Lab Demonstration 1
Fractals with Tensorflow

Shekhar “Shakes” Chandra

In this lab, we will study fractals and symmetry as a way of learning the basics of Tensorflow. The foundation of Tensorflow (TF) is based on n-dimensional arrays in a similar manner to Numpy. As a consequence, a number of functions from Numpy are also available in TF. These arrays are actually called Tensors, but for now we will refer to them as just arrays.

The lab is partitioned into three main parts. Firstly, we will look to introduce the main concepts of a TF program and plot useful mathematical functions. Then use TF to study the famous Mandelbrot and Julia sets in the complex plane. Finally, you will have a chance to implement a fractal of your choice from a list of known fractals using TF.

1 Part 1 (5 Marks)

We will use Jupyter notebook interface to write all our programs as they provide a nice front end to cloud based GPU clusters such as Google Colaboratory and AWS systems. Assuming you have watched the videos of these systems in logging in and starting up the Jupyter notebook, we will begin by implementing a simple Gaussian function via TF. Enter the following code block by block into the Jupyter notebook on the cloud system you’re using.

Begin by importing the two libraries we will be using for this part of the lab including TF

```
In [9]: import tensorflow as tf
        import numpy as np
```

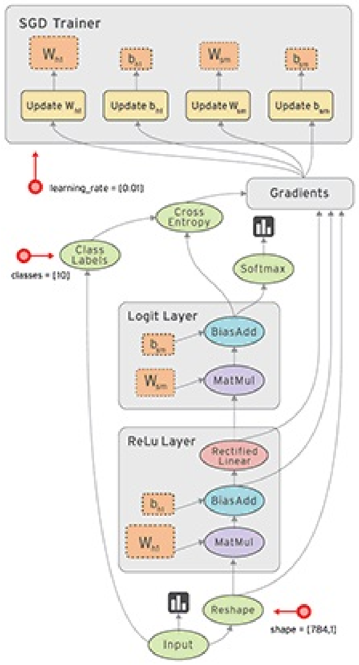
It is often useful to check/print the current version of the TF library installed, especially when using cloud based or cluster system. We can do this via the variable

```
In [10]: print("TF Version:", tf.__version__)

TF Version: 1.11.0
```

All TF programs execute by building a computational graph (see official page). All code in the script or notebook is used to build this graph first before actually executing the computation. Once the graph is built, it needs to be executed. This is done in TF as sessions. A session can be setup to run any part of the graph. This allows the execution of parts of the graph at different stages on different hardware.

Usually, it is best to run the graph in its entirety as soon as it is fully setup to fully exploit the hardware and avoid unnecessary data transfer between hardware components, such as the system



memory and the GPU. However, sometimes it is more appropriate to run the graph across multiple GPUs or other hardware components. Sessions provide this flexibility.

In our case, to make running simple algorithms, we can enable a default session and simplify using sessions as a single session that allows us to run any computation without specifying them explicitly. This is called an interactive mode and is ideal for debugging purposes and get immediate execution. We will use this mode so to execute the code as soon as individual lines are encountered.

```
In [11]: sess = tf.InteractiveSession()
```

Now that the default session has been setup, we can define the actual computation including equations and the image. We would like to define an image of the 2D Gaussian (or normal distribution). The traditional way would be to loop through each pixel and compute the value at that pixel based on the real position of the pixel with respect to the function being computed.

For example, the centre of the Gaussian would be the centre pixel of the image and its real position value would be (0,0) mathematically to give the maximum value of the distribution. This will however require writing loops and possibly involve errors involving out of bounds, as well as working out the spacings between pixels and their coordinates. It is more compact to write it letting Numpy handle the looping, especially when there are special function that can handle precisely this problem. Once such function is the mgrid function originally found in MATLAB.

```
In [12]: X, Y = np.mgrid[-4.0:4:0.01, -4.0:4:0.01]
```

Here, we have defined two grids of x-values and y-values with positions or coordinates from -4.0 to 4.0 at interval of 0.01. Then mgrid computes all the necessary parts related to the size of the grid and coordinates. Having defined the coordinates of the x and y points, we need to provide it as potential inputs to the graph. Since these grids are fixed and constant upon execution of the graph, we use the constant definition

```
In [13]: xs = tf.constant(X.astype(np.float32))
         ys = tf.constant(Y.astype(np.float32))
```

There are other ways to define inputs, including placeholders and variables. Having defined the inputs, we need to initialise the graph. In our case, we don't need to initialise the inputs, but this step also initialises other aspects of the graph and is an important step.

```
In [14]: tf.global_variables_initializer().run() #init variables
```

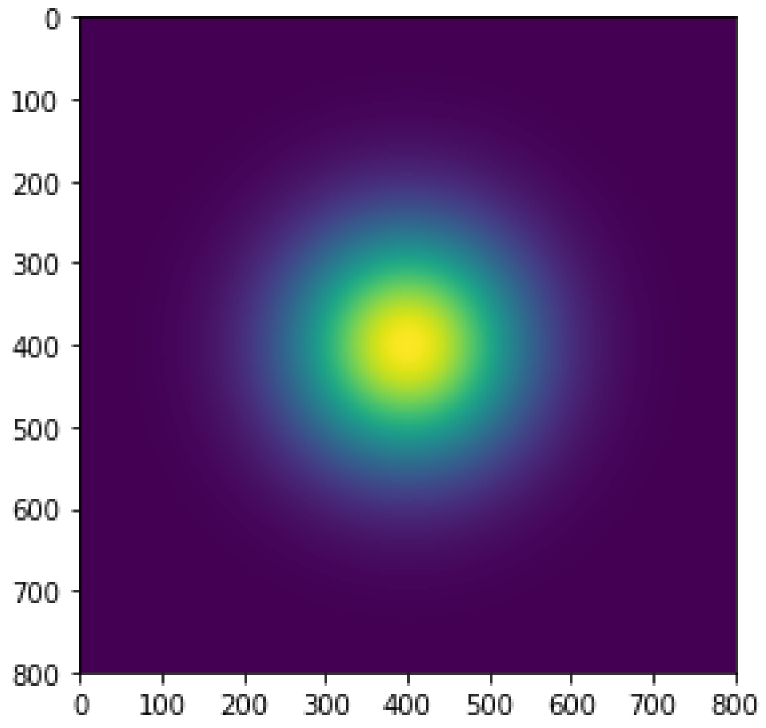
Now we can compute the actual Gaussian function $e^{-r^2/\sigma}$, where $r = x^2 + y^2$, via the TF exponential function using the TF constants we have defined previously

```
In [15]: # Compute Gaussian
         zs = tf.exp(-(xs**2+ys**2)/2.0)
```

Finally, we plot the result as an image using Matplotlib's imshow

```
In [16]: #plot
         import matplotlib.pyplot as plt

         plt.imshow(zs.eval())
         plt.tight_layout()
         plt.show()
```



An important distinction from Numpy execution is the eval() member call. In TF, this causes the computational graph to be executed, because up to this point, no actual computation has been done and the tensors zs etc. are empty.

To demonstrate this part of the lab, you must show your lab demonstrator the following items, noting the you will still need the Gaussian function later as well:

- Change the Gaussian function into a 2D sine or cosine function (2 Marks)
- What do you get when you multiply both the Gaussian and the sine/cosine function together? (2 Marks)

This is called modulation and you should get a Gabor filter, a mathematical function that is known to be a good approximation of a mammalian receptive field! We will use the theory of receptive fields later in the convolutional neural networks module of the course.

2 Part 2 (5 Marks)

In this part, we will be using TF to compute the Mandelbrot set, a fractal that is present in the complex plane. The essence of the Mandelbrot set is to compute which points in the complex plane converge when repeatedly squared or diverge (tend to infinity). In other words, we are going to compute the equation $z_{n+1} = z_n + c$ iteratively for the point c with $z_0 = 0$. For example for the point c, at the first iteration we have $z_1 = 0 + c$ then $z_2 = z_1 + c$ and so on. To construct the fractal, we will colour the points that converge to black and the remaining points coloured based on the rate of divergence.

Consider the point $c = 0.5 + 0.5j$ in the complex plane. Its magnitude $\mid c \mid$ is less than unity and repeatedly squaring this number makes the result smaller and smaller. In fact, it tends to zero and so we conclude that this point converges and colour it black. Other points whose magnitude is greater than may in fact diverge, so we may colour these using colours that denote the rate at which they are diverging.

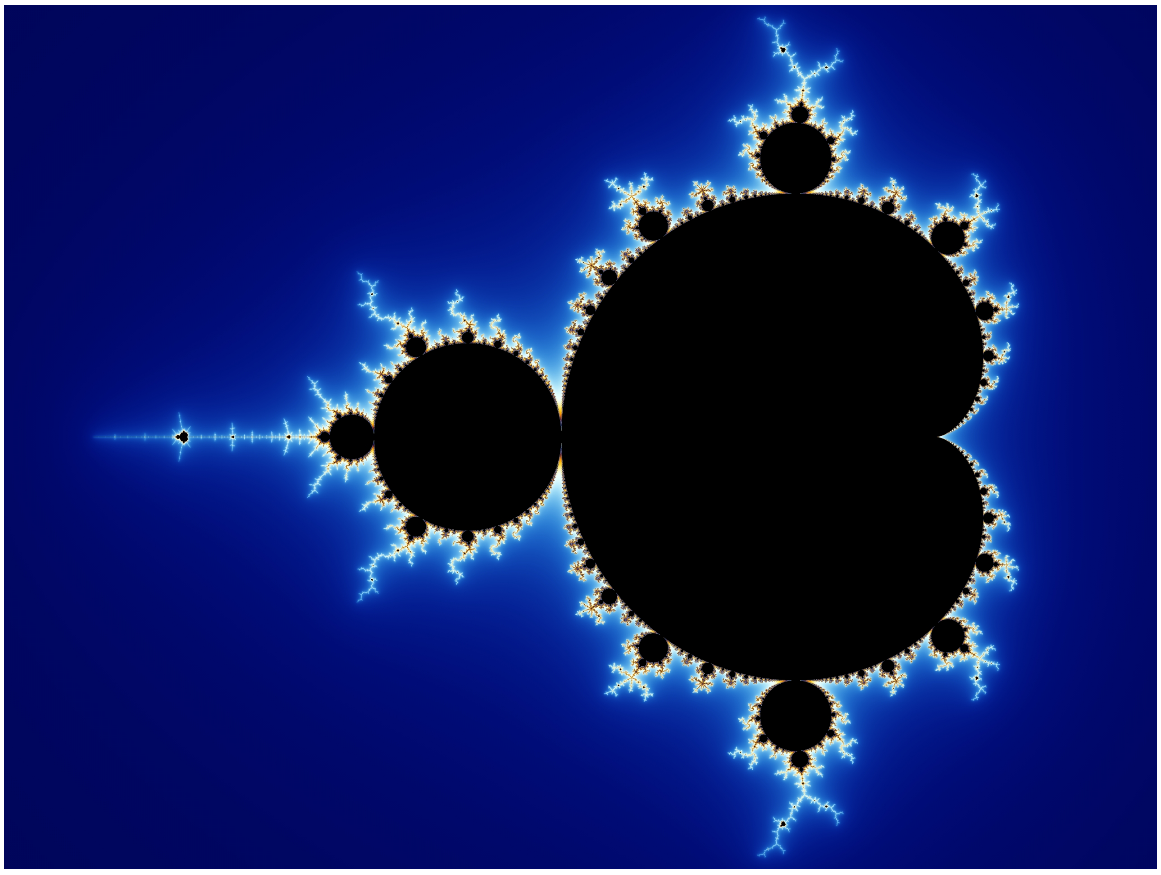
Begin by importing the two libraries we will be using for this part of the lab including TF

```
In [22]: import tensorflow as tf
import numpy as np
```

We create an interactive session to simplify session handling within TF for now. New versions of TF support Eager execution, basically making the execution indistinguishable from a library like Numpy. But keeping sessions is a helpful reminder of the computational graph that lies underneath while learning TF for the first time.

```
In [23]: sess = tf.InteractiveSession()
```

```
E:\Dev\WinPython-64bit-3.6.6.2Qt5\python-3.6.6.amd64\lib\site-packages\tensorflow\python\clien
warnings.warn('An interactive session is already active. This can '
```



Mandelbrot Set

Like the previous part, we create the grid coordinates using mgrid, this time it is the complex plane Z

```
In [24]: # Use NumPy to create a 2D array of complex numbers on [-2,2]x[-2,2]
Y, X = np.mgrid[-1.3:1.3:0.005, -2:1:0.005]
Z = X+1j*Y
```

Adjusting the grid resolution and position will allow us to look deeper into the Mandelbrot set. We will explore this later. Now we define the TF inputs to the graph, but now because z_{n+1} will be changing of time, it needs to be a variable. We also keep track of the eventual result (a surrogate for the rate of divergence) as n.

```
In [25]: xs = tf.constant(Z.astype(np.complex64))
zs = tf.Variable(xs)
ns = tf.Variable(tf.zeros_like(xs, tf.float32))
```

Having defined the inputs, we need to initialise the graph.

```
In [26]: tf.global_variables_initializer().run() #init variables
```

Now we encode the equation $z_{n+1} = z_n + c$ into the graph for the Mandelbrot set

```
In [27]: #Mandelbrot Set
#Compute the new values of z: z^2 + x
zs_ = zs*zs + xs
```

Once z_{n+1} is computed, we need to check whether it has converged or not

```
In [28]: # Have we diverged with this new value?
not_diverged = tf.abs(zs_) < 4
```

We then reassign z_n for the next iteration by $z_{n+1} \rightarrow z_n$ and then add to the n counter to keep track of the rate of divergence. We use the group function within TF to simply calling the lines as a single call

```
In [29]: # Operation to update the zs and the iteration count.
#
# Note: We keep computing zs after they diverge! This
#       is very wasteful! There are better, if a little
#       less simple, ways to do this.
#
step = tf.group( zs.assign(zs_), ns.assign_add(tf.cast(not_diverged, tf.float32)) )
```

Once setup, we run it iteratively for 200 iterations

```
In [30]: #run
for i in range(200):
        step.run()
```

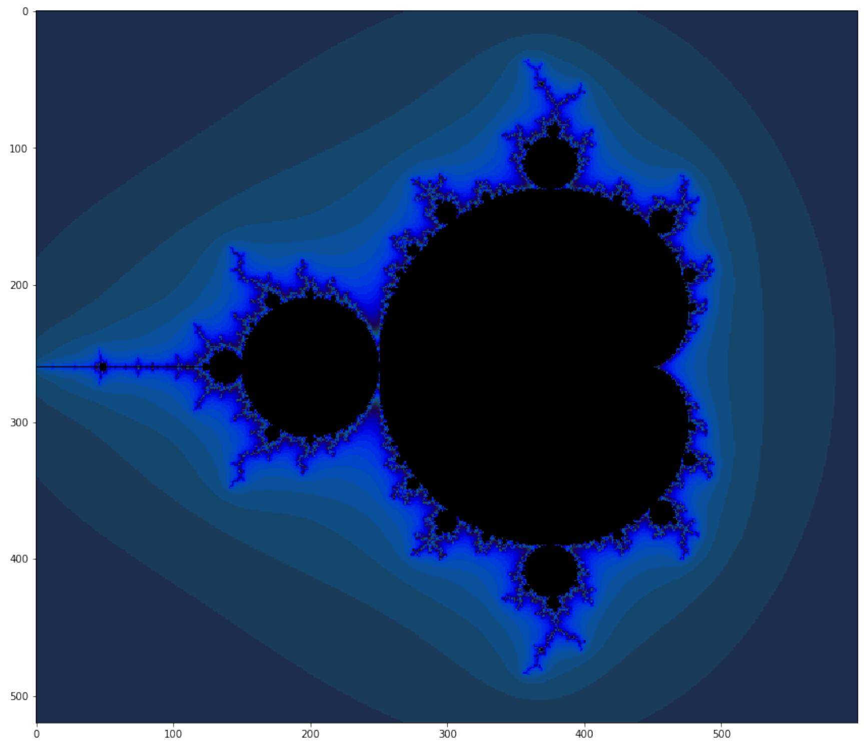
Finally plot the result via the n counter. We use a colour map for the counter and cast it to a 8-bit integer so that we can save it as an image file (such as a PNG) if we wanted to.

```
In [31]: #plot
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(16,10))

def processFractal(a):
        """Display an array of iteration counts as a
             colorful picture of a fractal."""
        a_cyclic = (6.28*a/20.0).reshape(list(a.shape)+[1])
        img = np.concatenate([10+20*np.cos(a_cyclic),
        30+50*np.sin(a_cyclic),
        155-80*np.cos(a_cyclic)], 2)
        img[a==a.max()] = 0
        a = img
        a = np.uint8(np.clip(a, 0, 255))
        return a

plt.imshow(processFractal(ns.eval()))
plt.tight_layout(pad=0)
plt.show()
```



To demonstrate this part of the lab, you must show your lab demonstrator the following items:

- High resolution computation of the set by decreasing the mgrid spacing. This may increase the computation time significantly, so choose a value that balances quality of the image and time spent.
- Zoom to another part of the Mandelbrot set and compute the image for it.
- Modify the code so to show a Julia set rather than the Mandelbrot set.

```
In [32]: sess.close()
```

3 Part 3 (5 Marks)

For this part, choose a fractal of your choice and implement using TF. Best entries will be added to the PatternFlow project! There are a number of potential fractals that you could explore including

- those from this list on Wikipedia.
- the fractal that Shakes discovered!
- those in the text book - Fractals for the Classroom. See for example chapters 2.10, 3.6, 4.6, 6.6.

You will need to demonstrate your code and resulting fractal to the demonstrator and justify that it uses TF in a major component within the algorithm of the fractal (3 Marks). Marks will be given for adequately commenting the code (1 Mark) and utilising parallelism with TF in reasonable some way (1 Mark).