Queen Mary
University of London

DR ADRIAN BEVAN

PRACTICAL MACHINE LEARNING

INTRODUCTORY TENSORFLOW

LECTURE PLAN

- Introduction
  - Graphs
  - Shapes
  - constants, variables and placeholders
- Ops
- Mathematical ops
  - element wise operations
  - metrics computed from ensembles of data
  - matrix operations
- Random numbers
- Generating data
- Summary
- Suggested Reading

These slides focus on the use of tensor flow data types and operations in order to ensure that there is a common background to build on for the remaining weeks.

Once we have covered these basics we will move on to machine learning with TensorFlow.

QMUL Summer School:	https://www.qmul.ac.uk/summer-school/
Practical Machine Learning QMplus Page:	https://qmplus.qmul.ac.uk/course/view.php?id=10006

INTRODUCTION

- In TensorFlow the user sets up a graph for a calculation.
- Then that graph is executed on an ensemble of data.
- This means that you have to think a bit differently about how you approach a problem when you write code.
  - This can take a bit of getting used to.
- We will use the following throughout

```
import tensorflow as tf
```

INTRODUCTION

- For evaluation of some of the more advanced tensorflow scripts we will work with, we also need to allow for the OpenMP library to be loaded multiple times.
- This can be done by adding

```
import os

os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

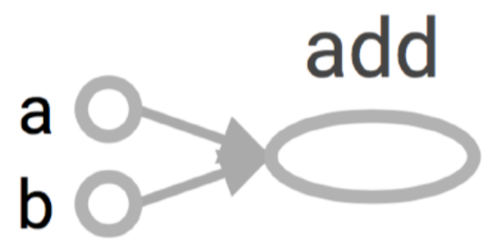
- If we don't do this, Spyder will stop evaluating scripts part way through (when model training) without any error or warning. If you use PyCharm you will get an error, and if you use a terminal, then the script will execute without any problem (on a Mac).

INTRODUCTION: GRAPHS

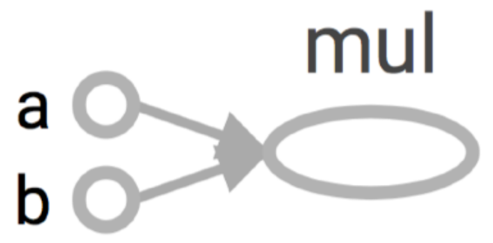
- Conceptually the graph is an implementation of the calculation that you wish to perform, that is abstracted from the data that you wish to operate on.
- e.g. consider the operation of multiplying the number 3 by a scale factor of 5:
  - y = 5 * 3.
- If we want to be able to define a rule to arbitrarily scale some number or an ensemble of numbers by that scale factor, then we could write:
  - y = 5 * x.
- Here x is not specified, and we could interpret that as a scalar. We could also consider x and y as a higher dimensional representation of data such as a vector or tensor.

INTRODUCTION: GRAPHS

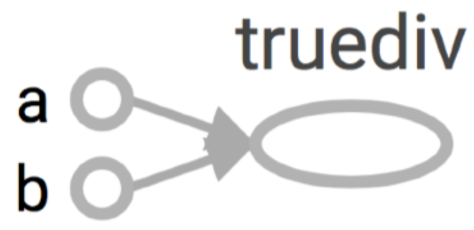
- Consider the following graphs:



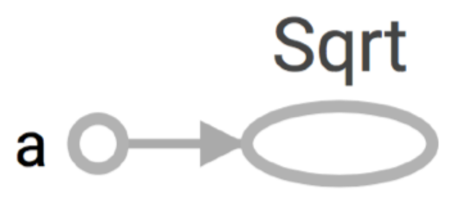
$$y = a + b$$



$$y = a \times b$$



$$y = a/b$$



$$y = \sqrt{a}$$

- The graphs for your scripts can be inspected using TensorBoard.

INTRODUCTION: SHAPES

TensorFlow types depend on shape; this allows graphs to be abstracted in such a way that is independent of the data representation.

Scalars, vectors matrices and tensors are loosely referred to as tensors for the purposes of model implementation.

- **Scalar:**
  - A single number.
  - The rank of a scalar is 0.

$$x = 3$$

- **Vector:**
  - A 1D collection of numbers.
  - The rank of a vector is 1.

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

- **Matrix:**
  - A 2D collection of numbers.
  - The rank of a matrix is 2.

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

- **Tensor:**
  - A higher dimensional collection of numbers.
  - The rank of a tensor is ≥3.

e.g. a Rank 3 tensor can be visualised by taking an NxM matrix and allowing each element to become a vector of dimension P (so this would be a NxMxP tensor of rank 3).

https://www.tensorflow.org/programmers_guide/tensors

INTRODUCTION

- Tensor flow has its own types, operations (ops) and higher level objects. These slides will focus mostly on the types and ops.
- We will encounter
  - constants:
  - Variables:
  - placeholders:
    - This is a placeholder for a tensor. We will use these with feed_dict when training models.
- There are also SparseTensors (we won't use these, but they can help implement efficient calculations for some problems).

https://www.tensorflow.org/programmers_guide/tensors

INTRODUCTION: CONSTANTS

Constants have a fixed value that is not changeable

- We can define constants of a given shape using `tf.constant(args)`:
  - Using a list of values to create a constant tensor

```
b          = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=tf.float32, name="b")
```

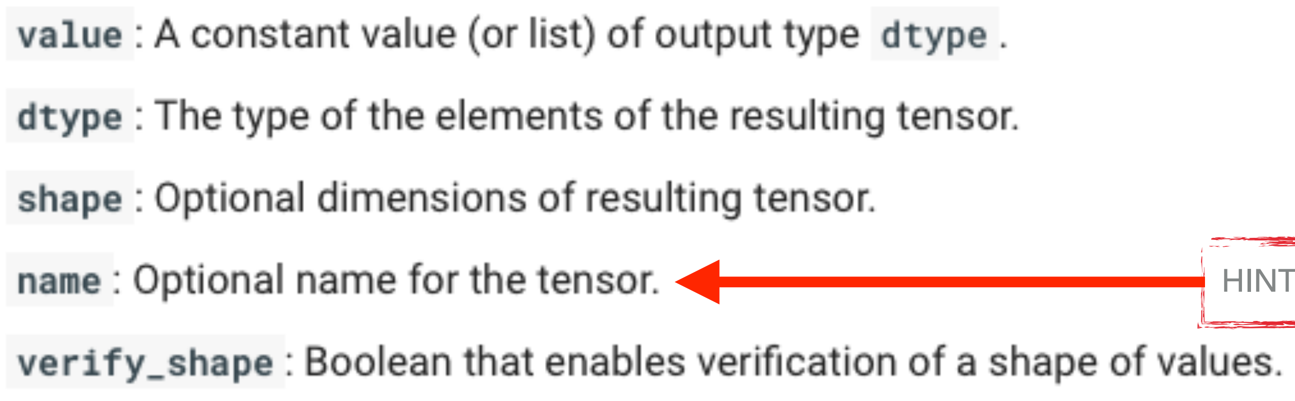
  - NumPy arrays can also be used for this purpose.

```
x = np.array([[1, 2], [3, 4]], name="x")
c = tf.constant(x, name="c")
```

  - tensors can be created with a pre-defined shape (no need for a NumPy array:

```
d = tensor = tf.constant(-1.0, shape=[2, 2], name="d")
```

- args:



`value` : A constant value (or list) of output type `dtype` .

`dtype` : The type of the elements of the resulting tensor.

`shape` : Optional dimensions of resulting tensor.

`name` : Optional name for the tensor. ← HINT: assign names to help understand graph descriptions.

`verify_shape` : Boolean that enables verification of a shape of values.

https://www.tensorflow.org/api_docs/python/tf/constant

INTRODUCTION: VARIABLES

Variables can be changed after initialisation

- A tf Variable is created similarly with `tf.Variable(args)`

```
tensorVar = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], name="tensorVar")
```

- The structure of the tensor can be printed out, but graph evaluation can not proceed without variable initialisation (see later).

- Having created a variable you can change it using ops (see later). For example:

```
tensorVar.assign(tensorVar + 1)
tensorVar.assign_add(1)
```

  both add 1.0 to the original value of the tensor elements.

- Args: see [1].

[1] https://www.tensorflow.org/api_docs/python/tf/Variable

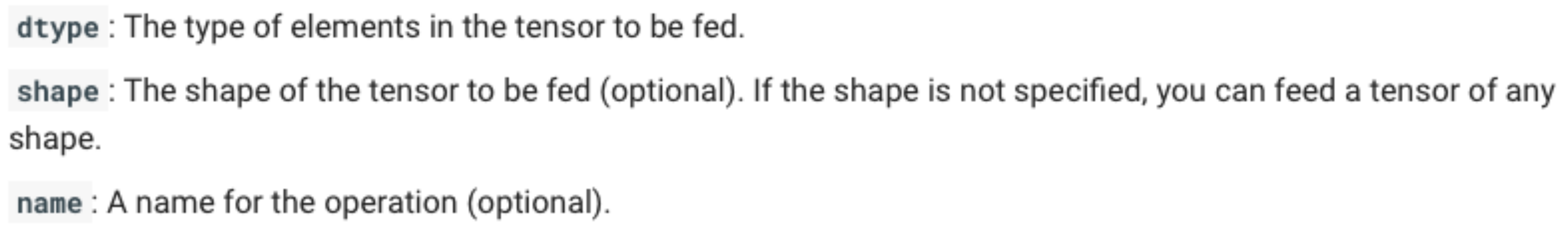
INTRODUCTION: PLACEHOLDERS

placeholders don't have a fully defined shape

- These are tensors that need to be fed using feed_dict when being evaluated and created using `tf.placeholder(args)`
- e.g. consider a placeholder tensor for a 10x10 matrix with graph y=x.x

```
xp = tf.placeholder(tf.float32, shape=(10, 10), name="xp")
y = tf.matmul(xp, xp)
rand_array = np.random.rand(10, 10)
print(sess.run(y, feed_dict={xp: rand_array}))
```

- Allows the user to evaluate a sub-set of of data examples during a training cycle.
- Shapes can be incomplete (e.g. don't have to specify the number of examples ahead of training a model)
- args:



`dtype` : The type of elements in the tensor to be fed.

`shape` : The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.

`name` : A name for the operation (optional).

https://www.tensorflow.org/api_docs/python/tf/placeholder

INTRODUCTION: EVALUATION

- Having defined a graph, evaluation of that graph on data is required. This can be done in one of several ways
  - Using a session:

```
sess = tf.Session()
sess.run(tensor)
```

  - One can specify a session as default:

```
sess = tf.Session()
with sess.as_default():
     tensor.eval()
```

INTRODUCTION: EVALUATION

- Having defined a graph, evaluation of that graph on data is required. This can be done in one of several ways
  - Using a session with variables:

```
sess = tf.Session()
init  = tf.global_variables_initializer()
sess.run(init)
sess.run(tensor)
```

  - One can specify a session as default (with variables):

```
sess = tf.Session()
init  = tf.global_variables_initializer()
sess.run(init)
with sess.as_default():
     tensor.eval()
```

INTRODUCTION: EVALUATION

- Having defined a graph, evaluation of that graph on data is required. This can be done in one of several ways
  - Using a session with variables:

```
sess = tf.Session()
sess.run(tensor.initializer)
sess.run(tensor)
```

  - One can specify a session as default (with variables):

```
sess = tf.Session()
with sess.as_default():
     sess.run(tensor.initializer)
     tensor.eval()
```

INTRODUCTION: EVALUATION

- Having defined a graph, evaluation of that graph on data is required. This can be done in one of several ways
  - Using a
  - One ca riables):

The steps involved in this process are:

1. Create the graph
2. Initialise the variables in the graph
3. Evaluate the graph

```
    tensor.eval()
```

INTRODUCTION: EVALUATION

- If you try to use a variable that has not been initialised you will get an error like the following

```
Traceback (most recent call last):
  File "./vars.py", line 72, in <module>
    print("tensorVar2           = ", sess.run(tensorVar2))
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 766, in run
    run_metadata_ptr)
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 964, in _run
    feed_dict_string, options, run_metadata)
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 1014, in _do_run
    target_list, options, run_metadata)
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 1034, in _do_call
    raise type(e)(node_def, op, message)
tensorflow.python.framework.errors_impl.FailedPreconditionError: Attempting to use uninitialized value tensorVar2
     [[Node: _send_tensorVar2_0 = _Send[T=DT_INT32, client_terminated=true, recv_device="/job:localhost/replica:0/task:
0/cpu:0", send_device="/job:localhost/replica:0/task:0/cpu:0", send_device_incarnation=-4751424724121413910,
tensor_name="tensorVar2:0", _device="/job:localhost/replica:0/task:0/cpu:0"](tensorVar2)]]
```

INTRODUCTION: EVALUATION

- If you try to use a variable that has not been initialised you will get an error like the following

Where the problem is

```
Traceback (most recent call last):
  File "./vars.py", line 72, in <module>
    print("tensorVar2            = ", sess.run(tensorVar2))
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 766, in run
    run_metadata_ptr)
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 964, in _run
    feed_dict_string, options, run_metadata)
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 1014, in _do_run
    target_list, options, run_metadata)
  File "/Library/Python/2.7/site-packages/tensorflow/python/client/session.py", line 1034, in _do_call
    raise type(e)(node_def, op, message)
tensorflow.python.framework.errors_impl.FailedPreconditionError: Attempting to use uninitialized value tensorVar2
     [[Node: _send_tensorVar2_0 = _Send[T=DT_INT32, client_terminated=true, recv_device="/job:localhost/replica:0/task:
0/cpu:0", send_device="/job:localhost/replica:0/task:0/cpu:0", send_device_incarnation=-4751424724121413910,
tensor_name="tensorVar2:0", _device="/job:localhost/replica:0/task:0/cpu:0"](tensorVar2)]]
```

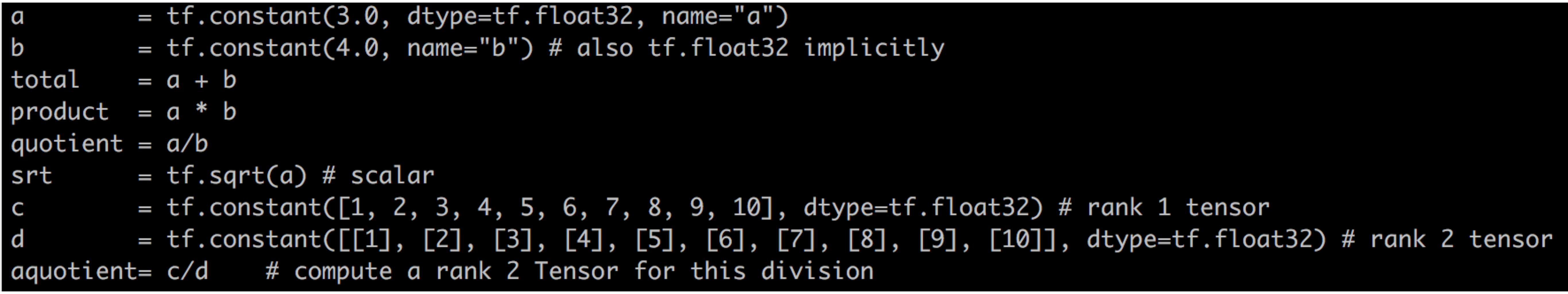
What the problem is

name of the tensor. If you don't specify this value when declaring your tensor you will get a default name assigned e.g. Variable or Variable_N. This makes debugging your scripts harder work (and this is one of the reasons why I recommend naming your tensors).

OPS

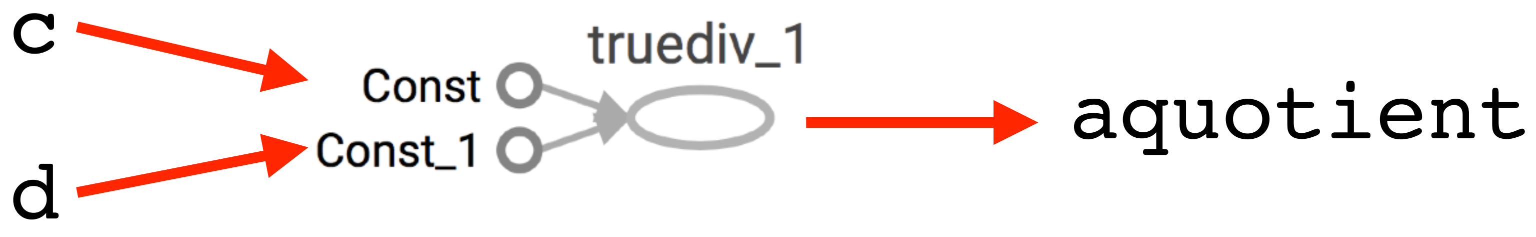
- `ops.py, ops2.py`

Uses tensorflow



```
a        = tf.constant(3.0, dtype=tf.float32, name="a")
b        = tf.constant(4.0, name="b") # also tf.float32 implicitly
total    = a + b
product  = a * b
quotient = a/b
srt      = tf.sqrt(a) # scalar
c        = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=tf.float32) # rank 1 tensor
d        = tf.constant([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]], dtype=tf.float32) # rank 2 tensor
aquotient= c/d    # compute a rank 2 Tensor for this division
```

- This script defines two constants, a and b, and then computes the sum, product, ratio of these and the square root of a.
- It also defines 2 tensors and computes the ratio of both of these.
- Note that c and d do not have names. These will have default names of Const and Const_1 when inspecting graphs using TensorBoard.

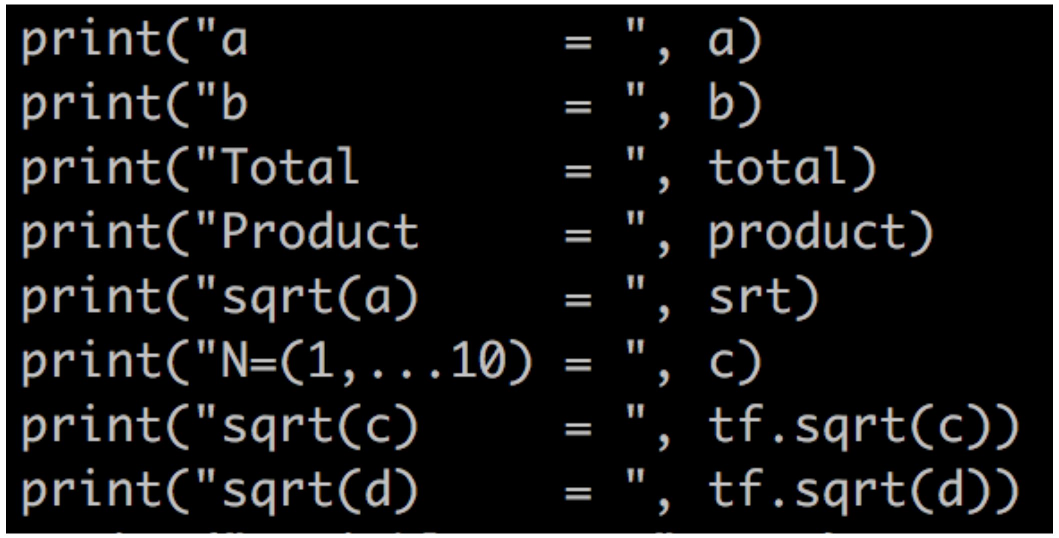


OPS

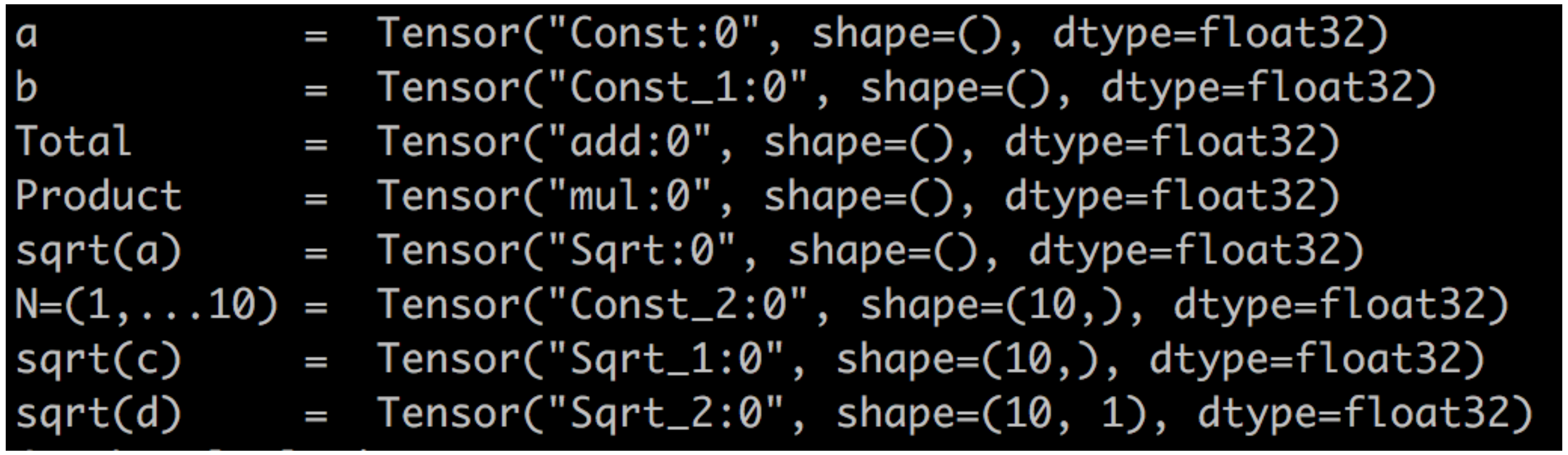
- ops.py, ops2.py

Uses tensorflow

- Printing tensors provides information about their shape, type and name.



```
print("a            = ", a)
print("b            = ", b)
print("Total        = ", total)
print("Product      = ", product)
print("sqrt(a)      = ", srt)
print("N=(1,...10) = ", c)
print("sqrt(c)      = ", tf.sqrt(c))
print("sqrt(d)      = ", tf.sqrt(d))
```



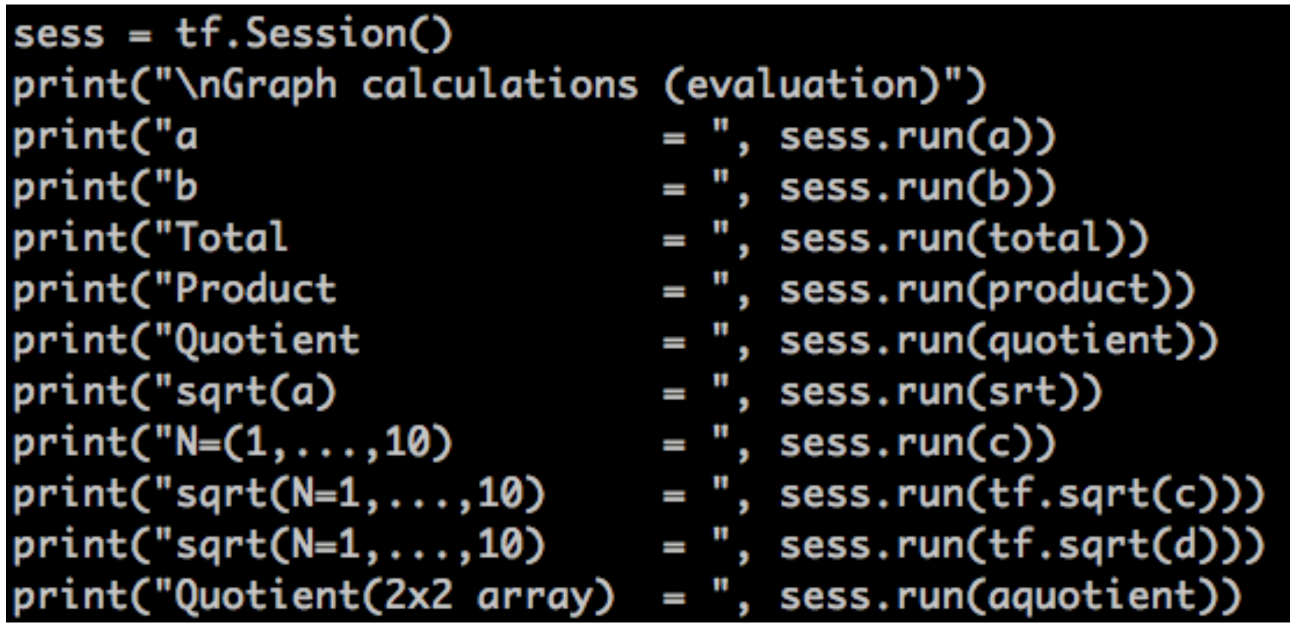
```
a            =  Tensor("Const:0", shape=(), dtype=float32)
b            =  Tensor("Const_1:0", shape=(), dtype=float32)
Total        =  Tensor("add:0", shape=(), dtype=float32)
Product      =  Tensor("mul:0", shape=(), dtype=float32)
sqrt(a)      =  Tensor("Sqrt:0", shape=(), dtype=float32)
N=(1,...10) =  Tensor("Const_2:0", shape=(10,), dtype=float32)
sqrt(c)      =  Tensor("Sqrt_1:0", shape=(10,), dtype=float32)
sqrt(d)      =  Tensor("Sqrt_2:0", shape=(10, 1), dtype=float32)
```

OPS

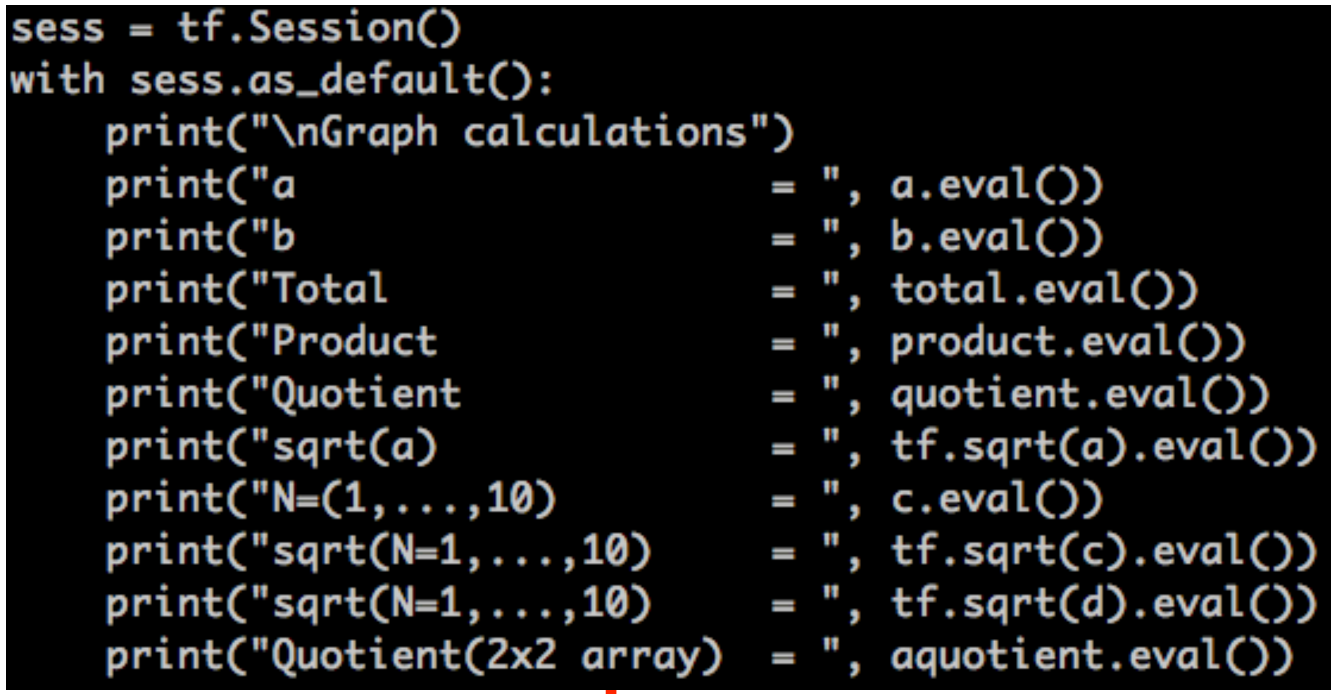
- `ops.py, ops2.py`

Uses tensorflow

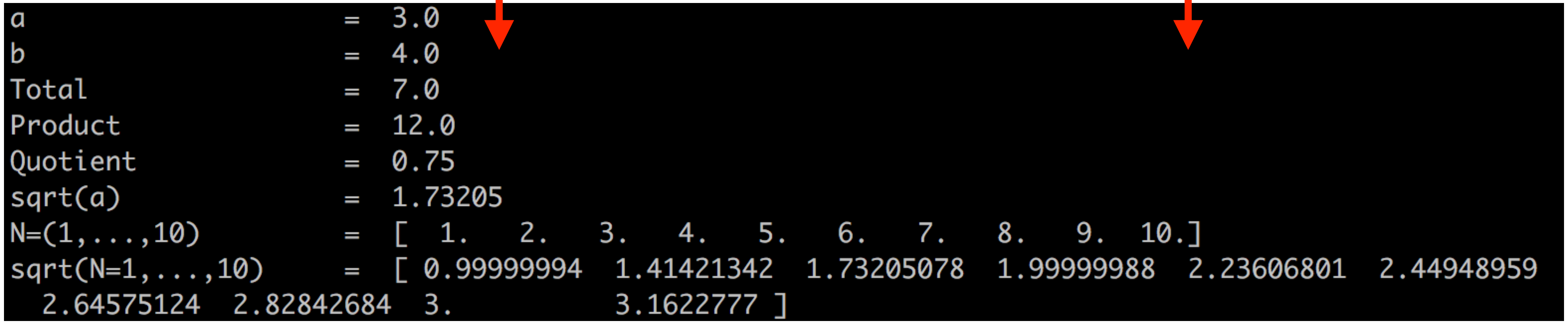
- need to use `sess.run(tensor)` to print out the values of the tensors; or specify the default session and eval().



```
sess = tf.Session()
print("\nGraph calculations (evaluation)")
print("a                       = ", sess.run(a))
print("b                       = ", sess.run(b))
print("Total                   = ", sess.run(total))
print("Product                 = ", sess.run(product))
print("Quotient                = ", sess.run(quotient))
print("sqrt(a)                 = ", sess.run(srt))
print("N=(1,...,10)            = ", sess.run(c))
print("sqrt(N=1,...,10)        = ", sess.run(tf.sqrt(c)))
print("sqrt(N=1,...,10)        = ", sess.run(tf.sqrt(d)))
print("Quotient(2x2 array)     = ", sess.run(aquotient))
```



```
sess = tf.Session()
with sess.as_default():
    print("\nGraph calculations")
    print("a                       = ", a.eval())
    print("b                       = ", b.eval())
    print("Total                   = ", total.eval())
    print("Product                 = ", product.eval())
    print("Quotient                = ", quotient.eval())
    print("sqrt(a)                 = ", tf.sqrt(a).eval())
    print("N=(1,...,10)            = ", c.eval())
    print("sqrt(N=1,...,10)        = ", tf.sqrt(c).eval())
    print("sqrt(N=1,...,10)        = ", tf.sqrt(d).eval())
    print("Quotient(2x2 array)     = ", aquotient.eval())
```

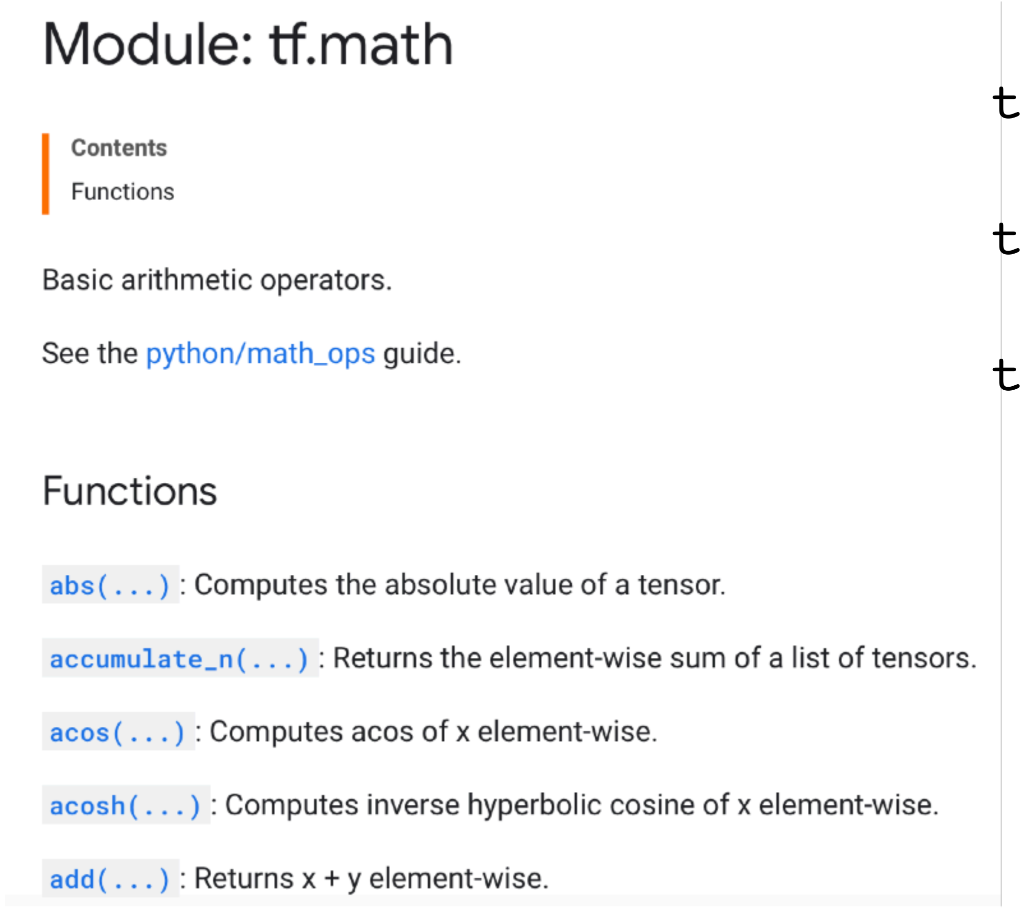


```
a                    =  3.0
b                    =  4.0
Total                =  7.0
Product              =  12.0
Quotient             =  0.75
sqrt(a)              =  1.73205
N=(1,...,10)         =  [  1.    2.    3.    4.    5.    6.    7.    8.    9.   10.]
sqrt(N=1,...,10)     =  [ 0.99999994  1.41421342  1.73205078  1.99999988  2.23606801  2.44948959
  2.64575124  2.82842684  3.          3.1622777 ]
```

etc.

MATHEMATICAL OPS: ELEMENT WISE

- The operators +, -, * and / are defined for tensors so that element wise computations are performed on the data.
- Mathematical operations are also defined (like tf.sqrt that we have already seen).



Module: tf.math

Contents
Functions

Basic arithmetic operators.

See the python/math_ops guide.

Functions

`abs(...)`: Computes the absolute value of a tensor.

`accumulate_n(...)`: Returns the element-wise sum of a list of tensors.

`acos(...)`: Computes acos of x element-wise.

`acosh(...)`: Computes inverse hyperbolic cosine of x element-wise.

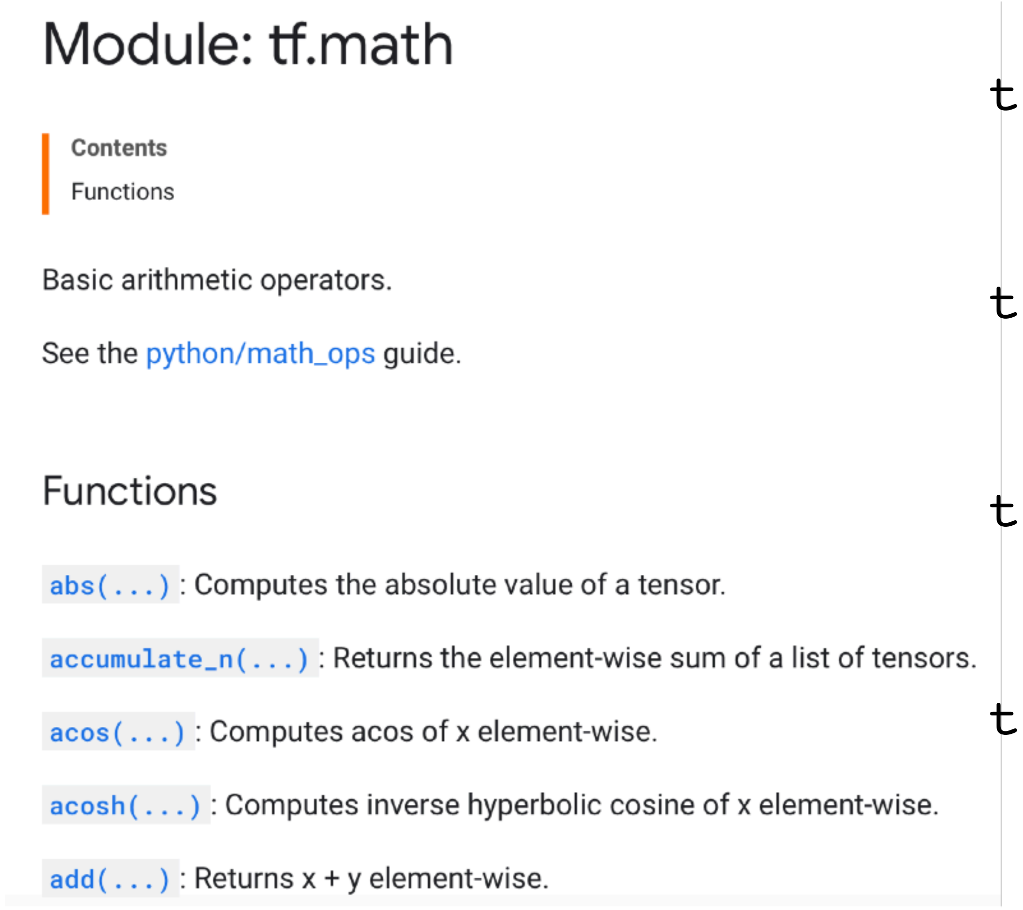
`add(...)`: Returns x + y element-wise.

`tf.abs`:	compute the magnitude of a complex number
`tf.acos`:	element wise computation of acos (inverse cosine)
`tf.acosh`:	element wise computation of acosh (inverse hyperbolic cosine)

https://www.tensorflow.org/api_docs/

MATHEMATICAL OPS: ELEMENT WISE ‣ `ops3.py`

- The operators +, -, * and / are defined for tensors so that element wise computations are performed on the data.
- Mathematical operations are also defined (like tf.sqrt that we have already seen).


Module: tf.math

Contents
Functions

Basic arithmetic operators.

See the python/math_ops guide.

Functions

`abs(...)`: Computes the absolute value of a tensor.

`accumulate_n(...)`: Returns the element-wise sum of a list of tensors.

`acos(...)`: Computes acos of x element-wise.

`acosh(...)`: Computes inverse hyperbolic cosine of x element-wise.

`add(...)`: Returns x + y element-wise.

`tf.add`:	element wise computation of the sum of two tensors (equivalent to +)
`tf.div`:	element wise computation of the sum of two tensors (equivalent to /)
`tf.multiply`:	element wise computation of the sum of two tensors (equivalent to *)
`tf.subtract`:	element wise computation of the sum of two tensors (equivalent to -)

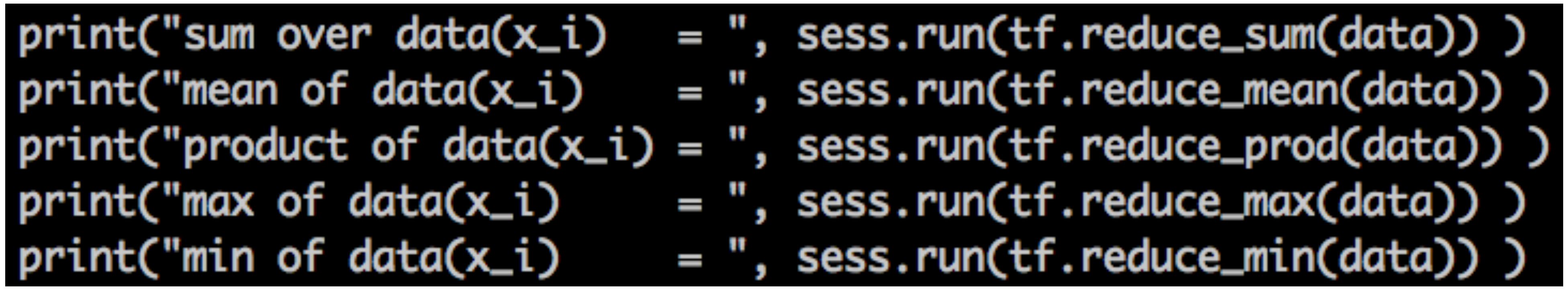
etc.

https://www.tensorflow.org/api_docs/

MATHEMATICAL OPS: METRICS

- `ops_math.py`

- In addition to element wise operations that are defined, TensorFlow has a number of figure of merit (metric) computations defined, including:



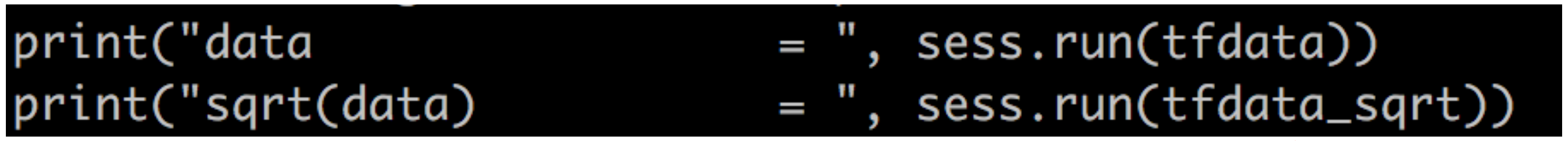
```
print("sum over data(x_i)    = ", sess.run(tf.reduce_sum(data)) )
print("mean of data(x_i)     = ", sess.run(tf.reduce_mean(data)) )
print("product of data(x_i) = ", sess.run(tf.reduce_prod(data)) )
print("max of data(x_i)      = ", sess.run(tf.reduce_max(data)) )
print("min of data(x_i)      = ", sess.run(tf.reduce_min(data)) )
```

- For some data (this example uses a numpy array, it also works with tensors) we can compute the sum, mean, product … of the elements.

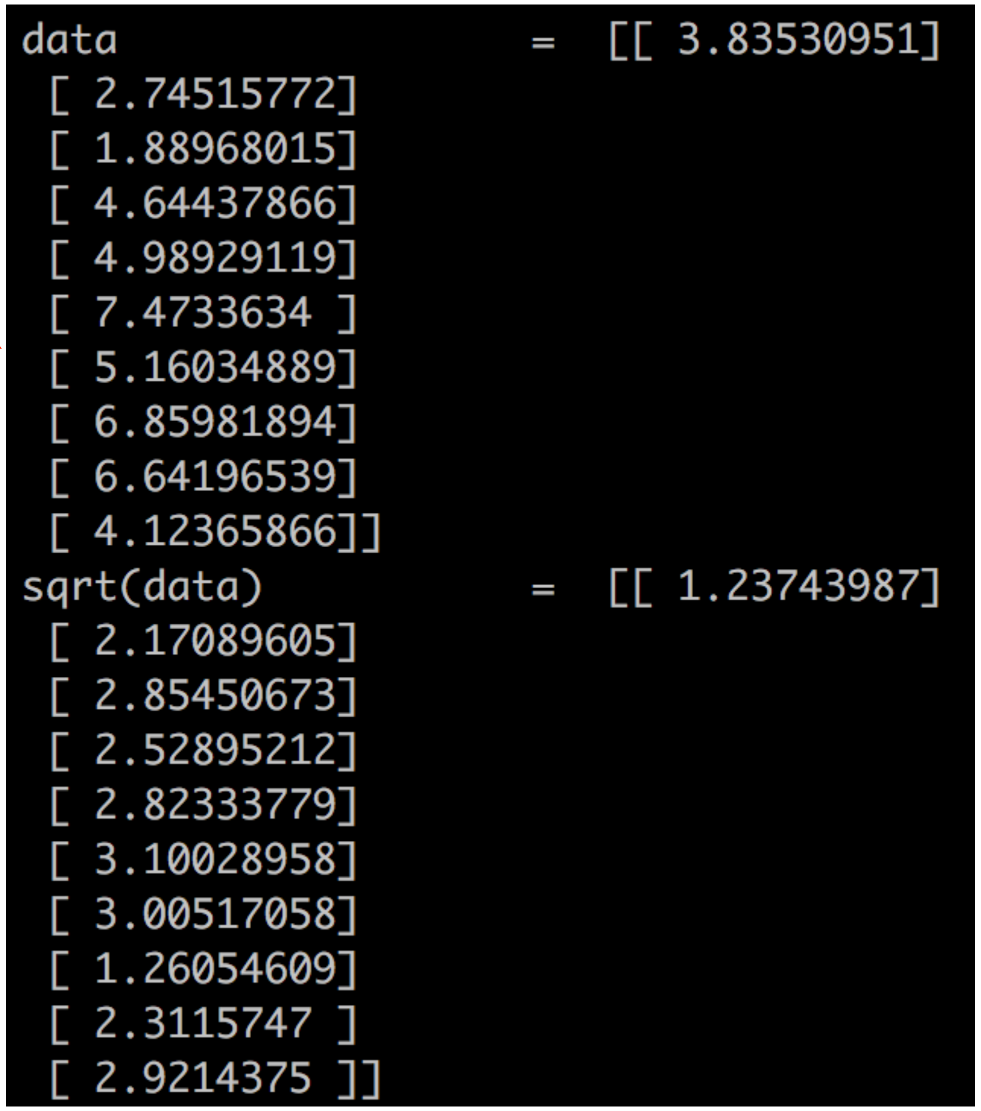
- e.g. the op reduce_sum is equivalent to $\sum_{i=1}^{N_{data}} x_i$.

MATHEMATICAL OPS: METRICS

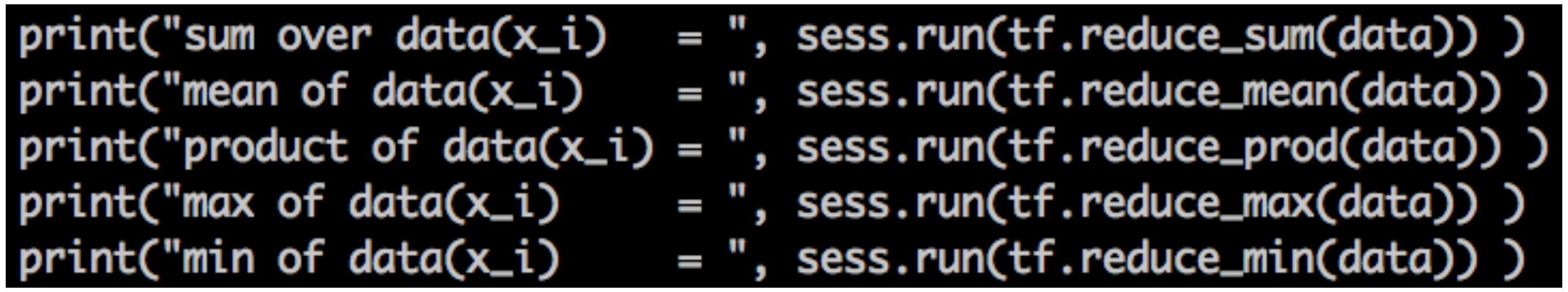
- ops_math.py



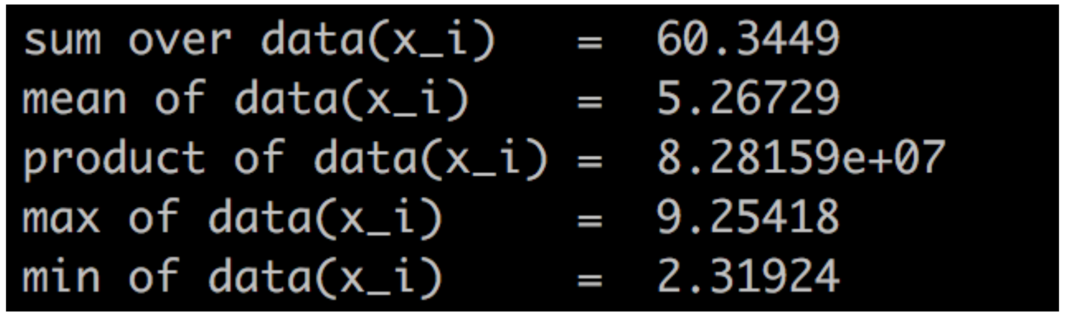
```
print("data                  = ", sess.run(tfdata))
print("sqrt(data)            = ", sess.run(tfdata_sqrt))
```



```
data                 =  [[ 3.83530951]
 [ 2.74515772]
 [ 1.88968015]
 [ 4.64437866]
 [ 4.98929119]
 [ 7.4733634 ]
 [ 5.16034889]
 [ 6.85981894]
 [ 6.64196539]
 [ 4.12365866]]
sqrt(data)           =  [[ 1.23743987]
 [ 2.17089605]
 [ 2.85450673]
 [ 2.52895212]
 [ 2.82333779]
 [ 3.10028958]
 [ 3.00517058]
 [ 1.26054609]
 [ 2.3115747 ]
 [ 2.9214375 ]]
```



```
print("sum over data(x_i)   = ", sess.run(tf.reduce_sum(data)) )
print("mean of data(x_i)    = ", sess.run(tf.reduce_mean(data)) )
print("product of data(x_i) = ", sess.run(tf.reduce_prod(data)) )
print("max of data(x_i)     = ", sess.run(tf.reduce_max(data)) )
print("min of data(x_i)     = ", sess.run(tf.reduce_min(data)) )
```



```
sum over data(x_i)    =  60.3449
mean of data(x_i)     =  5.26729
product of data(x_i)  =  8.28159e+07
max of data(x_i)      =  9.25418
min of data(x_i)      =  2.31924
```

MATHEMATICAL OPS: MATRIX OPERATIONS

- `ops_matrix.py`

- Many of the ops required for models involve matrix multiplication.
- Data will generally have more than one feature (dimension) e.g. a 2D image has an x and a y value for each pixel and a colour value (grey scale) or three colour values (R, G, B image).
- Models will process that information by multiplying the input data by some scale factor (called a weight).
- One can think of the weights as individual numbers that are used to multiply individual elements, and to sum over all operations required for the computation.
- It is convenient to write these as a matrix multiplication.

MATHEMATICAL OPS: MATRIX OPERATIONS

- `ops_matrix.py`

- Useful TensorFlow ops for matrix manipulation include:
  - matmul: `c = tf.matmul(a, b)`
  - matrix_determinant: `detA = tf.matrix_determinant(a)`
  - matrix_inverse: `invA = tf.matrix_inverse(a)`

Note that the matrix_inverse does not work with integer matrices

RANDOM NUMBERS

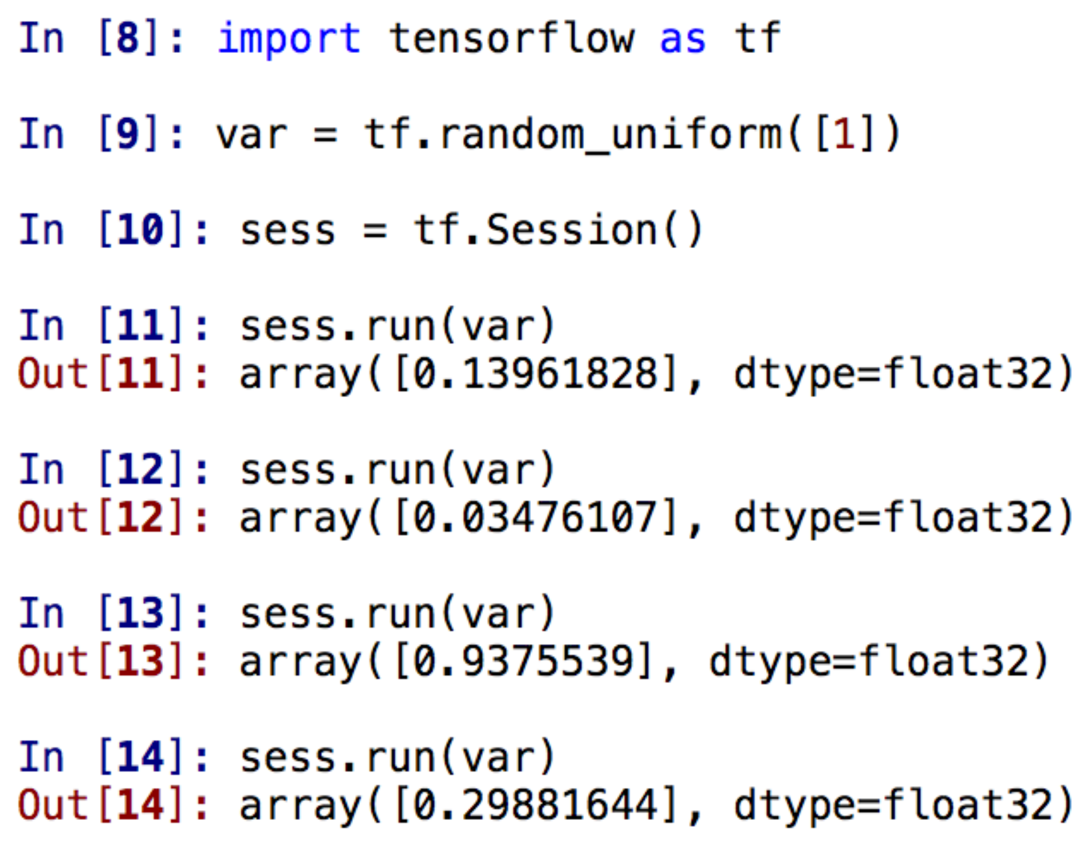
- Random number generation is useful for two things:
  - Weight parameter initialisation
  - Generating simulated data sets that can be used for testing models.
- TensorFlow has a number of functions to generate tensors of random numbers with different distributions:
  - `tf.random.rand`
  - `tf.random_normal`
  - `tf.truncated_normal`
  - `tf.random_uniform`
  - `tf.random_shuffle`
  - `tf.random_crop`
  - `tf.multinomial`
  - `tf.random_gamma`

In addition to these `tf.set_random_seed` can be used to ensure that the random number sequence is reproducible.

https://www.tensorflow.org/api_docs/python/tf/random

RANDOM NUMBERS

- Problem - generate a random number and repeatedly evaluate the tf Variable to see what happens.



```
In [8]: import tensorflow as tf

In [9]: var = tf.random_uniform([1])

In [10]: sess = tf.Session()

In [11]: sess.run(var)
Out[11]: array([0.13961828], dtype=float32)

In [12]: sess.run(var)
Out[12]: array([0.03476107], dtype=float32)

In [13]: sess.run(var)
Out[13]: array([0.9375539], dtype=float32)

In [14]: sess.run(var)
Out[14]: array([0.29881644], dtype=float32)
```

Create a tf Variable with shape [1].

This scalar evaluates to 0.1396… the first time it is evaluated.

It gives 0.03476… the second time

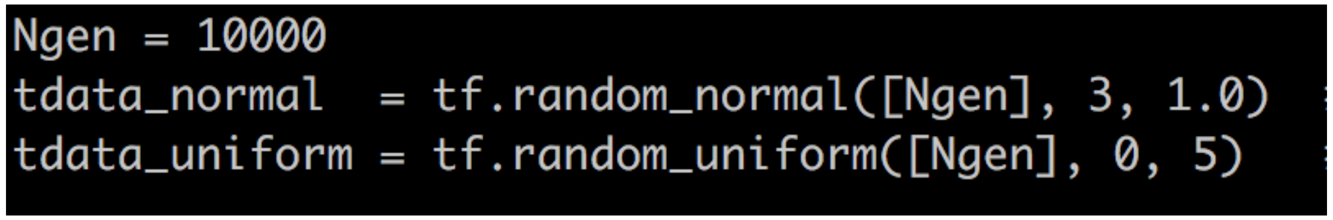
and changes every subsequent time it is evaluated.

https://www.tensorflow.org/api_docs/python/tf/random

GENERATING DATA ▸ `generating_data.py`

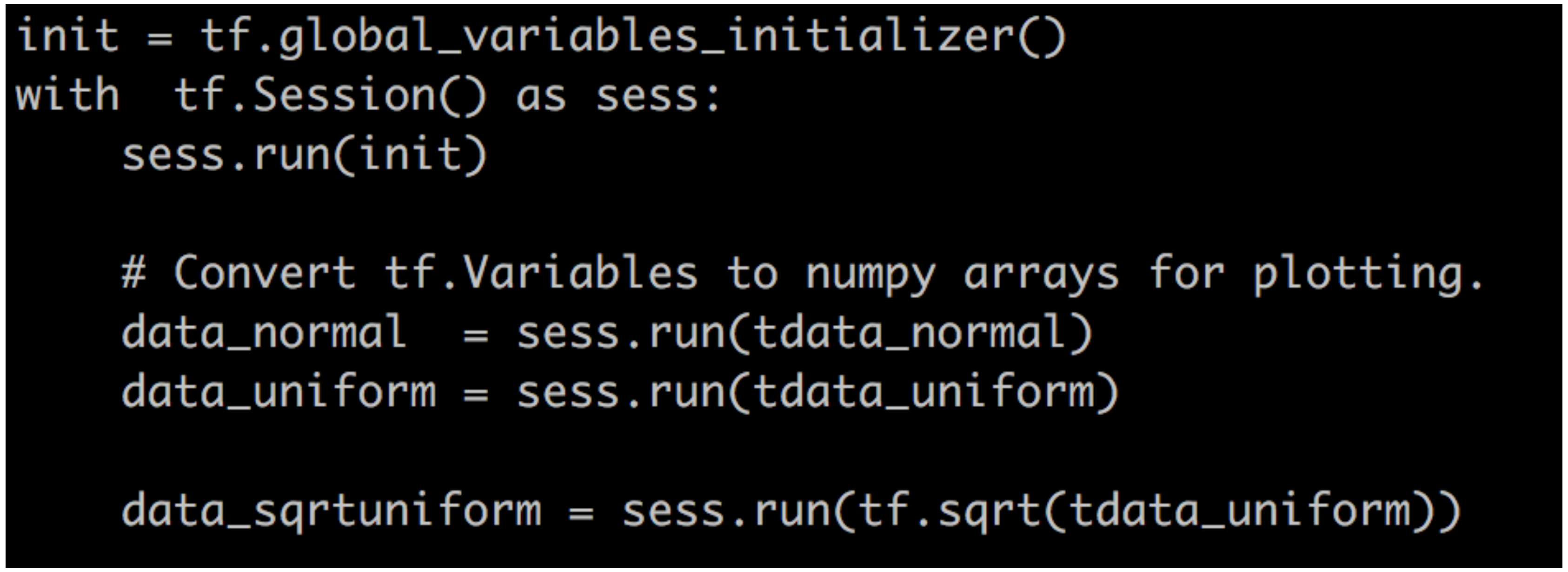
Uses tensorflow and matplotlib.pyplot

- Generate 10k examples to plot: define the graph:



```
Ngen = 10000
tdata_normal  = tf.random_normal([Ngen], 3, 1.0)
tdata_uniform = tf.random_uniform([Ngen], 0, 5)
```

- Run the computation



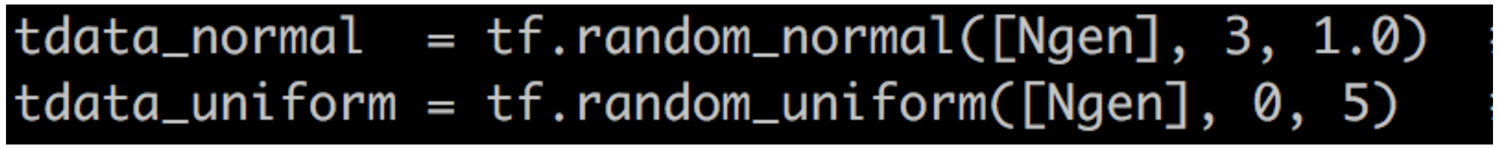
```
init = tf.global_variables_initializer()
with  tf.Session() as sess:
    sess.run(init)

    # Convert tf.Variables to numpy arrays for plotting.
    data_normal  = sess.run(tdata_normal)
    data_uniform = sess.run(tdata_uniform)

    data_sqrtuniform = sess.run(tf.sqrt(tdata_uniform))
```

- `data_normal`, `data_uniform` and `data_sqrtuniform` are numpy ND arrays of the corresponding Tensor.

GENERATING DATA ▸ generating_data.py

Uses tensorflow and matplotlib.pyplot

- Note: If you use tensorflow variables to represent the random numbers:

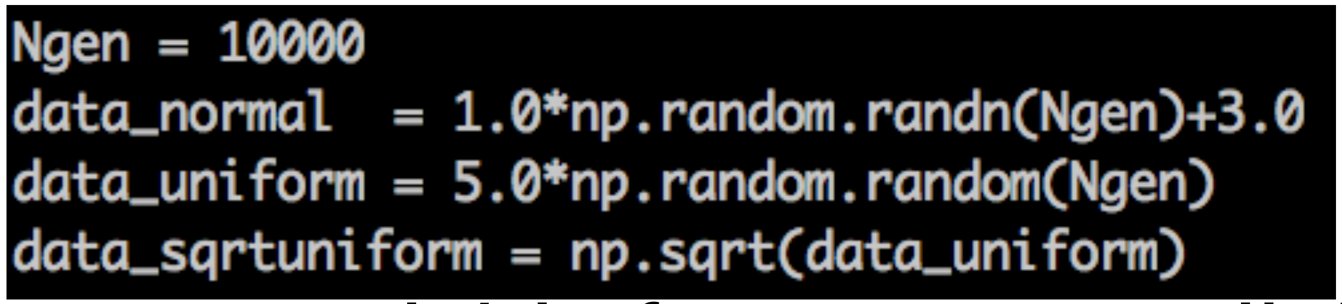


```
tdata_normal  = tf.random_normal([Ngen], 3, 1.0)
tdata_uniform = tf.random_uniform([Ngen], 0, 5)
```

- The graph will be updated each evaluation step.
- This will result in new random numbers being generated each step… which is not necessarily what you want.
- Conversion via numpy arrays, or generating random numbers using numpy arrays avoids this issue.

GENERATING DATA ‣ `generating_data_numpy.py`

- For a single ensemble of random numbers it is better to use numpy, or to evaluate the tf Variable and assign that to a numpy array.



```
Ngen = 10000
data_normal   = 1.0*np.random.randn(Ngen)+3.0
data_uniform = 5.0*np.random.random(Ngen)
data_sqrtuniform = np.sqrt(data_uniform)
```

- NumPy has many available functions to call that will yield randomly sampled numbers
  - `numpy.random.random`
  - `numpy.random.randn`
  - `numpy.random.logistic`
  - `numpy.random.exponential`
  - `numpy.random.poisson`
  - `numpy.random.multinomial`
  - `numpy.random.gamma`

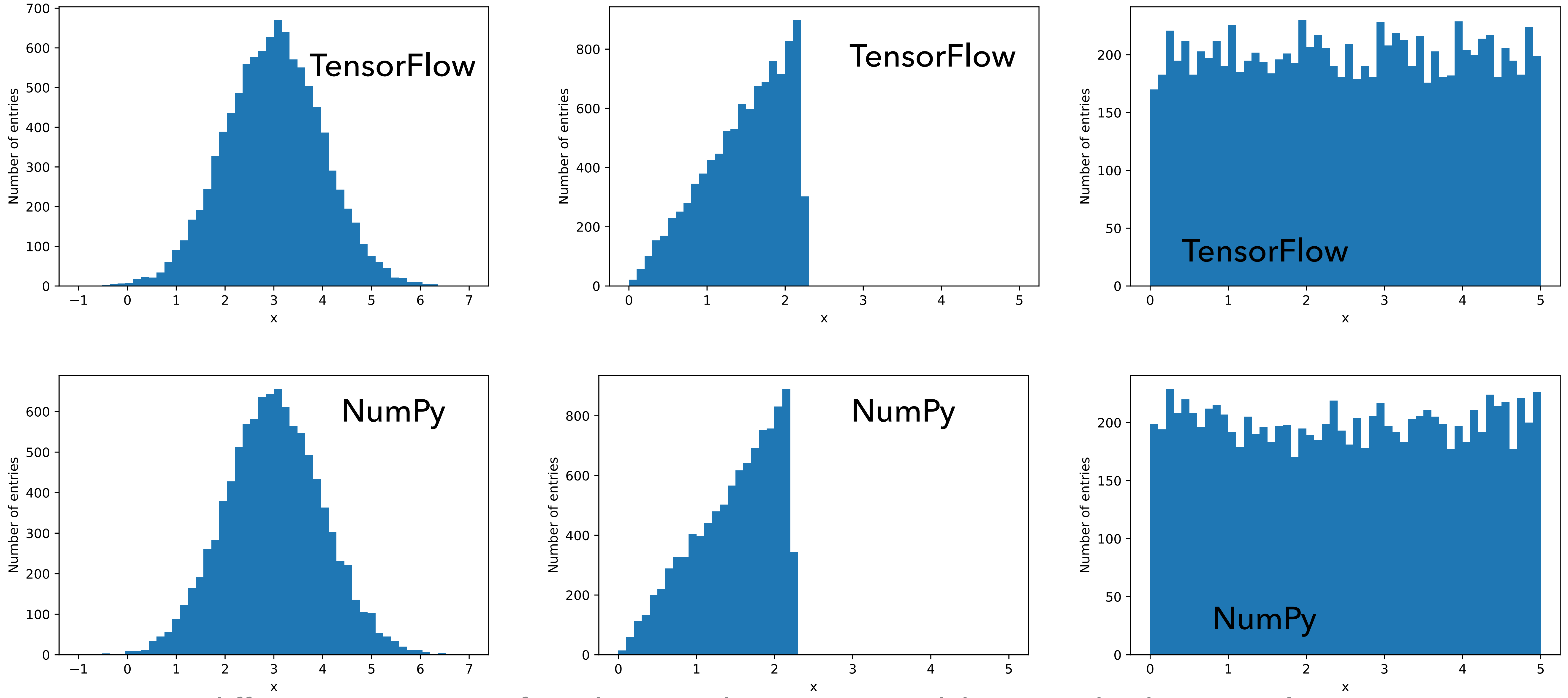
https://docs.scipy.org/doc/numpy/reference/routines.random.html

GENERATING DATA

- `generating_data.py`
- `generating_data_numpy.py`

Uses tensorflow or numpy and matplotlib.pyplot

- This script uses matplotlib.pyplot to make three pdf files, one for each distribution.



Here we see different sequences of random numbers generated, however both TensorFlow and NumPy are generating data, sampling from the same underlying distributions.

SUMMARY

- The TensorFlow programming approach, defining the graph and then performing computation according to that graph has been discussed.
  - TensorBoard helps us inspect model graphs, which is very useful for more complicated models.
- We have explored the use of variables and constants in TensorFlow.
- Looked at a variety of operations, ranging from element wise, through to matrix operations on tensors.
- Introduced random numbers (useful for hyperparameter [HP] initialisation when we come to discuss machine learning).
- Generated some data that can be used for analysis.
  - These basic building bocks (along with the Python we have already covered) will allow us to start exploring machine learning in the next set of coding slides.

SUGGESTED READING

- There are a number of books on TensorFlow available, but as this is a quickly changing framework, the most useful reference for you will be the TensorFlow website, and web searches for any errors that you might encounter.
- e.g.
  - https://www.tensorflow.org/get_started/
  - https://www.tensorflow.org/api_docs/python/
  - https://stackoverflow.com