 is a 3-darray – $N^3 = N \times N \times N$ *aka triad*
  - Rank-4 is a 4-darray – $N^4 = N \times N \times N \times N$ *aka tetrad*
  - *etc*

# Linear algebra

color image is 3rd-order tensor



# Linear algebra

- Rank of a matrix is the maximum number of linearly independent column (or row) vectors in the matrix, denoted as: *rk(A)* or *rank(A)*

  - How to find the rank:
    - They are generally: 0,1,2 and 3:

      Rank(A) = 0 when matrix is null

      Rank(A) = 1 when every sub-matrix of A is singular or $\det(A_n) = 0$

      Rank(A) = 2 when A is singular or $|A| = 0$, **and** at least one of its sub-matrix is $|A_1| \neq 0$

      Rank(A) = 3 when A is non-singular or $|A| \neq 0$

Example:

$$A = \begin{bmatrix} -1 & -1 & 0 \\ 4 & 2 & 2 \\ 3 & 1 & 2 \end{bmatrix},$$

1) det(A) = (-1)(2*2-2*1)-(-1)(4*2-2*3)+(0)(4*1-2*3) = -2 +2 -0 = 0

   => $|A| = 0$ is singular

2) $\det(A_1) = 4*1-2*3 = 4-6 = -2$ -- > *$A_1$ is a sub-matrix of A*

   => $|A_1| \neq 0$ and is non-singular

Therefore Rank(A) = 2

# Linear algebra

- Rank of a matrix is the maximum number of linearly independent column (or row) vectors in the matrix, denoted as: *rk(A)* or *rank(A)*
  - How to find the rank:
    - They are generally: 0,1,2 and 3:

      Rank(A) = 0 when matrix is null

      Rank(A) = 1 when every sub-matrix of A is singular or $\det(A_n) = 0$

      Rank(A) = 2 when A is singular or $|A| = 0$, **and** at least one of its sub-matrix is $|A_1| \neq 0$

      Rank(A) = 3 when A is non-singular or $|A| \neq 0$

Example:

$$A = \begin{bmatrix} -1 & -1 & 0 \\ 4 & 2 & 2 \\ 3 & 1 & 2 \end{bmatrix}$$

Python

```
In [1]: from numpy import linalg as lin

In [2]: from numpy import matrix as mt

In [3]: A=mt([[-1,-1,0],[4,2,2],[3,1,2]])

In [4]: A
Out[4]:
matrix([[-1, -1,  0],
        [ 4,  2,  2],
        [ 3,  1,  2]])

In [5]: lin.matrix_rank(A)
Out[5]: 2
```

# NumPy data type objects

- Data type objects
  - NymPy supports much larger variety of types than what the standard Python implementation does:

| Number type | Data type | Description |
|---|---|---|
| Booleans | bool, bool8, bool_ | Boolean (True or False) stored as a byte – 8 bits |
| Integers | byte | compatible: C char – 8 bits |
| | short | compatible: C short – 16 bits |
| | int, int0, int_ | Default integer type (same as C `long`; normally either `int32` or `int64`) – 64 bits |
| | longlong | compatible: C long long – 64 bits |
| | intc | Identical to C `int` – 32 bits |
| | intp | Integer used for indexing (same as C `size_t`) – 64 bits |
| | int8 | Byte (-128 to 127) – 8 bits |
| | int16 | Integer (-32768 to 32767) – 16 bits |
| | int32 | Integer (-2147483648 to 2147483647) – 32 bits |
| | int64 | Integer (-9223372036854775808 to 9223372036854775807) – 64 bits |
| Unsigned integers | uint, uint0 | Python int compatible, unsigned – 64 bits |
| | ubyte | compatible: C unsigned char, unsigned – 8 bits |
| | ushort | compatible: C unsigned short, unsigned – 16 bits |
| | ulonglong | compatible: C long long, unsigned – 64 bits |
| | uintp | large enough to fit a pointer – 64 bits |
| | uintc | compatible: C unsigned int – 32 bits |
| | uint8 | Unsigned integer (0 to 255) – 8 bits |
| | uint16 | Unsigned integer (0 to 65535) – 16 bits |
| | uint32 | Unsigned integer (0 to 4294967295) – 32 bits |
| | uint64 | Unsigned integer (0 to 18446744073709551615) – 64 bits |

# NumPy data type objects

- Data type objects
  - NymPy supports much larger variety of types than what the standard Python implementation does:

| Number type | Data type | Description |
|---|---|---|
| Floating-point numbers | half | compatible: C short – 16 bits |
| | single | compatible: C float – 32 bits |
| | double | compatible: C double – 64 bits |
| | longfloat | compatible: C long float – 128 bits |
| | float_ | Shorthand for `float64` – 64 bits |
| | float16 | Half precision float: sign bit, 5 bits exponent, 10 bits mantissa |
| | float32 | Single precision float: sign bit, 8 bits exponent, 23 bits mantissa |
| | float64 | Double precision float: sign bit, 11 bits exponent, 52 bits mantissa |
| | float128 | 128 bits |
| Complex floating-point numbers | csingle | 64 bits |
| | complex, complex_ | Shorthand for `complex128` – 128 bits |
| | complex64 | Complex number, represented by two 32-bit floats (real and imaginary components) |
| | complex128 | Complex number, represented by two 64-bit floats (real and imaginary components) |
| | complex256 | two 256 bit floats |

  - To check how many bits each type occupies, use one of these notations:
    1) (np.dtype(np.<type>).itemsize)*8
    2) np.<type>().itemsize*8

# NumPy data type objects

- Data type objects
  - the difference between signed and unsigned integers and long type variables is:
    - the signed and unsigned types are of the same size
    - the signed can represent equal amount of values around the '0' thus representing equal amount of positive and negative numbers
    - the unsigned can not represent any negative numbers, but can represent double the amount of total positive numbers as compared to the signed type
    - for 32-bit int we have:

      int: –2147483648 to 2147483647

      uint: 0 to 4294967295
    - for 64-bit long we have:

      long: -9223372036854775808 to 9223372036854775807

      ulong: 0 to 18446744073709551615

# TensorFlow

- Below is the full list of data types available in TensorFlow:

```
@tf_export("DType")
class DType(object):
  """Represents the type of the elements in a `Tensor`.

  The following `DType` objects are defined:

  * `tf.float16`: 16-bit half-precision floating-point.
  * `tf.float32`: 32-bit single-precision floating-point.
  * `tf.float64`: 64-bit double-precision floating-point.
  * `tf.bfloat16`: 16-bit truncated floating-point.
  * `tf.complex64`: 64-bit single-precision complex.
  * `tf.complex128`: 128-bit double-precision complex.
  * `tf.int8`: 8-bit signed integer.
  * `tf.uint8`: 8-bit unsigned integer.
  * `tf.uint16`: 16-bit unsigned integer.
  * `tf.uint32`: 32-bit unsigned integer.
  * `tf.uint64`: 64-bit unsigned integer.
  * `tf.int16`: 16-bit signed integer.
  * `tf.int32`: 32-bit signed integer.
  * `tf.int64`: 64-bit signed integer.
  * `tf.bool`: Boolean.
  * `tf.string`: String.
  * `tf.qint8`: Quantized 8-bit signed integer.
  * `tf.quint8`: Quantized 8-bit unsigned integer.
  * `tf.qint16`: Quantized 16-bit signed integer.
  * `tf.quint16`: Quantized 16-bit unsigned integer.
  * `tf.qint32`: Quantized 32-bit signed integer.
  * `tf.resource`: Handle to a mutable resource.
  * `tf.variant`: Values of arbitrary types.
```

# TensorFlow

- Examples:

```
In [1]: import tensorflow as tf

In [2]: import numpy as np

In [3]: sess = tf.Session()

In [4]: a = 92

In [5]: type(a)
Out[5]: int

In [6]: a
Out[6]: 92

In [7]: a = tf.convert_to_tensor(a, dtype = tf.float16)

In [8]: type(a)
Out[8]: tensorflow.python.framework.ops.Tensor
```

```
In [9]: a
Out[9]: <tf.Tensor 'Const:0' shape=() dtype=float16>

In [10]: sess.run(a)
Out[10]: 92.0

In [11]: type(sess.run(a))
Out[11]: numpy.float16

In [12]: a = sess.run(a)

In [13]: type(a)
Out[13]: numpy.float16

In [14]: a
Out[14]: 92.0

In [15]: a = tf.cast(a, tf.float32)

In [16]: type(a)
Out[16]: tensorflow.python.framework.ops.Tensor
```

```
In [22]: a = tf.cast(a, np.float64)

In [23]: type(a)
Out[23]: tensorflow.python.framework.ops.Tensor

In [24]: type(sess.run(a))
Out[24]: numpy.float64
In [27]: a = np.int16(sess.run(a))

In [28]: type(a)
Out[28]: numpy.int16

In [29]: a
Out[29]: 92
```

... try it in class

# TensorFlow

- Few extra notes:
  - Graphs save computation time
  - Graphs break computation into small pieces to facilitates auto-differentiation
  - Graphs handle distributed computation, that is they spread the work across multiple CPUs, GPUs, or devices
  - Many machine learning models are taught and visualized as directed graphs
  - Nodes in the graph are called **ops** (short for **operations**)
  - An **op** takes zero or more Tensors, performs some computation, and produces zero or more Tensors
  - **Sessions** gives us the **environment** to perform **operations** on our **tensor data**

# TensorFlow

- Few extra notes on types:

  - Using Python types to specify Tensor objects is quick and easy, and it is useful for prototyping ideas
  - However, there is an important and unfortunate downside to doing it this way:

    - TensorFlow has a plethora of data types, but basic Python types lack the ability to explicitly state what kind of data type we'd like to use!

    - Instead, TensorFlow has to infer which data type it was meant

    - TensorFlow is tightly integrated with NumPy, the scientific computing package designed for manipulating ndarrays

# TensorFlow

- Few extra notes on types:

1. In TensorFlow, all data passed from node to node are Tensor objects

2. TensorFlow *Operations* are able to look at standard Python types, such as integers and strings, and automatically convert them into tensors

3. There are a variety of ways to create Tensor objects manually (that is, without reading it in from an external data source)

4. … so let's see them next: …

# TensorFlow

- Few extra notes on types:

  - TensorFlow can take in Python numbers, booleans, strings, or lists of any of the above
  - Single values will be converted to a 0-D Tensor (or scalar)
  - Lists of values will be converted to a 1-D Tensor (vector)
  - Lists of lists of values will be converted to a 2-D Tensor (matrix), and so on

# TensorFlow

- Here is a small chart showcasing this:

```
t_0 = 50                                  # Treated as 0-D Tensor,
or "scalar"

t_1 = [b"apple", b"peach", b"grape"] # Treated as 1-D Tensor,
or "vector"

t_2 = [[True, False, False],          # Treated as 2-D Tensor,
or "matrix"
       [False, False, True],
       [False, True, False]]

t_3 = [[ [0, 0], [0, 1], [0, 2] ],    # Treated as 3-D Tensor
       [ [1, 0], [1, 1], [1, 2] ],
       [ [2, 0], [2, 1], [2, 2] ]]
...
```

# TensorFlow

- Few extra notes on types (cont.):
  - TensorFlow's data types are based on those from NumPy
  - Tensors are just a superset of matrices!
  - In fact, the statement np.int32 == tf.int32 returns True!
  - Any NumPy array can be passed into any TensorFlow Op, and the beauty is that you can easily specify the data type you need with minimal effort

  The String data types problem:

  - For numeric and boolean types, TensorFlow and NumPy dtypes match perfectly well
  - However, tf.string does not have an exact match in NumPy due to the way NumPy handles strings
  - That said, TensorFlow can import string arrays from NumPy perfectly fine - just don't specify a dtype in NumPy!

# TensorFlow

- Unlike Python, where a string can be treated as a list of characters, TensorFlow's tf.strings are indivisible values

  Example:

  ‘x’ is a Tensor with shape (2,) and each element inside is a variable length string

  x = tf.constant(["This is a string", "This is another string"])

- TensorFlow provides the tf.decode_raw operator that takes tf.string tensor as input, but can decode the string into any other primitive data type

- To interpret the string as a tensor of characters, you can do:

```
In [13]: x = tf.constant("This is string")

In [14]: x = tf.decode_raw(x, tf.uint8)

In [15]: y = x[:4]

In [16]: sess = tf.InteractiveSession()

In [17]: print(y.eval())
[ 84 104 105 115]
```

# TensorFlow

- Few extra notes on types (cont.):
  - You can use the functionality of the numpy library both before and after running your graph, as the tensors returned from Session.run **<u>are</u>** NumPy arrays
  - Here's an example of how to create NumPy arrays, mirroring the previous example:

```
import numpy as np  # Don't forget to import NumPy!

# 0-D Tensor with 32-bit integer data type
t_0 = np.array(50, dtype=np.int32)

# 1-D Tensor with byte string data type
# Note: don't explicitly specify dtype when using strings in
NumPy
t_1 = np.array([b"apple", b"peach", b"grape"])

# 1-D Tensor with boolean data type
t_2 = np.array([[True, False, False],
                [False, False, True],
```

Source: TensorFlow for Machine Intelligence

# TensorFlow

- Few extra notes on types (cont.):

  - You can use the functionality of the numpy library both before and after running your graph, as the tensors returned from Session.run **are** NumPy arrays
  - Here's an example of how to create NumPy arrays, mirroring the previous example:

```
                [False, True, False]],
                dtype=np.bool)

# 3-D Tensor with 64-bit integer data type
t_3 = np.array([[ [0, 0], [0, 1], [0, 2] ],
                [ [1, 0], [1, 1], [1, 2] ],
                [ [2, 0], [2, 1], [2, 2] ]],
                dtype=np.int64)
...
```

# Tensor Shape

- The shape in TensorFlow terminology describes both:
  - the number of dimensions in a tensor as well as
  - the length of each dimension

- Tensor shapes can either be Python lists or tuples containing an ordered set of integers:

  - there are as many numbers in the list as there are dimensions, and each number describes the length of its corresponding dimension

    **Example:**

    the list [2, 3] describes the shape of a 2-D tensor of length 2 in its first dimension and length 3 in its second dimension

    * Note that either: - tuples (wrapped with parentheses () ) or
    - lists (wrapped with brackets [] ) can be used to define shapes

# Tensor Shape

- Some examples:

```
# Shapes that specify a 0-D Tensor (scalar)
# e.g. any single number: 7, 1, 3, 4, etc.
s_0_list = []
s_0_tuple = ()

# Shape that describes a vector of length 3
# e.g. [1, 2, 3]
s_1 = [3]
```

```
# Shape that describes a 3-by-2 matrix
# e.g [[1 ,2],
#      [3, 4],
#      [5, 6]]
s_2 = (3, 2)
```

Source: TensorFlow for Machine Intelligence

# Tensor Shape

- In addition to being able to specify fixed lengths to each dimension, we are also able assign a flexible length by passing in None as a dimension's value
- This will tell TensorFlow to allow a tensor of any shape
- That is, a tensor with any amount of dimensions and any length for each dimension:

```
# Shape for a vector of any length:
s_1_flex = [None]

# Shape for a matrix that is any amount of rows tall, and 3
columns wide:
s_2_flex = (None, 3)

# Shape of a 3-D Tensor with length 2 in its first dimension,
and variable-
# length in its second and third dimensions:
s_3_flex = [2, None, None]

# Shape that could be any Tensor
s_any = None
```

Source: TensorFlow for Machine Intelligence

# Tensor Shape

- If we ever need to figure out the shape of a tensor in the middle of our graph, we can use the tf.shape Op

- It simply takes in the Tensor object we'd like to find the shape for, and returns it as an int32 vector:

```
import tensorflow as tf

# ...create some sort of mystery tensor

# Find the shape of the mystery tensor
shape = tf.shape(mystery_tensor, name="mystery_shape")
```

** Remember that tf.shape, like any other Operation, doesn't run until it is executed inside of a Session*

# TensorFlow

- Examples:

```python
## Tensor Types
import tensorflow as tf
import numpy as np

# Define a 2x2 matrix in 3 different ways
m1 = [[1.0, 2.0], [3.0, 4.0]]                                  # <class 'list'>
m2 = np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32) # <class 'np.ndarray'>
m3 = tf.constant([[1.0, 2.0], [3.0, 4.0]])                     # <class 'tensorflow'>
print(type(m1))
print(type(m2))
print(type(m3))

# Create tensor objects out of various types
t1 = tf.convert_to_tensor(m1, dtype=tf.float32)           # <class 'tensorflow'>
t2 = tf.convert_to_tensor(m2, dtype=tf.float32)           # <class 'tensorflow'>
t3 = tf.convert_to_tensor(m3, dtype=tf.float32)           # <class 'tensorflow'>
print(type(t1))
print(type(t2))
print(type(t3))
```

Python

```
In [1]: (executing cell "Tensor Types" (line 2 of "types.py"))
<class 'list'>
<class 'numpy.ndarray'>
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

Python

```
In [9]: m3.op.name
Out[9]: 'Const'
```

... try it in class

# TensorFlow

- Before you can use a variable, it must be initialized
- Most high-level frameworks such as tf.contrib.slim, tf.estimator.Estimator and Keras automatically initialize variables
- If you are explicitly creating your own graphs and sessions, you must explicitly initialize the variables
- To initialize all trainable variables in one go, before training starts, call:

  ```
  session.run(tf.global_variables_initializer())
  ```

- To ask which variables have still not been initialized:

  ```
  session.run(my_variable.initializer)
  ```

# TensorFlow

- Examples:

```
## A simple matrix and the InteractiveSession:
import tensorflow as tf
sess = tf.InteractiveSession()

matrix = tf.constant([[5., 6.]])
negMatrix = tf.negative(matrix)

result = negMatrix.eval()
print(result)
sess.close()
```

```
[[-5. -6.]]
```

- **tf.InteractiveSession**:
    - The **<u>only difference</u>** with a regular `Session` is that an `InteractiveSession` installs itself as the default session on construction.
    - For example:

```
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# We can just use 'c.eval()' without passing 'sess'
print(c.eval())
sess.close()
```

… try it in class

# TensorFlow

- Examples:

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

# Create a boolean variable called `spike` to detect sudden 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 True
Spike False
Spike True
```

... try it in class

# TensorFlow

- Examples:

```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

- Examples:

```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

- Examples:

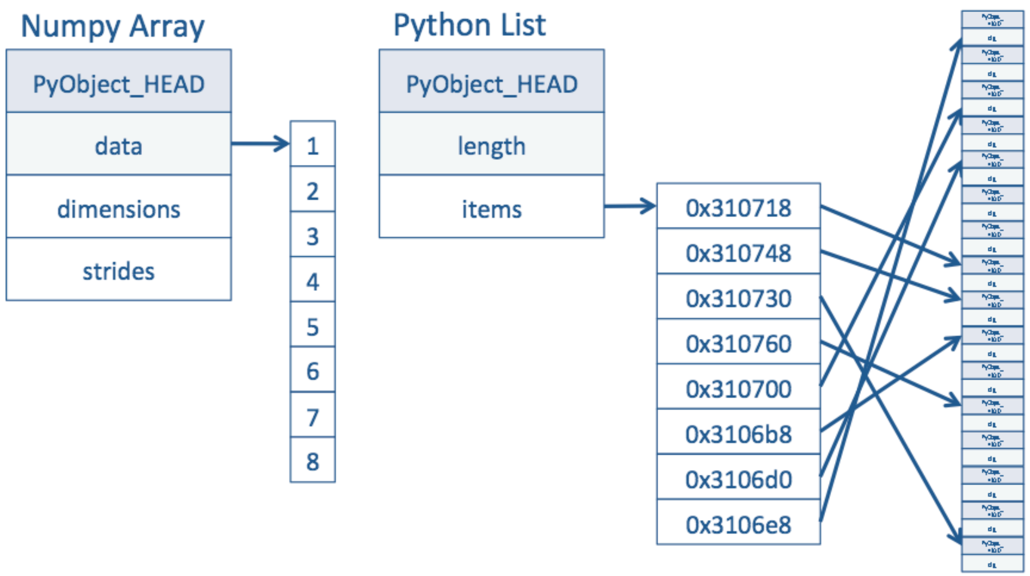
```python
## Log example:
import tensorflow as tf

# 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



# NumPy arrays recap

- NumPy arrays

  Example:

```
In [23]: a = np.array([[12, 34, 41], [54, 62, 18], [72, 84, 96]], np.int16)

In [24]: a
Out[24]:
array([[12, 34, 41],
       [54, 62, 18],
       [72, 84, 96]], dtype=int16)

In [25]: a.size
Out[25]: 9

In [26]: a.shape
Out[26]: (3, 3)

In [27]: type(a)
Out[27]: numpy.ndarray

In [28]: a.dtype
Out[28]: dtype('int16')

In [29]: a[2,2]   # this is how we index a particular elemnt in the array (#9)
Out[29]: 96

In [30]: b = a[0,:]

In [31]: b
Out[31]: array([12, 34, 41], dtype=int16)

In [32]: b.shape
Out[32]: (3,)

In [33]: b[2] = 88   # this is how we reassign another value to a member in the array

In [34]: a[2,2] = 99 # the change above also affects the original array

In [35]: a
Out[35]:
array([[12, 34, 88],
       [54, 62, 18],
       [72, 84, 99]], dtype=int16)

In [36]: b
Out[36]: array([12, 34, 88], dtype=int16)
```

… try it in class

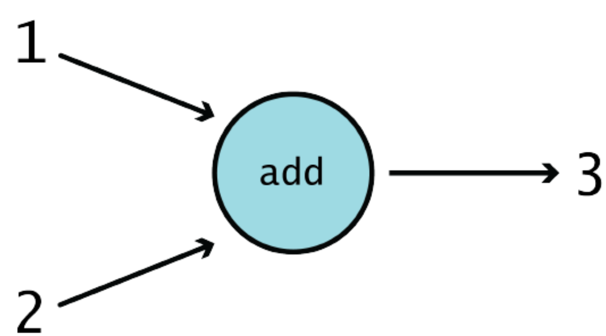
# TensorFlow

- Let’s discuss the basics of computation graphs without the context of Tensor-Flow

- This includes:
  - defining nodes
  - defining edges
  - dependencies
  - examples to illustrate key principles

# TensorFlow

- 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

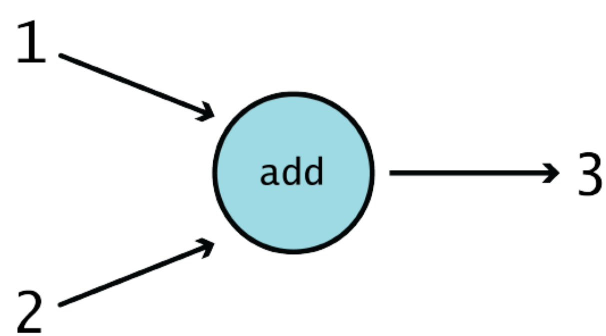



$$f(1,2) = 1 + 2 = 3$$

Source: TensorFlow for Machine Intelligence

# TensorFlow

- 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.

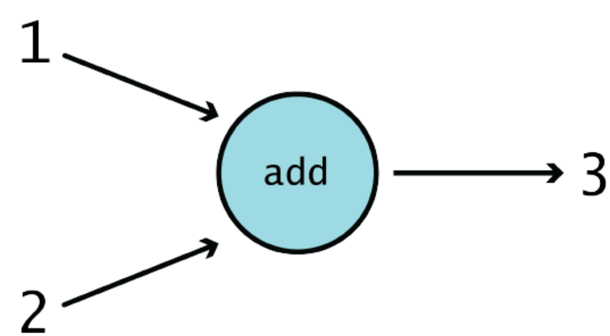



$$f(1,2) = 1 + 2 = 3$$

Source: TensorFlow for Machine Intelligence

# TensorFlow

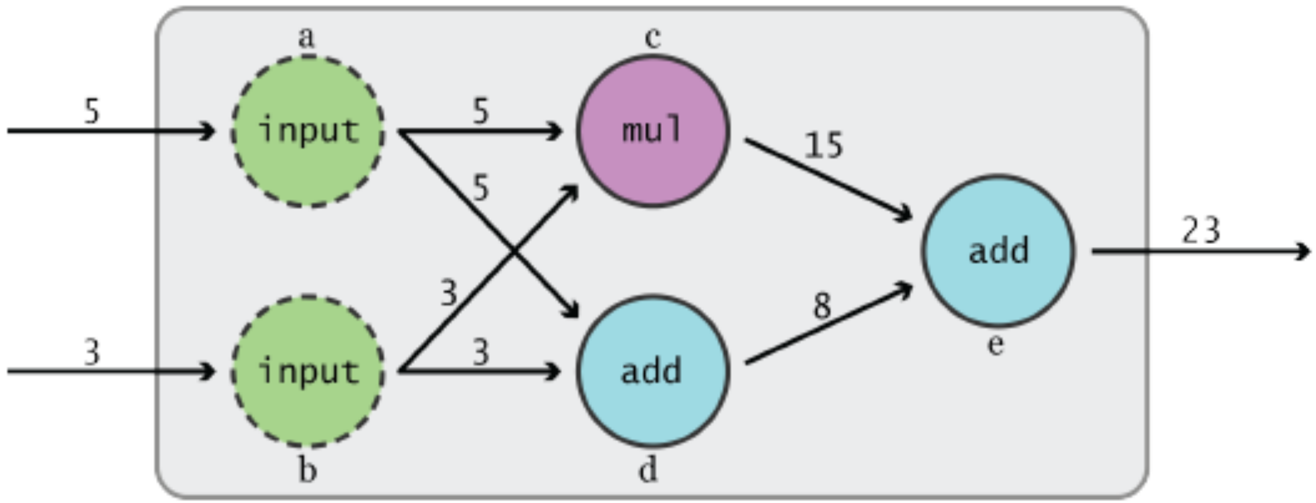
- Graph basics:
  - Edges: are the actual values that get passed to and from *Operations*, and are typically drawn as arrows
  - In the "add" example, the inputs 1 and 2 are both edges leading into the node, while the output 3 is an edge leading out of the node
  - We can think of edges as the link between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




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

Source: TensorFlow for Machine Intelligence

# TensorFlow




Source: TensorFlow for Machine Intelligence

# TensorFlow

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

$$a = input_1;\ b = input_2$$

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

$$e = c + d$$

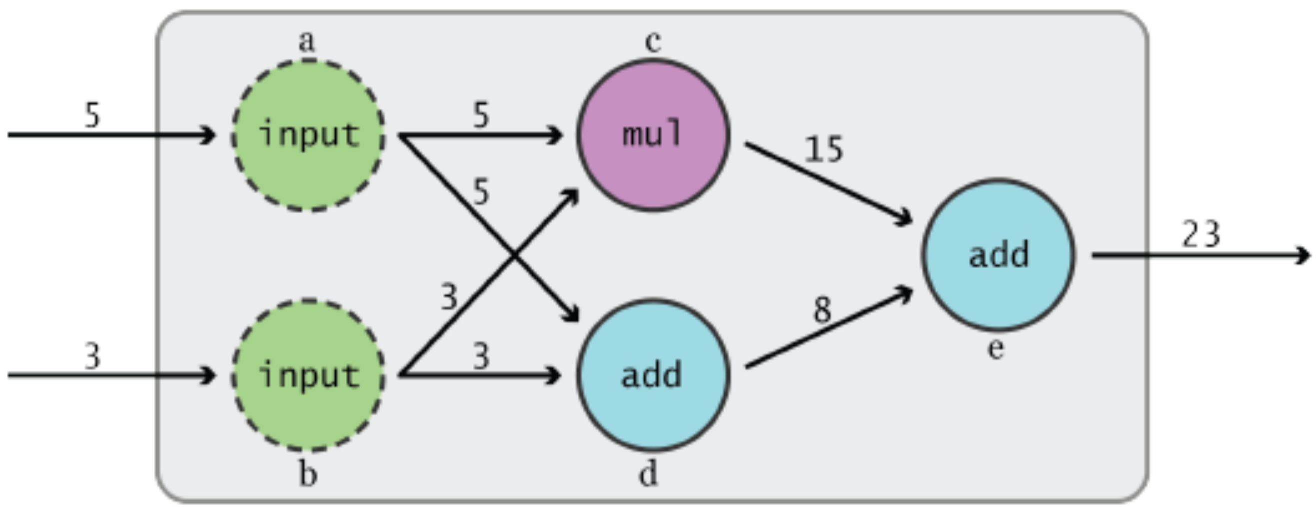
to solve $e$ for $a = 5$ and $b = 3$,

$$a = 5;\ b = 3$$

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

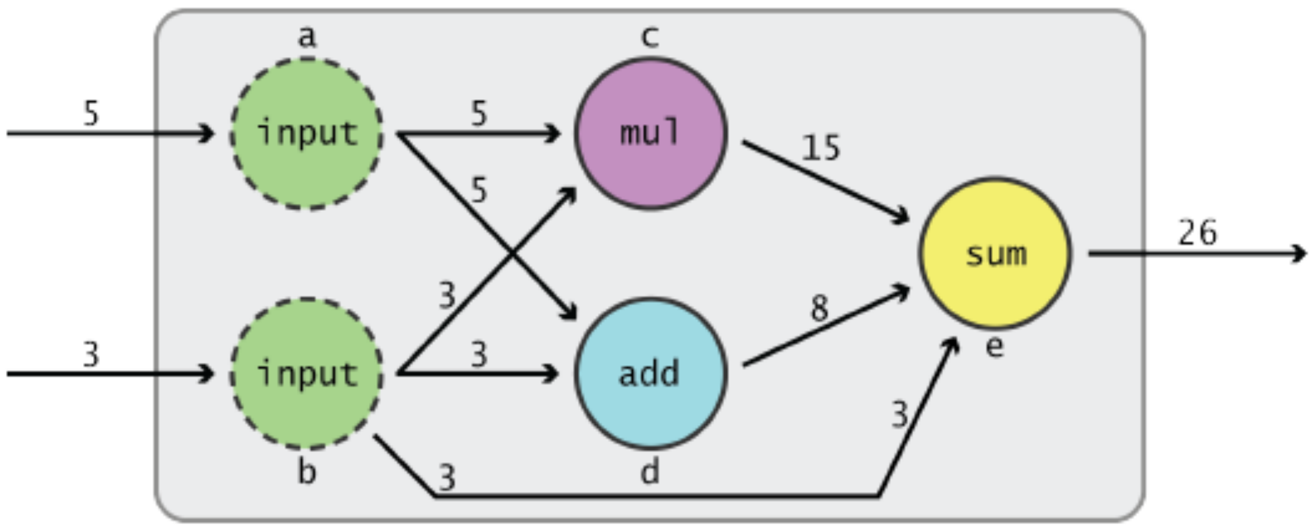
$$e = (5 \cdot 3) + (5 + 3)$$

$$e = 15 + 8 = 23$$




Source: TensorFlow for Machine Intelligence

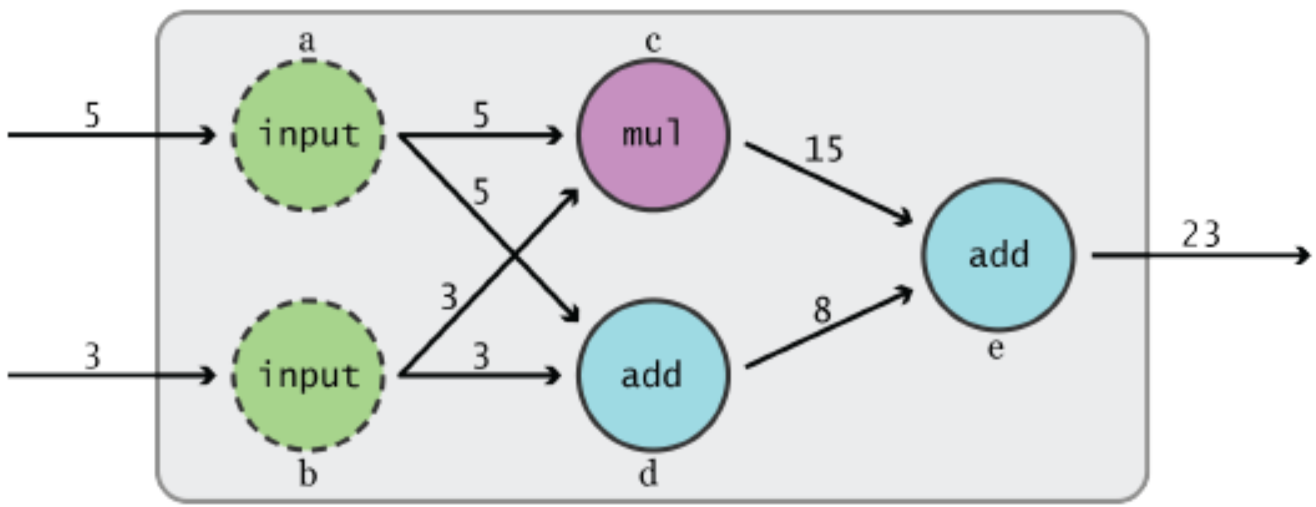
# TensorFlow




Source: TensorFlow for Machine Intelligence

# 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*

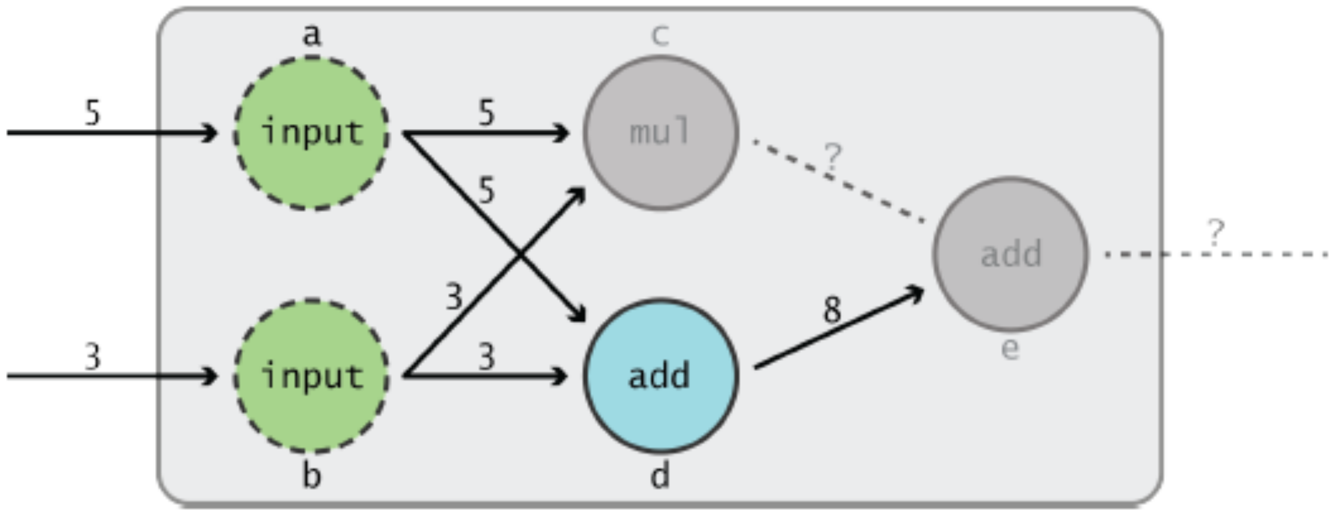



Source: TensorFlow for Machine Intelligence

# TensorFlow

- Graph basics:
  - Dependencies: …

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

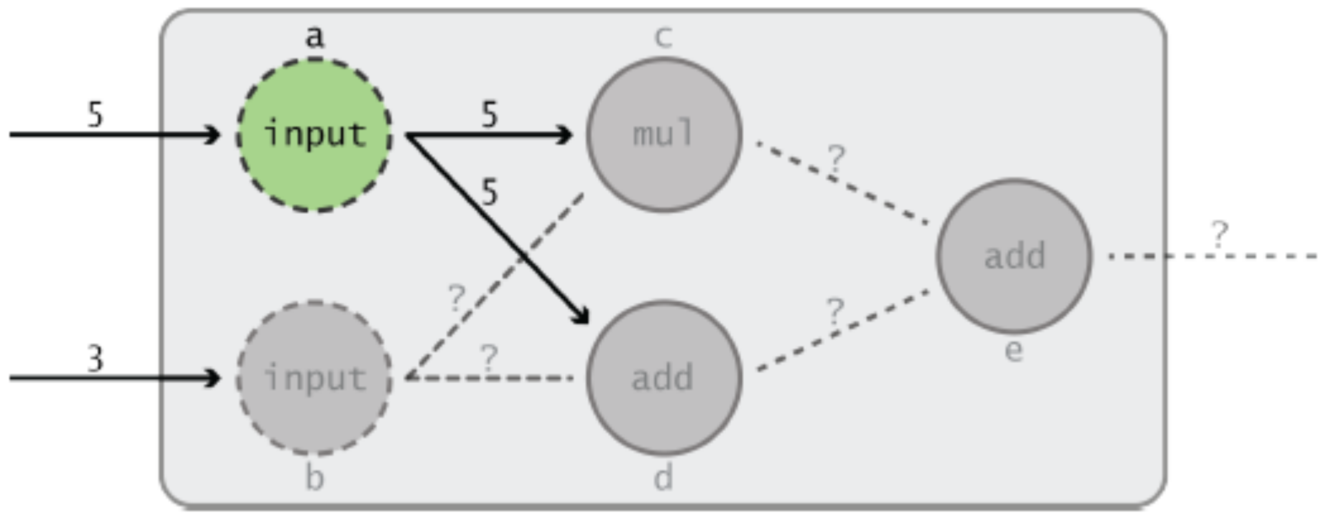



Source: TensorFlow for Machine Intelligence

# TensorFlow

- Graph basics:
  - Dependencies: …

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

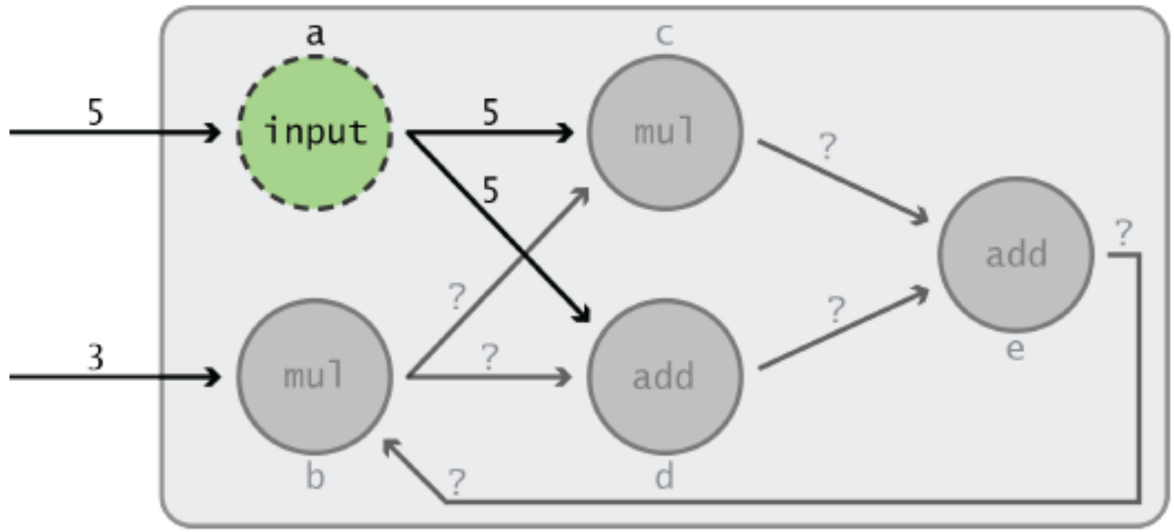



Source: TensorFlow for Machine Intelligence

# TensorFlow

- Graph basics:
  - Dependencies: …

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



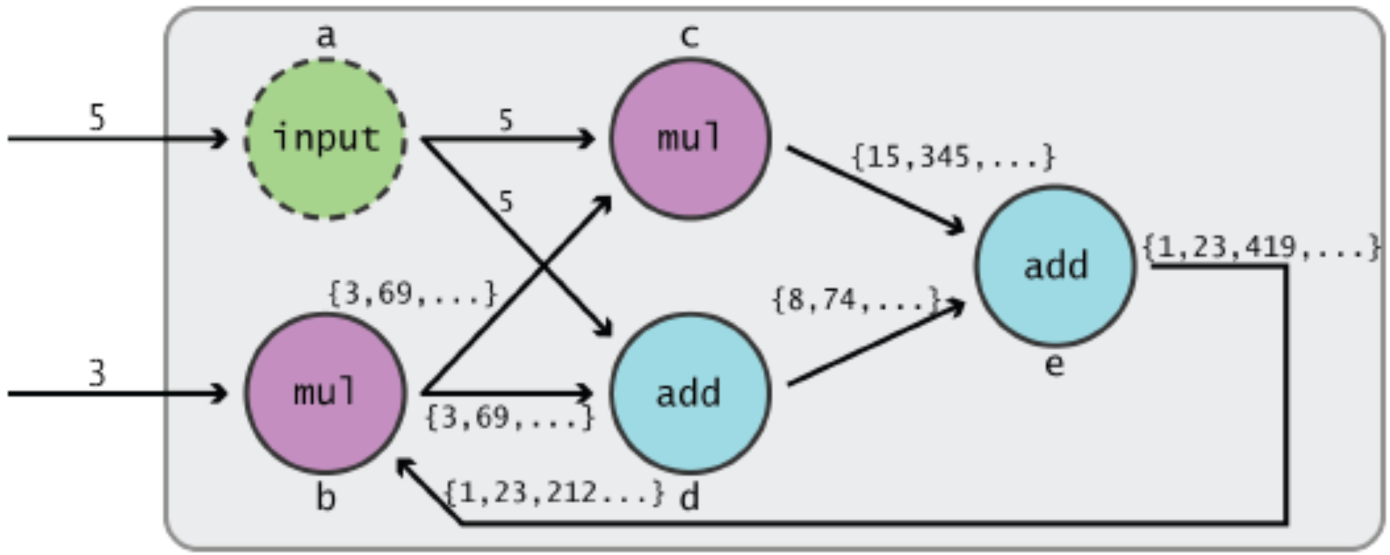

Source: TensorFlow for Machine Intelligence

# 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:

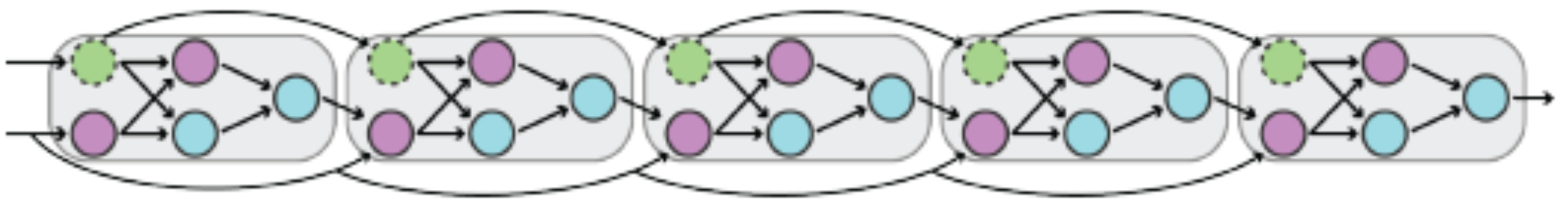



Source: TensorFlow for Machine Intelligence

# TensorFlow

- Graph basics:
    - Dependencies: …

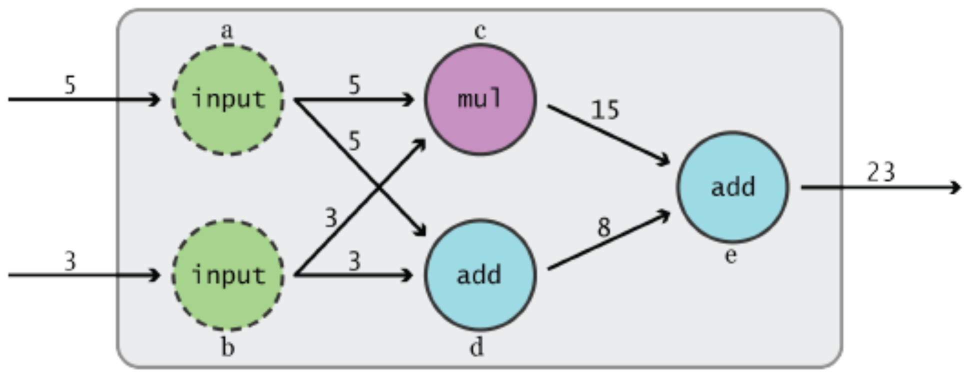
      By unrolling our graph like this, we can simulate useful cyclical dependencies while maintaining a deterministic computation.



Source: TensorFlow for Machine Intelligence

# TensorFlow

- Building our first graph in TensorFlow:




simple_graph.py

```
## 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")
```

Source: TensorFlow for Machine Intelligence

# TensorFlow

- 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 have to 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:

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)
```

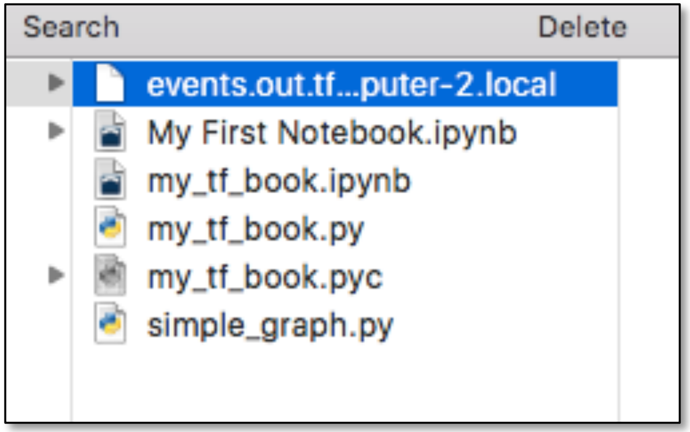
Docstring:
Writes `Summary` protocol
buffers to event files.

'/path/to/logs'

# TensorFlow

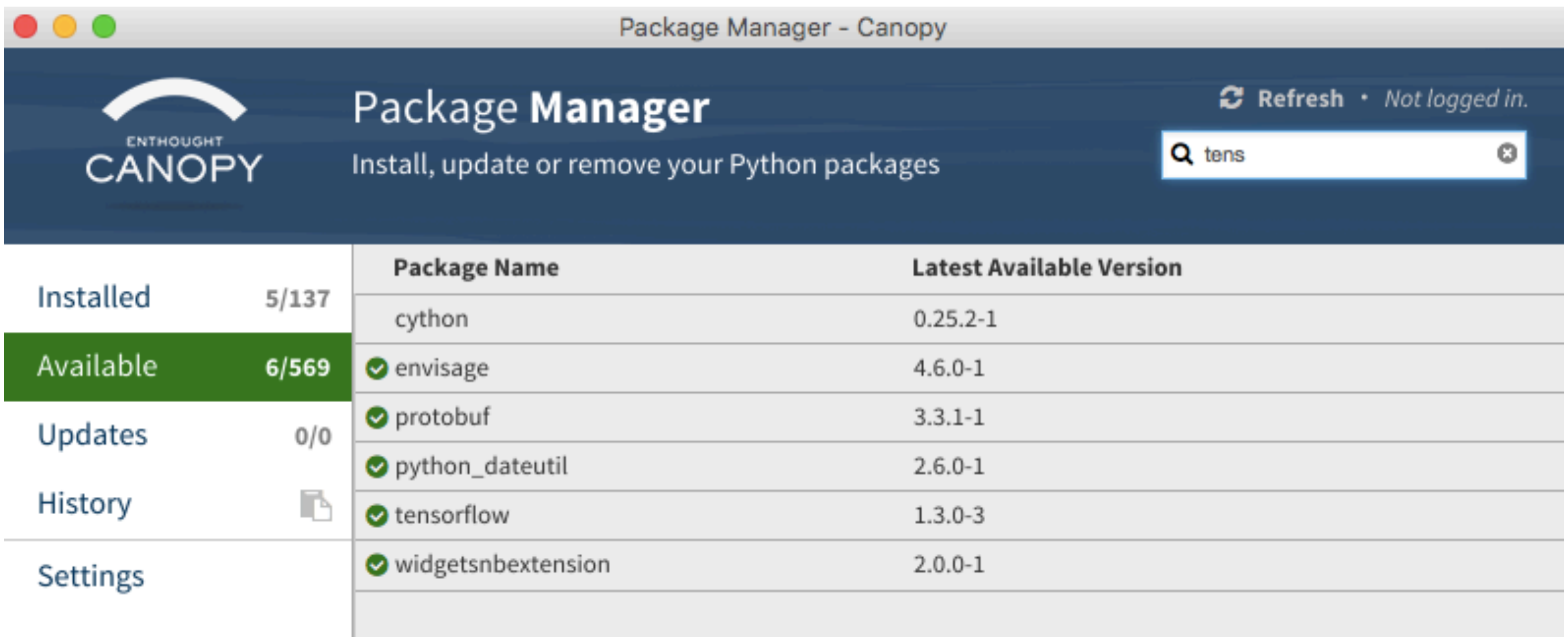
- Let's construct the actual graph using TensorBoard:

  - Once we run the previous code, a file with the session is generated in our current folder:



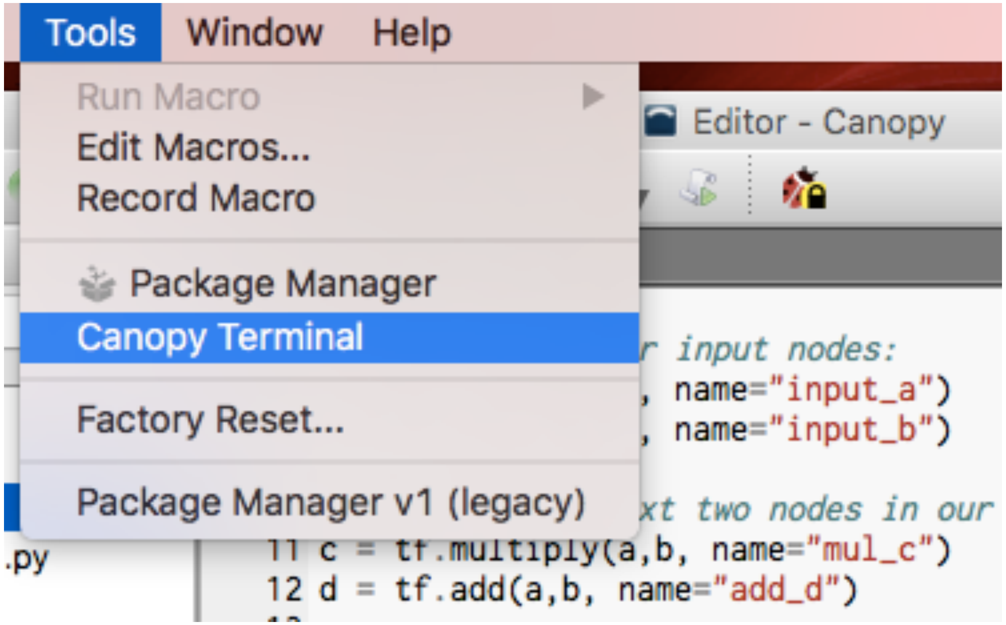
# TensorFlow

- Let’s construct the actual graph using TensorBoard:
  - Before we continues we need to check if we have TensorBoard installed in our system:



# TensorFlow

- 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

- Let’s construct the actual graph using TensorBoard:
  - We then type: pip list to check for installed packages:

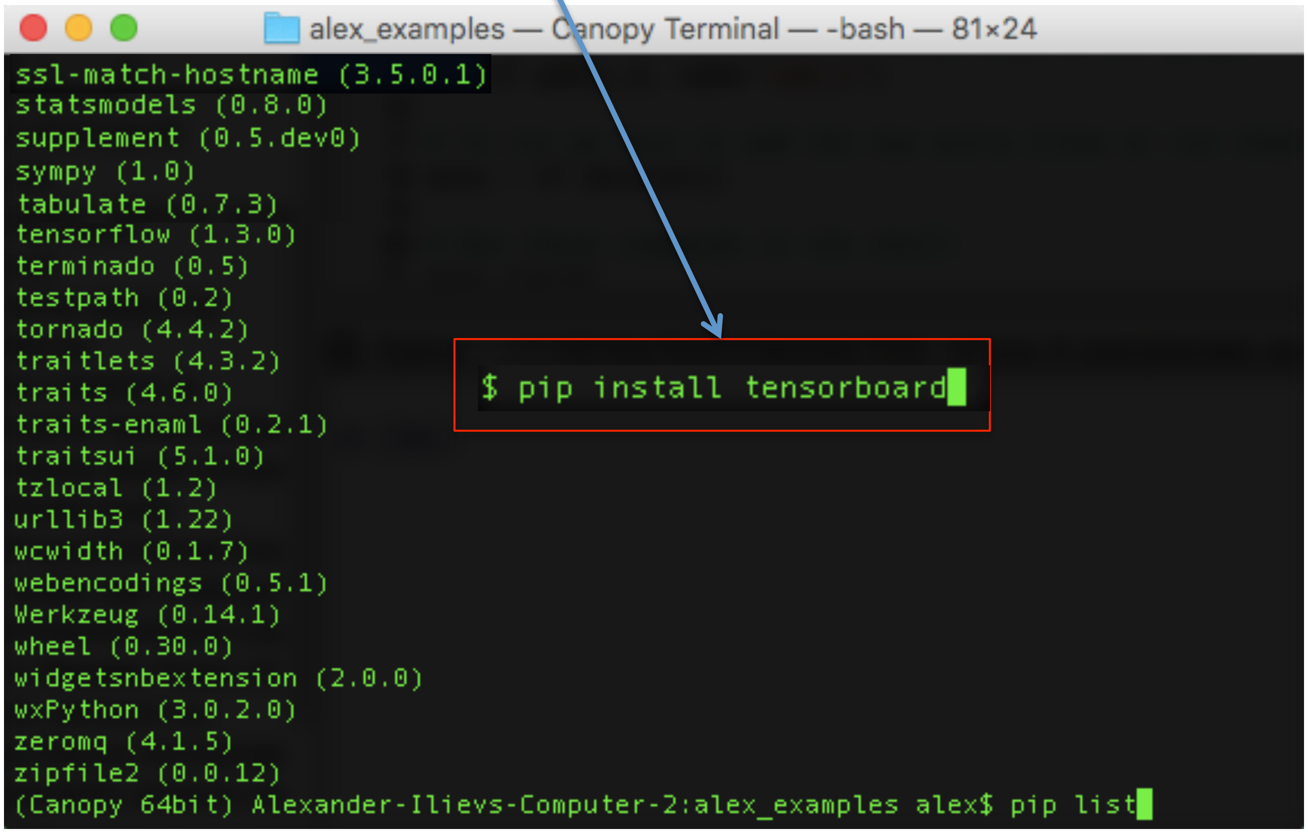
TensorBoard is not found in the list

alex_examples — Canopy Terminal — -bash — 81×24

```
ssl-match-hostname (3.5.0.1)
statsmodels (0.8.0)
supplement (0.5.dev0)
sympy (1.0)
tabulate (0.7.3)
tensorflow (1.3.0)
terminado (0.5)
testpath (0.2)
tornado (4.4.2)
traitlets (4.3.2)
traits (4.6.0)
traits-enaml (0.2.1)
traitsui (5.1.0)
tzlocal (1.2)
urllib3 (1.22)
wcwidth (0.1.7)
webencodings (0.5.1)
Werkzeug (0.14.1)
wheel (0.30.0)
widgetsnbextension (2.0.0)
wxPython (3.0.2.0)
zeromq (4.1.5)
zipfile2 (0.0.12)
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ pip list
```

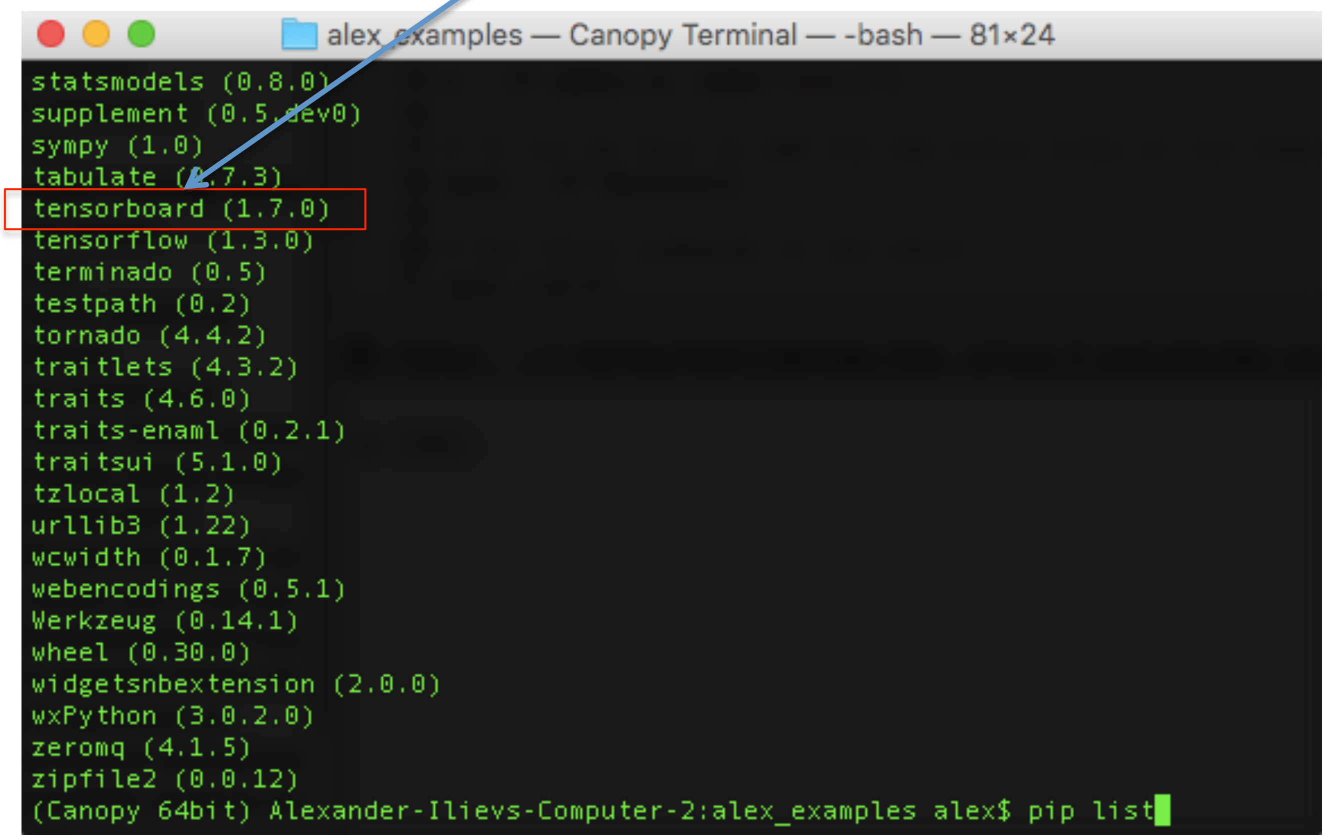
# TensorFlow

- Let's construct the actual graph using TensorBoard:
  - We then type: pip install tensorboard in the terminal



# TensorFlow

- Let’s construct the actual graph using TensorBoard:
  - We check again: tensorboard is now installed



# TensorFlow

- 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:

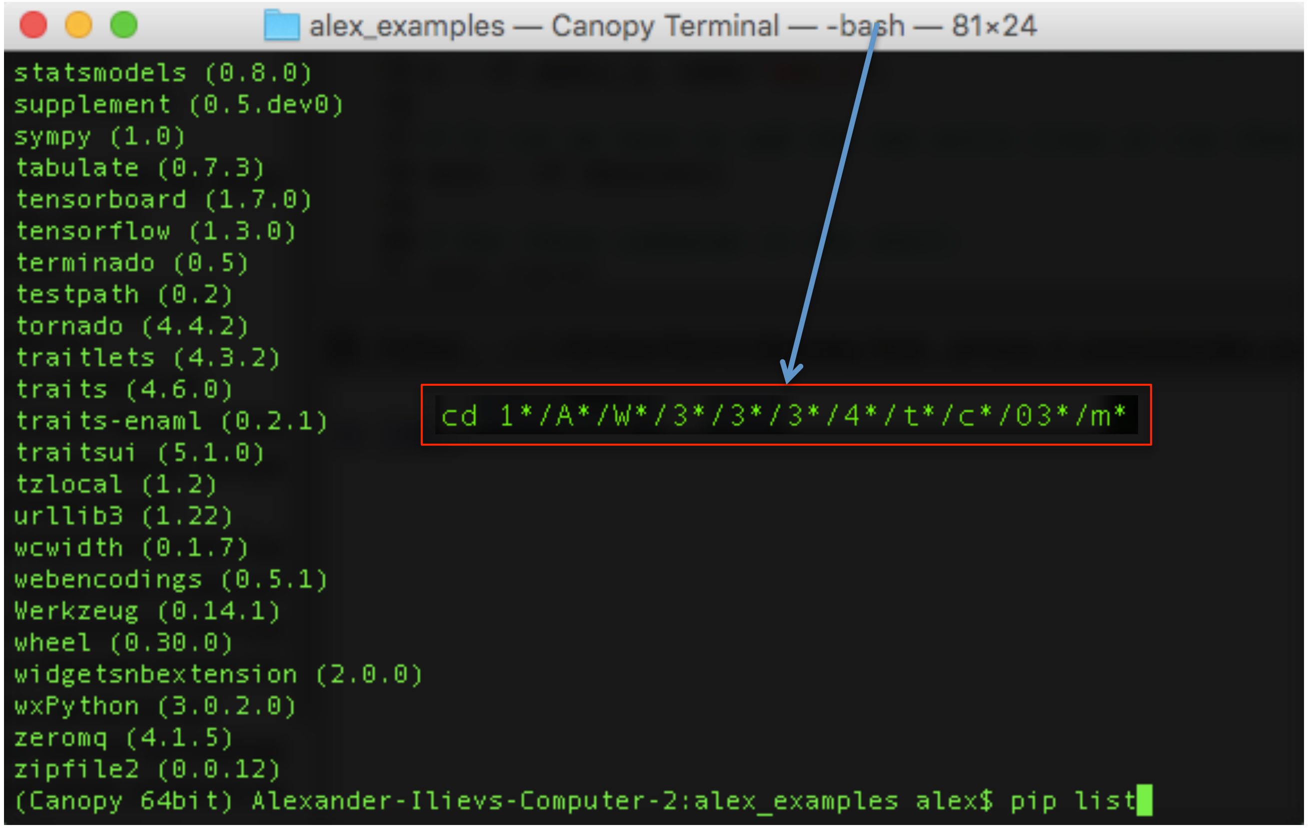
```
tensorboard                    1.6.0
tensorflow                     1.6.0
```

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

- Let’s construct the actual graph using TensorBoard:
    - We must be operating in the correct directory before we go on:



# TensorFlow

- Let's construct the actual graph using TensorBoard:

  - We must be operating in the correct directory before we go on:

```
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ pwd
/Users/alex/1.HD/Alex/Work/3.Berkeley Extension/3. final course material/3. Deep Learning using Python/code/more_tf_examples/alex_examples
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ ls -la
total 80
drwxr-xr-x  10 alex  501   340 Mar 31 15:34 .
drwxr-xr-x   9 alex  501   306 Mar  3 17:38 ..
-rw-r--r--@  1 alex  501  6148 Mar 31 15:34 .DS_Store
drwxr-xr-x   6 alex  501   204 Mar  3 16:42 .ipynb_checkpoints
-rw-r--r--   1 alex  501  1316 Mar  3 16:12 My First Notebook.ipynb
-rw-r--r--   1 alex  501  1139 Mar 31 15:32 events.out.tfevents.1522528323.Alexander-Ilievs-Computer-2.local
-rw-r--r--   1 alex  501  8698 Mar  4 13:05 my_tf_book.ipynb
-rw-r--r--   1 alex  501   251 Mar  3 17:25 my_tf_book.py
-rw-r--r--   1 alex  501   430 Mar  3 17:26 my_tf_book.pyc
-rw-r--r--   1 alex  501   607 Mar 31 14:43 simple_graph.py
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$
```

  - Then we make sure the graph summary file was created

# TensorFlow

- 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 ./
```

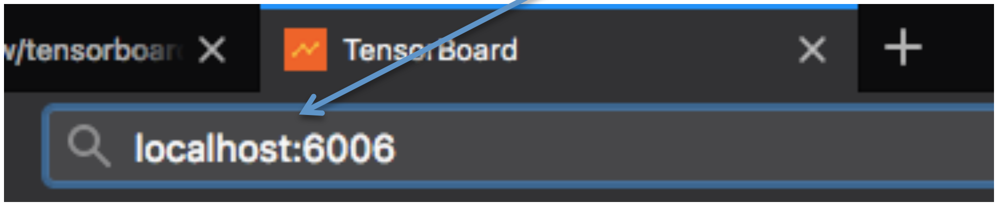
# TensorFlow

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

```
alex_examples — Canopy Terminal — tensorboard --logdir ./ --event_file events.out.tfevents.152252832...
[(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ ls
My First Notebook.ipynb                                          my_tf_book.py
events.out.tfevents.1522528323.Alexander-Ilievs-Computer-2.local      my_tf_book.pyc
my_tf_book.ipynb                                                 simple_graph.py
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ pwd
/Users/alex/1.HD/Alex/Work/3.Berkeley Extension/3. final course material/3. Deep Learning using Python
/code/more_tf_examples/alex_examples
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ ls -la
total 80
drwxr-xr-x  10 alex  501   340 Mar 31 15:34 .
drwxr-xr-x   9 alex  501   306 Mar  3 17:38 ..
-rw-r--r--@  1 alex  501  6148 Mar 31 15:34 .DS_Store
drwxr-xr-x   6 alex  501   204 Mar  3 16:42 .ipynb_checkpoints
-rw-r--r--   1 alex  501  1316 Mar  3 16:12 My First Notebook.ipynb
-rw-r--r--   1 alex  501  1139 Mar 31 15:32 events.out.tfevents.1522528323.Alexander-Ilievs-Computer-2
.local
-rw-r--r--   1 alex  501  8698 Mar  4 13:05 my_tf_book.ipynb
-rw-r--r--   1 alex  501   251 Mar  3 17:25 my_tf_book.py
-rw-r--r--   1 alex  501   430 Mar  3 17:26 my_tf_book.pyc
-rw-r--r--   1 alex  501   607 Mar 31 14:43 simple_graph.py
(Canopy 64bit) Alexander-Ilievs-Computer-2:alex_examples alex$ tensorboard --logdir ./ --event_file ev
ents.out.tfevents.1522528323.Alexander-Ilievs-Computer-2.local
TensorBoard 1.7.0 at http://Alexander-Ilievs-Computer-2.local:6006 (Press CTRL+C to quit)
W0331 15:43:15.306705 Thread-1 application.py:272] path /[[_dataImageSrc]] not found, sending 404
W0331 15:43:15.383185 Thread-1 application.py:272] path /[[_imageURL]] not found, sending 404
```

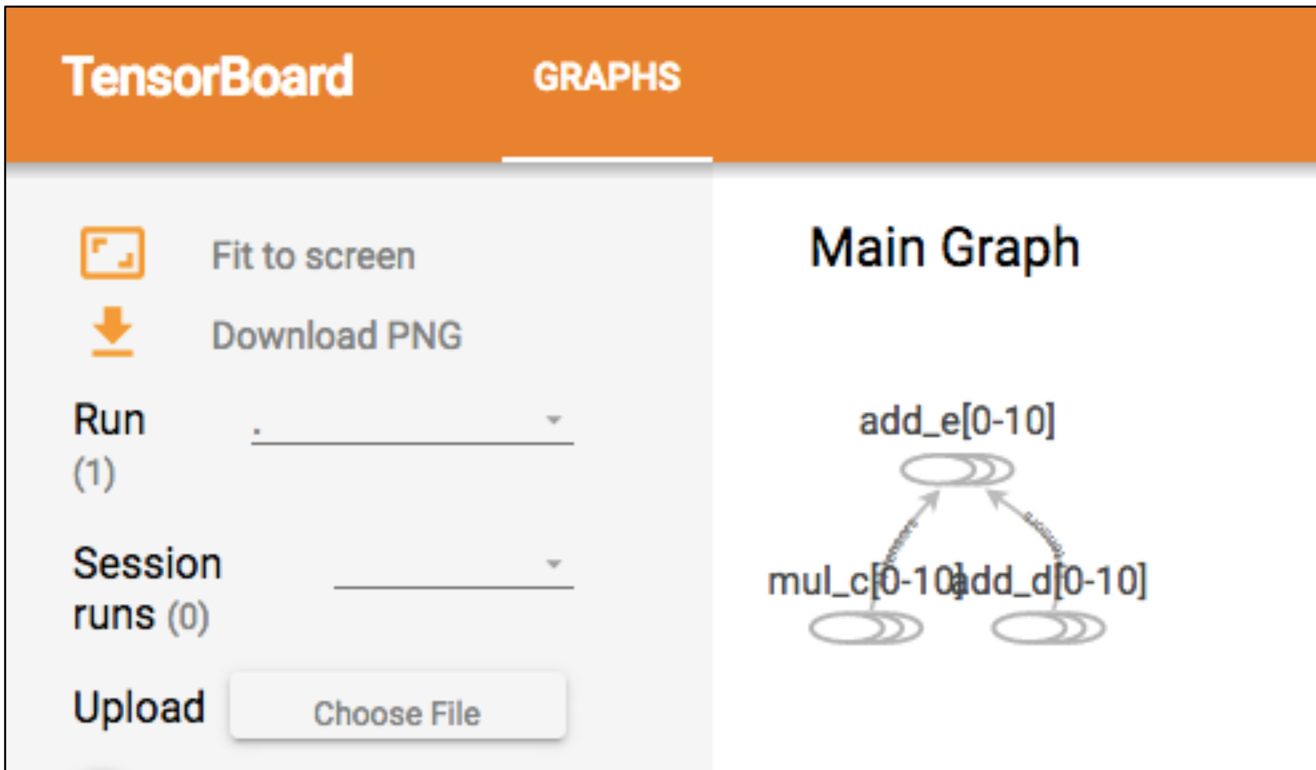
# TensorFlow

- Let’s construct the actual graph using TensorBoard:
    - Next, we open our browser and take a look at the graph by typing:



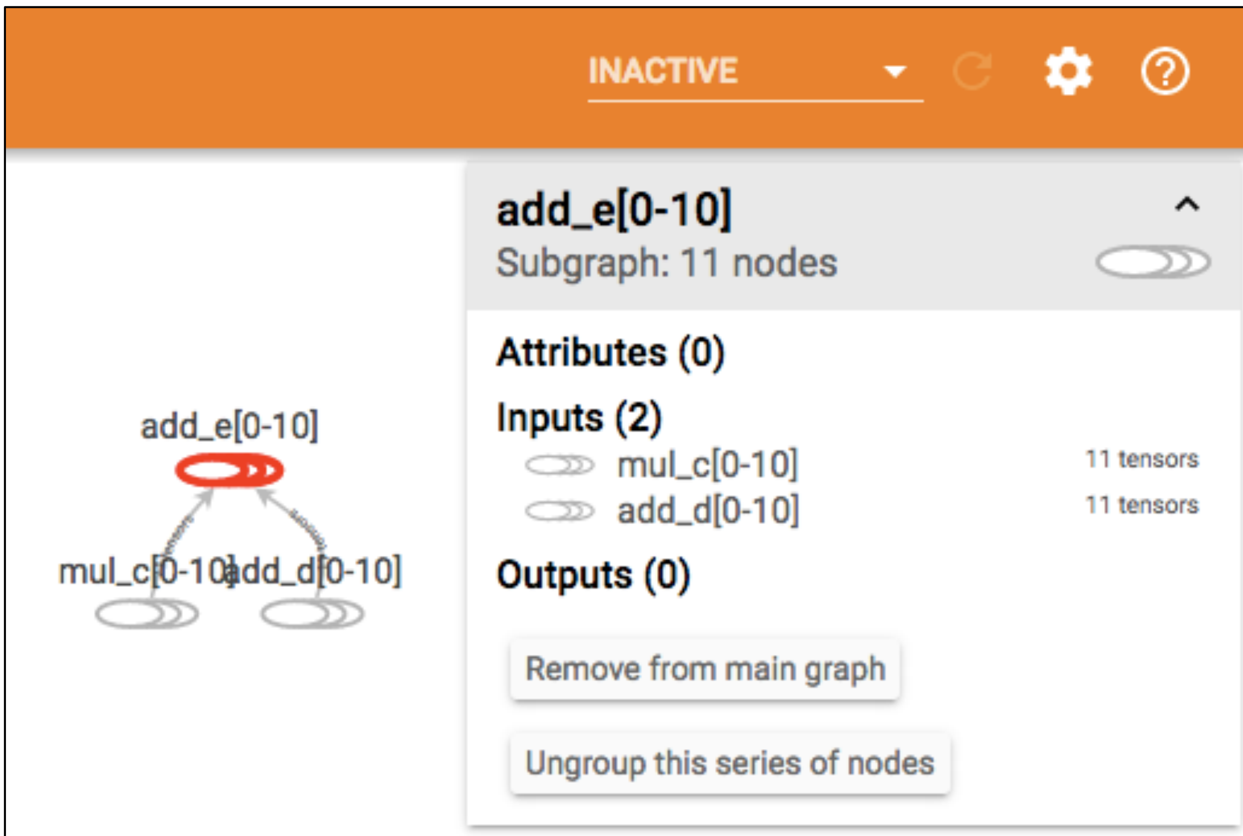
# TensorFlow

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



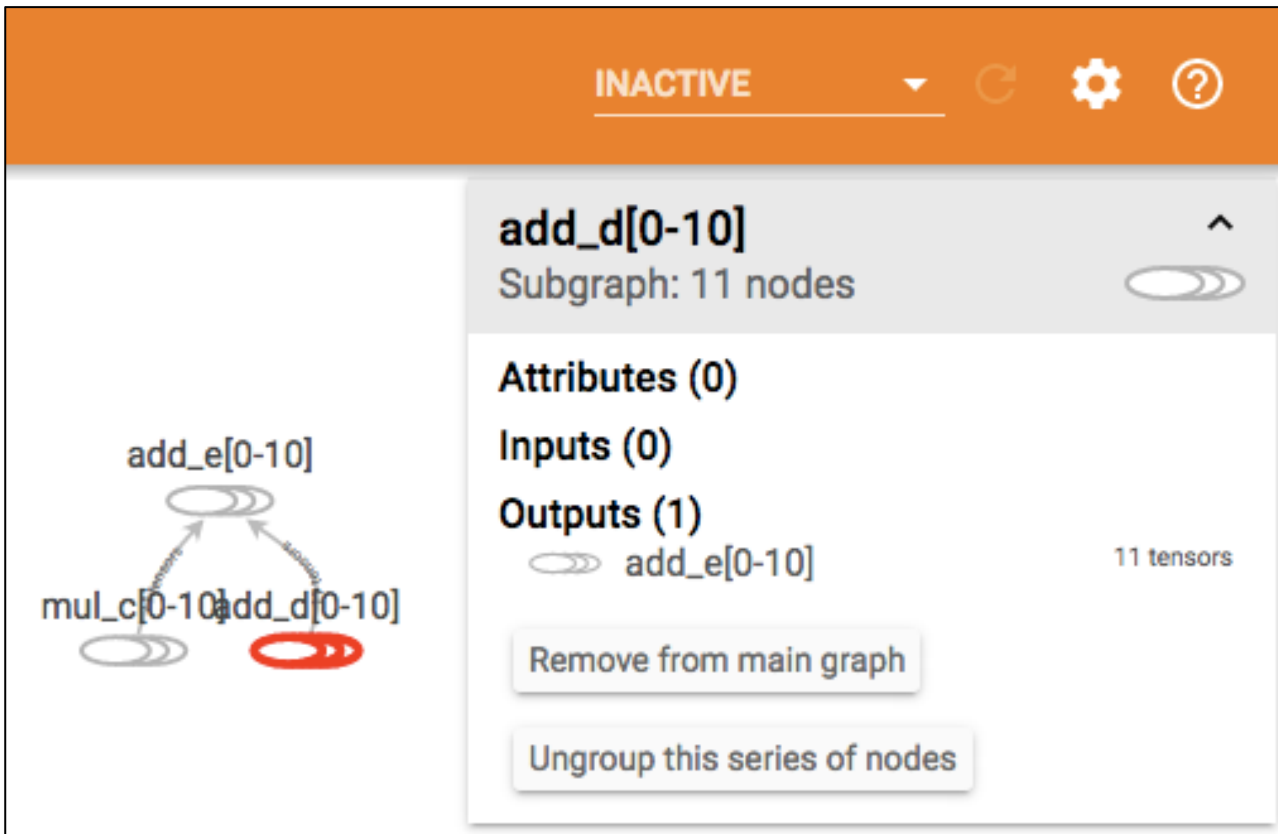
# TensorFlow

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



# TensorFlow

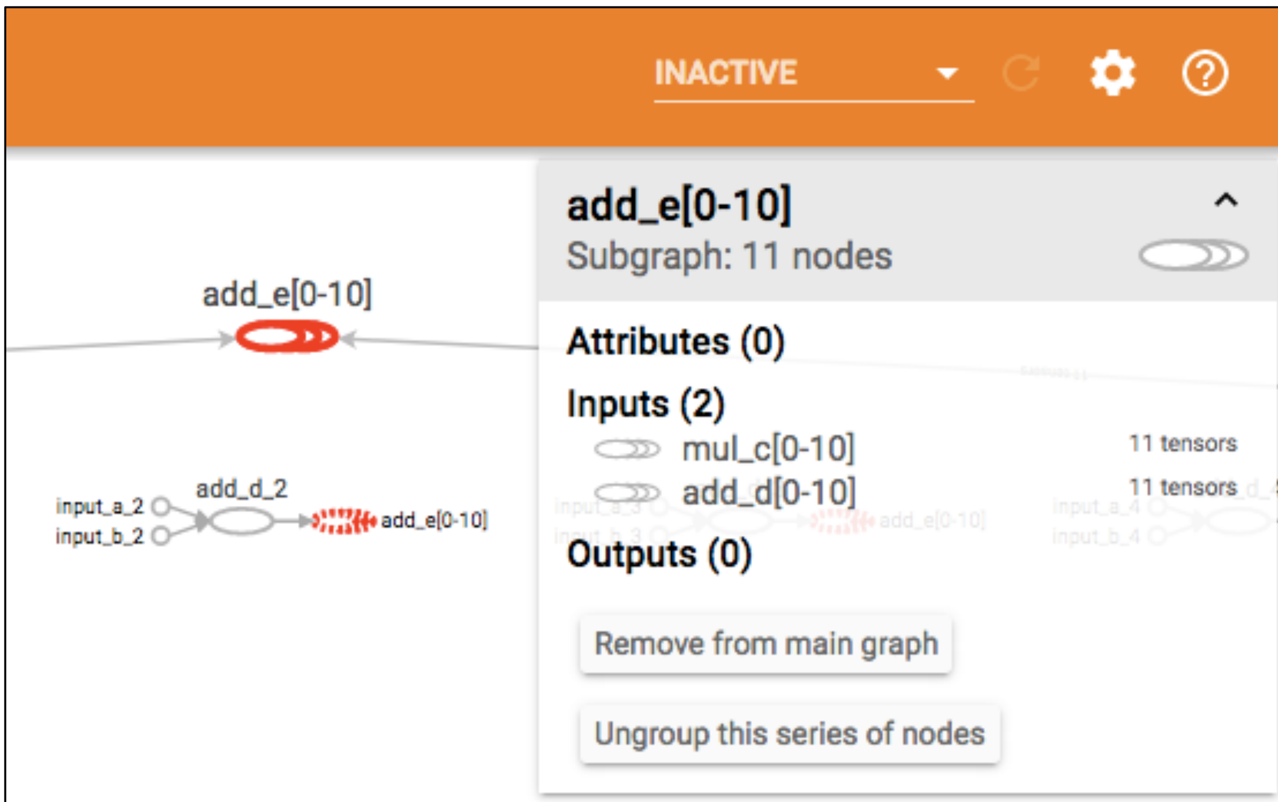
- Let’s construct the actual graph using TensorBoard:
  - Simply click on any of the nodes to inspect them more closely:



# TensorFlow

- Let's construct the actual graph using TensorBoard:

  - Simply click on any of the nodes to inspect them more closely:



# TensorFlow

- Let's construct the actual graph using TensorBoard:

  - Once you 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()
```

  - 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

- Instead of having two separate input nodes, we can have a single input node that can take in a vector (or 1-D tensor) of numbers
- This graph has several advantages over our previous example:

1. The client only has to send input to a single node, which simplifies using the graph
2. The nodes that directly depend on the input now only have to keep track of one dependency instead of two
3. We now have the option of making the graph take in vectors of any length, if we'd like. This would make the graph more flexible