Lab Demonstration 1
Fractals

Shekhar "Shakes" Chandra

Version 2.0

The foundation of deep learning and artificial intelligence (AI) models are libraries such as Tensorflow (TF) and PyTorch, which are based on n-dimensional arrays in a similar manner to Numpy called Tensors. In this lab, we will study fractals, self-similarity and symmetry as a way of learning the basics of Tensors, which are essential for deep learning content covered later in the course. As a consequence, a number of functions from Numpy are also available in these frameworks.

The lab is partitioned into three main parts. Firstly, we will introduce the main concepts of a PyTorch program and plot useful mathematical functions. Then use PyTorch 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 either TF or PyTorch.

Use of Artificial Intelligence

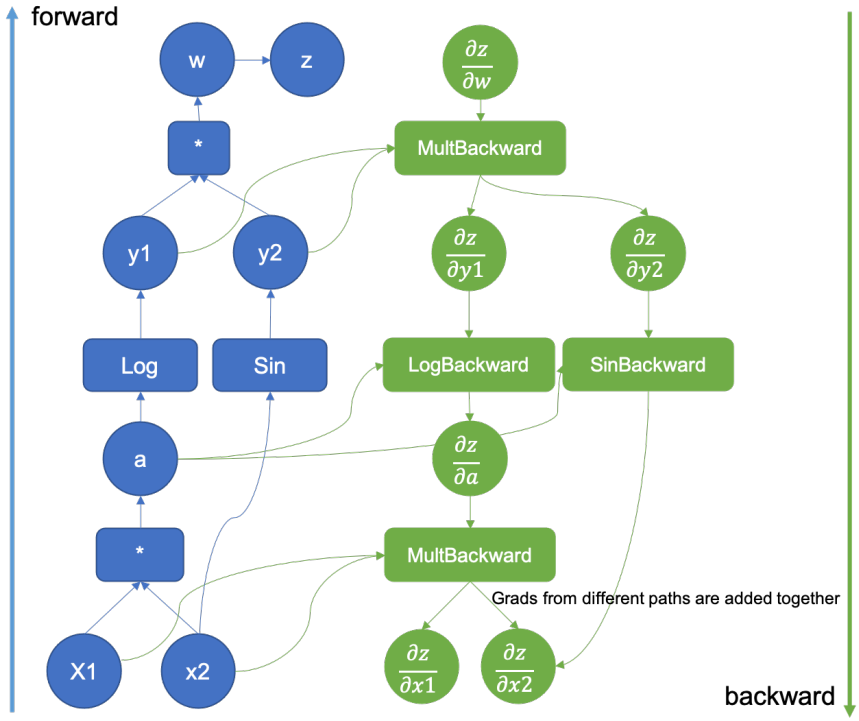
This lab is designed to incorporate the use of AI models to assist in your learning experiences. Models such as GitHub Co-Pilot (available for free to students), Microsoft Co-Pilot, Google Gemini, OpenAI ChatGPT (or equivalent) or Claude will be permitted as described in the instructions below. The goal is to show how these models can be useful in understanding and improving your learning and workflows, but to understand their limitations and common problems with their use as well.

Getting Started

To complete the instructions below, you will need either a Python notebook (either locally via something like WinPython/Anaconda or Google Collaboratory, see YouTube Playlist for instructions) or a working Python environment. Anaconda Python is pre-installed on the lab computers provided during the sessions that also have graphics processing units (GPUs) in them that can be used for computations. The teaching team will provide instructions to set these up during your practical sessions if you haven't already done so.

1 Part 1 (2 Marks)

You will write all our programs using Python scripts or notebooks. We will begin by implementing a simple Gaussian function via PyTorch. Enter the following code line by line into your newly created Python script on system you're using.



1.1 Tasks

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

```
import torch
import numpy as np
```

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

```
print("PyTorch Version:", torch.__version__)
```

Hopefully your output eventually in the terminal should look like below:

```
PyTorch Version: 2.0.1
```

Once your PyTorch environment is verified, it is good to ensure that the right computational device has been set and is available to use when needed:

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

All PyTorch 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 and execution can be optimized for the device that is selected.

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.

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:

```
# grid for computing image, subdivide the space
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 as tensors:

```
# load into PyTorch tensors
x = torch.Tensor(X)
y = torch.Tensor(Y)
```

Having defined the inputs, we need to convert the Numpy arrays to PyTorch tensors and then move it to the graphical processing unit (GPU).

```
# transfer to the GPU device
x = x.to(device)
y = y.to(device)
```

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

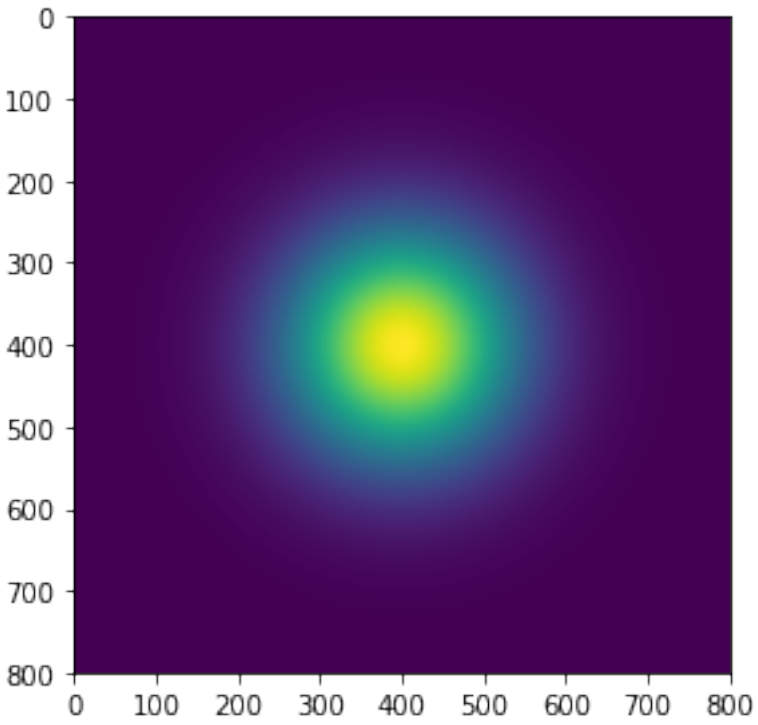
```
# Compute Gaussian
z = torch.exp(-(x**2+y**2)/2.0)
```

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

```
#plot
import matplotlib.pyplot as plt

plt.imshow(z.cpu().numpy())#Updated!

plt.tight_layout()
plt.show()
```



An important distinction from Numpy execution is the numpy() member call to convert the tensor back to a numpy array on the CPU for plotting. In PyTorch, this causes the computational graph to be executed, because up to this point, no actual computation has been done and the tensors z etc. are empty.

1.2 AI Tasks

Given the outputs from the previous section, replicate the output of the Gaussian function using an AI model that uses Numpy. You can use a prompt such as:

Generate a Python script to plot a 2D Gaussian function using Numpy and Matplotlib

Note any extra prompts/chat you need to replicate the plot in Numpy. Then ask the AI model to convert this script to PyTorch and to use its Tensors instead of Numpy. Again note any additional prompts/chat required and how seemless or problematic it was to replicate the tasks from the previous section.

Now that you have a Gaussian function generated by AI, use similar code to create a 2D sine function (i.e. a sune function from PyTorch whose angle is dependent on x and y coordinates of the pixels) and another plot of the resulting Tensor that shows the 'stripes' of this function.

1.3 Demonstration

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

- Change the Gaussian function into a 2D sine or cosine function (1 Mark)

- What do you get when you multiply both the Gaussian and the sine/cosine function together and visualise it? (1 Mark)

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 (2 Marks)

In this part, we will be using PyTorch 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 $z_{n+1} = z_n^2 + c$ 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.

2.1 Tasks

Begin by importing the two libraries we will be using for this part of the lab as usual

```
import torch
import numpy as np
```

Once your PyTorch environment is verified, it is good to ensure that the right computational device has been set and is available to use when needed:

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

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

```
# 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]
```

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 PyTorch inputs to the graph, but now because z_{n+1} will be changing of time, it needs to be a variable tensor. We also keep track of the eventual result (a surrogate for the rate of divergence) as n.

```
# load into PyTorch tensors
x = torch.Tensor(X)
y = torch.Tensor(Y)
z = torch.complex(x, y) #important!
zs = z.clone() #Updated!
ns = torch.zeros_like(z)
```

Next we ensure the tensors are copied to the GPU

```
# transfer to the GPU device
z = z.to(device)
zs = zs.to(device)
ns = ns.to(device)
```

Now we encode the equation $z_{n+1} = z_n^2 + c$ into the graph for the Mandelbrot set over 200 iterations. Once z_{n+1} is computed, we need to check whether it has converged or not. We then reassign z_n for the next iteration by $z_{n+1} \to z_n$ and then add to the n counter to keep track of the rate of divergence.

```
#Mandelbrot Set
for i in range(200):
    #Compute the new values of z: z^2 + x
    zs_ = zs*zs + z
    #Have we diverged with this new value?
    not_diverged = torch.abs(zs_) < 4.0
    #Update variables to compute
    ns += not_diverged
    zs = zs_
```

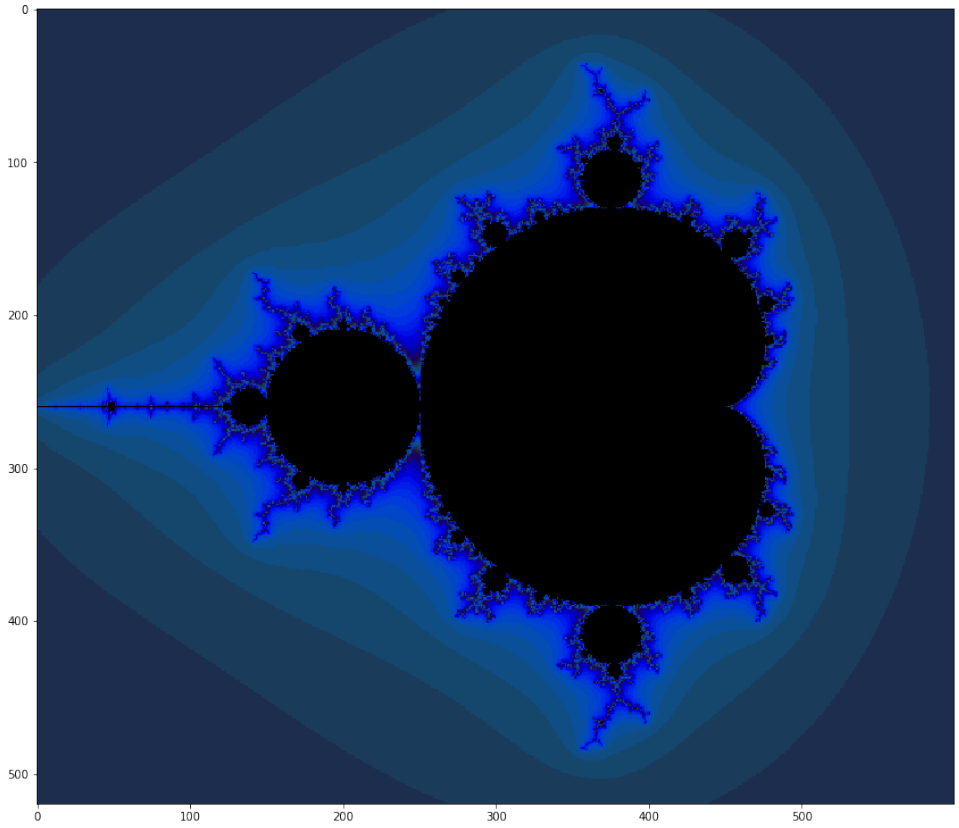
Finally plot the result via the n counter. We use a colour map for the counter and cast it to a 8-bit integer so that we can save it as an image file (such as a PNG) if we wanted to.

```
#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.cpu().numpy()))
plt.tight_layout(pad=0)
plt.show()
```



2.2 AI Tasks

How good is the AI model in generating the Mandelbrot set implemented in PyTorch and that runs on the GPU (i.e. seems to run fast and allows fast rendering) via prompts? Were there any issues encountered and did you need to modify the code, and if so, what were they?

2.3 Demonstration

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 and zooming to another part of the Mandelbrot set and compute the image for it. This may increase the computation time significantly, so choose a value that balances quality of the image and time spent. (1 Mark)
- Modify the code so to show a Julia set rather than the Mandelbrot set. (1 Mark)

3 Part 3 (6 Marks)

For this part, you will learn to ue Git, choose a fractal of your choice and implement it using PyTorch or TF. You will be making your own open-source GitHub repository for your fractal!

3.1 Introduction to Git Course (2 Marks)

For the first part of this section, please complete the Git Short Course: Introduction to Version Control with Git. Firstly, register with edX in order to access the short course. If you do not already have an edX account linked to your UQ student account, you will need to create an account. Additional instructions on accessing this short course (including most up-to-date links, as these change every semester) will be provided as a Blackboard or Eds Discussion board post.

3.2 Fractal Implementation (4 Marks)

Setup a GitHub account using your UQ student email if you do not already have one. If you have a previous GitHub account, you may be asked to verify that this is your personal account during the demo.

3.3 Tasks

Create a new repository for your fractal implementation and use this repository for your fractal code. Once you have completed implementing your fractal, push your commits to this repository so that it shows up on the GitHub page for that repository. There are a number of potential fractals that you could explore, including

- those from this list on Wikipedia (excluding those already covered in this lab sheet).
- 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.

If you have an idea for a fractal algorithm that you would like to attempt that is not on these lists, please enquire with the lab demonstrator or the course coordinator.

Important Notes

- **Note that fractals close to or similar to the Mandelbrot set are not acceptable. Fractals that are not substantially different from those in this document may also not be acceptable. Please check with the teaching team to be sure.**
- **If you use AI models to generate Fractal code and plots etc. You must document all of your prompts and the outputs/reasoning of the model. You must also additionally demonstrate a substantial analysis of the fractal in question. For example, generate code to look at the fractal dimension and show this as part of the output. Or incorporate different visualisations and colours of the fractal outputs.**

- **The teaching team reserves the right to reduce marks significantly if you are not able to show that a reasonable effort has been put into your fractal project. For example, fractal projects that only use a single prompt to generate the results may result in a loss of significant marks.**

3.4 Demonstration

You will need to demonstrate our fractal project by

1. showing the resulting fractal code and its output to the demonstrator and justifying that it uses PyTorch/TF in a major component within the algorithm of the fractal utilising parallelism with PyTorch/TF in a reasonable way (3 Marks)

2. showing that the fractal code is available on a GitHub repository on your account. You may be asked to verify that you own the GitHub account. (1 Mark)