lab1

February 27, 2023

1 A.I. Assignment 1

Set up and familiarize with some Python packages. A simple application with random numbers.

For this assignment you have these tasks:

1. Prepare the working environment
2. Get familiarized with Numpy (create and manipulate arrays)
3. Get familiarized with Matplotlib (display various graphics)
4. Get familiarized with Pytorch tensors (creation and manipulation)
5. Get familiarized with PIL/PILLOW (python image library)
6. An application: Bertrand's paradox (probability)

1.1 Task 1 - Prepare the working environment

In order to present the solved tasks from the laboratories a Jupyter notebook is preferred. For this prepare the working environment as follows:

Install Anaconda distribution and navigator. Create a new environment aiclasses and inside this environment install: numpy, matplotlib, pytorch, pillow.

Environments in Python are like sandboxes that have different versions of Python and/or packages installed in them. You can create, export, list, remove, and update environments. Anaconda allows easy management for these. Here are the links for the packages' documentations if you need further references:

Pytorch Pillow Numpy Matplotlib

After these setups install jupyter notebook and launch it. Create a new notebook related to python 3.

1.2 Task 2 – Get familiarized with Numpy (create and manipulate arrays)

[1]:
```
import numpy as np
```

[2]:
```
my_array = np.array([1, 2, 3, 4])
# works as it would with a standard list
len(my_array)
```

[2]:
```
4
```

The shape array of an array is very useful (we'll see more of it later when we talk about 2D arrays – matrices – and higher-dimensional arrays).

[3]:
```
my_array.shape
```

[3]:
```
(4,)
```

Numpy arrays are typed. This means that by default, all the elements will be assumed to be of the same type (e.g., integer, float, String).

[4]:
```
my_array.dtype
```

[4]:
```
dtype('int32')
```

Numpy arrays have similar functionality as lists! Below, we compute the length, slice the array, and iterate through it (one could identically perform the same with a list).

[7]:
```
print(len(my_array))
print(my_array[2:4])
for element in my_array:
    print(element)
```

```
4
[3 4]
1
2
3
4
```

There are two ways to manipulate numpy arrays:

1. by using the numpy module's methods (e.g., `np.mean()`)
2. by applying the function `np.mean()` with the numpy array as an argument.

[8]:
```
print(my_array.mean())
print(np.mean(my_array))
```

```
2.5
2.5
```

There are many other efficient ways to construct numpy arrays. Here are some commonly used numpy array constructors. Read more details in the numpy documentation.

[9]:
```
np.ones(10) # generates 10 floating point ones
```

[9]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Numpy gains a lot of its efficiency from being typed. That is, all elements in the array have the same type, such as integer or floating point. The default type, as can be seen above, is a float.

(Each float uses either 32 or 64 bits of memory, depending on if the code is running a 32-bit or 64-bit machine, respectively).

[10]:
```
np.dtype(float).itemsize # in bytes (remember, 1 byte = 8 bits)
```

[10]:
```
8
```

[11]:
```
np.ones(10, dtype='int') # generates 10 integer ones
```

[11]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

[12]:
```
np.zeros(10)
```

[12]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Often, you will want random numbers. Use the random constructor!

[13]:
```
np.random.random(10) # uniform from [0,1]
```

[13]:
```
array([0.50929916, 0.99934203, 0.02767909, 0.79934657, 0.38465927,
       0.56439711, 0.52433838, 0.08889742, 0.18240995, 0.36137475])
```

You can generate random numbers from a normal distribution with mean 0 and variance 1:

[14]:
```
normal_array = np.random.randn(1000)
print("The sample mean and standard devation are %f and %f, respectively." %(np.
 ↪mean(normal_array), np.std(normal_array)))
```

```
The sample mean and standard devation are -0.034809 and 0.999227, respectively.
```

[15]:
```
len(normal_array)
```

[15]:
```
1000
```

You can sample with and without replacement from an array. Let's first construct a list with evenly-spaced values:

[17]:
```
grid = np.arange(0., 1.01, 0.1)
grid
```

[17]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Without replacement

[18]:
```
np.random.choice(grid, 5, replace=False)
```

[18]:
```
array([0.1, 0.9, 0.3, 1. , 0.6])
```

[19]:
```
np.random.choice(grid, 20, replace=False)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-9eae7c9e97b5> in <module>
----> 1 np.random.choice(grid, 20, replace=False)

mtrand.pyx in numpy.random.mtrand.RandomState.choice()

ValueError: Cannot take a larger sample than population when 'replace=False'
```

With replacement:

[20]:
```
np.random.choice(grid, 20, replace=True)
```

[20]:
```
array([0.5, 0.7, 0.5, 0.2, 0.5, 0.8, 0.7, 0.4, 0.6, 0. , 0.3, 0. , 0.7,
       1. , 0. , 0.9, 0.1, 0.9, 0.2, 0.6])
```

Let's create 1,000 points between -10 and 10

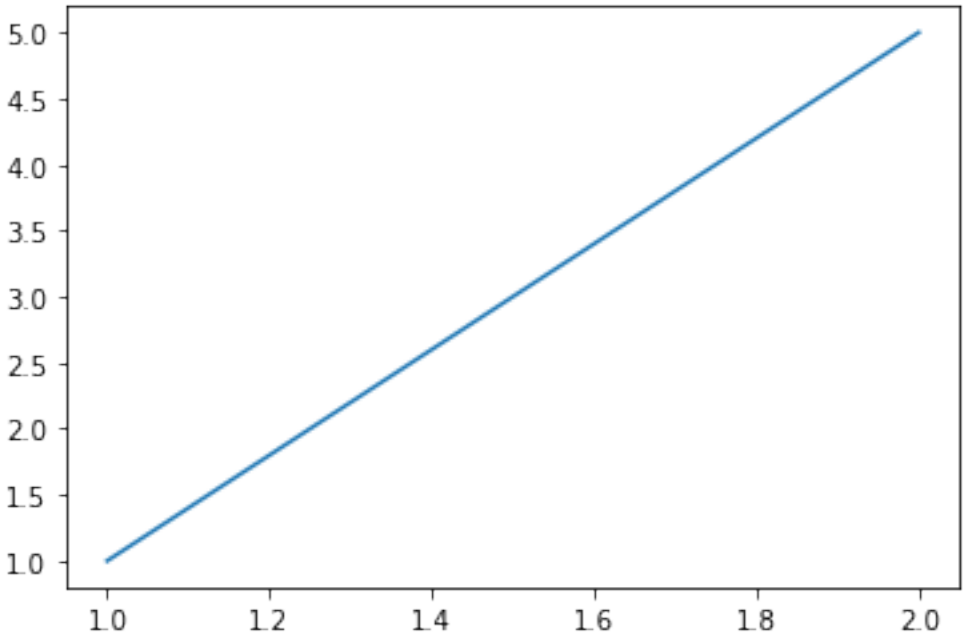
[25]:
```
x = np.linspace(-10, 10, 1000) # linspace() returns evenly-spaced numbers over a␣
 ↪specified interval
x[-5:], x[:5]
```

[25]:
```
(array([ 9.91991992,  9.93993994,  9.95995996,  9.97997998, 10.        ]),
 array([-10.        , -9.97997998, -9.95995996, -9.93993994,
        -9.91991992]))
```

1.3 Task 3 – Get familiarized with Matplotlib (display various graphics)

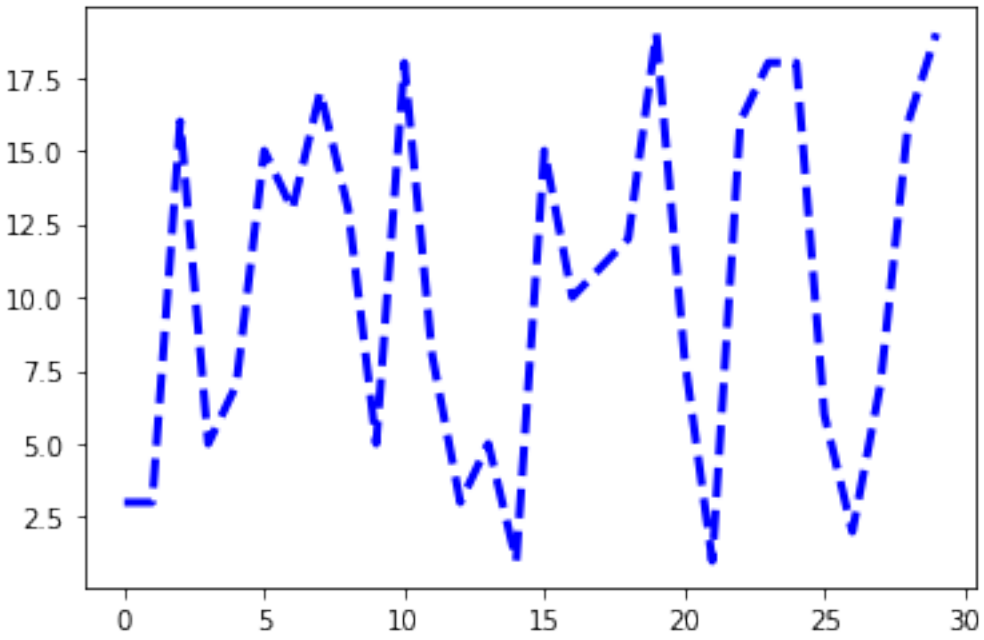
The plot() function is used to draw points (markers) in a diagram. By default, the plot() function draws a line from point to point. The function takes parameters for specifying points in the diagram. Parameter 1 is an array containing the points on the x-axis. Parameter 2 is an array containing the points on the y-axis.

[26]:
```
import matplotlib.pyplot as plt
x = [1, 2]
y = [1, 5]
plt.plot(x, y)
plt.show()
```



Other example where we modify the line style and the color:

[27]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randint(low=1, high=20, size=30)
plt.plot(x, color = 'blue', linewidth=3, linestyle='dashed')
plt.show()
```



1.4 Task 4 – Get familiarized with Pytorch tensors (creation and manipulation)

[28]:
```
import torch
```

Creation Examples:

[31]:
```
x = torch.empty(3, 4)
print(type(x))
print(x)
```

```
<class 'torch.Tensor'>
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

[32]:
```
zeros = torch.zeros(2, 3)
print(zeros)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

[33]:
```
ones = torch.ones(2, 3)
print(ones)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

[34]:
```
torch.manual_seed(1729)
random = torch.rand(2, 3)
print(random)
```

```
tensor([[0.3126, 0.3791, 0.3087],
        [0.0736, 0.4216, 0.0691]])
```

Observe the last example with the seed specified. Run the following example and observe the "random" values generated:

[35]:
```
torch.manual_seed(1)
random1 = torch.rand(2, 3)
print(random1)

random2 = torch.rand(2, 3)
print(random2)

torch.manual_seed(1)
random3 = torch.rand(2, 3)
print(random3)
```

```
random4 = torch.rand(2, 3)
print(random4)
```

```
tensor([[0.7576, 0.2793, 0.4031],
        [0.7347, 0.0293, 0.7999]])
tensor([[0.3971, 0.7544, 0.5695],
        [0.4388, 0.6387, 0.5247]])
tensor([[0.7576, 0.2793, 0.4031],
        [0.7347, 0.0293, 0.7999]])
tensor([[0.3971, 0.7544, 0.5695],
        [0.4388, 0.6387, 0.5247]])
```

Tensor Shapes

On performing operations on two or more tensors, they will need to be of the same shape - that is, having the same number of dimensions and the same number of cells in each dimension. For that, we have the `torch.*_like()` methods:

[37]:
```
x = torch.empty(2, 2, 3)
print(x.shape)
print(x)

empty_like_x = torch.empty_like(x)
print(empty_like_x.shape)
print(empty_like_x)

zeros_like_x = torch.zeros_like(x)
print(zeros_like_x.shape)
print(zeros_like_x)

ones_like_x = torch.ones_like(x)
print(ones_like_x.shape)
print(ones_like_x)

rand_like_x = torch.rand_like(x)
print(rand_like_x.shape)
print(rand_like_x)
```

```
torch.Size([2, 2, 3])
tensor([[[0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.]]])
torch.Size([2, 2, 3])
tensor([[[0., 0., 0.],
         [0., 0., 0.]],
```

```
        [[0., 0., 0.],
         [0., 0., 0.]]])
torch.Size([2, 2, 3])
tensor([[[0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.]]])
torch.Size([2, 2, 3])
tensor([[[1., 1., 1.],
         [1., 1., 1.]],

        [[1., 1., 1.],
         [1., 1., 1.]]])
torch.Size([2, 2, 3])
tensor([[[0.6826, 0.3051, 0.4635],
         [0.4550, 0.5725, 0.4980]],

        [[0.9371, 0.6556, 0.3138],
         [0.1980, 0.4162, 0.2843]]])
```

Moving to GPU

First, we should check whether a GPU is available, with the is_available() method.

If you do not have a CUDA-compatible GPU and CUDA drivers installed, the executable cells in this section will not execute any GPU-related code.

[40]:
```
if torch.cuda.is_available():
    print('We have a GPU!')
else:
    print('Sorry, CPU only.')
```

```
Sorry, CPU only.
```

A common way to handle this situation is this:

[41]:
```
if torch.cuda.is_available():
    my_device = torch.device('cuda')
else:
    my_device = torch.device('cpu')
print('Device: {}'.format(my_device))

x = torch.rand(2, 2, device=my_device)
print(x)
```

```
Device: cpu
tensor([[0.3398, 0.5239],
        [0.7981, 0.7718]])
```

1.5 Task 5 – Get familiarized with PIL/PILLOW (python image library)

A. Display an image with pillow (the image must be in the same folder for these examples to run, and we considered the name of the image file `opera.jpg`):

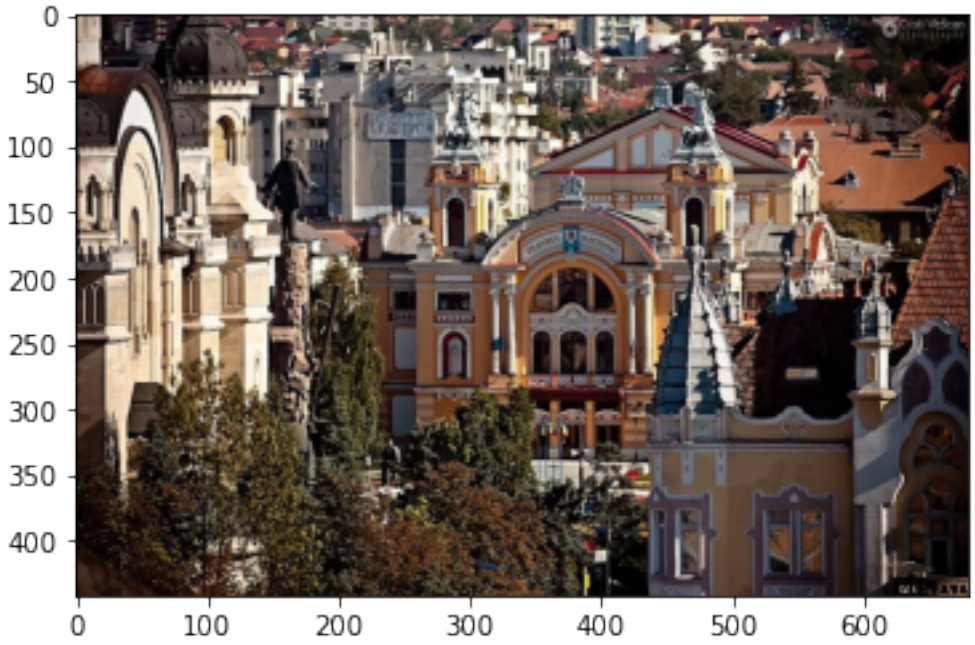
[45]:
```
# load and show an image with Pillow
from PIL import Image
# load the image
image = Image.open('opera.jpg')
# summarize some details about the image
print(image.format)
print(image.mode)
print(image.size)
# show the image
image.show()
```

```
JPEG
RGB
(680, 442)
```

B. Convert the image to a numpy array:

[46]:
```
# load and display an image with Matplotlib
from matplotlib import image
from matplotlib import pyplot
# load image as pixel array
data = image.imread('opera.jpg')
# summarize shape of the pixel array
print(data.dtype)
print(data.shape)
# display the array of pixels as an image
pyplot.imshow(data)
pyplot.show()
```

```
uint8
(442, 680, 3)
```



C. Resize an image to a specific dimension:

[47]:
```
# create a thumbnail of an image
from PIL import Image
# load the image
image = Image.open('opera.jpg')
# report the size of the image
print(image.size)
# create a thumbnail and preserve aspect ratio
image.thumbnail((100,100))
# report the size of the thumbnail
print(image.size)
```

```
(680, 442)
(100, 65)
```

D. Other operations can be found at the address How to Load and Manipulate Images for Deep Learning in Python With PIL/Pillow

1.6 Task 6 – An application: Bertrand's paradox (probability)

Consider the Bertrand paradox.

For each of the three cases generate a sample of 100 elements (chords) and compute the ratio of the chords bigger than the side of the equilateral triangle inscribed in the circle.

The circle radius is equal with 1 and the set of chords will contain the cartesian coordinates of the edges of the chords. The origin of the coordinate system is in the origin of the circle.

Draw using matplotlib each set of chords for each case respectively.

[ ]: