BMVA Summer School Python Introduction

Neill D. F. Campbell (with input from Tom Haines and Mihaela Rosca)

25th June 2018

Introduction

This document provides a brief introduction to python for numerical computation followed by an illustration of the `TensorFlow` toolkit from Google that provides a powerful interface for the large-scale numerical programming used in modern computer vision and machine learning. There are a number of pointers to recommended online resources for `python` beginners and for those switching from other numerical languages such as `Matlab` and `R`.

Python Beginners!

If you have never used python before I would recommend looking through the online guide (also a book) Automate the Boring Stuff https://automatetheboringstuff.com. Chapters 1 to 10 provide a good overview of the language with few assumptions about background programming knowledge.

The official `python` website https://docs.python.org/3/ provides tutorials and references for the language and native libraries and is recommended as the first port of call for reference material since it will be kept up-to-date and often gives recommendations for the *"pythonic way"* of doing something - a term used to indicate best practice as suggested by the programming community. I recommend exercising caution when taking suggestions from random sites since they may be misleading and there are few guarantees of code quality. Some sites are reasonably well curated (such as stack overflow https://stackoverflow.com or the "LearnPython" subredit https://www.reddit.com/r/learnpython/) and are therefore more reliable.

Python 2 vs 3

There are two main versions of `python` in active use: the version 2 and version 3 branch. I will present all the material using version 3 python however the majority of the syntax is shared between the two versions. If you have a specific reason to use version 2 (compatibility with some other libraries for example) then I hope it will be OK to keep an eye out for any syntax changes, otherwise I would recommend using version 3. If you have old code from version 2 and would like to update it to version 3 then please see the official python guide at https://docs.python.org/3/howto/pyporting.html.

iPython, Jupyter and Google Colaboratory Notebooks

A great way to do prototyping in `python` is to use the *iPython* or *Jupyter* notebook framework. This consists of a python server running on the local or a remote computer and a web interface (accessed by a standard browser) that you use to enter code and run it. Those familiar with `Matlab` or `R` graphical interfaces will find this a familiar

concept where your code is divided up into a series of "cells" and you can execute the cells in sequence and see the results for each cell individually. This provides an interactive programming experience that is very useful for writing new code and debugging. Once you are happy with your code you can always add it to a `.py` file to turn it into a library (to be called from other programs) or run it as a script.

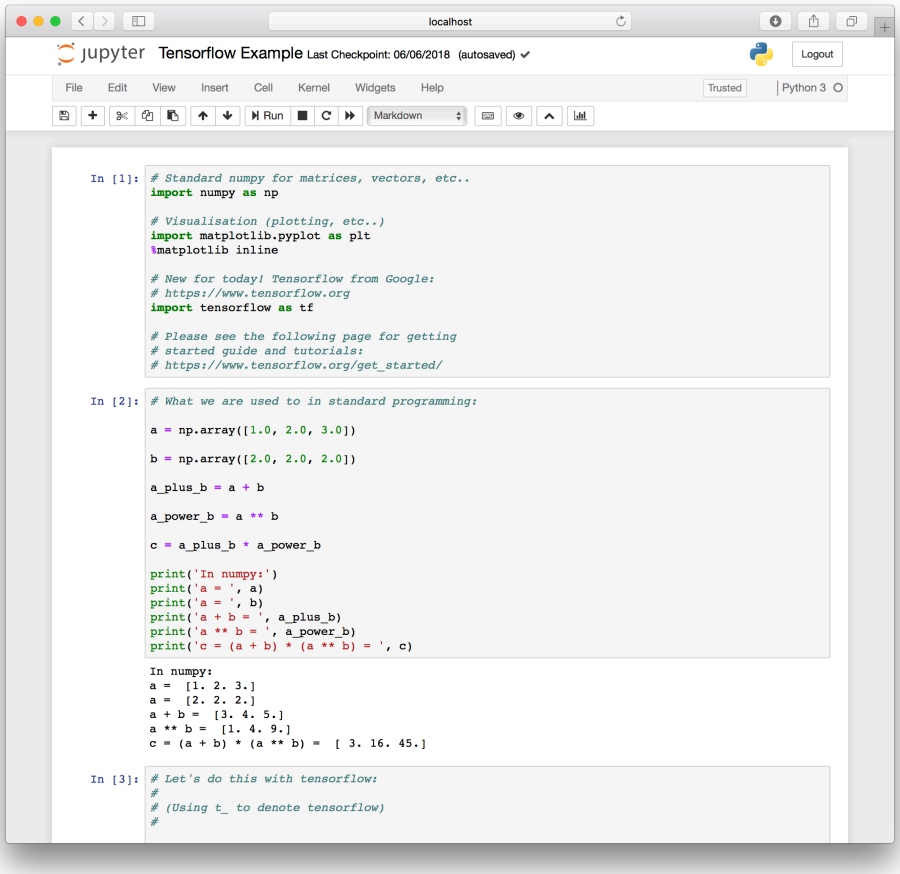

Figure 1: A screenshot of a running jupyter notebook.

Figure 1 shows a screenshot of a jupyter notebook in action. The notebook is running inside a browser. You are free to enter python code in the cells (the grey boxes denoted by "In[#]:" on the left hand side) and the output is shown underneath (e.g. the text between input cells 2 and 3).

Google have recently launched a new, freely available cloud service called colaboratory (https://colab.research.google.com) that runs a notebook server in the cloud. The same notebook as before is shown in colab in Figure 2.

Aside: *Local Installation of Jupyter*

If you want to install jupyter on your own computer you can follow the standard guide from the jupyter website http://jupyter.org/install. They recommend using the anaconda framework if you want to install `python` and jupyter for the first time. This should work across Windows, Linux and Mac. If you already have `python` and `pip` installed and comfortable with running things from the terminal I would recommend creating a virtual environment and installing jupyter with `pip`: **(Please ignore the following if you don't know what the commands mean!)**

```
# Create a new environments folder in your home directory
cd ~
mkdir environments
# Create a python environment called "jupyter-env" in the folder
```

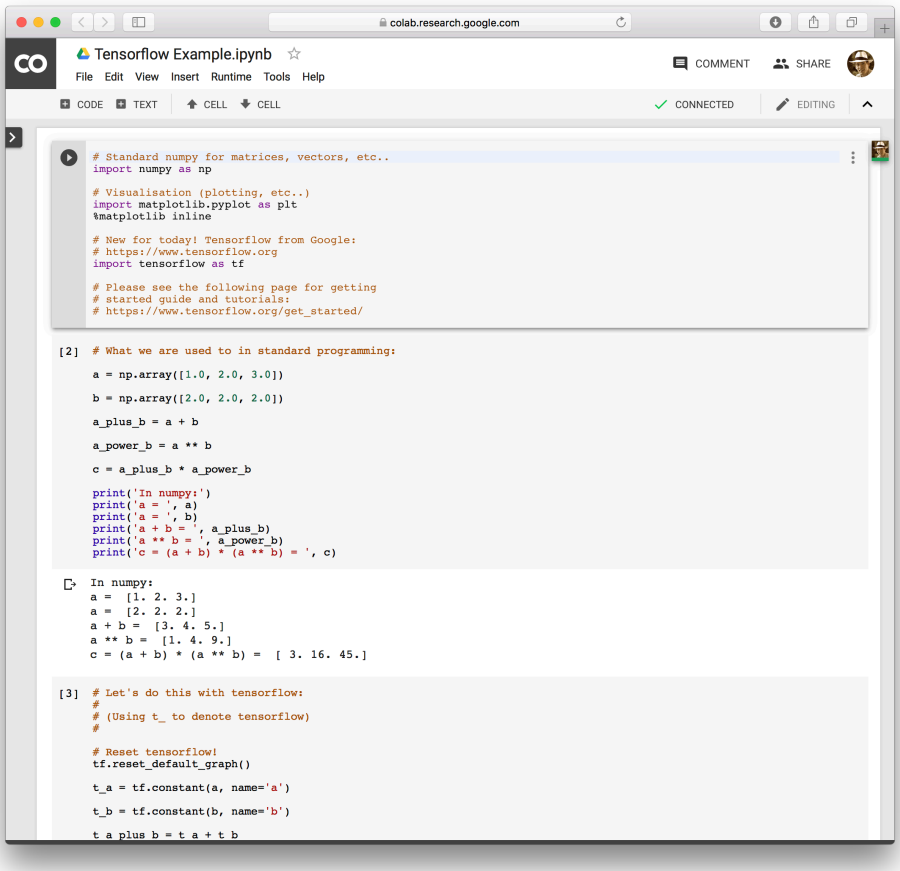

Figure 2: A screenshot of the same notebook running in the cloud on the new Google colaboratory notebook server. The resulting notebook can be saved locally on your computer or on your Google Drive if you have a gmail account. Notebooks can be shared an edited by multiple people in a similar manner to Google Docs.

```
cd environments
python3 -m venv jupyter-env
# Activate the environment
source ~/environments/jupyter-env/bin/activate
# Upgrade pip and install jupyter notebook and some numerical libraries
pip3 install --upgrade pip
pip3 install ipython numpy scipy matplotlib jupyter
```

To use the virtual environment you can then open a new terminal, activate the environment and run the jupyter server:

```
# Activate the environment
source ~/environments/jupyter-env/bin/activate
# Run the server
jupyter notebook
```

Numerical Python

In addition to being a general purpose, high-level programming language, `python` also provides special packages for scientific and numerical programming. The main packages are `numpy`, `scipy` and `matplotlib` which provide numerical and scientific methods as well as plotting functions respectively. With these packages, `python` provides a similar environment to `Matlab` and `R` with the advantages of running natively with the rest of the python infrastructure that provides all sorts of other packages (for example, easy access to the `opencv` computer vision libraries). It also provides a native interface to high-level machine learning libraries such as `TensorFlow`, `Caffe`, `PyTorch` and `Theano`. We will be looking at `TensorFlow` in this lab session but the *philosophy* of the programming environment extends to the other libraries.

Aside: *NumPy for Matlab or R Users*

There are a number of tutorials for programmers who are comfortable with other numerical languages moving to use `numpy`. If you are a `Matlab` programmer you might like to look at NumPy for Matlab users from the `scipy` website or the Matlab to NumPy Cheat Sheet which maintains a list of the `numpy` equivalent syntax for a range of common `Matlab` operations. The same can be found for `R` at NumPy for R users.

NumPy Overview

The `numpy` package provides a range of mathematical operations but in particular it provides linear algebra operations on vectors, matrices and tensors that are very useful for computer vision problems. It also removes the need for the `maths` module in the main python library so no need to import and use those old functions. We usually import the `numpy` module with the *alias* `np` so `np.some_function` can be used to call a `numpy` function.

The central object in `numpy` is the `np.ndarray` object. This is an N-dimensional array that can be used to store vectors, matrices or higher order tensors. They are much more efficient to compute with than nested `lists` from standard python. The arrays also have an explicit datatype for the contents of the array denoted as the `dtype`. We must be careful when creating and performing operations arrays that we are using the correct `dtype`. There are mappings to standard numerical types such that `dtype=np.float32` is a single precision floating point (e.g. `float` in C), `dtype=np.float64` is a double precision floating point (e.g. `double` in C) and `dtype=np.int` will be a integer of the native type on the current machine (usually 64-bit).

```
import numpy as np
a_vector = np.array([1.0, 2.0, 3.0])
a_matrix = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
print('a_vector =\n', a_vector)
print('a_matrix =\n', a_matrix)
a_vector_of_ints = np.array([1, 2, 3])
print('dtype of a_vector is', a_vector.dtype)
print('dtype of a_vector_of_ints is', a_vector_of_ints.dtype)
```

```
Output:
a_vector =
 [1. 2. 3.]
a_matrix =
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

```
dtype of a_vector is float64
dtype of a_vector_of_ints is int64
```

Getting Help with Functions

All `numpy` functions have documentation on the official website (NumPy and SciPy Documentation). The reference page for NumPy can be accessed directly here NumPy Reference.

Inside the notebook, you can access help directly by entering the function name followed by a question mark in a blank cell and running it. This will bring up a help panel with details on the function as shown in Figure 3. You can also access context sensitive help by pressing the `[tab]` key after the opening parenthesis of a function call or in a module or object (e.g. `np.[tab]`) to bring up a dropdown box with documentation or a list of module/class members. An example is given in Figure 4.

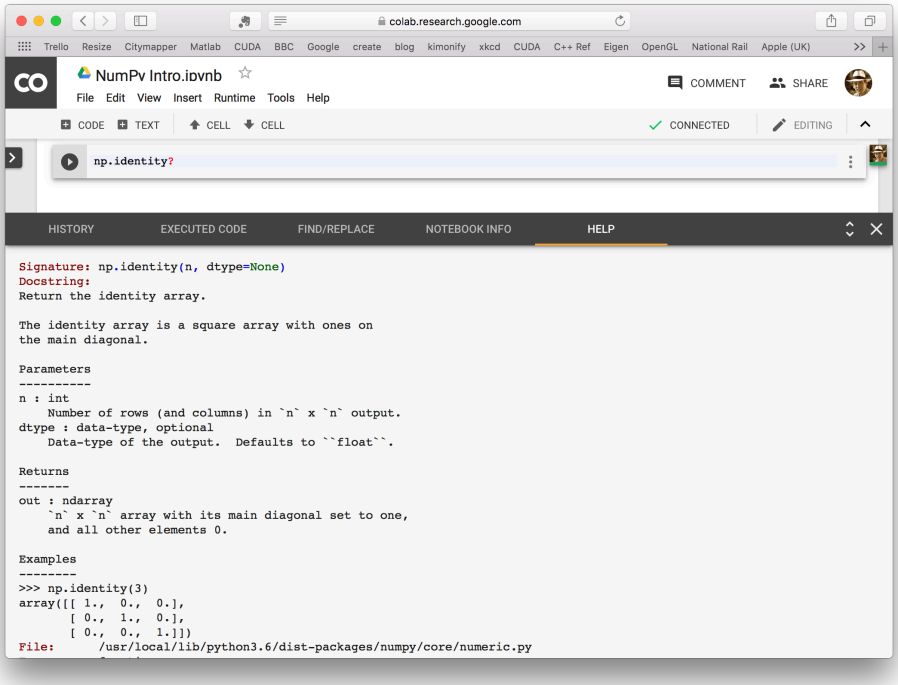

Figure 3: Accessing help on functions inside a notebook by entering the function name followed by a "`?`" in a cell and then running it. This will bring up a help panel at the bottom of the screen that you can scroll through and read. The panel can be closed with the cross on the right hand side of its title bar.

Constructors

There are a number of ways to construct arrays (similar in concept to construction in `Matlab` and `R`). The `np.array()` method above can be used to convert (nested) lists. The following are also valid:

```
np.empty(size) # Creates an empty array of size (list or tuple)
np.zeros([M, N]) # Creates an M x N matrix of zeros
np.ones([N]) # Creates a N long vector of ones
np.identity(size) # Creates an identity matrix of size (list or tuple)
np.fill(data) # Use this after np.empty() to fill array with data
```

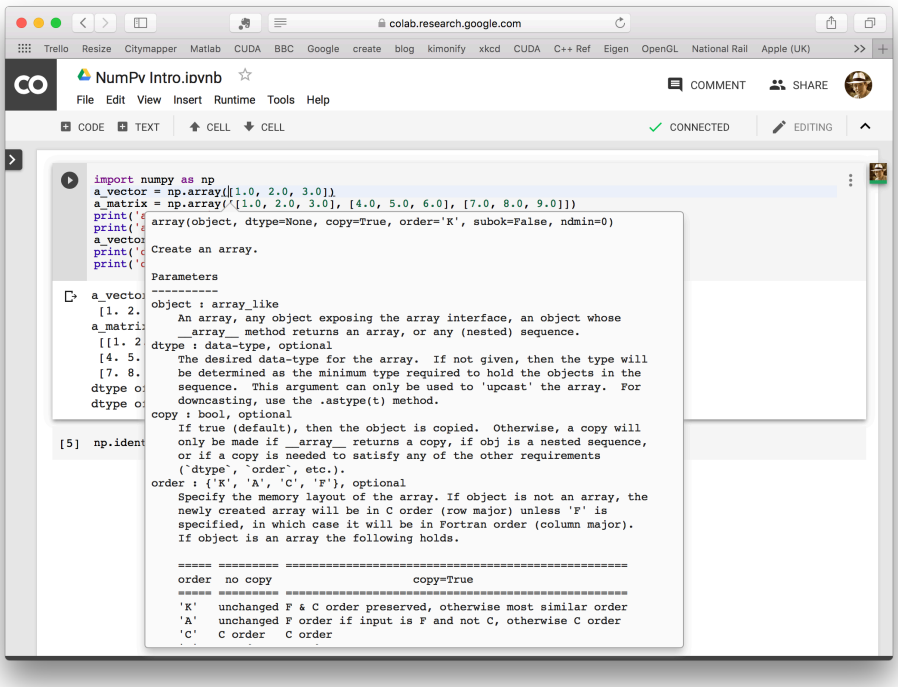

Figure 4: Context sensitive help is also available by pressing the `[tab]` key which will bring up help on the arguments of a function or a list of members of the class/module.

Operations

The standard operations on arrays are element-wise. This means that sizes of arrays need to match - there are *"broadcasting"* rules, which we will look at shortly, that apply when sizes to do not match (e.g. `a_matrix + 1.0` will add one to all entries in `a_matrix`). **Note:** While it makes sense for operations like addition and subtraction to be element-wise, we need to be careful with multiplication. The $*$ operator will perform element-wise multiplication whereas the `.dot()` operator will perform **matrix multiplication**:

```
a_vector = np.array([1.0, 2.0, 3.0])
a_matrix = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
# Create a 3x3 matrix of random integers..
b_matrix = np.random.randint(size=[3, 3], low=-5, high=5)
print('a_vector =\n', a_vector)
print('a_matrix =\n', a_matrix)
print('b_matrix =\n', b_matrix)
# Element-wise multiplication
print('a_matrix * b_matrix =\n', a_matrix * b_matrix)
# Matrix multiplication
print('a_matrix.dot(b_matrix) =\n', a_matrix.dot(b_matrix))
# Matrix vector multiplication
print('a_matrix.dot(a_vector) =\n', a_matrix.dot(a_vector))
# Matrix transpose
print('a_matrix.T =\n', a_matrix.T)
# Note the output of operations on a floating point array
# and an array of integers is a floating point array..
```

Output:

```
a_vector =
 [1. 2. 3.]
a_matrix =
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
b_matrix =
 [[ 0 -3  4]
 [-3  4 -4]
 [ 3 -2  4]]
a_matrix * b_matrix =
 [[  0.  -6.  12.]
 [-12.  20. -24.]
 [ 21. -16.  36.]]
a_matrix.dot(b_matrix) =
 [[ 3. -1.  8.]
 [ 3. -4. 20.]
 [ 3. -7. 32.]]
a_matrix.dot(a_vector) =
 [14. 32. 50.]
a_matrix.T =
 [[1. 4. 7.]
 [2. 5. 8.]
 [3. 6. 9.]]
```

Indexing and Slicing

Note: Unlike some languages (e.g. `Matlab`) `python` (sensibly!) uses zero based indexing and this applies to `numpy` as well. Indexing is performed with square brackets such as `a_matrix[index0, index1]`. A colon `:` can be used to indicate all of the elements in a dimension or with numbers to specify a range. For example, `a_vector[n:m]` will return entries $\mathfrak{n}$ (starting at zero) to $(\mathfrak{m}-1)$, so $(\mathfrak{m}-\mathfrak{n})$ elements. It is important to remember that the end of the range should be one after the index that you want (this is standard python notation for ranges).
If you leave a number out it defaults to the start of the dimension or the end of the dimension respectively so that `a_vector[:m]` returns the first $\mathfrak{m}$ elements and `a_vector[3:]` returns all the elements from the 4^{th} to the end. Negative indices at the end count backwards from the end so `a_vector[:-1]` returns all but the last element. Indexing starts from the left-most dimension so if you have a $2\times4\times3\times6$ tensor array and you specify two indices, the appropriate 3×6 sub-array will be returned (the remaining entries are all assumed to be `:`). Let's see some examples:

```
a_vector = np.array([n+1 for n in range(6)])
print('a_vector =', a_vector)
print('a_vector[3:] =', a_vector[3:])
print('a_vector[:-2] =', a_vector[:-2])
```

```
Output:
a_vector = [1 2 3 4 5 6]
a_vector[3:] = [4 5 6]
a_vector[:-2] = [1 2 3 4]
```

```
print('a_matrix =\n', a_matrix)
# These next two are the same!
print('a_matrix[0] =\n', a_matrix[0])
print('a_matrix[0,:] =\n', a_matrix[0,:])
print('a_matrix[:,1:2] =\n', a_matrix[:,1:2])
print('a_matrix[1:,:2] =\n', a_matrix[1:,:2])
```

```
Output:
a_matrix =
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
a_matrix[0] =
 [1. 2. 3.]
a_matrix[0,:] =
 [1. 2. 3.]
a_matrix[:,1:2] =
 [[2.]
 [5.]
 [8.]]
a_matrix[1:,:2] =
 [[4. 5.]
 [7. 8.]]
```

Helpful Tip: If in doubt about indexing and slicing then make a random tensor of the right size in a new notebook cell (using `np.random.randn([A,B,C])` to make an $A \times B \times C$ tensor) and then try the slicing printing out the resulting arrays and their shapes to make sure you have got the correct slice. *This is a common source of bugs since it is easy to make a mistake - it is always worth double checking!*

You can also use *logical indexing* when a Boolean array of the same size as the array (after broadcasting rules have been applied) specifies that an element should be returned or ignored if the corresponding Boolean element is `True` or `False`:

```
a_vector = np.random.randint(size=[6], low=-10, high=10)
print('a_vector =', a_vector)
logical_index = np.array([True, False, True, True, False, True])
print('a_vector[logical_index] =', a_vector[logical_index])
# The logical index can be the result of a Boolean operation..
print('a_vector[a_vector > 0] =', a_vector[a_vector > 0])
```

```
Output:
a_vector = [ 8  6 -6  1 -5  8]
a_vector[logical_index] = [ 8 -6  1  8]
a_vector[a_vector > 0] = [8 6 1 8]
```

Concatenation

Arrays can be joined together as long as their shapes match appropriately (after broadcasting rules have been applied). The standard operation for this is the `np.concatenate()` operation that takes a list of arrays to combine and an axis (dimension) to combine over:

```
a_matrix = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
a_vector = np.array([10.0, 20.0, 30.0])
b_vector = np.array([6.0, 5.0, 4.0])
print('a_matrix =\n', a_matrix)
print('a_matrix.shape =', a_matrix.shape)
# Note: we need to specify the new dimension in the vector
# to ensure the shapes match when we are concatenating..
joined = np.concatenate([a_matrix, a_vector[np.newaxis,:]], axis=0)
print('joined =\n', joined)
# There are also shorter notations specifically for row- and
# column-wise concatenations:
print('np.r_[a_vector, b_vector] =\n', np.r_[a_vector, b_vector])
print('np.c_[a_vector, b_vector] =\n', np.c_[a_vector, b_vector])
# The np.stack function will stack ND arrays of the same size
# to return an array of dimension N+1..
print('np.stack([a_vector, b_vector]) =\n', np.stack([a_vector, b_vector]))
```

```
Output:
a_matrix =
 [[1. 2. 3.]
 [4. 5. 6.]]
a_matrix.shape = (2, 3)
joined =
 [[ 1.  2.  3.]
 [ 4.  5.  6.]
 [10. 20. 30.]]
np.r_[a_vector, b_vector] =
 [10. 20. 30.  6.  5.  4.]
np.c_[a_vector, b_vector] =
 [[10.  6.]
 [20.  5.]
 [30.  4.]]
np.stack([a_vector, b_vector]) =
 [[10. 20. 30.]
 [ 6.  5.  4.]]
```

Vectorisation

As with many other numerical programming languages, linear algebra operations are much more efficient when performed in a vectorised manner. This is the specification of operations over arrays directly rather than the individual elements. The simplest example might be summing two vectors. Both of the following are equivalent but the vectorised operation is much faster:

```
a_vector = np.array([1.0, 2.0, 3.0])
b_vector = np.array([6.0, 5.0, 4.0])
# Create a vector of length 3 to hold the output..
c_loop_result = np.zeros([3])
# Sum the vectors with a loop..
for n in range(3):
```

```
    c_loop_result[n] = a_vector[n] + b_vector[n]
print('c_loop_result = \n', c_loop_result)
# Vectorised approach
print('c_vectorised_result = \n', a_vector + b_vector)
```

```
Output:
c_loop_result =
 [7. 7. 7.]
c_vectorised_result =
 [7. 7. 7.]
```

Aside: The above is not a pythonic way to perform the loop since we had to explicitly state the size of the array. The following shows the use of the `zip` and `enumerate` functions to perform the same loop in a better fashion (although remember the vectorised operation is the proper way for `numpy` arrays):

```
# We can loop over the elements of an array directly..
for a in a_vector:
    print('direct loop a =', a)
# We can also get the current index using enumerate..
for n, a in enumerate(a_vector):
    print('enumerate loop n =', n, ', a =', a)
# We can loop over elements of multiple arrays
# if they are the same size using zip..
for (a, b) in zip(a_vector, b_vector):
    print('zip loop a =', a, ', b =', b)
# We can combine these all together to make our
# previous loop more "pythonic"..
c_loop_result.fill(0.0)
for n, (a, b) in enumerate(zip(a_vector, b_vector)):
    c_loop_result[n] = a + b
print('c_loop_result =', c_loop_result)
```

```
Output:
direct loop a = 1.0
direct loop a = 2.0
direct loop a = 3.0
enumerate loop n = 0 , a = 1.0
enumerate loop n = 1 , a = 2.0
enumerate loop n = 2 , a = 3.0
zip loop a = 1.0 , b = 6.0
zip loop a = 2.0 , b = 5.0
zip loop a = 3.0 , b = 4.0
c_loop_result = [7. 7. 7.]
```

```
# NOTE: You cannot change the value provide by the array iterator.
# The c_loop_result below will not be correct..
c_loop_result.fill(0.0)
for (a, b, c) in zip(a_vector, b_vector, c_loop_result):
    c = a + b
print('c_loop_result =', c_loop_result)
```

```
Output:
c_loop_result = [0. 0. 0.]
```

Note: Standard *list comprehensions* in python will not return `numpy` arrays by default:

```
a_vector = np.array([1.0, 2.0, 3.0])
# The following will not be a numpy array..
new_vector = [a * 2.0 for a in a_vector]
print('new_vector =', new_vector)
print('type(new_vector) =', type(new_vector))
# ..unless we explicitly cast it to be one..
np_new_vector = np.array([a * 2.0 for a in a_vector])
print('np_new_vector =', np_new_vector)
print('type(np_new_vector) =', type(np_new_vector))
# Better to use a vectorised call..
better_new_vector = a_vector * 2.0
print('better_new_vector =', better_new_vector)
```

```
Output:
new_vector = [2.0, 4.0, 6.0]
type(new_vector) = <class 'list'>
np_new_vector = [2. 4. 6.]
type(np_new_vector) = <class 'numpy.ndarray'>
better_new_vector = [2. 4. 6.]
```

Vectorised code will always be faster than the equivalent loop operation since it allows for more efficient implementations to be used (e.g. `BLAS` library calls - the same library used by `Matlab`). It is not necessarily true that vectorised code will be clearer to someone reading the program, however, and it can be easy to make mistakes.

Helpful Tip: When writing a complicated piece of vectorised code, go to a new notebook and write a for loop version that you are sure is correct. Then use this code to check your vectorised version on some test data. Once you are happy the vectorised code is correct you can add the vectorised function into your main code. This is an example of a concept called *unit testing* and is very good practice when writing research code. We are often trying new ideas and we don't know if our algorithm will work or how well it should perform. If we want to see that our idea works we need to be confident that the code performs correctly so that the results we obtain can be trusted and are not spurious due to errors in the code.

Broadcasting

Every array has a set size that can be accessed by the `.shape` property. What happens when we perform an operation that expects arrays to be of a certain shape but they are not? These are the *broadcasting* rules we mentioned above. There are two key rules that are applied:

1. If two arrays have different numbers of dimensions then pad out the lower dimensional one with length 1 dimensions at the start (now both have the same number of dimensions).
2. If you match a length 1 dimension to one of length $N > 1$ then the length 1 dimension acts as though the value is repeated N times (this is performed in a memory efficient manner).

We have actually already seen some examples of broadcasting in previous code but let's look at some formal examples now:

```
a_vector = np.array([1.0, 2.0, 3.0])
a_longer_vector = np.ones([6])
a_matrix = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
# The shape property returns the dimensionalities..
print('a_vector.shape =', a_vector.shape)
print('a_longer_vector.shape =', a_longer_vector.shape)
print('a_matrix.shape =', a_matrix.shape)
# The following will give an error since the shapes do not match
result = a_vector + a_longer_vector
```

```
Output:
a_vector.shape = (3,)
a_longer_vector.shape = (6,)
a_matrix.shape = (3, 3)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-59a4c434611d> in <module>()
      7 print('a_matrix.shape =', a_matrix.shape)
      8 # The following will give an error since the shapes do not match
----> 9 result = a_vector + a_longer_vector

ValueError: operands could not be broadcast together with shapes (3,) (6,)
```

```
# What happens when you add a 3x3 matrix and a 3 vector?
broadcast_result = a_matrix + a_vector
print('a_vector =', a_vector)
print('a_matrix =\n', a_matrix)
print('broadcast_result =\n', broadcast_result)
```

```
Output:
a_vector = [1. 2. 3.]
a_matrix =
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
broadcast_result =
 [[ 2.  4.  6.]
 [ 5.  7.  9.]
 [ 8. 10. 12.]]
```

How was this obtained?

```
# First apply rule one to insert a new dimension at the
# start of the vector so that it is also 2D..
print('a_vector.shape =', a_vector.shape)
extended_vector = a_vector[np.newaxis, :]
print('extended_vector.shape =', extended_vector.shape)
# Now apply rule two - the new first dimension must be repeated
# 3 times to match the 3x3 matrix
repeated_vector = np.repeat(extended_vector, repeats=3, axis=0)
print('repeated_vector.shape =', repeated_vector.shape)
```

```
print('repeated_vector =\n', repeated_vector)
check_broadcast_result = a_matrix + repeated_vector
print('check_broadcast_result =\n', check_broadcast_result)
```

```
Output:
a_vector.shape = (3,)
extended_vector.shape = (1, 3)
repeated_vector.shape = (3, 3)
repeated_vector =
 [[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]]
check_broadcast_result =
 [[ 2.  4.  6.]
 [ 5.  7.  9.]
 [ 8. 10. 12.]]
```

We can also be explicit about which dimension to repeat under the broadcast rules by using the `np.newaxis` command to specify which dimension to add. Then the first broadcast rule will be skipped and the second will repeat the dimension as appropriate:

```
# What if I want to change the broadcast dimension?
# We can make it explicit using np.newaxis..
broadcast_second_dim = a_matrix + a_vector[:, np.newaxis]
print('a_vector =', a_vector)
print('a_matrix =\n', a_matrix)
print('broadcast_second_dim =\n', broadcast_second_dim)
```

```
Output:
a_vector = [1. 2. 3.]
a_matrix =
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
broadcast_second_dim =
 [[ 2.  3.  4.]
 [ 6.  7.  8.]
 [10. 11. 12.]]
```

TensorFlow

The main purpose of this lab is to introduce a new paradigm for numerical programming - the use of computational graphs. Traditional numerical programming (e.g. `numpy`, `Matlab`, `R`) is performed using *imperative* programming where the commands are executed in the sequence specified in the source code and the operations are evaluated directly. Instead, frameworks such as `TensorFlow` provide an interface for *declarative* where the operations are not evaluated directly but are used to build up an object that will later perform the computations. This is best illustrated with an example so we will work through an example together and then there will an opportunity to try this framework out for yourself. For further information about `TensorFlow` there is a very detailed website including a getting started guide and tutorials https://www.tensorflow.org/get_started/.

Setting up

In a similar manner to `numpy` we import `TensorFlow` using the alias `tf` so all functions that start with `tf` and in `TensorFlow` and those that start `np` are in `numpy`.

```
# Standard numpy for matrices, vectors, etc..
import numpy as np
# Visualisation (plotting, etc..)
import matplotlib.pyplot as plt

# Tensorflow from Google:
# https://www.tensorflow.org
import tensorflow as tf
# The following works out if we are running on a
# local Jupyter server or in Google's colab..
try:
    in_colab = False
    import google.colab
    in_colab = True
except:
    pass
# Use the following to access tensorboard when running on colab
if in_colab:
    !pip install -U tensorboardcolab
    from tensorboardcolab import *
else:
    # Use to make plots appear inline with output in jupyter
    %matplotlib inline
```

Standard Programming

In standard programming we are used to everything being evaluated directly and in sequence; hopefully the following should not be too surprising!

```
# What we are used to in standard programming:
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a_plus_b = a + b
a_power_b = a ** b
c = a_plus_b * a_power_b
print('In numpy:')
print('a = ', a)
print('a = ', b)
print('a + b = ', a_plus_b)
print('a ** b = ', a_power_b)
print('c = (a + b) * (a ** b) = ', c)
```

```
Output:
In numpy:
a =  [1. 2. 3.]
```

```
a =  [2. 2. 2.]
a + b =  [3. 4. 5.]
a ** b =  [1. 4. 9.]
c = (a + b) * (a ** b) =  [ 3. 16. 45.]
```

Now in TensorFlow

What happens when we try the same in `TensorFlow`? (**Note:** We use the prefix `t_` to indicate `TensorFlow` variables for clarity but this is not a requirement)

```
# Let's do this with tensorflow!
# First reset tensorflow!
tf.reset_default_graph()
# Now run equivalent operations..
t_a = tf.constant(a, name='a')
t_b = tf.constant(b, name='b')
t_a_plus_b = t_a + t_b
t_a_power_b = t_a ** t_b
t_c = t_a_plus_b * t_a_power_b
print('In tensorflow:')
print('a = ', t_a)
print('a = ', t_b)
print('a + b = ', t_a_plus_b)
print('a ** b = ', t_a_power_b)
print('c = (a + b) * (a ** b) = ', t_c)
```

```
Output:
In tensorflow:
a =  Tensor("a:0", shape=(3,), dtype=float64)
a =  Tensor("b:0", shape=(3,), dtype=float64)
a + b =  Tensor("add:0", shape=(3,), dtype=float64)
a ** b =  Tensor("pow:0", shape=(3,), dtype=float64)
c = (a + b) * (a ** b) =  Tensor("mul:0", shape=(3,), dtype=float64)
```

We note that we get strange types out and the answers we had in `numpy` do not appear at all. This is because of the *declarative* interface of `TensorFlow`. Instead of performing the operations directly, we have specified a *computational graph* of operations that represent the computations we wish to perform. We can actually visualise this graph directly through the `tensorboard` interface provided by Google. The `name=` parameters help us during the visualisation.

Visualise the Graph

The following operations allow you to see the graph in `tensorboard` and the resulting graph is shown in Figure 5.

```
# Can ignore this code for now, this just creates a
# visualisation file so we can see what is going on!
with tf.Session() as session:
    if in_colab:
        tbc = TensorBoardColab()
        summary_file_writer = tbc.get_writer()
```

```
        summary_file_writer.add_graph(session.graph)
        summary_file_writer.flush()
        tbc.close()
    else:
        summary_file_writer = tf.summary.FileWriter(
            'visualisation_files', session.graph)
        summary_file_writer.flush()
# This has created a folder called "visualisation_files"
# with some data in it that we can view with the command:
#
# tensorboard --logdir=visualisation_files
#
# If in colab, the TensorBoardColab call will output a link to follow
```

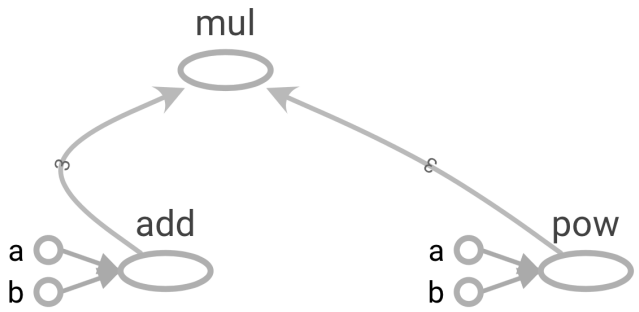

Figure 5: The computational graph corresponding to the `TensorFlow` operations of the same functions as the `numpy` code. If we look back at the code we can see that it makes sense. The two inputs `a` and `b` are combined together using both a sum operation and a power operation. The results of these two operations are then combined using a multiplication to produce the final output.

But Why?

So why would we want to do this? Well first we should check that it does what we think it should do:

```
# First - does it actually work?
# In order to use the graph we have to "run" it!
# In tensorflow, we need to run things inside of a session

# This line creates a session..
with tf.Session() as session:
    # Inside here we can use the "session" object created..

    # Let's run our graph to actually compute something!
    result = session.run(t_c)
    print("result = ", result)

# Outside of the "with" statement the session object is
# deleted and we can no longer use it
#
```

```
# This line would cause an error:
# result_again = session.run(t_c)
```

```
Output:
result =  [ 3. 16. 45.]
```

We can compare everything to ensure it's consistent:

```
# Let's check everything makes sense:
print('In numpy:')
print('a = ', a)
print('a = ', b)
print('a + b = ', a_plus_b)
print('a ** b = ', a_power_b)
print('c = (a + b) * (a ** b) = ', c)
with tf.Session() as session:
    print('In tensorflow session:')
    print('a = ', session.run(t_a))
    print('a = ', session.run(t_b))
    print('a + b = ', session.run(t_a_plus_b))
    print('a ** b = ', session.run(t_a_power_b))
    print('c = (a + b) * (a ** b) = ', session.run(t_c))
# Everything should be the same!
```

```
Output:
In numpy:
a =  [1. 2. 3.]
a =  [2. 2. 2.]
a + b =  [3. 4. 5.]
a ** b =  [1. 4. 9.]
c = (a + b) * (a ** b) =  [ 3. 16. 45.]
In tensorflow session:
a =  [1. 2. 3.]
a =  [2. 2. 2.]
a + b =  [3. 4. 5.]
a ** b =  [1. 4. 9.]
c = (a + b) * (a ** b) =  [ 3. 16. 45.]
```

So the computation graph seems to work but isn't it more effort than the `numpy` version?

Now we ask ourselves: What if we were doing an optimisation?

Performing an Optimisation

We want to fit a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ to a set of numbers $\mathbf{X} = \{x_0, x_1, \ldots, x_{N-1}\}$

If we assume the numbers are i.i.d. (identically and independently distributed) samples from a Gaussian then the likelihood of $\mathbf{X}$ is given by:

$$
\begin{aligned}
p(X) &= p(x_0) \cdot p(x_1) \cdot \cdots \cdot p(x_{N-1}) && (1)\\
&= \mathcal{N}(x_0 \,|\, \mu, \sigma^2) \cdot \mathcal{N}(x_1 \,|\, \mu, \sigma^2) \cdot \cdots \cdot \mathcal{N}(x_{N-1} \,|\, \mu, \sigma^2) && (2)\\
&= \prod_{n=0}^{N-1} \mathcal{N}(x_n \,|\, \mu, \sigma^2) && (3)\\
&= \prod_{n=0}^{N-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_n-\mu)^2}{2\sigma^2}\right) && (4)
\end{aligned}
$$

In our case we can find an analytic solution for

$$
\begin{aligned}
\mu^* &= \arg\max_\mu \log p(X) && (5)\\
{\sigma^*}^2 &= \arg\max_{\sigma^2} \log p(X) && (6)
\end{aligned}
$$

But let's pretend that the problem was more complicated and we needed to use *optimisation* to solve the problem.

To perform numerical optimisation you need to be able to calculate gradients of the objective function ($\log p(X)$) wrt the parameters that you are optimising (μ and σ^2).

Let's see how to do this in `TensorFlow`:

```
# First let's generate some numbers to fit the data to..
# How many values of x?
N = 20
# Pick the real mean and variance..
mu_true = 2.5
sigma_true = 1.5
x_n = np.random.normal(mu_true, sigma_true, N)
np.set_printoptions(precision=3, linewidth=50)
print('X = \n', np.transpose(x_n))
```

```
Output:
X =
 [1.331 3.775 4.025 1.505 1.199 1.839 2.445 4.915
 1.6   2.333 4.764 4.06  3.11  0.348 1.878 1.407
 4.618 4.927 1.862 1.095]
```

We are now going to build our tensorflow graph but we are going to account for the fact that μ and σ^2 are no longer constants since we wish to vary their values to find the maximum of $\log p(X)$. With numerical optimisation, we need to start with a guess for the values of μ and σ^2; in this case, we will start with

$$
\begin{aligned}
\mu_{\text{initial}} &= 1 && (7)\\
\sigma^2_{\text{initial}} &= 1 && (8)
\end{aligned}
$$

Top Tip! Care needs to be taken with σ since it can only be a positive value (unlike μ which can be any real number). In general, tensorflow variables can be positive or negative. In this example we square the value of `t_sigma` before using it to ensure that `t_sigma_2` is a positive value but we shouldn't, therefore, use the value for `t_sigma` directly in calculations..

As a reminder, we want to find:

$$\log p(X) = \sum_{n=0}^{N-1} -\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{(x_n - \mu)^2}{2\sigma^2} \tag{9}$$

```
# Reset tensorflow to remove our old a, b, etc..
tf.reset_default_graph()

# Our initial guesses..
mu_initial_guess = 1.0
sigma_initial_guess = np.sqrt(1.0)

# The data to fit to
t_x_n = tf.constant(x_n, name='X')

# Note: mu and sigma are now *variables* not constants!
# We need to specify their data type and initial value..
t_mu = tf.Variable(mu_initial_guess,
                   dtype=tf.float64,
                   name="mu")
t_sigma = tf.Variable(sigma_initial_guess,
                      dtype=tf.float64,
                      name="sigma")

# Note: this step is important - don't use t_sigma directly!!
t_sigma_2 = t_sigma ** 2.0

# Calculate log p(X) terms..
t_x_minus_mu_2 = (t_x_n - t_mu) ** 2.0
t_denom = 2.0 * t_sigma_2
t_sigma_term = - 0.5 * tf.log(2.0 * np.pi * t_sigma_2)
t_log_P_terms = t_sigma_term - (t_x_minus_mu_2 / t_denom)

# The sum is performed by a reduction in tensorflow
# (since a vector goes in and a scalar comes out)
# but this is effectively the same as np.sum(...)
t_log_P = tf.reduce_sum(t_log_P_terms)
```

```
# Let's just check that we calculated things correctly:
with tf.Session() as session:
    # IMPORTANT! Need to run this at the start to
    # initialise the values for the variables
    # t_mu and t_sigma. You will get an error if
    # you forget!
    session.run(tf.global_variables_initializer())

    test_value = session.run(t_log_P)
    print('Tensorflow log p(X) = ', test_value)
    print('(using initial guesses for mu and sigma)\n')
```

```
# Check with scipy..
from scipy.stats import norm
check_value = np.sum(norm.logpdf(x_n,
                                 mu_initial_guess,
                                 sigma_initial_guess))
print('Value from scipy stats package = ', check_value)
assert(np.isclose(test_value, check_value))
print('\nEverything working!')
```

```
Output:
Tensorflow log p(X) =  -66.43296888468996
(using initial guesses for mu and sigma)

Value from scipy stats package =  -66.43296888468996

Everything working!
```

Great, so everything is working. Well we have only checked for the initial parameters. We can also check that the values would be the same if we changed the parameter values:

```
# We can even go crazy and check with different
# values of the parameters..
mu_new_test_value = 3.3
sigma_new_test_value = 0.5

with tf.Session() as session:
    # IMPORTANT! (see above..)
    session.run(tf.global_variables_initializer())

    # Change the values of the variables while the
    # session is running..
    session.run(t_mu.assign(mu_new_test_value))
    session.run(t_sigma.assign(sigma_new_test_value))

    test_value = session.run(t_log_P)
    print('New tensorflow log p(X) = ', test_value)
    print('(using new values for mu and sigma)\n')

# Check with scipy..
from scipy.stats import norm
check_value = np.sum(norm.logpdf(x_n,
                                 mu_new_test_value,
                                 sigma_new_test_value))
print('New value from scipy stats package = ', check_value)
assert(np.isclose(test_value, check_value))
print('\nEverything working!')
```

```
Output:
New tensorflow log p(X) =  -104.42223609227118
(using new values for mu and sigma)
```

```
New value from scipy stats package =  -104.42223609227116

Everything working!
```

Calculating Gradients

So finally, we get to the advantage of `TensorFlow`!

Now, we can calculate the objective function and we can calculate the value of the objective when changing the input parameters.

This is great for optimisation (since we are going to need to change the parameters to increase the objective) but what we really need for the optimisation is to calculate the **gradient of the objective wrt to the parameters**. Let's see how to do that in `TensorFlow`:

```
with tf.Session() as session:
    # IMPORTANT! (see above..)
    session.run(tf.global_variables_initializer())

    t_gradient_wrt_mu = tf.gradients(t_log_P,
                                     t_mu)
    t_gradient_wrt_sigma = tf.gradients(t_log_P,
                                        t_sigma)

    grad_mu = session.run(t_gradient_wrt_mu)
    grad_sigma = session.run(t_gradient_wrt_sigma)
    print('Gradient wrt mu = ', grad_mu)
    print('Gradient wrt sigma = ', grad_sigma)
```

```
Output:
Gradient wrt mu =  [33.03373737424015]
Gradient wrt sigma =  [76.10839644119301]
```

So `TensorFlow` has calculated the gradients for use! We can check that result. Remember we have:

$$\log p(X) = \sum_{n=0}^{N-1} -\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{(x_n-\mu)^2}{2\sigma^2} \tag{10}$$

$$= -\frac{N}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(x_n-\mu)^2 \tag{11}$$

So for μ we have:

$$\frac{\partial \log p(X)}{\partial \mu} = -0 - \frac{1}{2\sigma^2}\frac{\partial}{\partial \mu}\sum_{n=0}^{N-1}(x_n - \mu)^2 \tag{12}$$

$$= -\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}\frac{\partial}{\partial \mu}(x_n - \mu)^2 \tag{13}$$

$$= -\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}2(x_n - \mu)\frac{\partial}{\partial \mu}(x_n - \mu) \tag{14}$$

$$= \frac{1}{\sigma^2}\sum_{n=0}^{N-1}(x_n - \mu) \tag{15}$$

where we used the *chain rule* a number of times. We can check using these results with `numpy`:

```
# numpy check of gradient wrt mu
grad_mu_check = np.sum(x_n - mu_initial_guess) / \
                (sigma_initial_guess ** 2)

print('Our analytic gradient wrt mu = ', grad_mu_check)
print('Tensorflow gradient wrt mu = ', grad_mu)
assert(np.isclose(grad_mu, grad_mu_check))
print('\nExcellent! tensorflow calculated the gradient for us :)')
```

```
Output:
Our analytic gradient wrt mu =  33.03373737424015
Tensorflow gradient wrt mu =  [33.03373737424015]

Excellent! tensorflow calculated the gradient for us :)
```

We should now all be in awe!!

This might seem like something trivial but hopefully you can see that actually quite a lot of maths and then coding went into determining the gradient.

In fact, you can do the same to check the value for the gradient wrt σ^2.

When we calculated the result using the chain rule. Since tensorflow built up a graph of the operations, it is able to apply the chain rule results for us automatically.

This:

$$\log p(X) = \sum_{n=0}^{N-1} -\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{(x_n - \mu)^2}{2\sigma^2} \tag{16}$$

has become the computational graph of Figure 6.

For example, the `pow` operation represents $r = a^b$ for the inputs a, b and result r. Tensorflow then knows that $\frac{\partial r}{\partial a} = ba^{b-1}$, and by chaining these operations together it can work backwards through the graph (from $\log p(X)$ at the top to μ at the bottom) to calculate the gradient.

Therefore, the tensorflow graph has multiple uses. A forward pass can calculate the objective for the current set of parameters and a backwards pass can calculate the gradients of an objective wrt any of the parameters.

Optimisation in TensorFlow

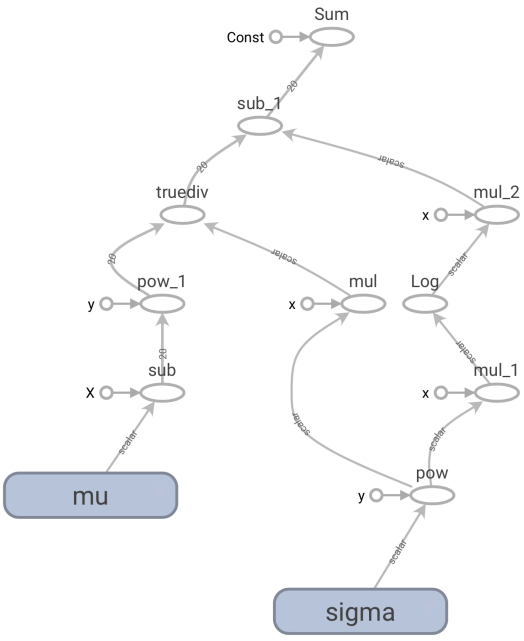

Figure 6: The computational graph corresponding to the calculation of the log likelihood function of Equation 16.

But the fun doesn't end here! In fact `TensorFlow` has actually done all the work to do the optimisation part, not just calculate the derivatives. So we can now run a full optimisation with our graph and it will use the gradients internally.

```
# Create a gradient descent optimiser that uses a
# certain step size (learning_rate)..
optimiser = tf.train.GradientDescentOptimizer(learning_rate=0.05)

# We want to maximise log p(X) therefore we
# need to minimise - log p(X)
t_objective = - t_log_P

# We want to optimise wrt mu and sigma
vars_to_optimise = [t_mu, t_sigma]

minimize_operation = optimiser.minimize(t_objective,
                                        var_list=vars_to_optimise)

# Number of iterations to perform
num_iterations = 50

with tf.Session() as session:
    # IMPORTANT! (see above..)
    session.run(tf.global_variables_initializer())

    # Run a number of iterations of gradient descent..
    for iteration in range(num_iterations):
        # At each iteration evaluate the minimize_operation
        # to perform the gradient descent step and also
        # keep track of the current value..
```

```
        step, cost = session.run([minimize_operation, t_log_P])

        # Print out the value of log P every 10 iterations..
        if ((iteration + 1) % 10 == 0):
            print('iter %4d, log P(X) = %0.3f' %
                  (iteration + 1, cost))

    # Get the final results of the optimisation..
    mu_optimised = session.run(t_mu)
    sigma_optimised = session.run(t_sigma)

    print('\nAfter optimisation:')
    print('Tensorflow mu = ', mu_optimised)
    print('Tensorflow sigma = ', sigma_optimised)

print('\nAnalytic estimates:')
print('Estimated mu = ', np.mean(x_n))
print('Estimated std = ', np.std(x_n))

print('\nGround truth values:')
print('True mu = ', mu_true)
print('True sigma = ', sigma_true)
```

```
Output:
iter   10, log P(X) = -43.105
iter   20, log P(X) = -35.690
iter   30, log P(X) = -35.690
iter   40, log P(X) = -35.690
iter   50, log P(X) = -35.690

After optimisation:
Tensorflow mu =  2.6516868687120074
Tensorflow sigma =  1.4413016026439345

Analytic estimates:
Estimated mu =  2.6516868687120074
Estimated std =  1.4413016026439343

Ground truth values:
True mu =  2.5
True sigma =  1.5
```

Excellent! We agree with the analytic estimate!

Using Different Data

Of course, the values don't match the true estimate since we didn't have a very large sample size.

What if we want to run again with more samples?

Unfortunately, we made `t_x_n` a constant at the start of our tensorflow code so now we can't change it. Instead, we could have made it a `placeholder`. This tells tensorflow "there will be some data here but I'm going to give it to you later".

How do we give the data later on?

We can provide the values to placeholders by specifying a "feed dictionary" to `session.run`. This means, "during this session use the following values to replace all the placeholders".

Let's do our example again:

```
# Reset tensorflow to remove our old a, b, etc..
tf.reset_default_graph()

# THIS TIME USE A PLACEHOLDER!
#
# The data to fit to is provided as a placeholder.
# We need to tell it what type of data we will provide..
t_x_n = tf.placeholder(dtype=tf.float64, name='X')

# EVERYTHING ELSE IS AS IT WAS BEFORE..

# Note: mu and sigma are now *variables* not constants!
# We need to specify their data type and initial value..
t_mu = tf.Variable(mu_initial_guess,
                   dtype=tf.float64,
                   name="mu")
t_sigma = tf.Variable(sigma_initial_guess,
                      dtype=tf.float64,
                      name="sigma")

# Note: this step is important - don't use t_sigma directly!!
t_sigma_2 = t_sigma ** 2.0

# Calculate log p(X) terms..

t_x_minus_mu_2 = (t_x_n - t_mu) ** 2.0
t_denom = 2.0 * t_sigma_2
t_sigma_term = - 0.5 * tf.log(2.0 * np.pi * t_sigma_2)

t_log_P_terms = t_sigma_term - (t_x_minus_mu_2 / t_denom)

# The sum is performed by a reduction in tensorflow
# (since a vector goes in and a scalar comes out)
# but this is effectively the same as np.sum(...)
t_log_P = tf.reduce_sum(t_log_P_terms)
```

```
# NOW WHEN WE RUN WE NEED TO FILL IN THE PLACEHOLDER..

# Create a gradient descent optimiser that uses a
# certain step size (learning_rate)..
```

```
optimiser = tf.train.GradientDescentOptimizer(learning_rate=0.05)

# We want to maximise log p(X) therefore we
# need to minimise - log p(X)
t_objective = - t_log_P

# We want to optimise wrt mu and sigma
vars_to_optimise = [t_mu, t_sigma]

minimize_operation = optimiser.minimize(t_objective,
                                        var_list=vars_to_optimise)

# Number of iterations to perform
num_iterations = 50

with tf.Session() as session:
    # IMPORTANT! (see above..)
    session.run(tf.global_variables_initializer())

    # Run a number of iterations of gradient descent..
    for iteration in range(num_iterations):
        # At each iteration evaluate the minimize_operation
        # to perform the gradient descent step and also
        # keep track of the current value..
        #
        # NEED TO ADD THE FEED DICTIONARY OTHERWISE WE
        # DON'T KNOW WHAT VALUE TO USE FOR t_x_n..
        #
        step, cost = session.run([minimize_operation, t_log_P],
                                 feed_dict={ t_x_n : x_n })

        # Print out the value of log P every 10 iterations..
        if ((iteration + 1) % 10 == 0):
            print('iter %4d, log P(X) = %0.3f' %
                  (iteration + 1, cost))

    # Get the final results of the optimisation..
    mu_optimised = session.run(t_mu)
    sigma_optimised = session.run(t_sigma)

    print('\nAfter optimisation:')
    print('Tensorflow mu = ', mu_optimised)
    print('Tensorflow sigma = ', sigma_optimised)

print('\nAnalytic estimates:')
print('Estimated mu = ', np.mean(x_n))
print('Estimated std = ', np.std(x_n))
```

```
print('\nGround truth values:')
print('True mu = ', mu_true)
print('True sigma = ', sigma_true)
```

```
Output:
iter   10, log P(X) = -43.105
iter   20, log P(X) = -35.690
iter   30, log P(X) = -35.690
iter   40, log P(X) = -35.690
iter   50, log P(X) = -35.690

After optimisation:
Tensorflow mu =  2.6516868687120074
Tensorflow sigma =  1.4413016026439345

Analytic estimates:
Estimated mu =  2.6516868687120074
Estimated std =  1.4413016026439343

Ground truth values:
True mu =  2.5
True sigma =  1.5
```

We can now even make this a function and call it with lots of different data:

```
def find_parameters_using_tensorflow(x_input,
                                     learning_rate=0.05):
    # Create a gradient descent optimiser that uses a
    # certain step size (learning_rate)..
    optimiser = tf.train.GradientDescentOptimizer(
        learning_rate=learning_rate)

    # We want to maximise log p(X) therefore we
    # need to minimise - log p(X)
    t_objective = - t_log_P

    # We want to optimise wrt mu and sigma
    vars_to_optimise = [t_mu, t_sigma]

    minimize_operation = optimiser.minimize(t_objective,
                                            var_list=vars_to_optimise)

    # Number of iterations to perform
    num_iterations = 50

    with tf.Session() as session:
        # IMPORTANT! (see above..)
        session.run(tf.global_variables_initializer())

        # Run a number of iterations of gradient descent..
```

```
        for iteration in range(num_iterations):
            # At each iteration evaluate the minimize_operation
            # to perform the gradient descent step and also
            # keep track of the current value..
            #
            # PASS THE ARGUMENT TO THE FUNCTION INTO THE FEED
            # DICTIONARY..
            #
            step, cost = session.run([minimize_operation, t_log_P],
                                     feed_dict={ t_x_n : x_input })

            # Print out the value of log P every 10 iterations..
            if ((iteration + 1) % 10 == 0):
                print('iter %4d, log P(X) = %0.3f' %
                      (iteration + 1, cost))

        # Get the final results of the optimisation..
        mu_optimised = session.run(t_mu)
        sigma_optimised = session.run(t_sigma)

    return mu_optimised, sigma_optimised
```

```
# Let's try with a larger N

N_bigger = 1000

x_bigger = np.random.normal(mu_true, sigma_true, N_bigger)

new_mu, new_sigma = find_parameters_using_tensorflow(x_bigger,
                                                     learning_rate=0.001)

print('Tensorflow estimates:')
print('Tensorflow mu = ', new_mu)
print('Tensorflow sigma = ', new_sigma)

print('\nAnalytic estimates:')
print('Estimated mu = ', np.mean(x_bigger))
print('Estimated std = ', np.std(x_bigger))

print('\nGround truth values:')
print('True mu = ', mu_true)
print('True sigma = ', sigma_true)
```

```
Output:
iter   10, log P(X) = -2017.966
iter   20, log P(X) = -1812.541
iter   30, log P(X) = -1812.541
iter   40, log P(X) = -1812.541
iter   50, log P(X) = -1812.541
```

```
Tensorflow estimates:
Tensorflow mu =  2.472980666304121
Tensorflow sigma =  1.4823118516374283

Analytic estimates:
Estimated mu =  2.472980666304121
Estimated std =  1.4823118516374285

Ground truth values:
True mu =  2.5
True sigma =  1.5
```

Advanced Topics

This ends our introduction to the new paradigm of declarative programming illustrated through `TensorFlow`. There are a whole range of more advanced topics to go and look at including:

- Visualising parts of computation (e.g. Tensorboard)
- Reusable components (e.g. modules for neural networks / classifiers / etc..)
- Run computations on the GPU instead of the CPU (often faster)
- Easy to scale; can distribute computations over an entire cluster!

For now, there is a separate notebook that you can workthrough on the colab site and try `TensorFlow` out for yourself!