# BMVA Summer School Python Introduction

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

12th July 2023

## Introduction

This document provides a brief introduction to python for numerical computation followed by an illustration of the `PyTorch` toolkit 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.

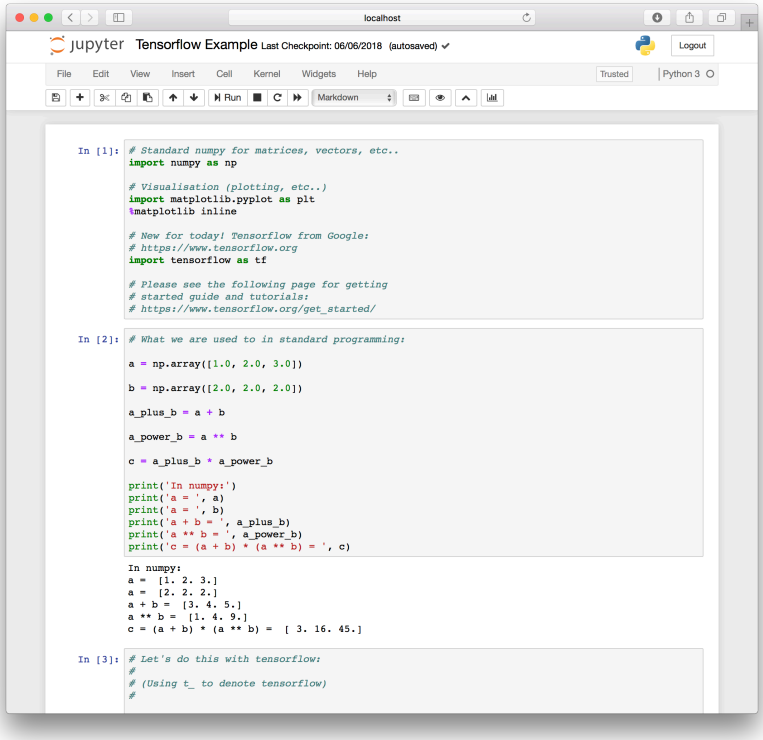


Figure 1: A screenshot of a running jupyter notebook.

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
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
```

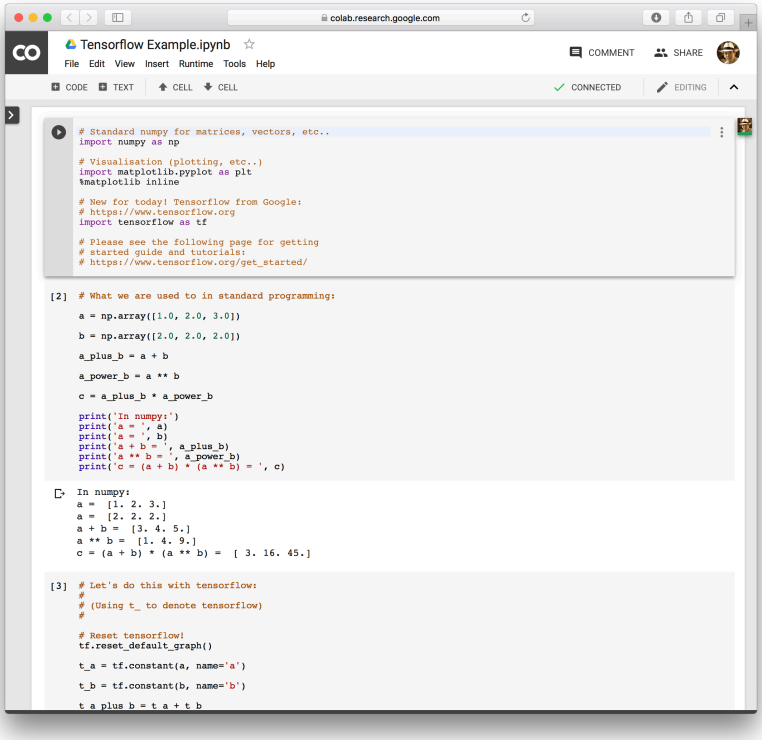


Figure 2: A screenshot of the same notebook running in the cloud on the new Google colaboratory notebook server. The resulting notebook can be saved locally on your computer or on your Google Drive if you have a gmail account. Notebooks can be shared an edited by multiple people in a similar manner to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

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 **`PyTorch`** 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.

### 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:

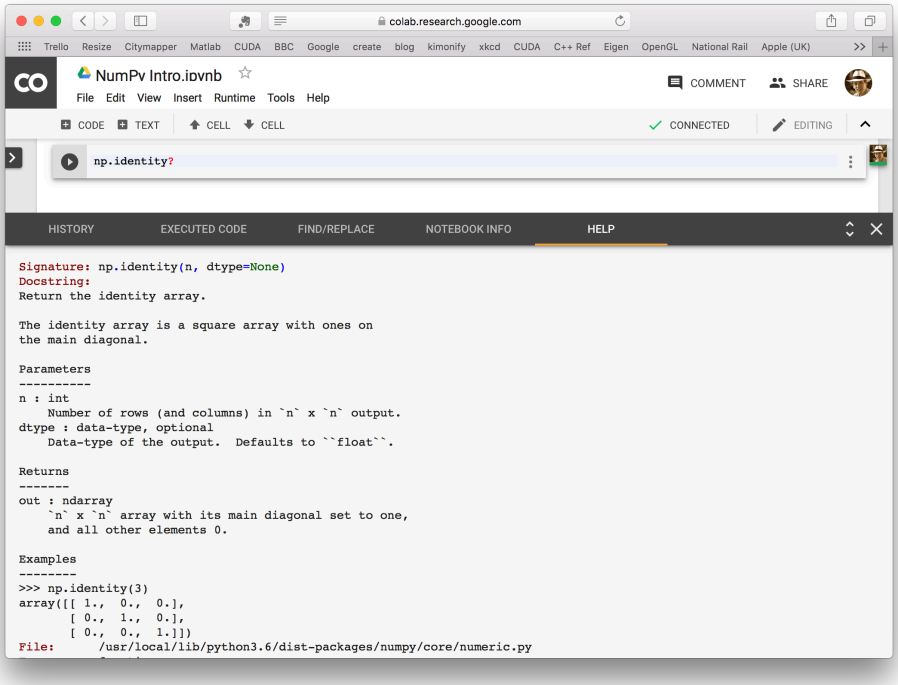


Figure 3: Accessing help on functions inside a notebook by entering the function name followed by a "**?**" in a cell and then running it. This will bring up a help panel at the bottom of the screen that you can scroll through and read. The panel can be closed with the cross on the right hand side of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

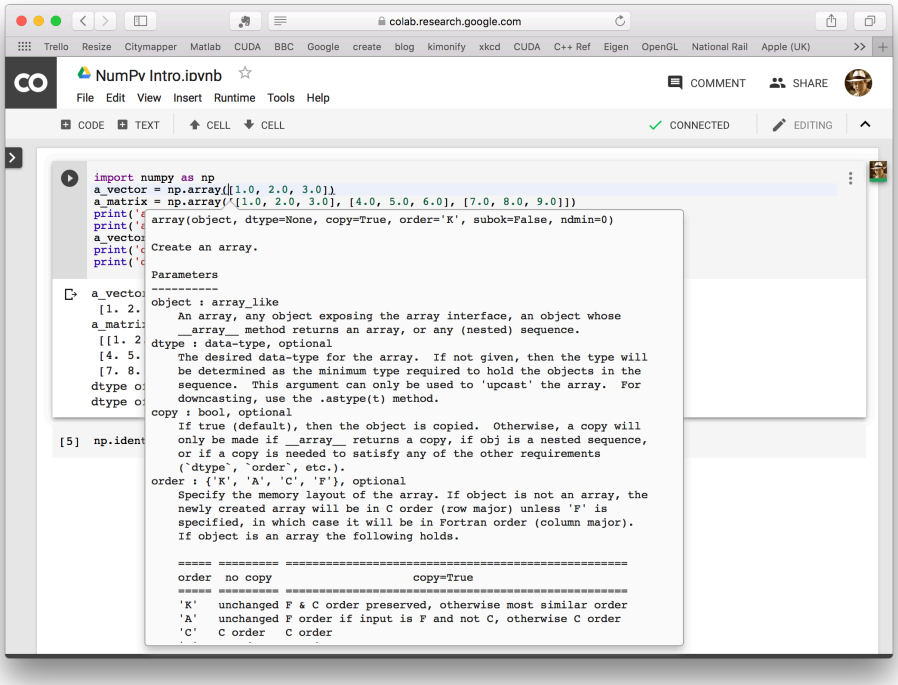


Figure 4: Context sensitive help is also available by pressing the `[tab]` key which will bring up help on the arguments of a function or a list of members of the class/module.

```
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
```

#### 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 $4^{\text{th}}$ 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 $3\times6$ 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.]]
```

# PyTorch

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. The PyTorch framework seeks the best of both worlds whereby the imperative environment is maintained (programmers are familiar with and it is easier to debug) whilst *implicitly* creating a computational graph in the background that can be used to provide features such as automatic differentiation and computation across devices (e.g. CPU or GPU).

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 `PyTorch` there is a very detailed website including a getting started guide and tutorials https://pytorch.org/tutorials/index.html.

## Setting up

In a similar manner to `numpy` we import `PyTorch` using its package name `torch` so all functions that start with `torch` and in `PyTorch` 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
# 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 torch and tensorboard when running on colab
if in_colab:
    !pip install -U torch torchvision torchviz tensorboardcolab
    from tensorboardcolab import *
else:
    # Use to make plots appear inline with output in jupyter
    %matplotlib inline
# New for today! Import PyTorch (refered to by package name torch)
import torch
# This is used for graph visualisation..
from torchviz import make_dot
```

### 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 Torch

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

```
# Let's do this with pytorch (or just torch):
t_a = torch.tensor(a)
t_b = torch.tensor(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 torch:')
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 torch:
a =  tensor([1., 2., 3.], dtype=torch.float64)
a =  tensor([2., 2., 2.], dtype=torch.float64)
a + b =  tensor([3., 4., 5.], dtype=torch.float64)
a ** b =  tensor([1., 4., 9.], dtype=torch.float64)
c = (a + b) * (a ** b) =  tensor([ 3., 16., 45.], dtype=torch.float64)
```

#### So why would we use torch instead of numpy?

So far, the two seem to operate in the same manner so why do we need torch?

In the background, torch is also builds a computational graph of the operations being performed. This will allow, amongst other things, the graph to be analysed and gradients to be computed automatically by going backwards through the graph applying the differentiation **chain rule** to propagate gradient information.

*Let's see an example in action!*

**Aside:** More information about how autograd works in `torch` can be found in the following links: Automatic Differentiation with torch.autograd and Autograd Mechanics.

```
# Let's do the same calculation as before but we will tell torch that we would
# like to calculate gradients by using the "requires_grad=True" argument when
# we create the pytorch tensors..
t_a = torch.tensor(a, requires_grad=True)
t_b = torch.tensor(b, requires_grad=True)
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('t_c = ', t_c)
```

```
Output:
t_c =  tensor([ 3., 16., 45.], dtype=torch.float64, grad_fn=<MulBackward0>)
```

**We notice t_c has a new attribute!**

If we look the printed output for t_c we notice there is a new piece of information, now t_c has an attribute called "grad_fn".

This indicates that there is a function associated with the tensor to propagate the gradient backward (part of the *computational graph* we mentioned before).

We notice that the grad_fn object is of type **MulBackward** which indicates that this is a function that calculates the gradient through a multiplication operation. We remember that t_c is the result of a multiplication: `t_c = t_a_plus_b * t_a_power_b` so we would expect the gradient to require an application of the chain rule via the derivative of a multiplication operation.

```
# We can visualise the computational graph for all the relevant terms:
print('t_c gradient function = ', t_c.grad_fn)
print('t_a_plus_b gradient function = ', t_a_plus_b.grad_fn)
print('t_a_power_b gradient function = ', t_a_power_b.grad_fn)
print('t_b gradient function = ', t_b.grad_fn)
print('t_a gradient function = ', t_a.grad_fn)
```

```
Output:
t_c gradient function =  <MulBackward0 object at 0x7f1e5d947dd0>
t_a_plus_b gradient function =  <AddBackward0 object at 0x7f1e5d947e10>
t_a_power_b gradient function =  <PowBackward1 object at 0x7f1e5d947ed0>
t_b gradient function =  None
t_a gradient function =  None
```

We note that when we get to t_a and t_b we no longer have any gradient functions since these are the starting points of the calculation (and not dependent on any other values).

Otherwise, we can see that a gradient function is associated with each operation using a suitable function. So

t_a_plus_b has **AddBackward** since it is the result of an addition operation. Equally, t_a_power_b has **PowBackward** since it is the result of taking a to the power of b.

**We can now see these gradient functions in action!** We will now make use of the gradient functions by running the *backward* operation in torch which tells the computational graph to propagate gradients.

Let us calculate the gradients of sum(t_c) wrt t_a and t_b. That is we want

$$\frac{ds}{da} \quad \text{and} \quad \frac{ds}{db}$$

where $s = \sum_i c_i$.

```
# Create a result to hold the scalar sum
t_s = torch.sum(t_c)
# Propagate gradients for t_s by calling backward..
t_s.backward()
# Now we can read out the gradients..
print('Gradient for t_a (ds/da) = ', t_a.grad)
print('Gradient for t_b (ds/db) = ', t_b.grad)
```

```
Output:
Gradient for t_a (ds/da) =  tensor([ 7., 20., 39.], dtype=torch.float64)
Gradient for t_b (ds/db) =  tensor([ 1.0000, 15.0904, 58.4376], dtype=torch.float64)
```

Let's check that these results make sense.

We have the following:

$$s = \sum_i c_i, \quad c_i = (a_i + b_i) \times {a_i}^{b_i}$$

Now if we remember our calculus rules, we will need to apply the following identities:

$$\frac{d}{dx}(a+b) = \frac{da}{dx} + \frac{db}{dx}$$

$$\frac{d}{dx}(a \times b) = \frac{da}{dx} \times b + a \times \frac{db}{dx}$$

$$\frac{d}{dx}(x^a) = a \times x^{a-1}$$

The final exponent rule (needed for t_b) is a little more involved:

$$\frac{d}{dx}(a^b) = \frac{d}{dx}(e^{b \ln a}) = (e^{b \ln a})\frac{d}{dx}(b \ln a) = a^b \left(\frac{db}{dx} \ln a + \frac{da}{dx}\frac{b}{a}\right)$$

So, we can mimic the work of torch for t_a:

$$\frac{dc_i}{da_i} = \frac{d}{da_i}[(a_i + b_i) \times {a_i}^{b_i}] = {a_i}^{b_i} \times \left[\frac{d}{da_i}(a_i + b_i)\right] + (a_i + b_i) \times \left[\frac{d}{da_i}({a_i}^{b_i})\right]$$

$$\Rightarrow \frac{dc_i}{da_i} = {a_i}^{b_i} + (a_i + b_i) \times (b_i \times {a_i}^{b_i - 1})$$

**Let's check the pytorch result..**

```
# Calculate our hand-derived gradient for t_a..
t_dc_da = (t_a ** t_b) + (t_a + t_b) * t_b * (t_a ** (t_b - 1.0))
print('Check t_dc_da = ', t_dc_da)
print('Torch Gradient for t_a = ', t_a.grad)
# Hopefully these two match!!
```

```
Output:
Check t_dc_da =  tensor([ 7., 20., 39.], dtype=torch.float64, grad_fn=<AddBackward0>)
Torch Gradient for t_a =  tensor([ 7., 20., 39.], dtype=torch.float64)
```

**So it seems to work!**

If you would like you can check the result for t_b as well..

Note: This is a more involved derivation (due to the more complex exponential rule); we can already see that even for our simple calculation, the gradient derivations can get very involved and so hand-calculation is error prone, time consuming and has to be changed with every modification to the original calculation. *The automatic differentiation in torch is really helping out!*

**Why do we need to calculate derivations?**

Well, what if we are doing an optimistion? Let's see a full motivating example..

## Example: fitting the parameters of a distribution

Suppose we want to fit a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ to a set of numbers $\mathrm{X} = \{\mathrm{x}_0, \mathrm{x}_1, \ldots, \mathrm{x}_{\mathrm{N}-1}\}$.

**We know that this fit can actually be calculated directly (one of the nice properties of the Gaussian distribution) but we will pretend that we cannot determine these parameters in closed-form and will perform nummerical optimisation to determine them - this way we can check our results with the analytic solution!**

If we assume the numbers are i.i.d. (indentically and independently distributed) samples from a Gaussian then the likelihood of X is given by:

$$
\begin{aligned}
\mathrm{p}(\mathrm{X}) &= \mathrm{p}(\mathrm{x}_0) \cdot \mathrm{p}(\mathrm{x}_1) \cdot \cdots \cdot \mathrm{p}(\mathrm{x}_{\mathrm{N}-1}) && (1)\\
&= \mathcal{N}(\mathrm{x}_0 \,|\, \mu, \sigma^2) \cdot \mathcal{N}(\mathrm{x}_1 \,|\, \mu, \sigma^2) \cdot \cdots \cdot \mathcal{N}(\mathrm{x}_{\mathrm{N}-1} \,|\, \mu, \sigma^2) && (2)\\
&= \prod_{\mathrm{n}=0}^{\mathrm{N}-1} \mathcal{N}(\mathrm{x}_\mathrm{n} \,|\, \mu, \sigma^2) && (3)\\
&= \prod_{\mathrm{n}=0}^{\mathrm{N}-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\mathrm{x}_\mathrm{n} - \mu)^2}{2\sigma^2}\right) && (4)
\end{aligned}
$$

**Top Tip!** When working with exponential family of distributions it often helps to work in the log domain..

$$
\log \mathrm{p}(\mathrm{X}) = \sum_{\mathrm{n}=0}^{\mathrm{N}-1} -\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{(\mathrm{x}_\mathrm{n} - \mu)^2}{2\sigma^2} \tag{5}
$$

So, we have the *maximum likelihood* fit to the parameters when we find the values of $\mu$ and $\sigma^2$ that maximise $\mathrm{p}(\mathrm{X})$ which (since $\log(\cdot)$ is a concave function) occurs at the same time that $\log \mathrm{p}(\mathrm{X})$ is maximised.

In our case we can find an analytic solution for

$$
\begin{aligned}
\mu^* &= \arg\max_\mu \log \mathrm{p}(\mathrm{X}) && (6)\\
\sigma^{*2} &= \arg\max_{\sigma^2} \log \mathrm{p}(\mathrm{X}) && (7)\\
& && (8)
\end{aligned}
$$

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

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

*Let's see how to do this in torch..*

```
# 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 =
 [ 1.563  2.28   0.209  4.216  1.281  1.223  3.576
  3.028  2.576  1.299  2.451  4.705  3.598  1.693
 -0.471  4.839  5.097  1.587  4.577  3.99 ]
```

We are now going to create our implicit torch computation graph but we are going to account for the fact that $\mu$ and $\sigma^2$ 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 $\mu$ and $\sigma^2$; in this case, we will start with

$$\mu_{\text{initial}} = 1 \tag{9}$$

$$\sigma^2_{\text{initial}} = 1 \tag{10}$$

**Top Tip!** Care needs to be taken with $\sigma$ since it can only be a positive value (unlike $\mu$ which can be any real number). In general, `torch` 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:

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

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

# The data to fit to (NOTE: this is our constant data so no gradients required)
t_x_n = torch.tensor(x_n)

# Note: mu and sigma are now *variables* not constants!
# We need to specify their data type and initial value..
t_mu = torch.tensor(mu_initial_guess, requires_grad=True)
t_sigma = torch.tensor(sigma_initial_guess, requires_grad=True)
```

```
# 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 * torch.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 torch
# (since a vector goes in and a scalar comes out)
# but this is effectively the same as np.sum(...)
t_log_P = torch.sum(t_log_P_terms)

# Let's just check that we calculated things correctly:

print('Torch log p(X) = ', t_log_P)
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)

# Check these are close (to nummerical precision - remember not to use
# equality when checking floating point numbers due to round-off error)..
assert(np.isclose(t_log_P.detach().numpy(), check_value))

print('\nEverything working!')
```

```
Output:
Torch log p(X) =  tensor(-70.9790, dtype=torch.float64, grad_fn=<SumBackward0>)
(using initial guesses for mu and sigma)

Value from scipy stats package =  -70.97903223521158

Everything working!
```

**Note:** torch.tensor values are not the same as numpy.arrays and so when passing torch values into numpy functions (as illustrated in the np.isclose call above). If we have a value that is current associated with a computational graph to calculate the gradient then we must also create a copy first by detaching it from the graph, hence the use of `t_log_P.detach().numpy()`

**This is great, but what if we want to use different parameter values?**

Well, we need to put our torch code into a function so that we can call it with different values. Let's put the code from above into a function:

```
# Create the log likelihood function taking the data and parameters as arguments
def torch_gaussian_log_likelihood(t_x_n, t_mu, t_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 * torch.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 torch
    # (since a vector goes in and a scalar comes out)
    # but this is effectively the same as np.sum(...)
    t_log_P = torch.sum(t_log_P_terms)

    return t_log_P

# Let's check again..
print('Torch log p(X) = ', torch_gaussian_log_likelihood(t_x_n, t_mu, t_sigma))
print('(using initial guesses for mu and sigma)\n')
print('Value from scipy stats package = ', check_value)
```

```
Output:
Torch log p(X) =  tensor(-70.9790, dtype=torch.float64, grad_fn=<SumBackward0>)
(using initial guesses for mu and sigma)

Value from scipy stats package =  -70.97903223521158
```

**So now let's use the power of pytorch!!**

To perform optimisation we need to know the gradient of the log likelihood with respect to the particular parameters $\mu$ and $\sigma$.

We know how to find these with torch using the backward from above!

```
t_log_P = torch_gaussian_log_likelihood(t_x_n, t_mu, t_sigma)
t_log_P.backward()
print('Gradient wrt mu = ', t_mu.grad)
print('Gradient wrt sigma = ', t_sigma.grad)
```

```
Output:
Gradient wrt mu =  tensor(33.3175)
Gradient wrt sigma =  tensor(85.2005, dtype=torch.float64)
```

Shall we check that result. Remember we have:

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

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

So for $\mu$ we have:

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

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

$$= -\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}2(x_n-\mu)\frac{\partial}{\partial\mu}(x_n-\mu) \tag{16}$$

$$= \frac{1}{\sigma^2}\sum_{n=0}^{N-1}(x_n-\mu) \tag{17}$$

where we used the chain rule a number of times..

```
# 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('Torch gradient wrt mu = ', t_mu.grad)
assert(np.isclose(t_mu.grad.detach().numpy(), grad_mu_check))
print('\nExcellent! torch calculated the gradient for us :)')
```

```
Output:
Our analytic gradient wrt mu =  33.31747153533429
Torch gradient wrt mu =  tensor(33.3175)

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

### Everyone should now 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 $\sigma^2$.

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

This:

$$\log p(X) = \sum_{n=0}^{N-1} -\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{(x_n-\mu)^2}{2\sigma^2} \tag{18}$$

has become a computational graph.

For example, the **pow** operation represents $r = a^b$ for the inputs $a, b$ and result $r$. As above, torch 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 $\mu$ at the bottom) to calculate the gradient.

Therefore, the procedure when operating with torch is always the same. A **forward pass** can calculate the objective for the current set of parameters and then a **backwards pass** can calculate the gradients of an objective wrt any of the parameters.

### Simple gradient descent using these gradients..

Let's use simple gradient descent to try to fit the values of our parameters to our data. Essentially, we start with some initial values and take a "step" in the direction of the (downhill) gradient in order to minimise the objective.

In our case, the parameters that we want will maximise the log likelihood so the objective function we must use is to minimise the **negative** log likelihood.

So the update rule will be:

$$\mu^{(k+1)} = \mu^{(k)} - \eta \frac{\partial}{\partial \mu} \log p(\mathbf{x})$$

where $\eta$ is some (small) step size and $k$ is the iteration index.

Let us try to do this loop a few times:

```
number_of_iterations = 100
step_size_eta = 1.0e-2

# The data to fit to (NOTE: this is our constant data so no gradients required)
t_x_n = torch.tensor(x_n)

# Note: mu and sigma are now *variables* not constants!
# We need to specify their data type and initial value..
t_mu = torch.tensor(mu_initial_guess, requires_grad=True)
t_sigma = torch.tensor(sigma_initial_guess, requires_grad=True)

for iteration in range(number_of_iterations + 1):
    # Perform the forward pass, calculate - log p(x)
    t_neg_log_likelihood = -1.0 * torch_gaussian_log_likelihood(t_x_n, t_mu, t_sigma)

    # Perform the backwards pass to get the gradients
    t_neg_log_likelihood.backward()

    # Temporarily disable gradient computations so that we can update the
    # parameter values (we don't want to differentiate the update!)..
    with torch.no_grad():
        # Update the parameters based on the step size..
        t_mu -= step_size_eta * t_mu.grad
        t_sigma -= step_size_eta * t_sigma.grad

        # IMPORTANT: Clear the gradients for next time..
        t_mu.grad.data.zero_()
```

```
        t_sigma.grad.data.zero_()

    # Print out the current values
    if iteration % 10 == 0:
        print('Iter %04d, NLL %0.2e, mu %.4f, sigma %.4f' %
              (iteration, t_neg_log_likelihood.detach().numpy(), t_mu, t_sigma))


print('\nAfter optimisation:')
print('Torch mu = ', t_mu)
print('Torch sigma = ', t_sigma)

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 0000, NLL 7.10e+01, mu 1.3332, sigma 1.8520
Iter 0010, NLL 3.98e+01, mu 1.9173, sigma 1.8621
Iter 0020, NLL 3.82e+01, mu 2.2708, sigma 1.7214
Iter 0030, NLL 3.77e+01, mu 2.4778, sigma 1.6262
Iter 0040, NLL 3.75e+01, mu 2.5820, sigma 1.5897
Iter 0050, NLL 3.75e+01, mu 2.6293, sigma 1.5795
Iter 0060, NLL 3.75e+01, mu 2.6500, sigma 1.5770
Iter 0070, NLL 3.75e+01, mu 2.6590, sigma 1.5765
Iter 0080, NLL 3.75e+01, mu 2.6629, sigma 1.5764
Iter 0090, NLL 3.75e+01, mu 2.6646, sigma 1.5764
Iter 0100, NLL 3.75e+01, mu 2.6653, sigma 1.5764

After optimisation:
Torch mu =  tensor(2.6653, requires_grad=True)
Torch sigma =  tensor(1.5764, dtype=torch.float64, requires_grad=True)

Analytic estimates:
Estimated mu =  2.6658735767667148
Estimated std =  1.5763538255551268

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

### Great - we can run an optimisation that automatically calculates the gradients!

So we now have everything coming together to perform a nummerical optimisation where we can perform gradient descent without having to write any code to calculate the gradients ourselves (yay!!).

The final stage is that there were a few technical gotchas in there around dealing with the parameter gradients and updating them (i.e. we had to temporaily suspend gradient calculation and ensure that we zeroed the gradient datastructures for each parameter.

Torch provides a nicer way to modularise this process into an object (or class) in python that provides a nicer interfact. Each module has a set of parameters to be optimised and a forward operation to be performed on data. The autograd operations will then take care of performing the backward operation and an optimiser can be called to control how the parameters are updated (e.g. something more complicated than simple gradient descent).

Let's wrap up our code in this interface.

### Putting models into modules..

Torch has a paradigm to make this procedure easy to work with based on deriving your own class based on a `module` that keeps parameters and the forward pass of the model together to allow easy optimisation.

```
# Create our own class derived from the torch module..
class MaximumLikelihoodGaussianModel(torch.nn.Module):
    # We must initialise our model - we specify our initial guesses for the
    # parameters..
    def __init__(self, mu_initial_guess, sigma_initial_guess):
        # Call the constructor for the torch.nn.Module super class..
        super().__init__()

        # We use the Parameter class (rather than tensors) for the module
        # but these behave in the same way..
        self.t_mu = torch.nn.Parameter(torch.tensor(mu_initial_guess))
        self.t_sigma = torch.nn.Parameter(torch.tensor(sigma_initial_guess))

    # This defines the forward operation on some data passed in
    def forward(self, t_x_n):

        # We will copy in the code from torch_gaussian_log_likelihood..

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

        # Calculate log p(X) terms..

        t_x_minus_mu_2 = (t_x_n - self.t_mu) ** 2.0
        t_denom = 2.0 * t_sigma_2
        t_sigma_term = - 0.5 * torch.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 torch
        # (since a vector goes in and a scalar comes out)
        # but this is effectively the same as np.sum(...)
        t_log_P = torch.sum(t_log_P_terms)
```

```
        # Remember to take the negative to return the negative log likelihood..
        return -1.0 * t_log_P
```

#### Now we can optimise our custom module

We can now use the standard optimisation approach for torch using our custom module. The steps are: - Define an instance of our module - Define an optimiser - Loop for each iteration: - Zero the gradients - Perform the forward pass - Perform the backward pass - Take an optimiser step

```
number_of_iterations = 100
# Note the step size is also referred to as the learning rate..
learning_rate = 1.0e-2

# The data to fit to (NOTE: this is our constant data so no gradients required)
t_x_n = torch.tensor(x_n)

# Create our model - it will initialise the paramters appropriately
gaussian_model = MaximumLikelihoodGaussianModel(mu_initial_guess, sigma_initial_guess)
gaussian_model.train()

# Create an optimiser for our model, the model has a datastructure of all
# the parameters to optimise that we will pass in (along with the learning rate)
optimizer = torch.optim.SGD(gaussian_model.parameters(), lr=learning_rate)

print([a for a in gaussian_model.parameters()])

for iteration in range(number_of_iterations + 1):
    # Perform the forward pass by calling the model with the data..
    t_neg_log_likelihood = gaussian_model(t_x_n)

    # Clear the gradients..
    optimizer.zero_grad()

    # Perform the backwards pass to calculate the gradients..
    t_neg_log_likelihood.backward()

    # Update the parameters via the optimiser..
    optimizer.step()

    # Print out the current values
    if iteration % 10 == 0:
        print('Iter %04d, NLL %0.2e, mu %.4f, sigma %.4f' %
              (iteration, t_neg_log_likelihood.item(),
               gaussian_model.t_mu, gaussian_model.t_sigma))


print('\nAfter optimisation:')
print('Torch mu = ', gaussian_model.t_mu.item())
print('Torch sigma = ', gaussian_model.t_sigma.item())
```

```
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 0000, NLL 7.10e+01, mu 1.3332, sigma 1.8520
Iter 0010, NLL 3.98e+01, mu 1.9173, sigma 1.8621
Iter 0020, NLL 3.82e+01, mu 2.2708, sigma 1.7214
Iter 0030, NLL 3.77e+01, mu 2.4778, sigma 1.6262
Iter 0040, NLL 3.75e+01, mu 2.5820, sigma 1.5897
Iter 0050, NLL 3.75e+01, mu 2.6293, sigma 1.5795
Iter 0060, NLL 3.75e+01, mu 2.6500, sigma 1.5770
Iter 0070, NLL 3.75e+01, mu 2.6590, sigma 1.5765
Iter 0080, NLL 3.75e+01, mu 2.6629, sigma 1.5764
Iter 0090, NLL 3.75e+01, mu 2.6646, sigma 1.5764
Iter 0100, NLL 3.75e+01, mu 2.6653, sigma 1.5764

After optimisation:
Torch mu =  2.6653215885162354
Torch sigma =  1.576355077492659

Analytic estimates:
Estimated mu =  2.6658735767667148
Estimated std =  1.5763538255551268

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

**Great - let's visualise our working model..**

Our new torch module allows us to perform the optimisation of our parameters using a standard approach (we can change the model and the rest of the code is unaltered).

**We can also visualise the computational graph created by our model with the follow code:**

```
make_dot(gaussian_model(t_x_n), params=dict(gaussian_model.named_parameters()))
```

Figure 5 shows us all the functions we would have to have implemented in order to calculate the gradients in the backward pass - we got all these for free using torch instead of numpy!!

## But the fun doesn't end here!

We can now run the same code again with different data just by changing the data we pass in to the model. This is how minibatching can be performed where we use a different subset of the data at each iteration (if we have very large datasets) to estimate the gradients to perform **stochastic gradient descent**.

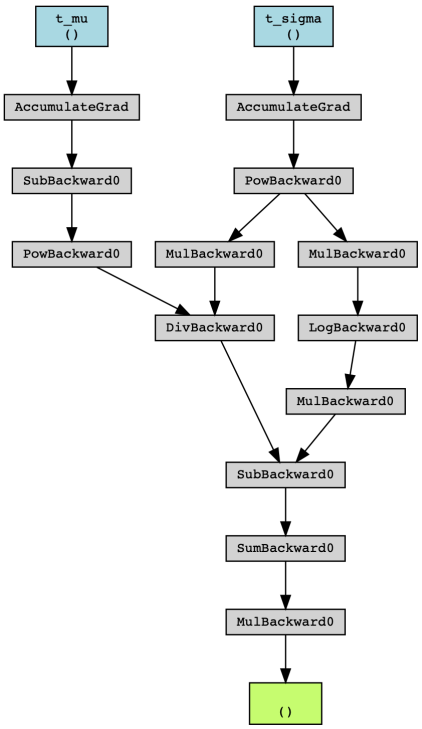


Figure 5: The computational graph corresponding to the `torch` operations for the backwards pass to calculate the gradients from the forward operations in the code.

**Aside:** More information on dataset loading: Datasets and Dataloaders

## All sorts of more advanced topics

- Visualise 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 `torch` out for yourself!

**More references:** Further material can be found in the offical PyTorch Tutorials and the PyTorch with Examples documentation.