Report on Learning Python

NumPy and Matplotlib using Google Colab

1. NumPy

Python's Numpy package is the foundation for scientific computing. It includes a high-performance multidimensional array object as well as utilities for manipulating them. The package provide similar functions to MATLAB.

Before using NumPy module, it is necessary to import. There are several ways to import NumPy. The standard approach of simple import is using the statement: import numpy. Contrasting to the standard importing method, it is also available to import as the followings: import numpy as np, and from numpy import *. Importing numpy as np is providing the name np for numpy. From numpy import * is equivalent to the standard approach since it is importing all functions in numpy.

```
import numpy
import numpy as np      # --> most commonly used
from numpy import *
```

A NumPy array is a grid of identical-type items indexed by a tuple of nonnegative integers. The array's rank is the number of dimensions; the shape is a tuple of numbers indicating the array's size along each dimension.

Arrays in numpy are similar to lists in Python, and they can be created from a list. They can form multi-dimension. These are the basic elements of array that can implement the kernels in the ConvNet, convolution.

```
import numpy as np

array = np.array([1, 4, 5, 8], float)
print(array)
type(array)

array3d = np.array([[1, 4, 5],
                    [3, 8, 2],
                    [7, 9, 6]], float)
```

As the advanced functions, arrays provide methods that can reshape the dimensions, copy the function, convert the array raw data to binary string, transpose the axes of arrays, concatenate

arrays and slice.

They are also available for simple and standard mathematical arithmetic to each array elements. It signifies that the math on array can compile with matching sizes. However, arrays that do not match will be broadcasted by Python to perform the operations. This automatic broadcasting provided in Python sometimes occurs error. The equivalent method can be done by using [newaxis, :] constant to reshape the array and apply math for each.

```
import numpy as np

# the inputs in the bracket represent; [start:end:step]

slicing1 = array3d[1, :]
# get the second element, indexing from the start and end
# the execution >>> array([3., 8., 2.])

slicing2 = array3d[:,2]
# get the third steps of every element (indexing start from 0)
# the execution >>> array([5., 2., 6.])

reshapeArray = np.array(range(6), float)
reshapeArray = reshapeArray.reshape((3, 2))
# the execution >>> array([[0., 1.],
#                          [2., 3.],
#                          [4., 5.])
```

The array operations mostly satisfies and are the same with the list functions provided by Python itself. It has sorting, calculating the means, obtaining minimum and maximum, and simple comparisons. The difference from list is that array has functions that return the indices of the minimum and maximum values, which are argmin and argmax, as well as computing the statistical quantities such as variance and standard deviation.

```
import numpy as np

# array operations provided similar to List
array3d.sum()
# np.sum(array3d)
array3d.prod()
# np.prod(array3d)
array3d.min()
array3d.max()
array3d.sort()

# argmin and argmax
array3d.argmin()
array3d.argmax()

# statistical quatities
array3d.mean()
```

```
array3d.var()   # variation
array3d.std()   # standard deviation
```

NumPy also provides many functions for performing standard vector and matrix multiplication routines. They are available for dot product examination with dot() method. It also generalizes to matrix multiplication, and possible for inner, outer and cross product implementations. They are utilizable in convolution and fully-connected layers of CNN, where it needs the dot product to convolute the image with filters. Not only the dot products, but it also provide the methods that can generate eigenvalues, eigen vectors, singular value decomposition and inverse of matrices.

```
import numpy as np

dotArray1 = np.array([1, 2, 3], float)
dotArray2 = np.array([0, 1, 1], float)

np.dot(dotArray1, dotArray2)
np.outer(dotArray1, dotArray2)
np.inner(dotArray1, dotArray2)
np.cross(dotArray1, dotArray2)
```

As for the last functionality of NumPy, it supplies methods for working with polynomial. It is possible to find the polynomial coefficients with given set of roots, or opposingly finding the roots of given polynomial coefficients. These functions are written with poly() and roots() methods. It is also able to integrate and derive the polynomial functions, evaluating the polynomial at a particular point, as well as fitting a polynomial of a specified order to a set of data by using polyint(), polyder(), polyval() and polyfit() correspondingly.

```
import numpy as np

np.poly([-1, 1, 1, 10])
# the execution >>> array([1, -11, 9, 11, -10])
# the list of elements [-1, 1, 1, 10] represent the roots of the polynomial x^(4)-11x^(3)+9x^(2)

np.roots([1, -11, 9, 11, -10])
# the execution >>> array([-1, 1, 1, 10])

np.polyint([1, 1, 1, 1])
# >>> array([0.25, 0.333333333, 0.5, 1., 0.])

np.polyder([0.25, 0.333333333, 0.5, 1., 0.])
# >>> array([1., 1., 1., 1.])

np.polyval([1, -2, 0, 2], 4)
# >>> 34

x = [1, 2, 3, 4, 5, 6, 7, 8]
y = [0, 2, 1, 3, 7, 10, 11, 19]
np.polyfit(x, y, 2)
# >>> array([ 0.375 , -0.88690476, 1.05357143])
```

2. Matplotlib

Matplotlib is a library for charting data, that is the most popular Python 2D graphics tool kit. It offers a simple way to view Python data as well as publication-quality figures in a variety of formats, which provides a plotting system similar to MATLAB.

Before any executions, %matplotlib magic is added to connect to a GUI loop. This is operated with “%matplotlib inline”.

```
%matplotlib inline
```

There are two main functions provided by Matplotlib. First of all, plotting 2D data in graphs is possible. Importing matplotlib.pyplot module and computing the 2D data with numpy ahead of the process is required. Then, plotting the output data with plot() method will allow the analysis. It is important to note that to visualize, it is necessary to use show() function before running the code. Equivalent to the MATLAB plotting functions, the labeling, putting the title and legends can be done with xlabel(), ylabel(), title() and legend() methods. When plotting, the module also provides the subplot function that data can be appeared in two different windows. The first two arguments assigned represent the height and width of the window, where the third value is for the window hierarchies for demonstration.

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')
```

```
25
26 # Set the second subplot as active, and make the second plot.
27 plt.subplot(2, 1, 2)
28 plt.plot(x, y_cos)
29 plt.title('Cosine')
30
31 # Show the figure.
32 plt.show()
```

For an additional function to view the image, scipy is a thing to learn. It provides is imshow() that displays images. The images are searched from the same folder hierarchy, and returns error if the image cannot be found. Although it can discover the subfolders from its current position, it cannot do such things for higher ordered folders. By using the function, images can be showed with modified colors by rescaling the image channels or displayed in numbers by printing the values read from the image.

```
 1 import numpy as np
 2 from scipy.misc import imread, imresize
 3 import matplotlib.pyplot as plt
 4
 5 img = imread('assets/cat.jpg')
 6 img_tinted = img * [1, 0.95, 0.9]
 7
 8 # Show the original image
 9 plt.subplot(1, 2, 1)
10 plt.imshow(img)
11
12 # Show the tinted image
13 plt.subplot(1, 2, 2)
14
15 # A slight gotcha with imshow is that it might give strange results
16 # if presented with data that is not uint8. To work around this, we
17 # explicitly cast the image to uint8 before displaying it.
18 plt.imshow(np.uint8(img_tinted))
19 plt.show()
```