CHAPTER 10

NumPy

NumPy is a Python library optimized for numerical computing. It bears close semblance with MATLAB and is equally as powerful when used in conjunction with other packages such as SciPy for various scientific functions, Matplotlib for visualization, and Pandas for data analysis. NumPy is short for numerical python.

NumPy's core strength lies in its ability to create and manipulate n-dimensional arrays. This is particularly critical for building machine learning and deep learning models. Data is often represented in a matrix-like grid of rows and columns, where each row represents an observation and each column a variable or feature. Hence, NumPy's 2-D array is a natural fit for storing and manipulating datasets.

This tutorial will cover the basics of NumPy to get you very comfortable working with the package and also get you to appreciate the thinking behind how NumPy works. This understanding forms a foundation from which one can extend and seek solutions from the NumPy reference documentation when a specific functionality is needed.

To begin using NumPy, we'll start by importing the NumPy module:

```
import numpy as np
```

NumPy 1-D Array

Let's create a simple 1-D NumPy array:

```
my_array = np.array([2,4,6,8,10])
my_array
'Output': array([ 2,  4,  6,  8, 10])
# the data-type of a NumPy array is the ndarray
type(my_array)
'Output': numpy.ndarray
```


E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-4470-8_10

```
# a NumPy 1-D array can also be seen a vector with 1 dimension
my_array.ndim
'Output': 1
# check the shape to get the number of rows and columns in the array \
# read as (rows, columns)
my_array.shape
'Output': (5,)
```

We can also create an array from a Python list.

```
my_list = [9, 5, 2, 7]
type(my_list)
'Output': list
# convert a list to a numpy array
list_to_array = np.array(my_list) # or np.asarray(my_list)
type(list_to_array)
'Output': numpy.ndarray
```

Let's explore other useful methods often employed for creating arrays.

```
# create an array from a range of numbers
np.arange(10)
'Output': [0 1 2 3 4 5 6 7 8 9]
# create an array from start to end (exclusive) via a step size - (start,
stop, step)
np.arange(2, 10, 2)
'Output': [2 4 6 8]
# create a range of points between two numbers
np.linspace(2, 10, 5)
'Output': array([  2.,   4.,   6.,   8.,  10.])
# create an array of ones
np.ones(5)
'Output': array([ 1.,  1.,  1.,  1.,  1.])
# create an array of zeros
np.zeros(5)
'Output': array([ 0.,  0.,  0.,  0.,  0.])
```

NumPy Datatypes

NumPy boasts a broad range of numerical datatypes in comparison with vanilla Python. This extended datatype support is useful for dealing with different kinds of signed and unsigned integer and floating-point numbers as well as booleans and complex numbers for scientific computation. NumPy datatypes include the **bool_**, **int**(8,16,32,64), **uint**(8,16,32,64), **float**(16,32,64), **complex**(64,128) as well as the **int_**, **float_**, and **complex_**, to mention just a few.

The datatypes with a _ appended are base Python datatypes converted to NumPy datatypes. The parameter **dtype** is used to assign a datatype to a NumPy function. The default NumPy type is **float_**. Also, NumPy infers contiguous arrays of the same type.

Let's explore a bit with NumPy datatypes:

```
# ints
my_ints = np.array([3, 7, 9, 11])
my_ints.dtype
'Output': dtype('int64')

# floats
my_floats = np.array([3., 7., 9., 11.])
my_floats.dtype
'Output': dtype('float64')

# non-contiguous types - default: float
my_array = np.array([3., 7., 9, 11])
my_array.dtype
'Output': dtype('float64')

# manually assigning datatypes
my_array = np.array([3, 7, 9, 11], dtype="float64")
my_array.dtype
'Output': dtype('float64')
```