PRACTICAL LINEAR ALGEBRA FOR MACHINE LEARNING

By Amirsina Torfi

Machine Learning Mindset Handbook

Practical Linear Algebra for Machine Learning

Author:
Amirsina Torfi

December, 2019

Certain entities or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the Machine Learning Mindset, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

Machine Learning Mindset
www.machinelearningmindset.com

Preface

Machine Learning is everywhere these days and a lot of fellows desire to learn it and even master it! This burning desire creates a sense of impatience. We are looking for shortcuts and willing to ONLY jump to the main concept. If you do a simple search on the web, you see thousands of people asking "How can I learn Machine Learning?", "What is the fastest approach to learn Machine Learning?", and "What are the best resources to start Machine Learning?" *Well, there is a problem here*. Mastering a branch of science is NOT just a feel-good exercise. It has its own requirements.

One of the most critical requirements for Machine Learning is Linear Algebra. Basically, the majority of Machine Learning is working with data and optimization. How can you want to learn those without Linear Algebra? How would you process and represent data without vectors and matrices? On the other hand, Linear Algebra is a branch of mathematics after all. A lot of people trying to avoid mathematics or have the temptation to "just learn as necessary." I agree with the second approach, though. *However, the bad news is*: You cannot escape Linear Algebra if you want to learn Machine Learning and Deep Learning. There is NO shortcut.

The good news is there are numerous resources out there. In fact, the availability of numerous resources made me ponder whether writing this book was necessary? I have been blogging about Machine Learning for a while and after searching and searching I realized there is a deficiency of an organized book which **(1)** teaches the most used Linear Algebra concepts in Machine Learning, **(2)** provides practical notions using everyday used programming languages such as Python, and **(3)** be concise and NOT unnecessarily lengthy.

In this book, you get all of what you need to learn about Linear Algebra that you need to master Machine Learning and Deep Learning.

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter, the goal is to investigate the importance of Linear Algebra in Machine Learning and motivate you to continue further.

1.1. Is Mathematics Painful?

Let me start with one fact: The majority of us may dislike math if not hate it! We try to avoid it at all costs and when we feel like we can't avoid it, it's like being in agony as our lungs blister with chemical burns. Am I right? We may have many different reasons, consciously or subconsciously:

- While reading it, it always looks like that we have to learn new things which we do not know how useful they are, except for understanding what we are reading!
- Why should I spend too much time one something that I don't receive an immediate benefit? What if I can skip the math and still be able to do the job?
- Maybe due to the conformity bias, It looks cool to say, "I hate mathematics."
- Maybe you had a bad teacher! I am not talking about being good or bad in actually knowing math; I am talking about a teacher that always discouraged you by criticizing your manner or questioning your ability! It usually happened. Right?
- Any other reason that you have in mind and I missed them ...

All in all, the list is too long. Many times understanding the root of this problem may not solve the problem. Perhaps it's good to only think about our motivations rather than forcing ourselves to like it! I personally believe there is no reason to like something if we have to do it. On the other hand, it's actually a skill: Doing things we hate to do! So, let's forget about liking math for a moment and try to just motivate ourselves to learn it at the level that we need!

A question still remains: **Do we really need to learn math if we desire to learn Machine Learning?** The short answer is: **(1)** yes, you need to learn some branches of mathematics, such as Linear Algebra, **(2)** you need to learn it as needed. There is no need to be a master!

1.2. The Motivation

Why Linear Algebra is Necessary for Machine Learning? Linear Algebra is a branch of math that is broadly used everywhere. It is, in essence, the study of vectors and linear functions and we use it to interpret and utilize Machine Learning. Knowing Linear Algebra enlighten you about how algorithms work at their core level, which helps you to employ an algorithm better or even make a decision whether that choice (the utilized algorithm/approach) is practical or not. An example is the K-nearest neighbor algorithm which is known to be one of the simplest Machine Learning algorithms. Even for that, you still cannot avoid knowing a minimal level of Linear Algebra, if you would like to know how to implement it.

Vectors and matrices are everywhere, especially in Deep Learning. Loosely speaking, Deep Learning is technically Machine Learning using Neural Networks and Big Data. Neural Networks are made of matrices, and they mainly operate under the laws of Linear Algebra. Henceforth, a good knowledge of linear algebra seems to be necessary for understanding and utilizing Machine Learning and Deep Learning algorithms and architectures. Let's not ignore what we can easily see: ***Linear Algebra is a MUST KNOWN for Machine Learning***.

But wait! Is that enough to exclaim the importance of Linear Algebra? Of course not. I like to address some situations that you need to know Linear Algebra, SERIOUSLY:

- **Working with vectors:** Can you ignore vectors and their associated characteristics while working on Machine Learning? I think NOT. They are everywhere. The foundation of Machine Learning is based on vectors and matrices.

- **Delving deep in an algorithm:** You can't possibly hit the root and gain a core understanding of Machine Learning algorithms without knowing Linear Algebra. You may say, "who cares about core understanding?" Yeah, you can ignore it, but what if your implementation is not working? What if you want to improve a method? What if you want to know more than average?! Then you need it so bad!!

- **You want to teach what you know:** Yes, that is very important. Assume you want to DO Machine Learning, whatever that "DOING" means. Eventually, you want to share your knowledge by any mean. Explaining to other folks, your manager, your students, and the list goes on. Of course, if you work in a company, it is unlikely that any of your managers care about "how your algorithm works and what is inside it?!" BUT, they care about the results. Assume you want to convince them something is NOT working. Then you need to explain in detail. If your managers neither

> care about why your algorithm works nor you can convince them by going into the details of Linear Algebra, then you MUST SIMPLY QUIT and chase your prosperity elsewhere!

Those lines where only simple examples to showcase the importance of Linear Algebra in Machine Learning. It is hard (if not impossible) to touch base on each and every aspect of Linear Algebra that are important in Machine Learning, in just ONE post. I am sure you will, but again, I suggest you to google ***"The importance of Linear Algebra in Machine Learning"*** to explore more!

It looks impossible to ignore, right? In the beginning, it may be like torture to force ourselves to dig deep into it. BUT, if you are audacious, what you will gain is amazing knowledge which facilitates your climbing the ladder of Machine Learning.

1.3. How to Use This Book?

This book aims to teach you the core concepts in a practical way. The focus is on the Linear Algebra topics that are mostly used in Machine Learning and Deep Learning. To further assist you in understanding the materials, for each notion, a practical Python code is provided. Do NOT just run the code. Try to understand everything by going further and modifying the codes to see what you will understand.

Chapter 2

Basic Linear Algebra Definitions

2.1. Introduction

In the process of practicing Machine Learning [1] you keep hearing buzzwords related to Linear Algebra. Vector, matrix, tensor? Those are basic Linear Algebra Definitions. In this chapter, some of the most commonly used Linear Algebra definitions are introduced.

This chapter aims to accomplish the following:

- Provide the basic definitions in Linear Algebra that we hear every day while dealing with Machine Learning.
- Describe the connection between those basic concepts.
- Demonstration of the commonly used notations.

If you already know Linear Algebra and/or familiar with basic definitions, you may still find some aspects of this article useful, such as the representation of "the commonly used notations."

2.2. Scalar and Vector

By using mathematical definition, a scalar is *"an element of a field, which is used to define a vector space, usually the field of real numbers."* Simply speaking, a scalar is just a number. We usually denote a scalar with a lower case symbol, such as a.

It is worth to have a comparison between scalars and vectors. Most probably, you are familiar with measures such as the velocity of a moving car and the direction of moving, which have both quantity and direction. Such elements are called vector as opposed to scalars, which have magnitude only. We denote vectors with bold symbols such as $\mathbf{a}$.

Subscripts usually denote the elements of the vector. For example the second element of the vector $\mathbf{v}$ is denoted as v_2. In general, we show a k-dimensional vector as an array of elements:

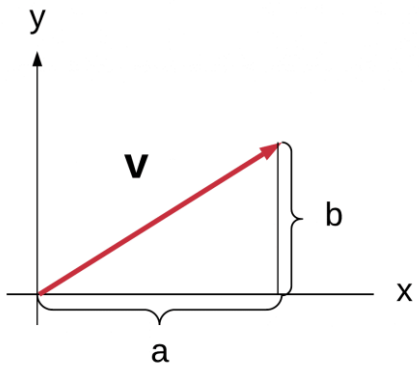

Fig. 2.1. The red arrow **v** is the vector and a and b are scalars denoting the magnitudes of **v** in horizontal and vertical directions.

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{k-1} \\ v_k \end{bmatrix}$$

NOTATION: We denote a k-dimensional vector as $\mathbf{v} \in \mathbb{R}^k$.

2.3. Matrix

In mathematics, a matrix is a rectangular array of elements, such as numbers organized in rows and columns. Usually, the matrices are denoted with bold uppercase, such as **M**. An example of a matrix is below:

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 3 \\ 4 & 11 & -0.2 \end{bmatrix}$$

We denote the matrix elements with subscripts. In the matrix **M**, the element in i^{th} row and j^{th} column is denoted as $\mathbf{M}_{i,j}$. So you can easily say for the example above, $\mathbf{M}_{1,2} = -1$ and $\mathbf{M}_{2,3} = -0.2$. Another important notation is when we desire to refer a row or a column as a whole, i.e., all elements in that particular row/column. In this case, we use ":" sign in the subscript. For example, assume we want to refer to all elements of i^{th} row for matrix **M**. We denote that row with $\mathbf{M}_{i,:}$. For matrix **A**, $\mathbf{A}_{1,:}$ is as below:

$$\mathbf{A}_{1,:} = \begin{bmatrix} 1 & -1 & 3 \end{bmatrix}$$

NOTE: We usually represent a vector using one column and multiple rows of elements. Henceforth, we can call a vector of size k as a $k \times 1$ matrix. In general, we can informally say vectors are special kind of matrices which are 1-dimensional.

NOTATION: We denote a matrix as $\mathbf{M} \in \mathbb{R}^{m \times n}$ in which m and n are the number of rows and columns, respectively.

2.4. Tensor

We usually represent data in machine learning, numerically. Tensors are used for such numerical representation. A tensor is simply a container that holds data in N-dimensional space. A Tensor can host and represent multiple matrices. An example is shown below:

$$\begin{bmatrix} \begin{bmatrix} 10 & 3 & 3.1 \\ 6 & 12 & -0.2 \end{bmatrix} & \begin{bmatrix} 1.0 & -1 & 3 \\ 4 & 11 & -2 \end{bmatrix} \\ \begin{bmatrix} 1 & -2 & 7 \\ 4. & 1.1 & 81.1 \end{bmatrix} & \begin{bmatrix} 1.1 & -2 & -2 \\ 4 & 4 & -5 \end{bmatrix} \end{bmatrix}$$

You can consider a simple 3D tensor as the data cube depicted below:

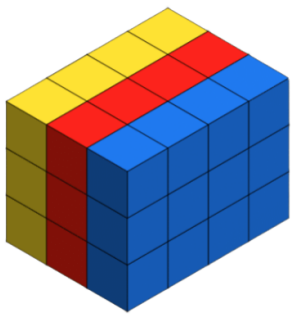

Fig. 2.2. In this tensor, each color represents a matrix and the concatenation of the matrices, form the tensor.

2.5. Conclusion

In this chapter, some of the most commonly used Linear Algebra definitions in Machine Learning are described. Scalar, vector, matrix, and tensor are what you hear the most. It is crucial to know them before proceeding to learn other concepts in Linear Algebra. For more details about the concepts in this chapter, you can refer to [2–6].

Chapter 3

An Introduction to NumPy

This chapter is dedicated to NumPy [7], one of the most important scientific computing libraries for Linear Algebra and henceforth, Machine Learning! As we will use it frequently to explain the concepts of Linear Algebra, it is important to know how to utilize it.

3.1. Introduction

In almost **any Python program code in Machine Learning** you see "NumPy" library being used! Why? Doing Machine Learning is impossible without Linear Algebra and Linear Algebra is formed by vectors, matrices, etc. Well, **NumPy is one of the best scientific computing packages, in particular, for Linear Algebra**. So that was the short answer to the "why" question earlier!

You can conduct a simple 30 minutes research to find out if I am right or wrong! Search through the web and look for Machine Learning projects and their available source codes. Check to see how many of them use Numpy. Check Python codes. See how many times you see the expression *"import numpy"* or *"import numpy as np"* on top of the code. Do it! It definitely motivates you unless you already are aware of the importance of Numpy.

Here are what you learn in this chapter:

- What is Numpy?
- How we can define arrays in Numpy?
- What are the basic operations?
- How to leverage Numpy for Machine Learning?
- What are the most used tips and tricks in using Numpy?

3.2. Data Types

Here, some of the most important Numpy numeric data types that you frequently encounter are described. Numpy supports numerous data types, perhaps even more than Python itself!

3.2.1 Basic Data Types

The most basic data types are as follows: integer ($ex: 1, 2, -10$), float ($ex: 1.1, 3.24, -7.00111$), complex ($ex: 1+2j, 2.1+3j, -2+1.4j$), and Boolean (ex: True, False). You can use Numpy to convert Python elements in many different ways such as changing an array type to another specific type. Let's start with the following examples:

```
# Import NumPy library
import NumPy as np

# Define a number
a = 10
print('Type, before converting: ', type(a))

# Change the type to float with NumPy
b = np.float64(a)
print('Type, after converting: ', type(b))

# Change the type to float with NumPy
c = float(a)
print('Type, after converting: ', type(c))
```

```
Type, before converting:  <class 'int'>
Type, after converting:  <class 'numpy.float64'>
Type, after converting:  <class 'float'>
```

Question

What is the difference between using "np.float64" and the Python "float" built-in function?

3.2.2 Type Conversion

Now, let's turn a list to a NumPy array. For now, let's just focus on the data types and conversion, later in this chapter, you will see how to define NumPy arrays in details.

```
# Import NumPy library
import numpy as np

# Define a Python list
a = [1, 2, 1.3]
print('Type "a": ', type(a))

# Turn Python list into a NumPy array of type np.int32 (Integer
    (-2147483648 to 2147483647))
```

```
b = np.int32(a)
print('Array "b": ', b)
print('Type array "b": ', type(b))
print('Array "b" data type: ', b.dtype)

# Turn Python list into a NumPy array of type np.float32 (same as
    Python float)
c = np.float32(a)
print('Array "c": ', c)
print('Type array "c": ', type(c))
print('Array "c" data type: ', c.dtype)
```

```
Type "a":  <class 'list'>
Array "b":  [1 2 1]
Type array "b":  <class 'numpy.ndarray'>
Array "b" data type:  int32
Array "c":  [1.  2.  1.3]
Type array "c":  <class 'numpy.ndarray'>
Array "c" data type:  float32
```

We used the **.dtype** NumPy method to realize what is the data type inside the array. The recommended way to change the type of a NumPy array is the usage of **.astype()** method. Take a look at the following example:

```
# Import NumPy library
import numpy as np

# Define a Python list
a = [1, 2, 4]
print('Type "a": ', type(a))

# Turn Python list into a NumPy array of type np.int32 (Integer
    (-2147483648 to 2147483647))
b = np.int32(a)
print('Array "b": ', b)
print('Type array "b": ', type(b))

# Turn Python list into a NumPy array of type np.float32 (same as
    Python float)
c = b.astype(np.float32)
print('Array "c": ', c)
print('Type array "c": ', type(c))
```

```
Type "a":  <class 'list'>
Array "b":  [1 2 4]
```

```
Type array "b":  <class 'numpy.ndarray'>
Array "c":  [1. 2. 4.]
Type array "c":  <class 'numpy.ndarray'>
```

3.3. Defining a NumPy Array

An array is basically a one- or multi-dimensional grid of values. In a NumPy array, in particular, all values are from the same type (integer, float). How we are going to define a NumPy array? For a NumPy array, we have the following definitions:

- **Rank:** The number of dimensions an array has.
- **Shape:** A tuple that indicates the number of elements in each dimension. Ex: The shape of an array being as (2,4,10) indicates that we have a three-dimensional array which has 2,4, and 10 elements in the first, second, and third dimension, respectively.

3.3.1 Create an array from a list

Now let's get started with Python. We start with the most common approach. Let's define create a NumPy array from a list:

```
# Import NumPy library
import numpy as np

# Define a Python list
mylist = [1, 2, 4, 8]

# Create a NumPy array from the list
numpy_array = np.array(mylist)
print('Array: ', numpy_array)
```

```
Array:  [1 2 4 8]
```

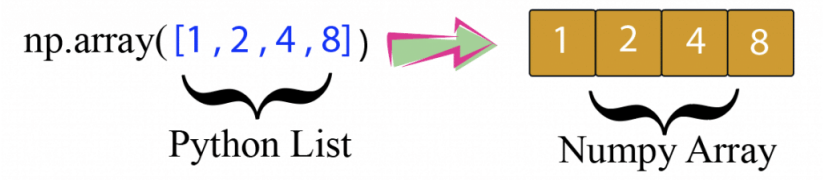

Fig. 3.1. Above, we used **np.array** function to transform a list into a NumPy array

What if we do not define a list and just input the numbers as below:

```
# Import NumPy library
import numpy as np

# Naively input the numbers
numpy_array = np.array(1,2,4,8)
print('Array: ', numpy_array)
```

Run the above code to see what you will get as the output.

Now, let's define a two-dimensional array:

```
# Import NumPy library
import numpy as np

# Naively input the numbers
row1 = [2,4,6,8]
row2 = [1,3,5,7]
numpy_array = np.array([row1,row2])
print('Array: ', numpy_array)

# Get the shape
print('Shape: ', numpy_array.shape)
```

```
Array:  [[2 4 6 8]
 [1 3 5 7]]
Shape:  (2, 4)
```

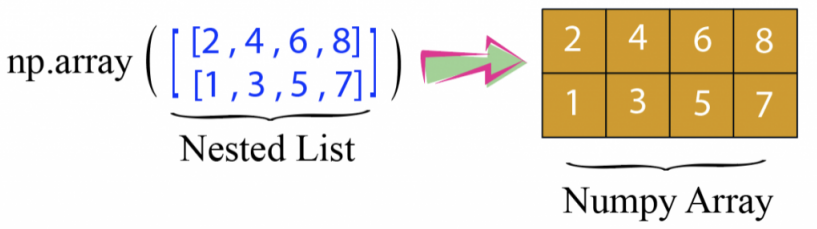

Fig. 3.2. Above, we transformed a list into a two-dimensional NumPy array

Let's take a look at the above code once again. We defined a matrix. The argument inside **np.array** is a list that each of its elements is another list (see figure above)! The inside lists denoted as row1 and row2 forms the rows of the matrix and MUST have the same size! Think why? That was a simple example to showcase how we can create arrays. The **.shape** method is used to return the NumPy array shape. The output above shows we have a matrix with two rows and four columns.

3.3.2 Special functions

We can create NumPy arrays using some special NumPy functions. I used a couple of them below for your reference:

```
# Import NumPy library
import numpy as np

### Defining arrays using special functions ###
# Arguments:
#   shape: The shape of the numpy array
#   dtype: Specifying the data type (not required)

# Defining an all-zero array
zeroArray = np.zeros(shape=(3,5), dtype=np.int16 )
print("zeroArray: ", zeroArray)

# Defining an all-one array
onesArray = np.ones(shape=(3,5), dtype=np.float32 )
print("onesArray: ", onesArray)

# Defining an array filled with one specific elements
fullArray = np.full(shape=(3,5), fill_value=4.2, dtype=np.float64 )
print("fullArray: ", fullArray)
```

```
zeroArray: [[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
onesArray: [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
fullArray: [[4.2 4.2 4.2 4.2 4.2]
 [4.2 4.2 4.2 4.2 4.2]
 [4.2 4.2 4.2 4.2 4.2]]
```

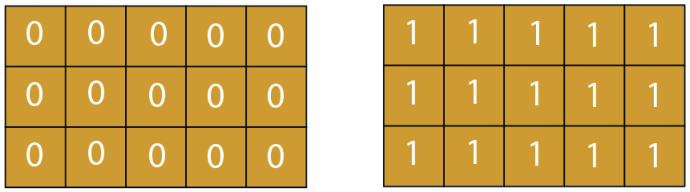

Fig. 3.3. Defining arrays filled with specific elements

One of the most important special functions is **np.arange** which is similar to Python range built-in function. An example of using **np.arrage** is as below:

```
# Import NumPy library
import numpy as np

# Define a NumPy array using np.arrage
# np.arrage defines an interval of numbers
# Arguments:
#   start: Starting of the interval.
#   stop: Ending of the interval.
#   step: Step size.
# NOTE: The interval includes "start" number but does NOT include "
   stop" number.
arr = np.arange(start=3,stop=10,step=2)
print("Array: ", arr)
```

```
Array:  [3 5 7 9]
```

Fig. 3.4. NumPy indexing starts from **0** to infinity! By defining np.arange(start=3,stop=10,step=2), start index is **3**. The stop index is **10** BUT it is NOT included in the range. Step size is **2** means every other index is picked, i.e., **we only pick green indexes and jump over orange ones**

3.3.3 Universal functions

NumPy, for performing element-wise operations, provides some universal functions. Take a look at some of the universal functions demonstrated below:

```
# Import NumPy library
import numpy as np

v = np.array([1, 3, 5])
w = np.array([2, 4, 6])

# Exponential opration
print('e^v= ', np.exp(v))

# The squere root of an array
print('v^{1/2}= ', np.sqrt(v))

# Adding two vectors
```

```
print('v+w= ', np.add(v, w))

# Sin and Cos of an array
print('Sin(v)= ', np.sin(v))
print('Cos(v)= ', np.cos(v))
```

Run the above code to see the result.

3.3.4 Random array

Sometimes, it is desired to define an array with random numbers. There are many ways to this task depends on the kind of random numbers we want to use. For example, do we want to generate random integers, float, etc? Take a look at the below approaches:

```
# Import NumPy library
import numpy as np

### Defining arrays with random elements ###
# Arguments:
#   size: The size of the numpy array

# Defining a random array
randArray = np.random.random(size=(2,3))
print("randArray: ", randArray)

# Defining a random array with integer elements
# high: The highest element that is allowed to be generated is ``
   high-1``
# low: The lowest integer that is allowed to be generated
randintArray = np.random.randint(low=1, high=5, size=(2,3))
print("randintArray: ", randintArray)
```

```
randArray:  [[0.65055144 0.12510875 0.09906024]
 [0.47333278 0.67082837 0.7673982 ]]
randintArray:  [[4 4 1]
 [1 2 2]]
```

CAVEAT

Since we generated a random vector, if you use the above Python code, you definitely will NOT get the results I reported above! It was obvious, though. Right?

3.4. Basic Arithmetic Operations

Let's first cover the basic arithmetic operations with some examples:

```
# Import NumPy library
import numpy as np

# Create two arrays
a = np.array([[1,5],[0,8]], dtype=np.float32)
b = np.array([[2,1],[5,2]], dtype=np.float32)
print('a= ', a)
print('b= ', b)

# Elementwise adding
# a + b and np.add(a, b) are the same.
# Check to see if a + b and np.add(a, b) are the same using ``assert
    ``.
# .all() method check if all elements of a matrix is True.
print('a + b= ', a + b)
assert((a + b == np.add(a, b)).all())

# Elementwise subtraction
print('a - b= ', a - b)
assert((a - b == np.subtract(a, b)).all())

# Elementwise multiplication
print('a * b= ', a * b)
assert((a * b == np.multiply(a, b)).all())

# Elementwise division
print('a / b= ',a / b)
assert((a / b == np.divide(a, b)).all())

# Elementwise square
print('a^2= ', np.square(a))
assert((a ** 2 == np.square(a)).all())
```

```
a=  [[1. 5.]
 [0. 8.]]
b=  [[2. 1.]
 [5. 2.]]
a + b=  [[ 3.  6.]
 [ 5. 10.]]
a - b=  [[-1.  4.]
 [-5.  6.]]
a * b=  [[ 2.  5.]
 [ 0. 16.]]
a / b=  [[0.5 5. ]
```

```
 [0.   4. ]]
a^2=  [[ 1. 25.]
 [ 0. 64.]]
 [1 2 2]]
```

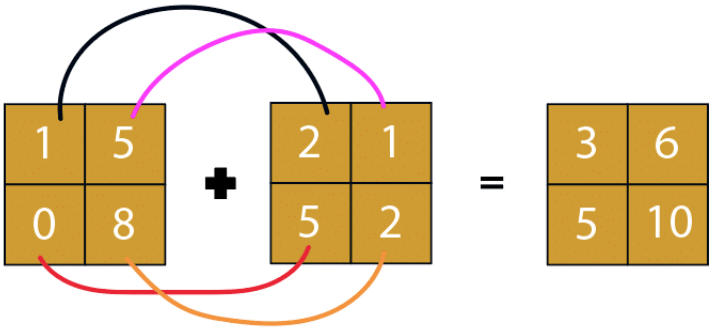

Fig. 3.5. Element-wise summation. In element-wise operations, the **operator performs on correspondent elements** from matrices

3.5. Array Manipulation

After becoming familiar with NumPy arrays, now it's time to learn how to play with arrays and learn some techniques.

3.5.1 Indexing

First, let's define and slice an array:

```
# Import NumPy library
import numpy as np

# Create a matrix
A = np.array([[2,1,3,4],[5,2,9,4],[5,2,10,1],[2,2,11,-1]])
print('A=\n', A)

# Extract the first two rows
# Remember 0:2 in indexing means {0,1} and does NOT include 2!
# Using : merely, means ALL!
print('The first two rows= \n', A[0:2,:])

# Extract the first three rows and the last two columns
# -2: means the second to the last to the end!
# 0:3 mean {0,1,2}
print('The first three rows and last two columns= \n', A[0:3,-2:])

```

```
18 # Let's point to one element
19 # The second row (index 1) and third column (index 2)
20 # Remember Python indexing starts from zero!!
21 print('The second row and third column= \n', A[1,2])
```

```
A=
 [[ 2  1  3  4]
 [ 5  2  9  4]
 [ 5  2 10  1]
 [ 2  2 11 -1]]
The first two rows=
 [[2 1 3 4]
 [5 2 9 4]]
The first three rows and last two columns=
 [[ 3  4]
 [ 9  4]
 [10  1]]
The second row and third column=
 9
```

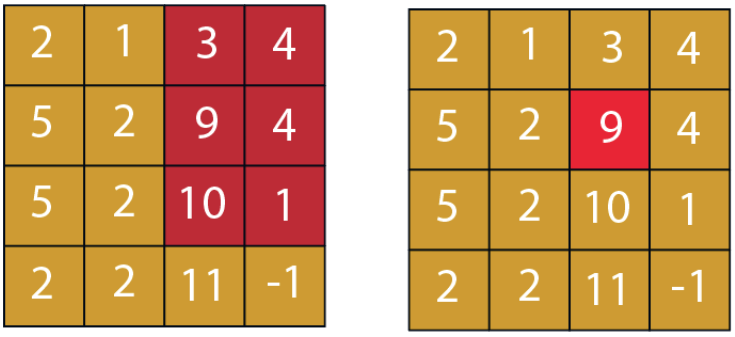

Fig. 3.6. Left figure: Shows the A[0:4,-2:] which is the first three rows and the last two columns. **Right figure:** Shows A[1,2] which is only one single element located in second row and third column.

Let's review the code above. At line **11**, I used sliced indexing by selecting from a range of indices. At line **21**, I only used integer indices. We can simply combine both, but there might be a difference in the output matrix ranking. Check the example below:

```
# Import NumPy library
import numpy as np

# Create a matrix
```

```
A = np.array([[2,1,3,4],[5,2,9,4],[5,2,10,1],[2,2,11,-1]])
print('A=\n', A)

# Extract the first row and all columns using two approaches
print('The first row with slice indexing= \n', A[0:1,:]) # Slice
    indexing
print('The first row with integer indexing= \n', A[0,:])  # Integer
    indexing
print('The first row shape slice indexing= \n', A[0:1,:].shape) #
    Slice indexing
print('The first row shape integer indexing= \n', A[0,:].shape)  #
    Integer indexing
```

```
A=
 [[ 2  1  3  4]
 [ 5  2  9  4]
 [ 5  2 10  1]
 [ 2  2 11 -1]]
The first row with slice indexing=
 [[2 1 3 4]]
The first row with integer indexing=
 [2 1 3 4]
The first row shape slice indexing=
 (1, 4)
The first row shape integer indexing=
 (4,)
```

If you see the results, the shape of the matrix would be different. Basically, **with slice indexing, we have a rank-2 matrix** and **with integer indexing, we will have a rank-1 matrix**. Be careful about this difference when you are dealing with NumPy indexing. We later talk about the matrix rank concept in details.

3.5.2 Shaping

```
# Import NumPy library
import numpy as np

# Create a matrix
A = np.array([[2,1,3,4],[5,2,9,4],[5,2,10,1]])
print('A=\n', A)
print('Shape of A=\n', A.shape)

# Reshape A to the new shape of (2,6)
B = A.reshape(2,6)
print("B: \n", B)

# Reshape A to the new shape of (2,x)
```

```
# If we use -1, the remaining dimension will be chosen automatically
    .
C = A.reshape(4,-1)
print("C: \n", C)

# Flatten operation
print("Flatten A: \n", A.ravel())
```

```
A=
 [[ 2  1  3  4]
 [ 5  2  9  4]
 [ 5  2 10  1]]
Shape of A=
 (3, 4)
B:
 [[ 2  1  3  4  5  2]
 [ 9  4  5  2 10  1]]
C:
 [[ 2  1  3]
 [ 4  5  2]
 [ 9  4  5]
 [ 2 10  1]]
Flatten A:
 [ 2  1  3  4  5  2  9  4  5  2 10  1]
```

The question is how reshaping operations work? Above we had the matrix $\mathbf{A}$ of size $(3,4)$ with 12 (3×4) total elements. When we use **np.reshape**, the default NumPy order is **"C-style"**, which is, the rightmost index "changes the fastest" for the processing operation. Let's use the above example of using **.ravel()** to flatten the matrix: The first element is obviously $\mathbf{A}_{0,0}$ and the next one is $\mathbf{A}_{0,1}$. The processing and creating the new array is as below when using **.ravel()**:

$$[\mathbf{A}_{0,0},\mathbf{A}_{0,1},\mathbf{A}_{0,2},\mathbf{A}_{0,3},\mathbf{A}_{1,0},\cdots,\mathbf{A}_{3,3},\mathbf{A}_{3,4}]$$

3.6. Conclusion

In this chapter, you learned how to use NumPy. You also realized how important it is for Machine Learning purposes. But clearly this introduction is NOT a panacea for all your NumPy needs nor it is flawless! You always need to explore more. You can learn more efficiently if you practice on your own. Use the above codes as a starter code and try to play around with them. If you would like to know more about NumPy, you can refer to [8–10].

Chapter 4

Matrix Operations

This chapter is dedicated to describe how we work with matrices and vectors. The discussion about vectors is straightforward and most of the content of this chapter is dedicated to matrix operations.

4.1. Introduction

Have you ever wondered why you need to know matrix operations? The answer is very simple: **To work with matrices**! Above all, the assumption is that you are familiar with the basic definitions (chapter 2). In this chapter, the most important matrix and vector operations that we need and frequently encounter in Machine Learning are are described.

4.2. Matrix Transpose

The transpose of a matrix is an operator which switches the row and column indices of the matrix. As a result, after transposing a matrix, we have a new matrix and is denoted as $\mathbf{A}^T$. Therefore, the i^{th} row, j^{th} column elements of $\mathbf{A}$ are the j^{th} row and i^{th} column elements of $\mathbf{A}^T$, respectively. Assume we show $\mathbf{A}^T$ with matrix $\mathbf{B}$, then we have the following:

$$\mathbf{B}_{j,i} = \mathbf{A}_{i,j}$$

We can have the following example to clarify better:

$$\begin{bmatrix} 2 & 3 & 1 \\ 3 & -4 & 2.2 \end{bmatrix}^T = \begin{bmatrix} 2 & 3 \\ 3 & -4 \\ 1 & -2.2 \end{bmatrix}$$

Calculation of a matrix transpose is very easy with Python. For instance, you can try the following code:

```
# Use Numpy package
import numpy as np
```

```
# Define a 3x2 matrix using np.array
A = np.array([[1, 2.2], [4, 7], [8, -2]])
"""A = [[ 1.   2.2]
       [ 4.   7. ]
       [ 8.  -2. ]] """

print("A is: {}".format(A))
print("The shape of A is: {}".format(A.shape))

# Use transpose() method
B = A.transpose()
"""B = [[ 1.   4.   8. ]
       [ 2.2  7.  -2. ]]"""

print("B is: {}".format(B))
print("The shape of B is: {}".format(B.shape))
```

```
A is: [[ 1.   2.2]
 [ 4.   7. ]
 [ 8.  -2. ]]
The shape of A is: (3, 2)
B is: [[ 1.   4.   8. ]
 [ 2.2  7.  -2. ]]
The shape of B is: (2, 3)
```

NOTE: It is worth noting that $(\mathbf{A}^T)^T = \mathbf{A}$.

4.3. Identity Matrix

Identity matrix, which is denoted as $\mathbf{I}_n$, is a square matrix (number of rows = number of columns = n) that all its elements along the main diagonal are 1's and the other elements are zero. For example, we can have the following 4×4 identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You can create the above matrix using the following code:

```
# Use Numpy package
import numpy as np
```

```
# Define an Identity matrix
# Ref: https://docs.scipy.org/doc/numpy/reference/generated/numpy.
    eye.html
A = np.eye(4)
```

4.4. Adding Operation

Adding two matrices is possible only if both matrices have the same shape. Assume $\mathbf{C} = \mathbf{A} + \mathbf{B}$. Let's get back to Python and define the same two matrices defined above. After that, we will add them together:

```
# Use Numpy package
import numpy as np

# Define a 3x2 matrix using np.array
A = np.array([[1, 2.2], [4, 7], [8, -2]])

# Use transpose() method
B = A.transpose()

# Create a matrix similar to A in shape but filled with random
    numbers
# Use *A.shape argument
A_like = np.random.randn(*A.shape)

# Add two matrices of the same shape
M = A + A_like
print("M equals to: ", M)

# Add two matrices with different shape
C = A + B
print("C equals to: ", C)
```

You should get the following output:

```
M equals to:  [[ 2.4212905   2.88158481]
 [ 4.34872344  5.01038501]
 [ 7.58194231 -2.0192284 ]]
---------------------------------------------------------------------------

ValueError                                Traceback (most recent
    call last)
<ipython-input-35-293b043ce9a9> in <module>()
     16
     17 # Add two matrices with different shape
---> 18 C = A + B
     19 print("C equals to: ", C)
```

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

What created the above error?

There are some important characteristics for adding matrices that you need to know:

- $\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$
- $(\mathbf{A}+\mathbf{B})+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$
- $\mathbf{A}+\mathbf{0}=\mathbf{0}+\mathbf{A}$
- $(\mathbf{A}+\mathbf{B})^T=\mathbf{A}^T+\mathbf{B}^T$

You can simply confirm the above properties with the following Python code:

```
# Use Numpy package
import numpy as np

# Define random 3x4 matrix using np.array
# Ref: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/
    numpy.random.randint.html
A = np.random.randint(10, size=(3, 4))
B = np.random.randint(10, size=(3, 4))
C = np.random.randint(10, size=(3, 4))

# np.all() test whether all array elements are True.
# More info: https://docs.scipy.org/doc/numpy/reference/generated/
    numpy.all.html
checkProperty = np.all(A + B == B + A)
if checkProperty: print('Property A + B == B + A is confirmed!')

checkProperty = np.all((A + B) + C == A + (B + C))
if checkProperty: print('Property (A + B) + C == A + (B + C) is
    confirmed!')

checkProperty = np.all(A + 0 == 0 + A)
if checkProperty: print('Property A + 0 == 0 + A is confirmed!')

checkProperty = np.all((A + B).transpose() == A.transpose() + B.
    transpose())
if checkProperty: print('Property (A + B)^T == A^T + B^T is
    confirmed!')
```

4.5. Scalar Multiplication

If we multiply a matrix by a number (a.k.a. scalar), the result equals to multipling every entry of the matrix by that specific scalar. For example, you can check the following calculations:

$$2 * \begin{bmatrix} 2 & 3 & 1 \\ 3 & -4 & 2.2 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 2 \\ 6 & -8 & 4.4 \end{bmatrix}$$

The properties of scalar and matrix multiplication are as below:

- $(\alpha+\beta)\mathbf{A} = \alpha\mathbf{A} + \beta\mathbf{A}$
- $\alpha(\mathbf{A}+\mathbf{B}) = \alpha\mathbf{A} + \alpha\mathbf{B}$
- $0.\mathbf{A} = 0$
- $1.\mathbf{A} = \mathbf{A}$

4.6. Matrix Multiplication

Assuming we like to multiply to matrices and calculate the output as $\mathbf{C} = \mathbf{AB}$. For doing so, the dimension of **A** and **B** matrices should match. **Here**, being matched does not mean being equal. **Matching means the number of columns** of **A** should equal the number of rows in **B**. An example is to assume the shape of **A** equal to $m \times q$. Then, the shape of **B** MUST be $q \times n$, so q is shared. As a result of such multiplication, the shape of **C** equal to $m \times n$.

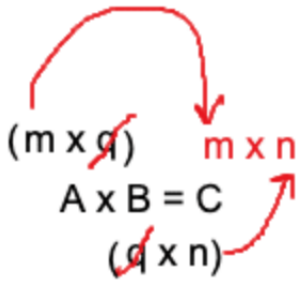

Fig. 4.1. Visualization of the dimention matching in matrix multiplication.

Check the following Python code to have a better understanding of matrix multiplication:

```
# Use Numpy package
import numpy as np

# Define two 3x4 random matrices using np.array
# Ref: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/
   numpy.random.randint.html
A = np.random.randint(10, size=(3, 4))
B = np.random.randint(10, size=(3, 4))

print('Shape A is: {}'.format(A.shape))
print('Shape B is: {}'.format(B.shape))

# Calculate the number of colums in A and number of rows in B
A_num_columns = A.shape[1]
B_num_rows = B.shape[0]

# Check the dimensions
if A_num_columns != B_num_rows: print('dimension mismatch')

# You should get an error as A_num_columns != B_num_rows
C = np.matmul(A , B)
```

What you will get as the output? Some of the above operations are valid and some of the are NOT...

Mathematically speaking, we can calculate different elements of the $\mathbf{C} = \mathbf{AB}$ as below:

$$C_{i,j} = \sum_m A_{i,m} B_{m,j} = A_{i,1} B_{1,j} + ... + A_{i,q} B_{q,j}$$

An illustrative example is shown below:

$$\begin{bmatrix} 2 & 3 & -1 \\ 7 & 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 2 & -1 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} 2\times 2+3\times 2-1\times 1 & \cdots \\ 7\times 2+0\times 2+2\times 1 & \cdots \end{bmatrix}$$

You can implement the above example with the following code and confirm the results:

```
# Use Numpy package
import numpy as np

# Define two matrices
A = np.array([[2,3,-1],[7,0,2]])
B = np.array([[2,1],[2,-1],[1,5]])

print('Shape A is: {}'.format(A.shape))
print('Shape B is: {}'.format(B.shape))

# Calculate the number of columns in A and number of rows in B
```

```
A_num_columns = A.shape[1]
B_num_rows = B.shape[0]

# Check the dimensions
if A_num_columns != B_num_rows: print('dimension mismatch')

# You should get an error as A_num_columns != B_num_rows
C = np.matmul(A , B)
print('C=AB= {}'.format(C))

# Instead of C=AB let's calculate C=BA
C = np.matmul(B , A)
print('C=BA= {}'.format(C))
```

The output will be:

```
Shape A is: (2, 3)
Shape B is: (3, 2)
C=AB= [[ 9 -6]
 [16 17]]
C=BA= [[11  6  0]
 [-3  6 -4]
 [37  3  9]]
```

You can clearly check that $\mathbf{AB} \neq \mathbf{BA}$. Properties of matrix multiplication are as below:

- $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$
- $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$
- $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$

You can check all the above properties with the following Python code:

```
# Use Numpy package
import numpy as np

# Define three random 3x3 matrix using np.array
# Ref: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/
   numpy.random.randint.html
A = np.random.randint(10, size=(3, 3))
B = np.random.randint(10, size=(3, 3))
C = np.random.randint(10, size=(3, 3))

# np.all() test whether all array elements are True.
# More info: https://docs.scipy.org/doc/numpy/reference/generated/
   numpy.all.html
checkProperty = np.all(np.matmul(A,np.matmul(B,C)) == np.matmul(np.
   matmul(A,B),C))
```

```
if checkProperty: print('Property A(BC) = (AB)C is confirmed!')

checkProperty = np.all(np.matmul(A,B+C) == np.matmul(A,B) + np.
    matmul(A,C))
if checkProperty: print('Property A(B+C) = AB + AC is confirmed!')

checkProperty = np.all(np.matmul(A,B).transpose() == np.matmul(B.
    transpose(),A.transpose()))
if checkProperty: print('Property (AB)^T = B^T.A^T is confirmed!')
```

It is important to address the matrix power. Assuming you see something like $\mathbf{A}^3$. This simply means we multiply the $\mathbf{A}$ matrix to its own for three times. It can be shown as below:

$$\mathbf{A}^3 = \mathbf{A} \times \mathbf{A} \times \mathbf{A}$$

BUT, the matrix **should be an square matrix**, e.g., the number of its rows and columns **MUST** be the same. Otherwise, the power operation would be invalid. Implement the following code to see what would happen:

```
# Calculate the matrix power for two square and non-square matrices.

# Use Numpy package
import numpy as np
# A: 3x3 SQUARE matrix using np.array
A = np.random.randint(10, size=(3, 3))
print('A has the shape of: {}'.format(A.shape))

# Calculate A^3
n=3
output = np.linalg.matrix_power(A, n)
print('Output has the shape of: {}'.format(output.shape))

# A: 3x4 NON-SQUARE matrix using np.array
B = np.random.randint(10, size=(3, 4))
print('B has the shape of: {}'.format(B.shape))

# Calculate A^3
n=3
output = np.linalg.matrix_power(B, n)
print('Output has the shape of: {}'.format(output.shape))
```

4.7. Matrix-Vector Multiplication

Let's take a look on how we can define matrices and vectors and multiplying them.

```
1 # Import Numpy library
```

```
2  import numpy as np
3
4  # Create two vectors and two matrices
5  v = np.array([0,8]).reshape(-1,1)
6  u = np.array([1,4]).reshape(-1,1)
7  A = np.array([[2,1],[5,2]])
8  B = np.array([[2,1],[5,2]])
9
10 # Dot porduct of two vectors with two approaches
11 print('v.u = ', v.dot(u.transpose()))
12 print('v.u = ', np.dot(v, u.transpose()))
13
14 # Porduct of a vector with a matrix
15 print('A.v = ', A.dot(v))
16 print('A.v = ', np.dot(A, v))
17
18 # Matrix product with three approaches
19 print('A.B = ', A.dot(B))
20 print('A.B = ', np.dot(A, B))
21 print('A.B = ', np.matmul(A,B))
```

Let's do a practice. Run the above code and answer the following questions:

Question

- What is the shape and rank of **v** and **u**?
- In lines 11 and 12, did we have to use "**.transpose()**"? Why?
- Instead of calculating '**v.u**' how would you calculate '**u.v**'?
- Take a look at lines 15 and 16. Instead of '**A.v**', can we calculate '**v.A**'?

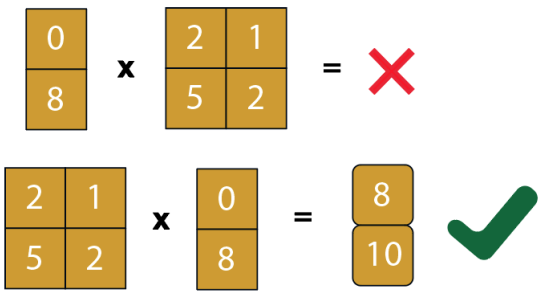

Fig. 4.2. You may see the visual answer to some of the questions above! Dimension matching is crucial in multiplying vectors and matrices.

NOTE: We usually represent a vector using one column and multiple rows of elements. Henceforth, we can call a vector of size k as a $k \times 1$ matrix. In general, we can informally say vectors are special kind of matrices which are 1-dimensional. $(\mathbf{A}^T)^T = \mathbf{A}$.

Similarly, to multiply the vector $\mathbf{v}$ to matrix $\mathbf{M}$, the dimensions must match. Assume our matrix $\mathbf{A}$ has the size of $m \times n$ and we like to calculate the $\mathbf{w} = \mathbf{A}\mathbf{v}$. What is the size of $\mathbf{v}$? Can you guess? The only acceptable size for $\mathbf{v}$ is $n \times 1$. So we have the following operation:

$$\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}^{m\times 1} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \cdots & \cdots & \vdots \\ A_{m1} & \cdots & \cdots & A_{mn} \end{bmatrix}^{m\times n} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}^{n\times 1}$$

You can implement a working example as below:

```
# Use Numpy package
import numpy as np
# A: 3x4 matrix using np.array
A = np.random.randint(10, size=(3, 4))

# B: A vector of size 4
v = np.random.randint(10, size=(4,))

print('Shape A is: {}'.format(A.shape))
print('Shape v is: {}'.format(v.shape))

# Calculate the number of colums in A and number of rows in B
A_num_columns = A.shape[1]
v_num_rows = v.shape[0]

# Check the dimensions
if A_num_columns != v_num_rows: print('dimension mismatch')

# You should get an error as A_num_columns != B_num_columns
w = np.matmul(A , v)
print('w=Av=', w)
print('The shape of w is:',w.shape)
```

4.8. Matrix Inverse

The inverse of matrix $\mathbf{A}$ exists and is shown with $\mathbf{A}^{-1}$ when both of the following conditions hold:

- **A** is a square matrix
- Multiplying **A** with its inverse results in identity matrix: $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$

If the matrix **A** is **invertible**, it is called non-singular. Otherwise, it is called **non-invertible** or **singular**. The concept of a matrix being singular is a relatively more advance concept and will be described in the following chapters.

You can practice the calculation of a matrix inverse using the following code:

```
# Use Numpy package
import numpy as np
# A: 3x3 SQUARE matrix using np.array
A = np.random.randint(10, size=(3, 3))
print('A has the shape of: {}'.format(A.shape))

# Caculate matrix inverse
ainv = np.linalg.inv(A)
print('The inverse of A is:',ainv)

# Check to see if multiplication of A and A^{-1} equals to identiy
    matrix I
if np.allclose(np.dot(A, ainv), np.eye(A.shape[0])): print('A is
    invertible!')
```

The following properties apply for matrix inversion operation:

- $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$
- $(\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T$
- $(\alpha\mathbf{A})^{-1} = (1/\alpha)\mathbf{A}^{-1}$

You can check the aforementioned properties using the following Python script:

```
# Use Numpy package
import numpy as np
# Define two 3x3 SQUARE matrix using np.array
A = np.random.randint(10, size=(3, 3))
B = np.random.randint(10, size=(3, 3))

# np.round(A,2): rounds the array A to two decimal points
# Check property (AB)^{-1} = B^{-1}.A^{-1}
left_side = np.linalg.inv(np.matmul(A,B))
right_side = np.matmul(np.linalg.inv(B),np.linalg.inv(A))
checkProperty = np.all(np.round(left_side, 2) == np.round(right_side
    , 2))
if checkProperty: print('Property (AB)^{-1} = B^{-1}.A^{-1} is
    confirmed!')
```

```
# Check property (A^{T})^{-1} = (A^{-1})^{T}
left_side = np.linalg.inv(np.transpose(A))
right_side = np.transpose(np.linalg.inv(A))
checkProperty = np.all(np.round(left_side, 2) == np.round(right_side
    , 2))
if checkProperty: print('Property (A^{T})^{-1} = (A^{-1})^{T} is
    confirmed!')

# np.dot is a very important function.
# Ref: https://docs.scipy.org/doc/numpy/reference/generated/numpy.
    dot.html
# Check property (alpha.A)^{-1} = (1/alpha)A^{-1}
alpha = float(3) # Any scalar (make sure to define a flaot)
left_side = np.linalg.inv(np.dot(alpha,A))
right_side = np.dot(1/alpha,np.linalg.inv(A))
checkProperty = np.all(np.round(left_side, 2) == np.round(right_side
    , 2))
if checkProperty: print('Property (alpha.A)^{-1} = (1/alpha)A^{-1}
    is confirmed!')
```

4.9. Matrix Trace

Assume we have a square matrix. Trace of a matrix is an operator that is the sum of its **diagonal elements** as below:

$$Tr(\mathbf{M}) = \sum_k \mathbf{M}_{k,k}$$

Working with the trace of a matrix, instead of the matrix itself, is easier for a lot of matrix calculations and mathematical proofs.

We have the following properties for the matrix trace operator:

- $Tr(\mathbf{M}) = Tr(\mathbf{M}^T)$
- $Tr(\mathbf{AB}) = Tr(\mathbf{BA})$ if both **AB** and **BA** exists and are valid.

4.10. Matrix Determinant

The **determinant** of the matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a scalar value that we compute from the elements of the matrix. The matrix **MUST** be square so we can compute the determinant. Furthermore, the matrix must be **non-singular**!

4.11. Special Matrices

There are special kinds of matrices that you may hear their names every day. I wanted to briefly describe some of them here.

Symmetric matrix

A symmetric matrix, equals its transpose as below:

$$\mathbf{M} = \mathbf{M}^T$$

An example would be the following matrix:

$$\begin{bmatrix} 1 & -1 \\ -1 & 3 \end{bmatrix}$$

Diagonal matrix

A diagonal matrix only have non-zero elements on its main diagonal. Everywhere else in the matrix, we have zero elements.

$$\mathbf{M}_{i,j} = 0, i \neq j$$

$$\mathbf{M}_{i,j} \neq 0, i = j$$

We can have the following example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

NOTE: A diagonal matrix, does not have to be a square matrix. The main diagonal of a matrix is formed with the elements $\mathbf{M}_{i,j}$ when $i = j$.

Orthogonal matrix

An Orthogonal matrix is a square matrix which has the following characteristic:

$$\mathbf{M}\mathbf{M}^T = \mathbf{M}^T\mathbf{M} = \mathbf{I}$$

Noted that if we multiply $\mathbf{M}$ with $\mathbf{M}^T$ and it results in the identity matrix, this implies that $\mathbf{M}^T = \mathbf{M}^{-1}$.

4.12. Conclusion

In this chapter, the important matrix operations that are commonly used in Machine Learning we explained. In addition, you learned how to code them in Python. In conclusion, you made a sense of the practical implementation of the operations in Python in addition to their theoretical interpretation. If you desire to dig more into matrices, you can refer to [4–6].

Chapter 5

Vector and Matrix Norms

In this chapter, the vector norms and matrix norms are explained. You will learn why they are essential, and what is their interpretation. Furthermore, you will learn how to implement them. We start with the definitions and ends it with the matrix norms. In fact, the majority of this chapter talks about the **vector norms** as they are more frequently used as opposed to **matrix norms**.

5.1. Introduction

In Machine Learning, we are dealing with training and evaluation all the time. It is needless to say it usually involves Linear Algebra, vectors, and matrices. In evaluating an element, such as loss functions, often, you need to **summarize all you need in one number**. Well, you use mean, standard deviation, etc. **BUT, how do you deal with vectors? How would you address the importance of a vector in terms of its elements? How would you evaluate the vector elements in terms of their contributions to the outcome?** We need some metrics to get a sense of the magnitude of a vector to see how much it affects the algorithm optimization, evaluation performance, etc. Here are the norms that come to play.

By the end of this chapter, you should know:

- What is the definition of the norm?
- The norms' properties
- Mostly used vector norms
- Definition of the matrix norm
- How to implement them in Python?

5.2. Vector Norm

We usually use the norms for vectors and rarely for matrices. At first, let's define what the norm of a vector is? A norm can be described as below:

- A function that operates on a vector (matrix) and returns a scalar element.
- A norm is denoted by $\mathscr{L}^q$ in which q shows the order of the norm and $p \geq 1, p \in \mathbb{R}$.
- The order a vector (matrix) is always a non-negative value.
- The intuition behind the norm is to measure a kind of distance.

Fig. 5.1. A norm usually indicates a distance or magnitude metric.

A norm is mathematically defined as below:

q-norm of a vector

$$\mathscr{L}^q(\mathbf{v}) = ||\mathbf{v}||_q = \left(\sum_k |v_k|^q\right)^{1/q}$$

The sign $|.|$ is an operation that outputs the absolute value of its argument. The example of which is $|-2| = 2$ and $|2| = 2$. You can implement $\mathscr{L}^q$ by the following Python code:

```
# Import Numpy package and the norm function
import numpy as np
```

```
from numpy.linalg import norm


# Define a vector
v = np.array([2,3,1,0])

# Take the q-norm which q=2
q = 2
v_norm = norm(v, ord=q)

# Print values
print('The vector: ', v)
print('The vector norm: ', v_norm)
```

```
The vector:  [2 3 1 0]
The vector norm:  3.7416573867739413
```

5.2.1 The Norm Function Properties

It is crucial to know the norms properties as we may need them in mathematical computation, especially mathematical proofs. A norm function $\mathscr{L}(.)$ has the following properties:

1. If the norm is zero, then the vector is all zero as well: $\mathscr{L}(\mathbf{v}) = 0 \Rightarrow \mathbf{v} = 0$
2. $\mathscr{L}(\mathbf{v}+\mathbf{w}) \leq \mathscr{L}(\mathbf{v}) + \mathscr{L}(\mathbf{w})$. This is called triangle inequality.
3. $\forall \beta \in \mathbb{R} : \mathscr{L}(\beta \mathbf{v}) = |\beta| \mathscr{L}(\mathbf{v})$

5.2.2 Proving the Properties (Advanced)

Properties **(1)** is easy to prove. As the norm add the absolute values of vector elements to any power, if the norm is zero, the only possible answer is that **all vector elements must be zero**.
We can prove property **(3)** as below:

$$\mathscr{L}^q(\beta \mathbf{v}) = \left(\sum_k |\beta v_k|^q \right)^{1/q} = \left(|\beta|^q \sum_k |v_k|^q \right)^{1/q} =$$

$$|\beta| \left(\sum_k |v_k|^q \right)^{1/q} = |\beta| \mathscr{L}^q(\mathbf{v})$$

We can mathematically prove property **(2)** as well. BUT, let's do it in an **empirical way**. Let's define our experiment as below:

1. We randomly generate two vectors **v** and **w**
2. We check the property that should be true.
3. Repeat the experiment for E=100 times.

This is *NOT scientific proof*. However, if $E = \infty$, then we can say we proved it. Right? Let's do it in **Python**:

```
# Import Numpy package and the norm function
import numpy as np
from numpy.linalg import norm

# Repeat experiments
E = 100  # Numper of experiments
q = 2    # Order of the norm
for i in range(E):
  # Define two random vector of size (1,5). Obviously v does not
    equal w!!
  v = np.random.rand(1,5)
  w = np.random.rand(1,5)

  propertyCheck = norm(v+w, ord=q) <= norm(v, ord=q) + norm(w, ord=q
    )
  if propertyCheck == False:
    print('Property is NOT correct')
```

So if the property (2) holds for all experiments, we expect the above Python code returns **NOTHING**. Think why?

5.3. Most Used Norms

In the previous section, I described what is the norm in general, and we implement it in Python. Here, I would like to discuss the norms that are mostly used in Machine Learning.

5.3.1 $\mathscr{L}^1$ norm

The $\mathscr{L}^1$ norm is technically the summation over the absolute values of a vector. The simple mathematical formulation is as below:

$$||\mathbf{v}||_1 = \sum_k |v_k|$$

In Machine Learning, we usually use $\mathscr{L}^1$ norm when the **sparsity of a vector matters**, i.e., when the essential factor is the non-zero elements of a matrix. **BUT why?** The $\mathscr{L}^1$ simply target the non-zero elements by adding them up.

5.3.2 $\mathscr{L}^2$ norm

$\mathscr{L}^2$ norm is also called the Euclidean norm which is the Euclidean distance of a vector to zero.
We can simply calculate it as below:

$$||\mathbf{v}||_2 = \sqrt{\sum_k |v_k|^2}$$

The $\mathscr{L}^2$ is commonly used in Machine Learning due to being differentiable, which is crucial for optimization purposes. Let's calculate $\mathscr{L}^2$ norm of a random vector with Python using two approaches. Both should lead to the same results:

```
# Import Numpy package and the norm function
import numpy as np
from numpy.linalg import norm

# Defining a random vector
v = np.random.rand(1,5)

# Calculate L-2 norm
sum_square = 0
for i in range(v.shape[1]):
  # Define two random vector of size (1,5). Obviously v does not
    equal w!!
  sum_square += np.square(v[0,i])
L2_norm_approach_1 = np.sqrt(sum_square)

# Calculate L-2 norm using numpy
L2_norm_approach_2 = norm(v, ord=2)

print('L2_norm: ', L2_norm_approach_1)
print('L2_norm with numpy:', L2_norm_approach_2)
```

You should get the same results!

5.3.3 Max norm

Well, you may not see this norm quite often. However, it is a kind of definition that you should be familiar with. The max norm is denoted with $\mathscr{L}^{\infty}$ and the mathematical formulation is as below:

$$||\mathbf{v}||_{\infty} = \max_{k} |v_k|$$

It simply returns the **maximum absolute value** in the vector elements.

5.4. Matrix Norm

For calculating the norm of a matrix, we have the unusual definition of **Frobenius norm** which is very similar to $\mathscr{L}^2$ norm of a vector and is as below:

$$||\mathbf{M}||_F = \sqrt{\sum_{i.j} |M_{i,j}|^2}$$

5.5. Conclusion

In this chapter, you learned what the norms are and how to implement them. You learned about the norm function properties and the most frequent norm functions. In Machine Learning, vector norms are used everywhere and they are one of the critical aspects of the optimization process.

Chapter 6

Linear Independence

This chapter in dedicated to the concept of linear independence of vectors and its associations with the solution of linear equation systems.

6.1. Introduction

What is the concept of vectors' linear independence? One simple example: Let's say you want to find the answer to the $\mathbf{Ax} = \mathbf{b}$. One may say we can simply find the answer by taking an inverse of $\mathbf{A}$ as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. NOT SO FAST! There is more in that? What if $\mathbf{A}$ does not have an inverse or is not square? You just let it go? Guess not.

That was just one example. In Machine Learning, it frequently happens that you want to **explore the correlation between vectors** to analyze them better. For **example**, you are dealing with a matrix which in essence, is formed by vectors. What would you know if you are not familiar with the concept of **linear independence**?

In this chapter, you will learn the following:

- The concept of linear independence/dependence!
- Examples of the linear dependence of vectors
- How to do it in Python using Numpy?
- Matrix ranks
- The solutions to a system of linear equations

6.2. The Concept

Assuming we have the set of $\{\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^p\} \in \mathbb{R}^q$ which are p column vectors of size q. Then, we call this set linear independent, if no vector exists that we can represent it as the linear combination of any other two vectors. Although, perhaps it is easier to define linear dependence.

Linear Dependency

A vector $\mathbf{v}^i$ is linear dependent if we can express it as the linear combination of another two vectors in the set, as below:

$$\mathbf{v}^i = \sum_{j \neq i} \alpha_j \mathbf{v}^j$$

In the above case, we say the set of $\{\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^p\}$ vectors are linearly dependent!

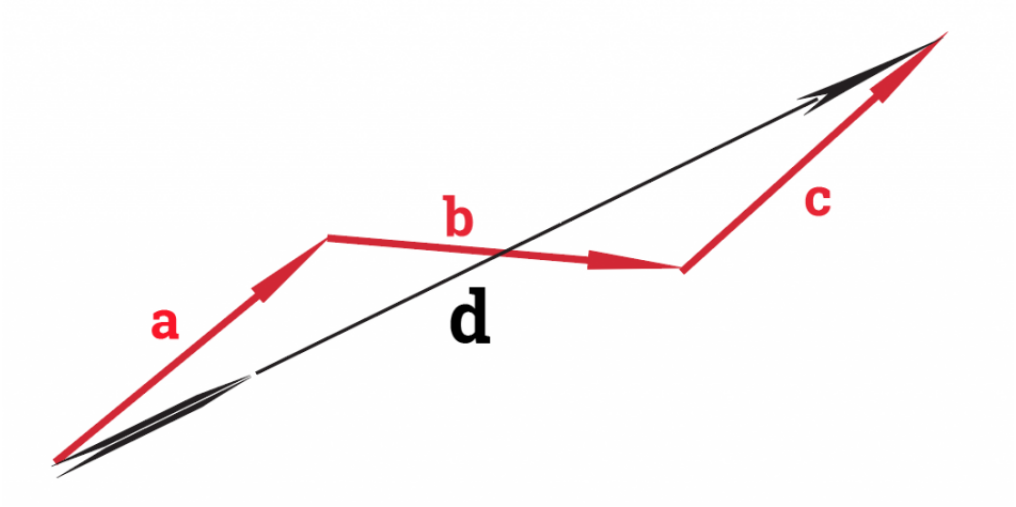

Fig. 6.1. Vector **d** is a linear combination of vectors **a**, **b**, and **c**. Actually, **d** = **a** + **b** + **c**.

6.3. Example

Consider the three vectors below:

$$\mathbf{v} = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}, \mathbf{u} = \begin{bmatrix} 0 \\ 3 \\ 1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 2 \\ 1 \\ 5 \end{bmatrix}$$

The above set is **linearly dependent. Why?** It is simple. Because $\mathbf{w} = 2\mathbf{v} + \mathbf{u}$. Let's do the above with Python and Numpy:

```
# Import Numpy library
import numpy as np

# Define three column vectors
v = np.array([1, -1, 2]).reshape(-1,1)
```

```
u = np.array([0, 3, 1]).reshape(-1,1)
w = np.array([2, 1, 5]).reshape(-1,1)

# Check the linear dependency with writing the equality
print('Does the equality w = 2v+u holds? Answer:', np.all(w == 2*v+u
    ))
```

Run the above code and see if Numpy confirms that or not!

6.4. The Relationship With Matrix Rank

The linear dependence of vectors has been discussed so far. Assume we have the matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$. The matrix has m rows and n columns. Let's focus on the columns. The n columns form n vectors. The j^{th} column is denoted with $\mathbf{M}_{:,j}$. So we have the set of n vectors $\{\mathbf{M}_{:,1}, \mathbf{M}_{:,2}, \ldots, \mathbf{M}_{:,n}\}$ as columns. The size of the largest subset of $\{\mathbf{M}_{:,1}, \mathbf{M}_{:,2}, \ldots, \mathbf{M}_{:,n}\}$ that its vectors are linearly independent, is called the **column rank** of the matrix $\mathbf{M}$. Considering the rows of $\mathbf{M}$, the size of the largest subset of rows that form a linearly independent set is called the **row rank** of the matrix $\mathbf{M}$.

We have the following properties for matrix ranks:

- For $\mathbf{M} \in \mathbb{R}^{m \times n}$, $rank(\mathbf{M}) \leq min(m,n)$. If $rank(M) = min(m,n)$, the matrix $\mathbf{M}$ is full rank.
- For $\mathbf{M} \in \mathbb{R}^{m \times n}$, $\mathbf{N} \in \mathbb{R}^{n \times q}$, and $\mathbf{H} = \mathbf{MN} \in \mathbb{R}^{m \times q}$, $rank(H) \leq min(rank(M), rank(N))$.
- For $\mathbf{M} \in \mathbb{R}^{m \times n}$, $\mathbf{N} \in \mathbb{R}^{m \times n}$, and $\mathbf{H} = \mathbf{M} + \mathbf{N} \in \mathbb{R}^{m \times n}$, $rank(H) \leq rank(M) + rank(N)$.

Check first and second property above with the following code:

```
# Import Numpy library
import numpy as np
from numpy.linalg import matrix_rank

# Define random 3x4 matrix using np.array
# Ref: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/
    numpy.random.randint.html
M = np.random.randint(10, size=(3, 4))
N = np.random.randint(10, size=(4, 3))

# np.all() test whether all array elements are True.
# More info: https://docs.scipy.org/doc/numpy/reference/generated/
    numpy.all.html
```

```
checkProperty = np.all(matrix_rank(M) <= min(M.shape[0],M.shape[1]))
if checkProperty: print('Property rank(M) <= min(M.shape[0],M.shape
    [1]) is confirmed!')

checkProperty = np.all(matrix_rank(np.matmul(M,N)) <= min(
    matrix_rank(M),matrix_rank(N)))
if checkProperty: print('Property rank(MN) <= min(rank(M),rank(N))
    is confirmed!')
```

Practice

Practice: Modify the above code and check property **(3)**.

6.5. Linear Equations

I talked about **linear dependency** and **matrix ranks**. After that, I would like to discuss their *application in finding the solution of linear equations*, which is of great importance. Consider the following equality which set a system of linear equations:

$$\mathbf{A}^{m\times n}\mathbf{x}^{n\times 1} = \mathbf{b}^{m\times 1}$$

Above, we see the matrix $\mathbf{A}$ is multiplied by the vector $\mathbf{x}$ and forms another vector $\mathbf{b}$. The above equality creates a set of **m-line** linear equations. Let's write line j for example:

$$\mathbf{A}_{j,1}\mathbf{x}_1 + \mathbf{A}_{j,2}\mathbf{x}_2 + \ldots + \mathbf{A}_{j,n}\mathbf{x}_n = \mathbf{b}_j, j \in \{1,2,\ldots,m\}$$

So the question is how many solution exists for the system of equations $\mathbf{Ax} = \mathbf{b}$. By solution I mean the possible values for the variables $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$. The answer is one of the following:

- There is **NO solution**.
- The system has **one unique solution**.
- We have **infinite** numbers of solutions.

NOTE: As you observed, having more than one BUT less than infinity solutions is NOT possible!

Theorem

If $\mathbf{u}$ and $\mathbf{v}$ are two solutions of the $\mathbf{Ax} = \mathbf{b}$ equation, then the specific linear combination of them as below, is a solution as well:

$$\mathbf{w} = \beta\mathbf{u} + (1-\beta)\mathbf{v}, \forall\beta \in \mathbb{R}$$

PROOF: Look at the below equations to see how $\mathbf{w}$ is also a solution:

$$\mathbf{Aw} = \mathbf{A}\{\beta\mathbf{v} + (1-\beta)\mathbf{u}\} = \beta\mathbf{Av} + \mathbf{Au} - \beta\mathbf{Au} = \beta\mathbf{b} + \mathbf{b} - \beta\mathbf{b} = b \ \blacktriangle$$

So the above proves shows that if we have more than one solution, then we can say we have infinite number of solutions! The **following two conditions** determine the number of solutions if there is at least one solution. Try to prove them based on what you learned so far:

- $\mathbf{A}^{m\times n}\mathbf{x}^{n\times 1} = \mathbf{b}^{m\times 1}$ has at least one solution if $m \leq n$ and Rank($\mathbf{A}$) = m.
- $\mathbf{A}^{m\times n}\mathbf{x}^{n\times 1} = \mathbf{b}^{m\times 1}$ has exactly one solution if $m = n$ and Rank($\mathbf{A}$) = m.

6.6. Conclusion

In this chapter, you learned the concept of linear independence of the vectors and their associates with the system of linear equations. This concept is crucial, especially in Machine Learning and optimization theory, in which we are dealing with all sorts of mathematical proofs necessary to justify why a method should work!

Chapter 7

Matrix Decomposition

This chapter is dedicated to explaining the concept of **matrix decomposition or factorization**, its definition, and the process. You will learn how you can decompose a matrix to its constituent elements.

7.1. Introduction

What is the ***benefit of decomposing a matrix***? What does that mean? When we *decompose anything*, we *break it into its constituent elements*. Assume we are going to disintegrate a **tool** (a car or a watch!). Such action helps us to understand the core particles and their tasks. Furthermore, it helps to have a better understanding of how that specific tool works and its characteristics! *Assume that the tool is a matrix which we would like to decompose*. There are different approaches to decompose a matrix. However, perhaps the most commonly used ones are matrix eigendecomposition and Singular Value Decomposition (SVD).

7.2. Matrix Eigendecomposition

In this section, I am going to show you the definition of eigendecomposition and the subsequent concepts necessary to understand it.

7.2.1 Eigenvector and Eigenvalue

Before we move on, we should know the definition of eigenvector and eigenvalue. The definition of eigenvector and eigenvalue are somehow connected.

Definition

Assuming we have the square matrix of $\mathbf{A} \in \mathbb{R}^{N \times N}$. The nonzero vector $\mathbf{v} \in \mathbb{R}^{N \times 1}$ is an eigenvector and scalar λ is its associated eigenvalue if we have:

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$$

From the above definition, it is clear than if $\mathbf{v}$ is an eigenvector, any vector $\alpha\mathbf{v}, \alpha \in \mathbb{R}$ is also an eigenvector with the same eigenvalue λ. Therefore, if we have one eigenvector, then we have infinite ones!

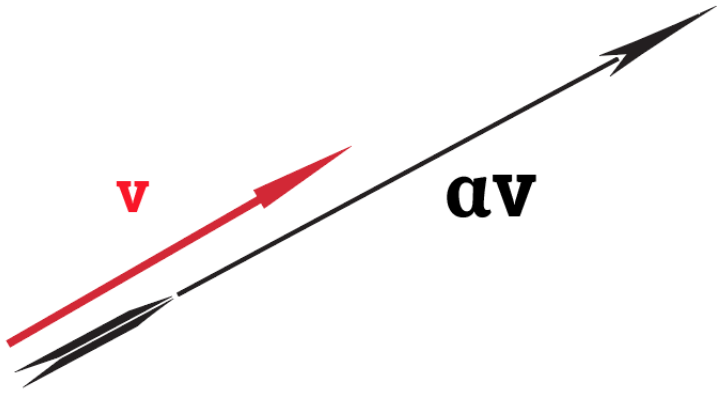

Fig. 7.1. Since $\mathbf{v}$ is an eigenvector, any vector $\alpha\mathbf{v}$ is also an eigenvector

Due to that, it is customary to only work with eigenvectors that have **unit norm**. It is simple to construct an eigenvector with the unit norm. Assume $\mathbf{v}$ is our eigenvector. Then, the following vector is also an eigenvector with the unit norm:

$$\mathbf{w} = \alpha\mathbf{v} = \frac{1}{||\mathbf{v}||}\mathbf{v}$$

where $||\mathbf{v}||$ is the norm of vector $\mathbf{v}$. We usually consider the euclidean norm.

7.2.2 The Process

Here, the process of how we decompose a matrix to its constituent elements is explained. It is called the **eigendecomposition of a matrix**.

Matrix Eigendecomposition

Assume we have the square matrix of $\mathbf{A} \in \mathbb{R}^{N \times N}$ which has N linear independent eigenvectors $\mathbf{v}^i, i \in 1, \ldots, N$. Then, we can factorize matrix $\mathbf{A}$ as below:

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$$

where $\mathbf{V} \in \mathbb{R}^{N \times N}$ is the square matrix whose j^{th} column is the eigenvector $\mathbf{v}^j$ of $\mathbf{A}$, and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{jj} = \lambda_j$.

Above, we basically concatenate eigenvectors $\mathbf{v}^i$ to form the $\mathbf{V}$ matrix as below:

$$\begin{bmatrix} | & | & \dots & | \\ | & | & \dots & | \\ \mathbf{v}^1 & \mathbf{v}^2 & \dots & \mathbf{v}^N \\ | & | & \dots & | \\ | & | & \dots & | \end{bmatrix}$$

7.2.3 Discussion on Matrix Eigendecomposition

Note that *we can only factorize **diagonalizable matrices** as above*. But the question is what is a diagonalizable matrix?

Diagnolizable Matrix

Assume we have the square matrix of $\mathbf{A} \in \mathbb{R}^{N \times N}$. It is called diagonalizable or nondefective if there exists an invertible matrix $\mathbf{H}$ such that $\mathbf{HAH}^{-1}$ is a diagonal matrix.

let's have a more precise definition of a matrix being singular or non-singular.

Singular Matrix

Assume we have the square matrix of $\mathbf{A} \in \mathbb{R}^{N \times N}$. It is called singular if and only if any of the eigenvalues (λ) are zero.

Under some circumstances, we can calculate the **matrix inverse** using the decomposition.

Matrix Inverse

Assume we have the square matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, it can be eigendecomposed and it is nonsingular. Therefore, we can calculate its inverse as below:

$$\mathbf{A}^{-1} = \mathbf{V}\blacksquare^{-1}\mathbf{V}^{-1}$$

Since Λ is diagonal, it inverse Λ^{-1} is also diagnoal and we can calculate it as:

$$\Lambda_{jj}^{-1} = \frac{1}{\lambda_j}$$

7.2.4 One Special Matrix Type and its Decomposition

We are interested to investigate *a special kind of matrix*: **Real symmetric matrix**. A real symmetric matrix is basically a symmetric matrix in which all elements belong to the space of real numbers $\mathbb{R}$.

For the **real symmetric matrix** of $\mathbf{A} \in \mathbb{R}^{N \times N}$, the eigenvalues are real numbers and we can choose eigenvectors in a way that they are orthogonal to each other. Then, we can factorize matrix $\mathbf{A}$ as below:

$$\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^T$$

where $\mathbf{Q}$ is an orthogonal matrix whose columns are the eigenvectors of $\mathbf{A}$, and Λ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{jj} = \lambda_j$.

7.2.5 Useful Properties

So far, the concepts and how we can decompose a matrix were explained. Let's see how we can leverage it. Assuming $\mathbf{A} \in \mathbb{R}^{N \times N}$:

- The determinant of the matrix $\mathbf{A}$ equals the product of its eigenvalues.
- The trace of the matrix $\mathbf{A}$ equals the summation of its eigenvalues.
- If the eigenvalues of $\mathbf{A}$ are λ_i, and $\mathbf{A}$ is non-singular, then the eigenvalues of $\mathbf{A}^{-1}$ are simply $\frac{1}{\lambda_i}$.
- The eigenvectors of $\mathbf{A}^{-1}$ are the same as the eigenvectors of $\mathbf{A}$.

7.2.6 Using Python

Now, let's do some practical work. I want to use Python and NumPy to compute eigenvalues and eigenvectors.

```
1  # Import NumPy library
2  import numpy as np
3
4  # Define random 4x4 matrix using np.array
5  # Ref: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/
       numpy.random.randint.html
6  N=4
7  A = np.random.randint(10, size=(N, N))
8  print('A:\n',A)
9
10 # Eigendecomposition
11 eigenvalues,eigenvectors= np.linalg.eig(A)
12
13 # Show values
14 print('Eigenvalues:', eigenvalues)
15 print('Eigenvectors:', eigenvectors)
16
17 # Create the diagonal matrix of \Lambda
18 Lambda = np.diag(eigenvalues)
19
20 # Create V, the matrix of eigenvectors
21 V = eigenvectors
22
23 # Check and Confirm the decomposition
24 A_ = np.matmul(np.matmul(V,Lambda),np.linalg.inv(V))
25 print('Computing A with decomposed elements:\n', A_)
```

Run the above code to see the results. Pretty simple. Right? In the above code, **line 24** aims to confirm if by using the decomposed elements $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$ we can reconstruct $\mathbf{A}$. What is your observation?

7.3. Singular Value Decomposition (SVD)

We previously discussed matrix eigendecomposition as an approach to decompose a matrix using its eigenvalues and eigendecomposition. **There was one issue though:** For matrix eigendecomposition, *the matrix **MUST** be square*. The Singular Value Decomposition (SVD) does NOT have this limitation, and it makes it even more useful and powerful compared to eigendecomposition.

7.3.1 Preliminary Definitions

Let's have a preliminary definition before we move on.

Conjugate Transpose of a Matrix

The conjugate transpose or Hermitian transpose of $\mathbf{M} \in \mathbb{C}^{m \times n}$ is obtained taking the transpose and then taking the complex conjugate of each entry of matrix $\mathbf{M}$. The resulting matrix is denoted by $\mathbf{M}^*$ or $\mathbf{M}^H$.
The set $\mathbb{C}$ is the field of the complex numbers.

Now, let's define a special kind of matrices.

Unitary Matrix

A complex square matrix $\mathbf{M} \in \mathbb{C}^{n \times n}$ is unitary if its conjugate transpose $\mathbf{M}^*$ is also its inverse. It can be shown as below:

$$\mathbf{M}\mathbf{M}^* = \mathbf{M}^*\mathbf{M} = \mathbf{I}$$

where $\mathbb{C}$ is the field of the complex numbers.

NOTE: In the above definition, if $\mathbf{M}$ belongs to the field of real numbers $\mathbb{R}$, the matrix $\mathbf{M}$ will be orthogonal and $\mathbf{M}^* = \mathbf{M}^\mathbf{T}$.

7.3.2 Matrix decomposition with SVD

Now, let's define the main concept, Singular Value Decomposition (SVD).

Singular Value Decomposition

Assume we have the matrix of $\mathbf{A} \in \mathbb{C}^{M \times N}$. Then, we can factorize matrix $\mathbf{A}$ as below:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^*$$

where $\mathbf{U}$ is an $M \times M$ and $\mathbf{V}$ is an $N \times N$ matrix and both are unitary. The matrix Σ is a diagonal $M \times N$ matrix with non-negative real numbers on the diagonal.

NOTE: The matrix Σ is a diagonal matrix of size $M \times N$. So it does not have to be square!

The **diagonal elements** of Σ are known as the **singular values** of the $\mathbf{A}$. The **columns** of $\mathbf{U}$ are known as the **left-singular** vectors and are the eigenvectors of $\mathbf{A}\mathbf{A}^*$. The **columns** of $\mathbf{V}$ are known as the **right-singular** vectors and are the eigenvectors of $\mathbf{A}^*\mathbf{A}$.

The diagonal matrix Σ is **uniquely determined** by $\mathbf{A}$. The nonzero singular values of $\mathbf{A}$ are the square roots of the eigenvalues $\mathbf{A}^*\mathbf{A}$. That's why they are non-negative!

7.3.3 A Practical Implementation

Let's compute the singular value decomposition of a matrix using Python and NumPy.

```
1  # Import Numpy library
2  import numpy as np
3
4  # Define random 3x4 matrix using np.array
5  # Ref: https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/
       numpy.random.randint.html
6  M=3
7  N=4
8  A = np.random.randint(10, size=(M, N))
9  print('A:\n',A)
10
11 # Singular Value Decomposition
12 # With full_matrices=True, U and Vt matrices will be squared.
13 U, S, Vt = np.linalg.svd(A, full_matrices=True)
14
15 # Show the shape of outputs
16 print('The shape of U:', U.shape)
17 print('The shape of S:', S.shape)
18 print('The shape of V^{T}:', Vt.shape)
19
20 # Create the diagonal matrix of \Lambda
21 Sigma = np.diag(S)
22
23 # Matrix A is fat! Sigma is not diagonal!
24 if Sigma.shape[1] != Vt.shape[0]:
25     zero_columns_count = Vt.shape[0] - Sigma.shape[1]
26     additive = np.zeros((Sigma.shape[0],zero_columns_count), dtype=np
       .float32)
27     Sigma = np.concatenate((Sigma,additive), axis=1)
28
29 # Matrix A is fat! Sigma is not diagonal!
30 if Sigma.shape[0] != U.shape[1]:
31     zero_rows_count = U.shape[1] - Sigma.shape[0]
32     additive = np.zeros((zero_rows_count,Sigma.shape[1]), dtype=np.
       float32)
33     Sigma = np.concatenate((Sigma,additive), axis=0)
34
35 # Check and Confirm the decomposition
36 A_ = np.matmul(np.matmul(U,Sigma),Vt)
37 print('Computing A with decomposed elements:\n', A_)
```

In the above code, the matrices' shapes is just get printed. Try to change the code above to see the outputs. Starting from **line 20**, the code reconstruct matrix **A** using the decomposed vector by SVD to confirm the computation!

7.3.4 Applications of SVD

Moore–Penrose Pseudoinverse

Before, digging into how SVD can help us with pseudoinverse calculations, let's talked about what is a Pseudoinverse?

Moore–Penrose Pseudoinverse

Assuming we have the matrix of $\mathbf{A} \in \mathbb{C}^{M \times N}$. Then, we have a pseudoinverse of $\mathbf{A}$, defined as $\mathbf{A}^+ \in \mathbb{C}^{M \times N}$, that satisfies all of the following criteria:

- $\mathbf{A}\mathbf{A}^+\mathbf{A} = \mathbf{A}$
- $\mathbf{A}^+\mathbf{A}\mathbf{A}^+ = \mathbf{A}^+$
- $(\mathbf{A}\mathbf{A}^+)^* = \mathbf{A}\mathbf{A}^+$
- $(\mathbf{A}^+\mathbf{A})^* = \mathbf{A}^+\mathbf{A}$

How we can *calculate the **Pseudoinverse** of a matrix using SVD*? If we have the SVD of a matrix $\mathbf{A}$ as $\mathbf{U}\Sigma\mathbf{V}^*$, then $\mathbf{A}^+$ can be calculated as below:

$$\mathbf{A}^+ = \mathbf{V}\Sigma^+\mathbf{U}^*$$

where Σ^+ is the pseudoinverse of Σ and we create it by replacing each non-zero diagonal entry σ in Σ by $\frac{1}{\sigma}$ and transposing the resulting matrix.

Matrix Rank Determination

Previously, the linear dependence and the matrix ranks were discussed. Now, as we have SVD, a more general approach compared to eigendecomposition, we can conclude the following directions.

Matrix Rank: Assume we have the SVD of a matrix $\mathbf{A}$ as $\mathbf{U}\Sigma\mathbf{V}^*$. The rank of $\mathbf{A}$ equals the number of non-zero singular values which is the number of non-zero diagonal elements in Σ.

7.4. Conclusion

In this chapter, the matrix decomposition (factorization) concept and two of its most used techniques were introduced. You learn what is matrix decomposition and how to do it. You also practiced to implement it in Python and NumPy. If you would like to know more about matrix decomposition, you can refer to [11, 12].

References

[1] Torfi A (2019) What is Machine Learning? An Introduction. Available at https://www.machinelearningmindset.com/what-is-machine-learning/.

[2] Schwartz M (1960) *Vector analysis: with applications to geometry and physics* (Harper), .

[3] Borisenko AI, et al. (1968) *Vector and tensor analysis with applications* (Courier Corporation), .

[4] Watkins DS (2004) *Fundamentals of matrix computations*. Vol. 64 (John Wiley & Sons), .

[5] Meyer CD (2000) *Matrix analysis and applied linear algebra*. Vol. 71 (Siam), .

[6] Lancaster P, Tismenetsky M (1985) *The theory of matrices: with applications* (Elsevier), .

[7] Oliphant TE (2006) *A guide to NumPy*. Vol. 1 (Trelgol Publishing USA), .

[8] VanderPlas J (2016) *Python data science handbook: essential tools for working with data* (" O'Reilly Media, Inc."), .

[9] Nunez-Iglesias J, Van Der Walt S, Walt S, Dashnow H (2017) *Elegant SciPy: The Art of Scientific Python* (" O'Reilly Media, Inc."), .

[10] Bressert E (2012) *SciPy and NumPy: an overview for developers* (" O'Reilly Media, Inc."), .

[11] Axler SJ (1997) *Linear algebra done right*. Vol. 2 (Springer), .

[12] Dym H (2013) *Linear algebra in action*. Vol. 78 (American Mathematical Soc.), .

Appendix

Appendix A: The Notations

a	A scalar
$\mathbf{v}$	vector
$\mathbf{A}$	matrix
$\mathbb{A}$	A set which can be the set of real number $\mathbb{R}$ for example
$\{1,...,n\}$	The set of all integers in the range of **1** to **n**
v_i	Element i of vector **v**
v_{-i}	All vector **v** elements except i
$A_{i,j}$	The element of row i and column j from the matrix **A**
A^T	The transpose of matrix **A**
A^H	The conjugate transpose of matrix **A**
$\|\|v\|\|_p$	The $\mathscr{L}^p$ norm of vector **v**

Appendix B: Python Environment

As you could see throughout the book, Python is used. Python is a programming language and needs an environment for being run. Instead of going through the numerous ways to setup a Python environment and installing the required packages, I would like to suggest using Google Colab. It provides you a nice environment with all the necessary packages installed. It also gives you FREE GPUs access! You will find it extremely valuable. In other words, invaluable!

Alphabetical Index