Mathematical Foundations for Artificial Intelligence using Python – Series 1

Dr. Thyagaraju G S

Contents

1. Linear Algebra using Python

1. Introduction to Linear Algebra
2. Linear Algebra and Machine Learning
3. Examples of Linear Algebra in Machine Learning
4. Introduction to Numpy Arrays
5. Index Slice and Reshape Arrays
6. Numpy Arrays Broad Casting
7. Vector and Vectors Matematics
8. Vector Norms

1. Linear Algebra using Python

9. Matrices and Matrix Arithmetic
10. Types of Matrices
11. Matrix Operation
12. Sparse Matrices
13. Tensor and Tensor Arithmetic
14. Matrix Decomposition
15. Eigen Decomposition
16. Singular Value Decomposition

1. Linear Algebra using Python

17. Introduction to Multivariate Statistics
18. Principal Component Analysis
19. Linear Regression
20. Basic Math Notation

1. Introduction

- Linear algebra is a field of mathematics that is universally agreed to be a prerequisite to a deeper understanding of machine learning.
- Linear algebra is the mathematics of data.
- Linear algebra has had a marked impact on the field of statistics.
- Linear algebra underlies many practical mathematical tools, such as Fourier series and computer graphics.

Introduction

- Matrices and vectors are the language of data. Linear algebra is about linear combinations.
- Linear algebra is the study of lines and planes, vector spaces and mappings that are required for linear transforms.
- A linear equation is just a series of terms and mathematical operations where some terms are unknown; for example:

$$y = 4 \times x + 1 \tag{1.1}$$

Equations like this are linear in that they describe a line on a two-dimensional graph

We can line up a system of equations with the same form with two or more unknowns; for example:

$$\begin{aligned} y &= 0.1 \times x_1 + 0.4 \times x_2 \\ y &= 0.3 \times x_1 + 0.9 \times x_2 \\ y &= 0.2 \times x_1 + 0.3 \times x_2 \\ &\dots \end{aligned} \tag{1.2}$$

We can write this compactly using linear algebra notation as:

$$y = A \cdot b \tag{1.3}$$

Linear Algebra and Statistics

Linear algebra is a valuable tool in other branches of mathematics, especially statistics. Some clear fingerprints of linear algebra on statistics and statistical methods include:

- Use of vector and matrix notation, especially with multivariate statistics.
- Solutions to least squares and weighted least squares, such as for linear regression.
- Estimates of mean and variance of data matrices.
- The covariance matrix that plays a key role in multinomial Gaussian distributions.
- Principal component analysis for data reduction that draws many of these elements together.

Applications of Linear Algebra

As linear algebra is the mathematics of data, the tools of linear algebra are used in many domains. Briefly they are:

- Matrices in Engineering, such as a line of springs.
- Graphs and Networks, such as analyzing networks.
- Markov Matrices, Population, and Economics, such as population growth.
- Linear Programming, the simplex optimization method.
- Fourier Series: Linear Algebra for functions, used widely in signal processing.
- Linear Algebra for statistics and probability, such as least squares for regression.
- Computer Graphics, such as the various translation, rescaling and rotation of images.

Quiz

Which of the following statements are wrong :

1. Linear algebra is the mathematics of data.
2. Linear algebra has had a marked impact on the field of statistics.
3. Linear algebra underlies many practical mathematical tools, such as Fourier series and computer graphics.
4. All above
5. None

2. Linear Algebra and Machine Learning

Linear algebra is a field of mathematics that could be called the mathematics of data. It is undeniably a pillar of the field of machine learning, and many recommend it as a prerequisite subject to study prior to getting started in machine learning.

2.1 Not everyone should learn linear algebra, that it depends where you are in your process of learning machine learning.

2.2 Reasons why a deeper understanding of linear algebra is required for intermediate machine learning practitioners.

2.1 Reasons to NOT Learn Linear Algebra

- **It's not required**. Having an appreciation for the abstract operations that underly some machine learning algorithms is not required in order to use machine learning as a tool to solve problems.
- **It's Slow :** Taking months to years to study an entire related field before machine learning will delay you achieving your goals of being able to work through predictive modeling problems.
- **It's a huge field:** Not all of linear algebra is relevant to theoretical machine learning, let alone applied machine learning.

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. However, because linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it.

— Page 31, *Deep Learning*, 2016.

2.2.1 Learn Linear Algebra Notation

You need to be able to read and write vector and matrix notation. Algorithms are described in books, papers and on websites using vector and matrix notation. Linear algebra is the mathematics of data and the notation allows you to describe operations on data precisely with specific operators. You need to be able to read and write this notation.

This skill will allow you to:

- Read descriptions of existing algorithms in textbooks.
- Internet and implement descriptions of new methods in research papers.
- Concisely describe your own methods to other practitioners.

2.2.2 Learn Linear Algebra Arithmetic

- You need to know how to add, subtract, and multiply scalars, vectors, and matrices.
- A challenge for newcomers to the field of linear algebra are operations such as **matrix multiplication and tensor multiplication**.
- An understanding of how vector and matrix operations are implemented is required as a part of being able to effectively read and write matrix notation.

2.2.3 Learn Linear Algebra for Statistics

- You must learn linear algebra in order to be able to learn statistics. Especially multivariate statistics.
- In order to be able to read and interpret statistics, you must learn the notation and operations of linear algebra.
- Modern statistics uses both the notation and tools of linear algebra to describe the tools and techniques of statistical methods.
- From vectors for the means and variances of data, to covariance matrices that describe the relationships between multiple Gaussian variables.

2.2.4 Learn Matrix Factorization

- Matrix factorization/decomposition is a key tool in linear algebra and used widely as an element of many more complex operations in both linear algebra (such as the matrix inverse) and machine learning (least squares).
- Further, there are a range of different matrix factorization methods, each with different strengths and capabilities, some of which you may recognize as ”machine learning” methods, such as Singular-Value Decomposition, or SVD for short, for data reduction.
- In order to read and interpret higher-order matrix operations, you must understand matrix factorization.

2.2.5 Learn Linear Least Squares

- Linear algebra was originally developed to solve systems of linear equations. These are equations where there are more equations than there are unknown variables. As a result, they are challenging to solve arithmetically because there is no single solution as there is no line or plane can fit the data without some error. Problems of this type can be framed as the minimization of squared error, called least squares, and can be recast in the language of linear algebra, called linear least squares.
- Linear least squares problems can be solved efficiently on computers using matrix operations such as matrix factorization.
- Least squares is most known for its role in the solution to linear regression models, but also plays a wider role in a range of machine learning algorithms.

Quiz

Which of the following statement is not related:

- Learn Linear Algebra Notation
- Learn Linear Algebra Arithmetic
- Learn Linear Algebra for Statistics
- Learn Matrix Factorization
- Learn Linear Least Squares
- Learn Linear Algebra for Calculus

3. Examples of Linear Algebra in Machine Learning

1. Dataset and Data Files
2. Images and Photographs
3. One Hot Encoding
4. Linear Regression
5. Regularization
6. Principal Component Analysis
7. Singular-Value Decomposition
8. Latent Semantic Analysis
9. Recommender Systems
10. Deep Learning

3.1 Dataset and Data Files

- In machine learning, you fit a model on a dataset. This is the table like set of numbers where each row represents an observation and each column represents a feature of the observation.
- Each Data is matrix X and vector y.

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa

3.2 Images and Photographs

- Each image that you work with is itself a table structure with a width and height and one pixel value in each cell for black and white images or 3 pixel values in each cell for a color image.
- A photo is yet another example of a matrix from linear algebra. Operations on the image, such as cropping, scaling, shearing and so on are all described using the notation and operations of linear algebra.

3.3 One Hot Encoding

```
red
green
blue
...
```

Listing 3.2: Example of a categorical variable.

Might be encoded as:

```
red, green, blue
  1,    0,     0
  0,    1,     0
  0,    0,     1
...
```

Listing 3.3: Example of a one hot encoded categorical variable.

3.4 Linear Regression

- Linear regression is an old method from statistics for describing the relationships between variables. Even the common way of summarizing the linear regression equation uses linear algebra notation:

$$y = A \cdot b$$

Where y is the output variable A is the dataset and b are the model coefficients.

3.5 Regularization

- A technique that is often used to encourage a model to minimize the size of coefficients while it is being fit on data is called regularization.
- Common implementations include the L2 and L1 forms of regularization.
- Both of these forms of regularization are in fact a measure of the magnitude or length of the coefficients as a vector and are methods lifted directly from linear algebra called the vector norm.

3.6 Principal Component Analysis

- Methods for automatically reducing the number of columns of a dataset are called dimensionality reduction, and perhaps the most popular is method is called the principal component analysis or PCA for short.
- This method is used in machine learning to create projections of high-dimensional data for both visualization and for training models.

3.7 Singular-Value Decomposition

- Another popular dimensionality reduction method is the singular-value decomposition method or SVD for short.
- It is a matrix factorization method from the field of linear algebra.
- It has wide use in linear algebra and can be used directly in applications such as feature selection, visualization, noise reduction and more.

3.8 Latent Semantic Analysis

- In the sub-field of machine learning for working with text data called natural language processing, it is common to represent documents as large matrices of word occurrences.
- For example, the columns of the matrix may be the known words in the vocabulary and rows may be sentences, paragraphs, pages or documents of text with cells in the matrix marked as the count or frequency of the number of times the word occurred.
- This is a sparse matrix representation of the text. Matrix factorization methods such as the singular-value decomposition can be applied to this sparse matrix which has the effect of distilling the representation down to its most relevant essence. Documents processed in thus way are much easier to compare, query and use as the basis for a supervised machine learning model.
- This form of data preparation is called Latent Semantic Analysis or LSA for short, and is also known by the name Latent Semantic Indexing or LSI.

3.9 Recommender Systems

- Predictive modeling problems that involve the recommendation of products are called recommender systems, a sub-field of machine learning. Examples include the recommendation of books based on previous purchases and purchases by customers like you on Amazon, and the recommendation of movies and TV shows to watch based on your viewing history and viewing history of subscribers like you on Netflix. The development of recommender systems is primarily concerned with linear algebra methods. A simple example is in the calculation of the similarity between sparse customer behavior vectors using distance measures such as Euclidean distance or dot products. Matrix factorization methods like the singular-value decomposition are used widely in recommender systems to distill item and user data to their essence for querying and searching and comparison.

3.10 Deep Learning

- Deep learning methods are routinely achieve state-of-the-art results on a range of challenging problems such as machine translation, photo captioning, speech recognition and much more.
- At their core, the execution of neural networks involves linear algebra data structures multiplied and added together.
- Scaled up to multiple dimensions, deep learning methods work with vectors, matrices and even tensors of inputs and coefficients, where a tensor is a matrix with more than two dimensions.
- Linear algebra is central to the description of deep learning methods via matrix notation to the implementation of deep learning methods such as Google's TensorFlow Python library that has the word "tensor" in its name.

4. Numpy Arrays

1. NumPy N-dimensional Array
2. Functions to Create Arrays
3. Combining Arrays

4.1 NumPy N-dimensional Array

- NumPy is a Python library that can be used for scientific and numerical applications and is the tool to use for linear algebra operations.
- The main data structure in NumPy is the ndarray, which is a shorthand name for N-dimensional array. When working with NumPy, data in an ndarray is simply referred to as an array. It is a fixed-sized array in memory that contains data of the same type, such as integers or floating point values.

Example1

```
# create array
from numpy import array
# create array
l = [1.0, 2.0, 3.0]
a = array(l)
# display array
print(a)
# display array shape
print(a.shape)
# display array data type
print(a.dtype)
```

```
[1. 2. 3.]
(3,)
float64
```

4.2 Functions to Create Arrays

- There are more convenience functions for creating fixed-sized arrays that you may encounter or be required to use

4.2.1 Empty

- The empty() function will create a new array of the specified shape. The argument to the function is an array or tuple that specifies the length of each dimension of the array to create.
- The values or content of the created array will be random and will need to be assigned before use.
- The example below creates an empty 3×3 two-dimensional array

```python
# create empty array
from numpy import empty
a = empty([3,3])
print(a)
```

```
[[1.03159552e-308 4.98979669e-111 6.62976411e-270]
 [4.89124989e-322 2.17292369e-311 1.82004088e-306]
 [8.90264492e-307 3.50919059e-191 1.06742258e-287]]
```

4.2.2 Zeros

- The zeros() function will create a new array of the specified size with the contents filled with zero values. The argument to the function is an array or tuple that specifies the length of each dimension of the array to create. The example below creates a 3×5 zero two-dimensional array

```python
# create zero array
from numpy import zeros
a = zeros([3,5])
print(a)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

4.2.3 Ones

- The ones() function will create a new array of the specified size with the contents filled with one values. The argument to the function is an array or tuple that specifies the length of each dimension of the array to create. The example below creates a 5-element one-dimensional array.

```
# create one array
from numpy import ones
a = ones([5])
print(a)
```

```
[1. 1. 1. 1. 1.]
```

4.3 Combining Arrays

- Vertical Stack
- Horizontal Stack

4.3.1 Vertical Stack

- Given two or more existing arrays, you can stack them vertically using the vstack() function. For example, given two one-dimensional arrays, you can create a new two-dimensional array with two rows by vertically stacking them.
- This is demonstrated in the example below

```python
# create array with vstack
from numpy import array
from numpy import vstack
# create first array
a1 = array([1,2,3])
print(a1)
# create second array
a2 = array([4,5,6])
print(a2)
# vertical stack
a3 = vstack((a1, a2))
print(a3)
print(a3.shape)
```

```
[1 2 3]
[4 5 6]
[[1 2 3]
 [4 5 6]]
(2, 3)
```

4.3.2 Horizontal Stack

- Given two or more existing arrays, you can stack them horizontally using the hstack() function. For example, given two one-dimensional arrays, you can create a new one-dimensional array or one row with the columns of the first and second arrays concatenated.
- This is demonstrated in the example below.

```python
# create array with hstack
from numpy import array
from numpy import hstack
# create first array
a1 = array([1,2,3])
print(a1)
# create second array
a2 = array([4,5,6])
print(a2)
# create horizontal stack
a3 = hstack((a1, a2))
print(a3)
print(a3.shape)
```

```
[1 2 3]
[4 5 6]
[1 2 3 4 5 6]
(6,)
```

5.Index, Slice and Reshape NumPy Arrays

- 1. From List to Arrays
- 2. Array Indexing
- 3. Array Slicing
- 4. Array Reshaping

5.1.From List to Arrays

One-Dimensional List to Array

```
# one dimensional list to arrays
# create one-dimensional array
from numpy import array
# list of data
data = [11, 22, 33, 44, 55]
# array of data
data = array(data)
print(data)
print(type(data))
```

```
[11 22 33 44 55]
<class 'numpy.ndarray'>
```

Two-Dimensional List of Lists to Array

```
# create two-dimensional array
from numpy import array
# list of data
data = [[11, 22],
        [33, 44],
        [55, 66]]
# array of data
data = array(data)
print(data)
print(type(data))
```

```
[[11 22]
 [33 44]
 [55 66]]
<class 'numpy.ndarray'>
```

5.2 Array Indexing

- Once your data is represented using a NumPy array, you can access it using indexing. Let’s look at some examples of accessing data via indexing.

One-Dimensional Indexing

```
# One-Dimensional Indexing
# index a one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
# index data
print(data[0])
print(data[4])
```

```
11
55
```

```python
# index array out of bounds
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
# index data
print(data[5])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent
<ipython-input-4-e856e9efc9a6> in <module>
      4 data = array([11, 22, 33, 44, 55])
      5 # index data
----> 6 print(data[5])

IndexError: index 5 is out of bounds for axis 0 with size 5
```

Two dimensional indexing

```
# negative array indexing
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
# index data
print(data[-1])
print(data[-5])
```

```
55
11
```

Two-Dimensional Indexing

```
# index two-dimensional array
from numpy import array
# define array
data = array([ [11, 22],
               [33, 44],
               [55, 66]])
# index data
print(data[0,0])
```

```
11
```

Example of indexing the first rown of a two-dimensional array.

```
# Example of indexing the first row of a two-dimensional array.
# index row of two-dimensional array
from numpy import array
# define array
data = array([ [11, 22], [33, 44], [55, 66]])
# index data
print(data[1,])
```

```
[33 44]
```

5.3 Array Slicing

- Structures like lists and NumPy arrays can be sliced. This means that a subsequence of the structure can be indexed and retrieved. This is most useful in machine learning when specifying input variables and output variables, or splitting training rows from testing rows. Slicing is specified using the colon operator : with a from and to index before and after the column respectively.
- The slice extends from the from index and ends one item before the to index.
- data[from:to]

One-Dimensional Slicing

```
# One-Dimensional Slicing
# slice a one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data[:])
```

```
[11 22 33 44 55]
```

Example of slicing a subset of a one-dimensional array.

```
:  # slice a subset of a one-dimensional array
   from numpy import array
   # define array
   data = array([11, 22, 33, 44, 55])
   print(data[0:1])
```

```
[11]
```

Negative Slicing

:
```
# negative slicing of a one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data[-2:])
```

```
[44 55]
```

Two-Dimensional Slicing

- Split Input and Output Features
  - It is common to split your loaded data into input variables (X) and the output variable (y).
  - We can do this by slicing all rows and all columns up to, but before the last column, then separately indexing the last column.
  - For the input features, we can select all rows and all columns except the last one by specifying : for in the rows index, and :-1 in the columns index.
    - X = [:, :-1]
  - For the output column, we can select all rows again using : and index just the last column by specifying the -1 index.
    - y = [:, -1]

```python
# split input and output data
from numpy import array
# define array
data = array([[11, 22, 33],
              [44, 55, 66],
              [77, 88, 99]])
# separate data
X, y = data[:, :-1], data[:, -1]
print(X)
print(y)
```

```
[[11 22]
 [44 55]
 [77 88]]
[33 66 99]
```

Split Train and Test Rows

- It is common to split a loaded dataset into separate train and test sets. This is a splitting of rows where some portion will be used to train the model and the remaining portion will be used to estimate the skill of the trained model. This would involve slicing all columns by specifying : in the second dimension index.
- The training dataset would be all rows from the beginning to the split point.
  - train = data[:split, :]
- The test dataset would be all rows starting from the split point to the end of the dimension.
  - test = data[split:, :]

```python
# split train and test data
from numpy import array
# define array
data = array([ [11, 22, 33],
               [44, 55, 66],
               [77, 88, 99]])
# separate data
split = 2
train,test = data[:split,:],data[split:,:]
print(train)
print(test)
```

```
[[11 22 33]
 [44 55 66]]
[[77 88 99]]
```

5.4 Array Reshaping

- After slicing your data, you may need to reshape it. For example, some libraries, such as scikit-learn, may require that a one-dimensional array of output variables (y) be shaped as a two-dimensional array with one column and outcomes for each column. Some algorithms, like the Long Short-Term Memory recurrent neural network in Keras, require input to be specified as a three-dimensional array comprised of samples, timesteps, and features. It is important to know how to reshape your NumPy arrays so that your data meets the expectation of specific Python libraries. We will look at these two examples

Data Shape

- NumPy arrays have a shape attribute that returns a tuple of the length of each dimension of the array.

Example of accessing shape for a one-dimensional array

```
# shape of one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data.shape)
```

```
(5,)
```

Example of accessing shape for a two-dimensional array.

```
:
# shape of a two-dimensional array
from numpy import array
# list of data
data = [[11, 22], [33, 44], [55, 66]]
# array of data
data = array(data)
print(data.shape)
```

```
(3, 2)
```

Example of accessing shape for a two-dimensional array in terms of rows and columns.

```
# row and column shape of two-dimensional array
from numpy import array
# list of data
data = [[11, 22],[33, 44], [55, 66]]
# array of data
data = array(data)
print('Rows: %d' % data.shape[0])
print('Cols: %d' % data.shape[1])
```

```
Rows: 3
Cols: 2
```

Reshape 1D to 2D Array

- It is common to need to reshape a one-dimensional array into a two-dimensional array with one column and multiple arrays. NumPy provides the reshape() function on the NumPy array object that can be used to reshape the data. The reshape() function takes a single argument that specifies the new shape of the array. In the case of reshaping a one-dimensional array into a two-dimensional array with one column, the tuple would be the shape of the array as the first dimension (data.shape[0]) and 1 for the second dimension.
- data = data.reshape((data.shape[0], 1))

Example of changing the shape of a one-dimensional array with the reshape() function.

```
# reshape 1D array to 2D
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data.shape)
# reshape
data = data.reshape((data.shape[0], 1))
print(data.shape)
```

```
(5,)
(5, 1)
```

```
print(data)
```

```
[[11]
 [22]
 [33]
 [44]
 [55]]
```

Reshape 2D to 3D Array

- It is common to need to reshape two-dimensional data where each row represents a sequence into a three-dimensional array for algorithms that expect multiple samples of one or more time steps and one or more features.
- This is clear with an example where each sequence has multiple time steps with one observation (feature) at each time step. We can use the sizes in the shape attribute on the array to specify the number of samples (rows) and columns (time steps) and fix the number of features at 1.
- data.reshape((data.shape[0], data.shape[1], 1))

Example of changing the shape of a two-dimensional array with the reshape() function.

```
# reshape 2D array to 3D
from numpy import array
# list of data
data = [[11, 22], [33, 44], [55, 66]]
# array of data
data = array(data)
print(data.shape)
# reshape
data = data.reshape((data.shape[0], data.shape[1], 1))
print(data.shape)
```

```
(3, 2)
(3, 2, 1)
```

```
print(data)
```

```
[[[11]
  [22]]

 [[33]
  [44]]

 [[55]
  [66]]]
```

5.5 NumPy Array Broadcasting

1. Limitation with Array Arithmetic
2. Broadcasting in NumPy
3. Limitations of Broadcasting

1.Limitation with Array Arithmetic

- Strictly, arithmetic may only be performed on arrays that have the same dimensions and dimensions with the same size. This means that a one-dimensional array with the length of 10 can only perform arithmetic with another one-dimensional array with the length 10. This limitation on array arithmetic is quite limiting indeed. Thankfully, NumPy provides a built-in workaround to allow arithmetic between arrays with differing sizes.

```python
# Limitation with Array
a = [1, 2, 3]
print(a)
b = [1, 2, 3]
print(b)
c = a + b
print(c)
c = [a[i] + b[i] for i in range(len(a))]
print(c)
```

```
[1, 2, 3]
[1, 2, 3]
[1, 2, 3, 1, 2, 3]
[2, 4, 6]
```

2. Array Broadcasting in Numpy

- Broadcasting is the name given to the method that NumPy uses to allow array arithmetic between arrays with a different shape or size.
- Although the technique was developed for NumPy, it has also been adopted more broadly in other numerical computational libraries, such as Theano, TensorFlow, and Octave.
- We can make broadcasting concrete by looking at three examples in NumPy. The examples in this section are not exhaustive, but instead are common to the types of broadcasting you may see or implement.

Scalar and One-Dimensional Array

- A single value or scalar can be used in arithmetic with a one-dimensional array. For example, we can imagine a one-dimensional array a with three values [a1,a2,a3] added to a scalar b.
  - a = [a1, a2, a3]
  - b
- The scalar will need to be broadcast across the one-dimensional array by duplicating the value it 2 more times
  - b = [b1, b2, b3]
- The two one-dimensional arrays can then be added directly
  - c = a + b
  - c = [a1 + b1, a2 + b2, a3 + b3]

Example of broadcasting a scalar to a one-dimensional array in NumPy.

```
# broadcast scalar to one-dimensional array
from numpy import array
# define array
a = array([1, 2, 3])
print(a)
# define scalar
b = 2
print(b)
# broadcast
c = a + b
print(c)
```

```
[1 2 3]
2
[3 4 5]
```

Scalar and Two-Dimensional Array

A scalar value can be used in arithmetic with a two-dimensional array. For example, we can imagine a two-dimensional array A with 2 rows and 3 columns added to the scalar b.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \qquad \mathbf{1}$$

The scalar will need to be broadcast across each row of the two-dimensional array by duplicating it 5 more times.

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix} \qquad \mathbf{2}$$

The two two-dimensional arrays can then be added directly.

$$C = A + B \qquad \textbf{3}$$

$$C = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & a_{1,3} + b_{1,3} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & a_{2,3} + b_{2,3} \end{pmatrix} \qquad \textbf{4}$$

Example of broadcasting a scalar to a two-dimensional array in NumPy

```
from numpy import array
# define array
A = array([ [1, 2, 3], [1, 2, 3]])
print(A)
# define scalar
b = 2
print(b)
# broadcast
C = A + b
print(C)
```

```
[[1 2 3]
 [1 2 3]]
2
[[3 4 5]
 [3 4 5]]
```

One-Dimensional and Two-Dimensional Arrays

A one-dimensional array can be used in arithmetic with a two-dimensional array. For example, we can imagine a two-dimensional array A with 2 rows and 3 columns added to a one-dimensional array b with 3 values.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

$$b = \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix}$$

The one-dimensional array is broadcast across each row of the two-dimensional array by creating a second copy to result in a new two-dimensional array B.

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix}$$

The two two-dimensional arrays can then be added directly.

$$C = A + B$$

$$C = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & a_{1,3} + b_{1,3} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & a_{2,3} + b_{2,3} \end{pmatrix}$$

Example of broadcasting a one-dimensional array to a two-dimensional array in NumPy.

```
from numpy import array
# define two-dimensional array
A = array([ [1, 2, 3], [1, 2, 3]])
print(A)
# define one-dimensional array
b = array([1, 2, 3])
print(b)
# broadcast
C = A + b
print(C)
```

```
[[1 2 3]
 [1 2 3]]
[1 2 3]
[[2 4 6]
 [2 4 6]]
```

3. Limitations of Broadcasting

- Broadcasting is a handy shortcut that proves very useful in practice when working with NumPy arrays.
- That being said, it does not work for all cases, and in fact imposes a strict rule that must be satisfied for broadcasting to be performed.
- Arithmetic, including broadcasting, can only be performed when the shape of each dimension in the arrays are equal or one has the dimension size of 1.

Limitations of Broadcasting

```
A.shape = (2 x 3)
b.shape = (3)
```

In effect, this becomes a comparison between:

```
A.shape = (2 x 3)
b.shape = (1 x 3)
```

This same notion applies to the comparison between a scalar that is treated as an array with the required number of dimensions:

```
A.shape = (2 x 3)
b.shape = (1)
```

This becomes a comparison between:

```
A.shape = (2 x 3)
b.shape = (1 x 1)
```

When the comparison fails, the broadcast cannot be performed, and an error is raised.

Limitations of Broadcasting

When the comparison fails, the broadcast cannot be performed, and an error is raised.

The example below attempts to broadcast a two-element array to a 2×3 array. This comparison is in effect:

```
A.shape = (2 x 3)
b.shape = (1 x 2)
```

We can see that the last dimensions (columns) do not match and we would expect the broadcast to fail. The example below demonstrates this in NumPy.

Example output of a broadcast error

```
# broadcasting error
from numpy import array
# define two-dimensional array
A = array([ [1, 2, 3], [1, 2, 3]])
print(A.shape)
# define one-dimensional array
b = array([1, 2])
print(b.shape)
# attempt broadcast
C = A + b
print(C)
```

```
(2, 3)
(2,)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-c484ca9d5287> in <module>
      8 print(b.shape)
      9 # attempt broadcast
---> 10 C = A + b
```

Example

```
#NO broadcasting error
from numpy import array
# define two-dimensional array
A = array([ [1, 2, 3], [1, 2, 3]])
print(A.shape)
# define one-dimensional array
b = array([1,])
print(b.shape)
# attempt broadcast
C = A + b
print(C)
```

```
(2, 3)
(1,)
[[2 3 4]
 [2 3 4]]
```

End of Part1

Series Will be Continued