

# Tensorflow Introduction

Neill D. F. Campbell

25<sup>th</sup> June 2018

## 1 Introduction

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 be 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/](https://www.tensorflow.org/get_started/).

## 2 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
```

### 3 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.]
```

### 4 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.

## 5 Visualise the Graph

The following operations allow you to see the graph in `tensorboard` and the resulting graph is shown in Figure 1.

```
# 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
```

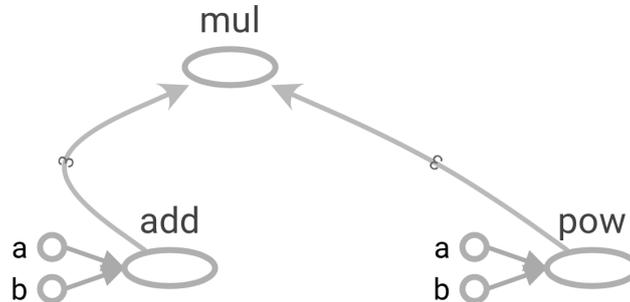


Figure 1: The computational graph corresponding to the `TensorFlow` operations of the same functions as the `numpy` code. If we look back at the code we can see that it makes sense. The two inputs `a` and `b` are combined together using both a sum operation and a power operation. The results of these two operations are then combined using a multiplication to produce the final output.

## 6 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?

## 7 Performing an Optimisation

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

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

$$p(X) = p(x_0) \cdot p(x_1) \cdot \dots \cdot p(x_{N-1}) \quad (1)$$

$$= \mathcal{N}(x_0 | \mu, \sigma^2) \cdot \mathcal{N}(x_1 | \mu, \sigma^2) \cdot \dots \cdot \mathcal{N}(x_{N-1} | \mu, \sigma^2) \quad (2)$$

$$= \prod_{n=0}^{N-1} \mathcal{N}(x_n | \mu, \sigma^2) \quad (3)$$

$$= \prod_{n=0}^{N-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_n - \mu)^2}{2\sigma^2}\right) \quad (4)$$

In our case we can find an analytic solution for

$$\mu^* = \arg \max_{\mu} \log p(X) \quad (5)$$

$$\sigma^{*2} = \arg \max_{\sigma^2} \log p(X) \quad (6)$$

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 ( $\mu$  and  $\sigma^2$ ).

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  $\mu$  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

$$\mu_{\text{initial}} = 1 \quad (7)$$

$$\sigma_{\text{initial}}^2 = 1 \quad (8)$$

**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, 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:

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

```

# 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
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!

## 8 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:

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

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

So for  $\mu$  we have:

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

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

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

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

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  $\sigma^2$ .

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:

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

has become the computational graph of Figure 2.

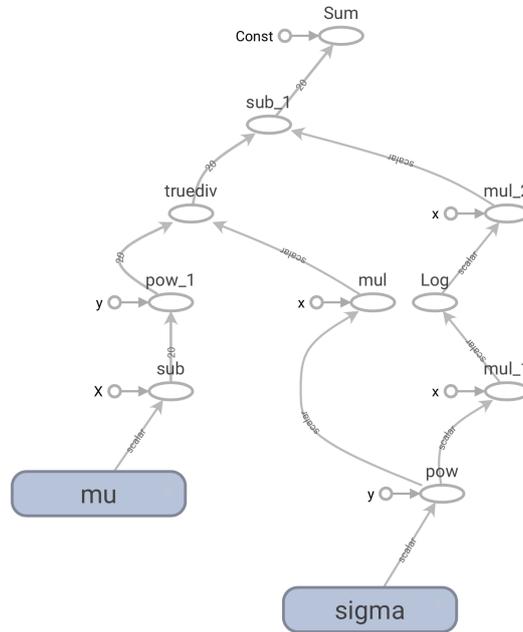


Figure 2: The computational graph corresponding to the calculation of the log likelihood function of Equation 16.

For example, the `pow` operation represents  $r = a^b$  for the inputs  $a, b$  and result  $r$ . TensorFlow then knows that  $\frac{\partial r}{\partial a} = b a^{b-1}$ , 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 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.

## 9 Optimisation in TensorFlow

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!

## 10 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

```

## 11 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!