Chapter 4

Introduction to NumPy Arrays

Arrays are the main data structure used in machine learning. In Python, arrays from the NumPy library, called N-dimensional arrays or the `ndarray`, are used as the primary data structure for representing data. In this tutorial, you will discover the N-dimensional array in NumPy for representing numerical and manipulating data in Python. After completing this tutorial, you will know:

- What the `ndarray` is and how to create and inspect an array in Python.
- Key functions for creating new empty arrays and arrays with default values.
- How to combine existing arrays to create new arrays.

Let's get started.

4.1 Tutorial Overview

This tutorial is divided into 3 parts; they are:

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

4.2 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.

The data type supported by an array can be accessed via the `dtype` attribute on the array. The dimensions of an array can be accessed via the `shape` attribute that returns a tuple describing the length of each dimension. There are a host of other attributes. A simple way

to create an array from data or simple Python data structures like a list is to use the `array()` function. The example below creates a Python list of 3 floating point values, then creates an `ndarray` from the list and access the arrays' shape and data type.

```
# 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)
```

Listing 4.1: Example of creating an array with the `array()` function.

Running the example prints the contents of the `ndarray`, the shape, which is a one-dimensional array with 3 elements, and the data type, which is a 64-bit floating point.

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

Listing 4.2: Sample output of creating an array with the `array()` function.

4.3 Functions to Create Arrays

There are more convenience functions for creating fixed-sized arrays that you may encounter or be required to use. Let's look at just a few.

4.3.1 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.

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

Listing 4.3: Example of creating an array with the `empty()` function.

Running the example prints the content of the empty array. Your specific array contents will vary.

```
[[  0.00000000e+000  0.00000000e+000  2.20802703e-314]
 [  2.20803350e-314  2.20803353e-314  2.20803356e-314]
 [  2.20803359e-314  2.20803362e-314  2.20803366e-314]]
```

Listing 4.4: Sample output of creating an array with the `empty()` function.

4.3.2 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.

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

Listing 4.5: Example of creating an array with the `zeros()` function.

Running the example prints the contents of the created zero array.

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

Listing 4.6: Sample output of creating an array with the `zeros()` function.

4.3.3 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)
```

Listing 4.7: Example of creating an array with the `ones()` function.

Running the example prints the contents of the created ones array.

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

Listing 4.8: Sample output of creating an array with the `ones()` function.

4.4 Combining Arrays

NumPy provides many functions to create new arrays from existing arrays. Let's look at two of the most popular functions you may need or encounter.

4.4.1 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.

```
# 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)
```

Listing 4.9: Example of creating an array from other arrays using the `vstack()` function.

Running the example first prints the two separately defined one-dimensional arrays. The arrays are vertically stacked resulting in a new 2×3 array, the contents and shape of which are printed.

```
[1 2 3]

[4 5 6]

[[1 2 3]
 [4 5 6]]

(2, 3)
```

Listing 4.10: Sample output of creating an array from other arrays with the `vstack()` function.

4.4.2 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.

```
# 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)
```

Listing 4.11: Example of creating an array from other arrays using the `hstack()` function.

Running the example first prints the two separately defined one-dimensional arrays. The arrays are then horizontally stacked resulting in a new one-dimensional array with 6 elements, the contents and shape of which are printed.

```
[1 2 3]

[4 5 6]

[1 2 3 4 5 6]

(6,)
```

Listing 4.12: Sample output of creating an array from other arrays with the `hstack()` function.

4.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- Experiment with the different ways of creating arrays to your own sizes or with new data.
- Locate and develop an example for 3 additional NumPy functions for creating arrays.
- Locate and develop an example for 3 additional NumPy functions for combining arrays.

If you explore any of these extensions, I'd love to know.

4.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

4.6.1 Books

- *Python for Data Analysis*, 2017.
  http://amzn.to/2B1sfXi
- *Elegant SciPy*, 2017.
  http://amzn.to/2yujXnT
- *Guide to NumPy*, 2015.
  http://amzn.to/2j3kEzd

4.6.2 References

- NumPy Reference.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/
- The N-dimensional array.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html
- Array creation routines.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.array-creation.html

4.6.3 API

- `numpy.array()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.array.html
- `numpy.empty()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.empty.html
- `numpy.zeros()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.zeros.html
- `numpy.ones()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ones.html
- `numpy.vstack()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.vstack.html
- `numpy.hstack()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.hstack.html

4.7 Summary

In this tutorial, you discovered the N-dimensional array in NumPy for representing numerical and manipulating data in Python. Specifically, you learned:

- What the `ndarray` is and how to create and inspect an array in Python.
- Key functions for creating new empty arrays and arrays with default values.
- How to combine existing arrays to create new arrays.

4.7.1 Next

In the next chapter you will discover how to slice, index, and reshape NumPy arrays.

Chapter 5

Index, Slice and Reshape NumPy Arrays

Machine learning data is represented as arrays. In Python, data is almost universally represented as NumPy arrays. If you are new to Python, you may be confused by some of the Pythonic ways of accessing data, such as negative indexing and array slicing. In this tutorial, you will discover how to manipulate and access your data correctly in NumPy arrays. After completing this tutorial, you will know:

- How to convert your list data to NumPy arrays.
- How to access data using Pythonic indexing and slicing.
- How to resize your data to meet the expectations of some machine learning APIs.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into 4 parts; they are:

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

5.2 From List to Arrays

In general, I recommend loading your data from file using Pandas or even NumPy functions. This section assumes you have loaded or generated your data by other means and it is now represented using Python lists. Let's look at converting your data in lists to NumPy arrays.

5.2.1 One-Dimensional List to Array

You may load your data or generate your data and have access to it as a list. You can convert a one-dimensional list of data to an array by calling the `array()` NumPy function.

```
# 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))
```

Listing 5.1: Example of creating one-dimensional array.

Running the example converts the one-dimensional list to a NumPy array.

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

Listing 5.2: Sample output of creating a one-dimensional array.

5.2.2 Two-Dimensional List of Lists to Array

It is more likely in machine learning that you will have two-dimensional data. That is a table of data where each row represents a new observation and each column a new feature. Perhaps you generated the data or loaded it using custom code and now you have a list of lists. Each list represents a new observation. You can convert your list of lists to a NumPy array the same way as above, by calling the `array()` function.

```
# 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))
```

Listing 5.3: Example of creating two-dimensional array.

Running the example shows the data successfully converted.

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

Listing 5.4: Sample output of creating a two-dimensional array.

5.3 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.

5.3.1 One-Dimensional Indexing

Generally, indexing works just like you would expect from your experience with other programming languages, like Java, C#, and C++. For example, you can access elements using the bracket operator [] specifying the zero-offset index for the value to retrieve.

```
# 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])
```

Listing 5.5: Example of indexing a one-dimensional array.

Running the example prints the first and last values in the array.

```
11
55
```

Listing 5.6: Sample output from indexing a one-dimensional array.

Specifying integers too large for the bound of the array will cause an error.

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

Listing 5.7: Example of an error when indexing a one-dimensional array.

Running the example prints the following error:

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

Listing 5.8: Sample error output from indexing a one-dimensional array.

One key difference is that you can use negative indexes to retrieve values offset from the end of the array. For example, the index -1 refers to the last item in the array. The index -2 returns the second last item all the way back to -5 for the first item in the current example.

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

Listing 5.9: Example of negative indexing a one-dimensional array.

Running the example prints the last and first items in the array.

```
55
11
```

Listing 5.10: Sample output from negative indexing a one-dimensional array.

5.3.2 Two-Dimensional Indexing

Indexing two-dimensional data is similar to indexing one-dimensional data, except that a comma is used to separate the index for each dimension.

```
data[0,0]
```

Listing 5.11: Example of indexing a two-dimensional array in Python.

This is different from C-based languages where a separate bracket operator is used for each dimension.

```
data[0][0]
```

Listing 5.12: Example of indexing a two-dimensional array in C-like languages.

For example, we can access the first row and the first column as follows:

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

Listing 5.13: Example of indexing a two-dimensional array.

Running the example prints the first item in the dataset.

```
11
```

Listing 5.14: Sample output from indexing a two-dimensional array.

If we are interested in all items in the first row, we could leave the second dimension index empty, for example:

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

Listing 5.15: Example of indexing the first column of a two-dimensional array.

This prints the first row of data.

```
[11 22]
```

Listing 5.16: Sample output from indexing the first column of a two-dimensional array.

5.4 Array Slicing

So far, so good; creating and indexing arrays looks familiar. Now we come to array slicing, and this is one feature that causes problems for beginners to Python and NumPy arrays. 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]
```

Listing 5.17: Example of the syntax of array slicing.

5.4.1 One-Dimensional Slicing

You can access all data in an array dimension by specifying the slice ':' with no indexes.

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

Listing 5.18: Example of slicing a one-dimensional array.

Running the example prints all elements in the array.

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

Listing 5.19: Sample output from slicing a one-dimensional array.

The first item of the array can be sliced by specifying a slice that starts at index 0 and ends at index 1 (one item before the *to* index).

```
# 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])
```

Listing 5.20: Example of slicing a subset of a one-dimensional array.

Running the example returns a sub-array with the first element.

```
[11]
```

Listing 5.21: Sample output from slicing a subset of a one-dimensional array.

We can also use negative indexes in slices. For example, we can slice the last two items in the list by starting the slice at -2 (the second last item) and not specifying a *to* index; that takes the slice to the end of the dimension.

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

Listing 5.22: Example of slicing with a negative index on a one-dimensional array.

Running the example returns a sub-array with the last two items only.

```
[44 55]
```

Listing 5.23: Sample output from slicing with a negative index on a one-dimensional array.

5.4.2 Two-Dimensional Slicing

Let's look at the two examples of two-dimensional slicing you are most likely to use in machine learning.

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]
```

Listing 5.24: Example of slicing input variables.

For the output column, we can select all rows again using `:` and index just the last column by specifying the -1 index.

```
y = [:, -1]
```

Listing 5.25: Example of slicing the output variable.

Putting all of this together, we can separate a 3-column 2D dataset into input and output data as follows:

```
# 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)
```

Listing 5.26: Example of slicing a dataset into input and output variables.

Running the example prints the separated X and y elements. Note that `X` is a 2D array and `y` is a 1D array.

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

Listing 5.27: Sample output slicing a dataset into input and output variables.

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, :]
```

Listing 5.28: Example of slicing a train set from a dataset.

The test dataset would be all rows starting from the split point to the end of the dimension.

```
test = data[split:, :]
```

Listing 5.29: Example of slicing a test set from a dataset.

Putting all of this together, we can split the dataset at the contrived split point of 2.

```
# 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)
```

Listing 5.30: Example of slicing a dataset into train and test subsets.

Running the example selects the first two rows for training and the last row for the test set.

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

Listing 5.31: Sample output slicing a dataset into train and test subsets.

5.5 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.

5.5.1 Data Shape

NumPy arrays have a `shape` attribute that returns a tuple of the length of each dimension of the array. For example:

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

Listing 5.32: Example of accessing `shape` for a one-dimensional array.

Running the example prints a tuple for the one dimension.

```
(5,)
```

Listing 5.33: Sample output `shape` for a one-dimensional array.

A tuple with two lengths is returned 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)
```

Listing 5.34: Example of accessing `shape` for a two-dimensional array.

Running the example returns a tuple with the number of rows and columns.

```
(3, 2)
```

Listing 5.35: Sample output `shape` for a two-dimensional array.

You can use the size of your array dimensions in the `shape` dimension, such as specifying parameters. The elements of the tuple can be accessed just like an array, with the 0th index for the number of rows and the 1st index for the number of columns. For example:

```
# 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])
```

Listing 5.36: Example of accessing `shape` for a two-dimensional array in terms of rows and columns.

Running the example accesses the specific size of each dimension.

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

Listing 5.37: Sample output `shape` for a two-dimensional array in terms of rows and columns.

5.5.2 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))
```

Listing 5.38: Example the `reshape()` function for reshaping from 1D to 2D data.

Putting this all together, we get the following worked example.

```
# 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)
```

Listing 5.39: Example of changing the shape of a one-dimensional array with the `reshape()` function.

Running the example prints the shape of the one-dimensional array, reshapes the array to have 5 rows with 1 column, then prints this new shape.

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

Listing 5.40: Sample output of changing the shape of a one-dimensional array with the `reshape()` function.

5.5.3 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. A good example is the LSTM recurrent neural network model in the Keras deep learning library. The reshape function can be used directly, specifying the new dimensionality. 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))
```

Listing 5.41: Example the `reshape()` function for reshaping from 2D to 3D data.

Putting this all together, we get the following worked example.

```
# 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)
```

Listing 5.42: Example of changing the shape of a two-dimensional array with the `reshape()` function.

Running the example first prints the size of each dimension in the 2D array, reshapes the array, then summarizes the shape of the new 3D array.

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

Listing 5.43: Sample output of changing the shape of a two-dimensional array with the `reshape()` function.

5.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- Develop one example of indexing, slicing and reshaping your own small data arrays.
- Load a small real dataset from CSV file and split it into input and output elements
- Load a small real dataset from CSV file and split it into train and test elements.

If you explore any of these extensions, I'd love to know.

5.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.7.1 Books

- *Python for Data Analysis*, 2017.
  http://amzn.to/2B1sfXi
- *Elegant SciPy*, 2017.
  http://amzn.to/2yujXnT
- *Guide to NumPy*, 2015.
  http://amzn.to/2j3kEzd

5.7.2 References

- NumPy Reference.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/
- The N-dimensional array.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html
- Array creation routines.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.array-creation.html
- NumPy Indexing.
  https://docs.scipy.org/doc/numpy-1.13.0/user/basics.indexing.html
- SciPy Indexing.
  https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html
- Indexing routines.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.indexing.html

5.7.3 API

- `numpy.array()` API.
  https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.array.html
- `numpy.reshape()` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html

5.8 Summary

In this tutorial, you discovered how to access and reshape data in NumPy arrays with Python. Specifically, you learned:

- How to convert your list data to NumPy arrays.

- How to access data using Pythonic indexing and slicing.
- How to resize your data to meet the expectations of some machine learning APIs.

5.8.1 Next

In the next chapter you will discover array broadcasting and the rules that govern it in Python.

Chapter 6

NumPy Array Broadcasting

Arrays with different sizes cannot be added, subtracted, or generally be used in arithmetic. A way to overcome this is to duplicate the smaller array so that it has the dimensionality and size as the larger array. This is called array broadcasting and is available in NumPy when performing array arithmetic, which can greatly reduce and simplify your code. In this tutorial, you will discover the concept of array broadcasting and how to implement it in NumPy. After completing this tutorial, you will know:

- The problem of arithmetic with arrays with different sizes.
- The solution of broadcasting and common examples in one and two dimensions.
- The rule of array broadcasting and when broadcasting fails.

Let's get started.

6.1 Tutorial Overview

This tutorial is divided into 4 parts; they are:

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

6.2 Limitation with Array Arithmetic

You can perform arithmetic directly on NumPy arrays, such as addition and subtraction. For example, two arrays can be added together to create a new array where the values at each index are added together. For example, an array a can be defined as $[1, 2, 3]$ and array b can be defined as $[1, 2, 3]$ and adding together will result in a new array with the values $[2, 4, 6]$.

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a + b
c = [1 + 1, 2 + 2, 3 + 3]
```

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.

6.3 Array Broadcasting

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. Broadcasting solves the problem of arithmetic between arrays of differing shapes by in effect replicating the smaller array along the last mismatched dimension.

> Vectors are built from components, which are ordinary numbers. You can think of a vector as a list of numbers, and vector algebra as operations performed on the numbers in the list.
>
> — *Broadcasting*, SciPy.org.

NumPy does not actually duplicate the smaller array; instead, it makes memory and computationally efficient use of existing structures in memory that in effect achieve the same result. The concept has also permeated linear algebra notation to simplify the explanation of simple operations.

> In the context of deep learning, we also use some less conventional notation. We allow the addition of matrix and a vector, yielding another matrix: $C = A + b$, where $C_{i,j} = A_{i,j} + b_j$. In other words, the vector b is added to each row of the matrix. This shorthand eliminates the need to define a matrix with b copied into each row before doing the addition. This implicit copying of b to many locations is called broadcasting.
>
> — Page 34, *Deep Learning*, 2016.

6.4 Broadcasting in NumPy

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.

6.4.1 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]
```

The example below demonstrates this 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)
```

Listing 6.1: Example of broadcasting a scalar to a one-dimensional array in NumPy.

Running the example first prints the defined one-dimensional array, then the scalar, followed by the result where the scalar is added to each value in the array.

```
[1 2 3]

2

[3 4 5]
```

Listing 6.2: Results from broadcasting a scalar to a one-dimensional array in NumPy.

6.4.2 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.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \tag{6.1}$$

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

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix} \tag{6.2}$$

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

$$C = A + B \tag{6.3}$$

$$C = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & a_{1,3} + b_{1,3} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & a_{2,3} + b_{2,3} \end{pmatrix} \tag{6.4}$$

The example below demonstrates this in NumPy.

```
# broadcast scalar to two-dimensional array
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)
```

Listing 6.3: Example of broadcasting a scalar to a two-dimensional array in NumPy.

Running the example first prints the defined two-dimensional array, then the scalar, then the result of the addition with the value 2 added to each value in the array.

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

2

[[3 4 5]
 [3 4 5]]
```

Listing 6.4: Results from broadcasting a scalar to a two-dimensional array in NumPy.

6.4.3 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.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \tag{6.5}$$

$$b = \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix} \tag{6.6}$$

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.

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix} \tag{6.7}$$

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

$$C = A + B \tag{6.8}$$

$$C = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & a_{1,3} + b_{1,3} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & a_{2,3} + b_{2,3} \end{pmatrix} \tag{6.9}$$

The example below demonstrates this in NumPy.

```
# broadcast one-dimensional array to two-dimensional array
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)
```

Listing 6.5: Example of broadcasting a one-dimensional array to a two-dimensional array in NumPy.

Running the example first prints the defined two-dimensional array, then the defined one-dimensional array, followed by the result C where in effect each value in the two-dimensional array is doubled.

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

[1 2 3]

[[2 4 6]
 [2 4 6]]
```

Listing 6.6: Results from broadcasting a one-dimensional to a two-dimensional array in NumPy.

6.5 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. The dimensions are considered in reverse order, starting with the trailing dimension; for example, looking at columns before rows in a two-dimensional case.

This make more sense when we consider that NumPy will in effect pad missing dimensions with a size of 1 when comparing arrays. Therefore, the comparison between a two-dimensional array A with 2 rows and 3 columns and a vector b with 3 elements:

```
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.

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.

```
# 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)
```

Listing 6.7: Example of broadcasting error in NumPy.

Running the example first prints the shapes of the arrays then raises an error when attempting to broadcast, as we expected.

```
(2, 3)
(2,)
ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

Listing 6.8: Example output of a broadcast error..

6.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- Create three new and different examples of broadcasting with NumPy arrays.
- Implement your own broadcasting function for manually broadcasting in one and two-dimensional cases.
- Benchmark NumPy broadcasting and your own custom broadcasting functions with one and two dimensional cases with very large arrays.

If you explore any of these extensions, I'd love to know.

6.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

6.7.1 Books

- Chapter 2, *Deep Learning*, 2016.
  http://amzn.to/2EnS7x5

6.7.2 Articles

- Broadcasting, NumPy API, SciPy.org.
  https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html
- Broadcasting semantics in TensorFlow.
  https://www.tensorflow.org/performance/xla/broadcasting
- Array Broadcasting in NumPy, EricsBroadcastingDoc.
  http://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc
- Broadcasting, Theano.
  http://deeplearning.net/software/theano/tutorial/broadcasting.html
- Broadcasting arrays in NumPy, 2015.
  https://eli.thegreenplace.net/2015/broadcasting-arrays-in-numpy/
- Broadcasting in Octave.
  https://www.gnu.org/software/octave/doc/v4.2.1/Broadcasting.html

6.8 Summary

In this tutorial, you discovered the concept of array broadcasting and how to implement in NumPy. Specifically, you learned:

- The problem of arithmetic with arrays with different sizes.
- The solution of broadcasting and common examples in one and two dimensions.
- The rule of array broadcasting and when broadcasting fails.

6.8.1 Next

This is the end of this part, in the next part you will discover vectors and matrices, the main data structures used in linear algebra.