Lecture 2

Learning outcomes:

- *Numerical Calculations with NumPy:*
- *Solving linear equations-Introduction to matrix operations*

Linear Algebra

- Linear algebra is an important branch of mathematics that deals with the study of linear equations and their representations in terms of matrices and vectors.

- NumPy is a popular Python library that provides a wide range of functions for performing linear algebra operations.

Solving Systems of Linear Equations Using NumPy

- Systems of linear equations are fundamental in various fields of science and engineering. NumPy, a powerful numerical computing library in Python, provides efficient methods to solve these systems.
- Example:
- Consider the system of equations:

$$\begin{cases} 2x + 3y = 8 \\ x - 4y = -2 \end{cases}$$

This can be written in matrix form as $Ax = B$:

$$\begin{bmatrix} 2 & 3 \\ 1 & -4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ -2 \end{bmatrix}$$

- Using NumPy, we can solve this system as follows:

```
import numpy as np

A = np.array([[2, 3], [1, -4]])
B = np.array([8, -2])

# Solving the system
solution = np.linalg.solve(A, B)
print("Solution:", solution)
```

✓**Output**:

Solution: [2.36363636 1.09090909]

Matrices

- A special subtype of a two-dimensional NumPy array is a *matrix*. A matrix is like an array except that matrix multiplication (and exponentiation) replaces element-by-element multiplication. Matrices are generated by the *matrix* function, which may also be abbreviated *mat*:

- Employing matrices enables us to execute linear algebra, which allows us to execute numerous operations on many numbers effectively using matrix operations in Python NumPy.

Construct a Matrix in NumPy

❑ np.array() function:

- To Construct a Matrix in NumPy, we may use the np.array() function.

Syntax:

numpy.array(object, dtype=None, **kwargs)

- In this function, we supply the Python object to be converted into a matrix (in our example, a list of lists) as well as the object's data type. It returns the NumPy array on which we will conduct matrix operations.
- To generate the aforementioned array in NumPy, use the np.array() method as shown below.

```
# Using np.array() function
import numpy as np
Array = np.array( [ [ 1, 2, 3 ],
                    [ 4, 5, 6 ],
                    [ 7, 8, 9 ] ] )
print(Array)
```

✓**Output**:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

❑ np.matrix() Function:

- To Construct a Matrix in NumPy, We may also use the np.matrix() Function

Syntax:

numpy.matrix(data, dtype=None, **kwargs)

- To transform our data into a NumPy array, we merely supply data and its data type to this method.
- Let us now use this approach to produce the aforementioned array for conducting matrix operations in python numpy.

```
# Using np.matrix() function
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                    [ 4, 5, 6 ],
                    [ 7, 8, 9 ] ] )
print(Array)
```

✓**Output**:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

- ✓Despite the fact that it was effective in producing matrices, it is no longer suggested to use this class, including for linear algebra.

- ✓Instead, utilize approach 1, which makes use of the np.array() function. It is quite likely that it will be removed from future NumPy versions.

- ✓Thus we can observe how the mathematical representation of a matrix is expressed in NumPy in both cases. Each row is surrounded by square brackets, and the entire array is surrounded by a set of brackets.

Operations on Matrices in NumPy

- For performing matrix operations in Python NumPy, there are several operations including:

- Addition of matrices
- Subtraction of matrices
- Multiply or divide a matrix by a scalar
- Finding the maximum value in the matrix
- Finding the minimum value in the matrix
- Sum of all values of a Numpy matrix
- Transpose a matrix
- Rank of a NumPy matrix

- Determinant of a square NumPy matrix.
- Inverse a matrix
- Convert a matrix to a list
- Vector norm
- Matrix Trace
- LU Decomposition
- QR Decomposition
- SVD (Singular Value Decomposition)
- Direct Solution of Linear Equations
- Least Squares Fitting
- Condition Number

Addition of Matrices

- The '+' operator is one of the most implemented matrix operations in Python NumPy can be used to execute addition on matrices. Consider the following example to better understand this.

```
# Addition of Matrices
import numpy as np
Array1 = np.matrix( [ [ 1, 2, 3 ],
                      [ 4, 5, 6 ],
                      [ 7, 8, 9 ] ] )
Array2 = np.matrix( [ [ 10, 11, 12 ],
                      [ 13, 14, 15 ],
                      [ 16, 17, 18 ] ] )
print("Output : \n",Array1+ Array2)
```

✓**Output:**

```
[[11 13 15]
[17 19 21]
[23 25 27]]
```

Subtraction of Matrices

- Subtraction is analogous to addition. We just need to switch from the '+' operator to the '-' operator. Consider the following example to better understand this.

```
# Subtraction of matrices
import numpy as np
Array1 = np.matrix( [ [ 1, 2, 3 ],
                      [ 4, 5, 6 ],
                      [ 7, 8, 9 ] ] )
Array2 = np.matrix( [ [ 10, 11, 12 ],
                      [ 13, 14, 15 ],
                      [ 16, 17, 18 ] ] )
print("Output : \n",Array2 - Array1)
```

✓**Output :**

```
[[9 9 9]
[9 9 9]
[9 9 9]]
```

NumPy matrix multiplication methods#

❑ There are three main ways to perform NumPy matrix multiplication:

- np.dot(array a, array b): returns the scalar or dot product of two arrays.
- np.matmul(array a, array b): returns the matrix product of two arrays.
- np.multiply(array a, array b): returns the element-wise matrix multiplication of two arrays.

Scalar multiplication or dot product with numpy.dot#

- Scalar multiplication is a simple form of matrix multiplication. A scalar is just a number, like 1, 2, or 3. In scalar multiplication, we multiply a scalar by a matrix. Each element in the matrix is multiplied by the scalar, which makes the output the same shape as the original matrix.

- With scalar multiplication, the order doesn’t matter. We’ll get the same result whether we multiply the scalar by the matrix or the matrix by the scalar.

▪ Let’s take a look at an example:

```
import numpy as np

A = 5
B = [[6, 7],
     [8, 9]]

print(np.dot(A,B))
```

✓**Output :**

```
[[30 35]
 [40 45]]
```

- When it comes to the product between two matrices, it may be done in two ways: scalar/dot product or cross product. Let us attempt to comprehend each of them thoroughly.

- Scalar/Dot Product

✓The inner product accepts two equal-sized vectors and produces a single integer (scalar). This is computed by multiplying the matching items in each vector and totaling the results. Vectors are one-dimensional NumPy arrays in NumPy.

✓We may obtain the inner product either using np.inner() or np.dot(). Both provide the same outcomes.

- Consider the following illustration:

```
# Scalar/ Dot product
import numpy as np
Array1 = np.array([1, 2, 3])
Array2 = np.array([4, 5, 6])
print("Dot product: \n",np.dot(Array2 , Array1))
print("Inner product: \n",np.inner(Array2 , Array1))
```

✓**Output**:

```
Dot product :
 32
Inner product :
 32
```

Matrix product with numpy.matmul#

- The matmul() function gives us the matrix product of two 2-d arrays. With this method, we can’t use scalar values for our input. If one of our arguments is a 1-d array, the function converts it into a NumPy matrix by appending a 1 to its dimension. This is removed after the multiplication is done.

- If one of our arguments is greater than 2-d, the function treats it as a stack of matrices in the last two indexes. The matmul() method is great for times when we’re unsure of what the dimensions of our matrices will be.

- Let’s look at some examples:

✓Multiplying a 2-d array by another 2-d array

```
import numpy as np
A = [[2, 4],
     [6, 8]]
B = [[1, 3],
     [5, 7]]
print(np.matmul(A,B))
```

✓**Output**:

```
[[22 34]
 [46 74]]
```

Element-wise matrix multiplication with numpy.multiply#

- The numpy.multiply() method takes two matrices as inputs and performs element-wise multiplication on them.

- Element-wise multiplication, or Hadamard Product, multiples every element of the first NumPy matrix by the equivalent element in the second matrix. When using this method, both matrices should have the same dimensions.

- Let’s look at an example:

```
import numpy as np
A = np.array([[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]])

B = np.array([[1, 2, 3, 4, 5], [5, 4, 3, 2, 1]])

print(np.multiply(A,B))
```

✓**Output**:

```
[[ 1  6 15 28 45]
 [10 16 18 16 10]]
```

Divide a Matrix by a Scalar

- A matrix's division by a scalar is similar to a scalar's division by a scalar.

```
# Divide a matrix by a scalar
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )
DivideByScalar = Array / 5
print("Output of division between an Array and scalar : \n", DivideByScalar)
```

```
The output of division between an Array and scalar :
 [[0.2 0.4 0.6]
 [0.8 1.  1.2]
 [1.4 1.6 1.8]]
```

Finding Maximum Value in the Matrix

- The np.amax() method may be used to obtain the maximum value along a certain axis. Its parameters are an array and an axis along which the greatest value is to be determined. Let's understand this with the help of an example:

```
# Finding the maximum value in the matrix
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )
print("Max value in the matrix : ", np.amax(Array))
print("Max value in the matrix along axis 0 : ", np.amax(Array, axis=0))
```

✓**Output**:

```
Max value in the matrix:  9
Max value in the matrix along axis 0 :  [[7 8 9]]
```

Finding Minimum Value in the Matrix

- It is very similar to the prior operation. The np.amin() function may be used to find the smallest value along a given axis.

- Its parameters are an array and an axis along which the least value is to be calculated. The amin() and amax() functions are considered to be very important for matrix operations in Python NumPy.

- Let us attempt to understand this with the assistance of an example:

```
# Finding the minimum value in the matrix
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )

print("Minimum value in the matrix : ", np.amin(Array))
print("Minimum value in the matrix along axis 0 : ", np.amin(Array, axis=0))
```

✓**Output**:

```
Minimum value in the matrix:  1
Minimum value in the matrix along axis 0 :  [[1 2 3]]
```

Sum of All Values of a NumPy Matrix

- The np.sum() function will be used to determine the sum of all the elements in the NumPy array.

- This function accepts a ndarray object as input, which represents the original matrix whose elements we wish to sum.

- It provides a 0-dimensional array or a scalar number that is the sum of all the array items. Consider the following scenario:

```
# Sum of all values of a NumPy matrix
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )

print("Sum of all values in the matrix : ", np.sum(Array))
```

✓**Output**:

```
Sum of all values in the matrix:  45
```

Transpose a Matrix

- A matrix's transposition is determined by swapping its rows and columns.

- To acquire the transpose, we can use the 'np.transpose()' function, NumPy 'ndarray.transpose()' function, or 'ndarray.T', a specific method that does not need parenthesis.

- Let us try to grasp this with an example.

```
# Transpose a matrix
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )
```

```
print("Array = ")
print(Array)
print("\nWith np.transpose(Array) function")
print(np.transpose(Array))

print("\nWith ndarray.transpose() method")
print(Array.transpose())

print("\nWith ndarray.T short form")
print(Array.T)
```

✓**Output**:

Array =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

With np.transpose(Array) function
[[1 4 7]
 [2 5 8]
 [3 6 9]]

With ndarray.transpose() method

```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

With ndarray.T short form

```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

✓You must have observed that all of the methods produced the same results.

Rank of a NumPy Matrix

- The dimensions of the vector space formed by a matrix's columns or rows are its rank. In other terms, it is the greatest number of linearly independent column vectors or row vectors.
- The matrix_rank() function from the NumPy linalg package may be used to determine the rank of a matrix.
- Consider the following example:

```
# Rank of a NumPy matrix
import numpy as np
Array = np.matrix( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )
```

```
print("Array = ")
print(Array)
print("\nRank:", np.linalg.matrix_rank(Array))
```

✓**Output**:

```
Array =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Rank: 2
```

Determinant of a Square NumPy Matrix

- The determinant of a square matrix may be determined using the NumPy linalg package's det() function.

- If the determinant is 0, the matrix cannot be inverted. In algebra, this is referred to as a singular matrix.

- Otherwise, if the determinant is not zero, the square matrix is invertible and is said to be non-singular in algebraic terms. Consider the following example, in which we will examine the determinant of a matrix.

```
# Determinant of a square NumPy matrix
import numpy as np
Array1 = np.array( [ [ 1, 2, 3 ],
                     [ 4, 5, 6 ],
                     [ 7, 8, 9 ] ] )
print("Array1 = ")
print(Array1)
print("\nDeterminant:", np.linalg.det(Array1))

Array2 = np.array([[2, 2, 1],
                   [1, 3, 1],
                   [1, 2, 2]])
print("Array2 = ")
print(Array2)
print("\nDeterminant:", np.linalg.det(Array2))
```

✓**Output**:

Array1 =

[[1 2 3]

 [4 5 6]

 [7 8 9]]

Determinant: 0.0

Array2 =

[[2 2 1]

 [1 3 1]

 [1 2 2]]

Determinant: 4.999999999999999

How to Inverse a Matrix using Numpy?

- As we saw in the last section, a square matrix can be either single or non-singular. Thus, if the matrix is non-singular, we compute its true inverse; otherwise, we compute its pseudo inverse.

- True Inverse

✓The inv() method of the NumPy linalg package may be used to get the true inverse of a square matrix. Consider the following example :

```
# True Inverse of NumPy Matrix
import numpy as np
Array = np.array([[2, 2, 1],
                  [1, 3, 1],
                  [1, 2, 2]])
print("Array = ")
print(Array)
print("\nDeterminant:", np.linalg.det(Array))
print("\nInverse of Array = ")
print(np.linalg.inv(Array))
```

✓**Output**:

Array =

[[2 2 1]

 [1 3 1]

 [1 2 2]]

Determinant: 4.999999999999999

Inverse of Array =

[[0.8 -0.4 -0.2]

 [-0.2 0.6 -0.2]

 [-0.2 -0.4 0.8]]

- ✓We should get an error if we try to calculate the true inverse of a singular matrix. For example:

Code:

```
import numpy as np
Array = np.array( [ [ 1, 2, 3 ],
                    [ 4, 5, 6 ],
                    [ 7, 8, 9 ] ] )
print("Array = ")
print(Array)
print("\nDeterminant:", np.linalg.det(Array))
print("\nInverse of Array = ")
print(np.linalg.inv(Array))
```

✓**Output:**

```
Array =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Determinant: 0.0

Inverse of Array =
------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-58-39a460124c92> in <module>()
      7 print("\nDeterminant:", np.linalg.det(Array))
      8 print("\nInverse of Array = ")
----> 9 print(np.linalg.inv(Array))
```

<__array_function__ internals> in inv(*args, **kwargs)

1 frames

/usr/local/lib/python3.7/dist-packages/numpy/linalg/linalg.py in _raise_linalgerror_singular(err, flag)

```
     86
     87 def _raise_linalgerror_singular(err, flag):
---> 88     raise LinAlgError("Singular matrix")
     89
     90 def _raise_linalgerror_nonposdef(err, flag):
```

LinAlgError: Singular matrix

Pseudo Inverse

- The pseudo-inverse may be generated even for singular matrices using the numpy linalg package's pinv() function. For example:

```
# Pseudo Inverse of NumPy Matrix
import numpy as np
Array = np.array( [ [ 1, 2, 3 ],
                    [ 4, 5, 6 ],
                    [ 7, 8, 9 ] ] )
print("Array = ")
print(Array)
print("\nDeterminant:", np.linalg.det(Array))
print("\nInverse of Array = ")
print(np.linalg.pinv(Array))
```

✓**Output**:

```
Array =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Determinant: 0.0

Inverse of Array =
[[-6.38888889e-01 -1.66666667e-01  3.05555556e-01]
 [-5.55555556e-02  3.78742005e-17  5.55555556e-02]
 [ 5.27777778e-01  1.66666667e-01 -1.94444444e-01]]
```

How to Convert a Matrix to a List?

- To transform the array into a list, we may use the NumPy ndarray tolist() method.

- If the array has more than one dimension, a stacked list is produced. A list containing the array items is returned for a one-dimensional array.

- The tolist() method takes no arguments. It's a straightforward method for transforming an array into a list format. Consider the following instance:

```
# Convert a NumPy matrix to a list
import numpy as np
Array = np.array( [ [ 1, 2, 3 ],
                    [ 4, 5, 6 ],
                    [ 7, 8, 9 ] ] )
print("Array = ")
print(Array)
print("\nlist :")
print(Array.tolist())
```

✓**Output**:

Array =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

list :
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Vector Norm using NumPy

- A vector's norm is a measurement of its distance to the origin in vector space. To obtain a vector norm, we employ the Numpy Python library method numpy.linalg.norm(). Consider the following example to better understand it.

```
# Vector norm of NumPy matrix
import numpy as np
Array = np.array( [ [ 1, 2, 3 ],
                    [ 4, 5, 6 ],
                    [ 7, 8, 9 ] ] )
print("Array = ")
print(Array)
print("\nVector Norm:", np.linalg.norm(Array))
```

✓**Output**:

Array =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Vector Norm: 16.881943016134134

Matrix Trace

- The trace is the sum of the diagonal elements. It only applies to square matrices. We get a single number as the trace.

```
# Define a square matrix
A = np.array([[1, 2], [3, 4]])
# Compute the trace of the matrix
trace_A = np.trace(A)
print("Trace of the Matrix:", trace_A)
```

✓**Output**:

Trace of the Matrix: 5

LU Decomposition

- LU decomposition breaks a matrix into two parts. One part is a lower triangular matrix (L). The other part is an upper triangular matrix (U). It helps solve linear least squares problems and find eigenvalues.

```
import numpy as np
from scipy.linalg import lu


# Define a matrix
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# LU Decomposition
P, L, U = lu(A)
# Display results
print("LU Decomposition:")
print("P matrix:\n", P)
print("L matrix:\n", L)
print("U matrix:\n", U)
```

✓**Output**:

LU Decomposition:
P matrix:
 [[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
L matrix:
 [[1.   0.   0.   ]
 [0.14285714 1.   0.   ]
 [0.57142857 0.5   1.   ]]
U matrix:
 [[7.00000000e+00 8.00000000e+00 9.00000000e+00]
 [0.00000000e+00 8.57142857e-01 1.71428571e+00]
 [0.00000000e+00 0.00000000e+00 -1.58603289e-16]]

QR Decomposition

- QR decomposition divides a matrix into two parts. One part is an orthogonal matrix (Q). The other part is an upper triangular matrix (R). It helps solve linear least squares problems and find eigenvalues.

```
import numpy as np
from scipy.linalg import qr
# Define a matrix
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# QR Decomposition
Q, R = qr(A)
# Display results
print("QR Decomposition:")
print("Q matrix:\n", Q)
print("R matrix:\n", R)
```

✓**Output**:

QR Decomposition:
Q matrix:
 [[-0.12309149 0.90453403 0.40824829]
 [-0.49236596 0.30151134 -0.81649658]
 [-0.86164044 -0.30151134 0.40824829]]
R matrix:
 [[-8.12403840e+00 -9.60113630e+00 -1.10782342e+01]
 [0.00000000e+00 9.04534034e-01 1.80906807e+00]
 [0.00000000e+00 0.00000000e+00 -1.11164740e-15]]

SVD (Singular Value Decomposition)

- SVD decomposes a matrix into three matrices: U, Σ, and V*. U and V* are orthogonal matrices. Σ is a diagonal matrix. It is useful in many applications like data reduction and solving linear systems.

```
import numpy as np
from scipy.linalg import svd
# Define a matrix
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Singular Value Decomposition
U, s, Vh = svd(A)
# Display results
print("SVD Decomposition:")
print("U matrix:\n", U)
print("Singular values:\n", s)
print("Vh matrix:\n", Vh)
```

✓**Output**:

SVD Decomposition:
U matrix:
[[-0.21483724 0.88723069 0.40824829]
[-0.52058739 0.24964395 -0.81649658]
[-0.82633754 -0.38794278 0.40824829]]
Singular values:
[1.68481034e+01 1.06836951e+00 1.40266540e-16]
Vh matrix:
[[-0.47967118 -0.57236779 -0.66506441]
[-0.77669099 -0.07568647 0.62531805]
[0.40824829 -0.81649658 0.40824829]]

Direct Solution of Linear Equations

- Find the values of variables that satisfy equations in a system. Each equation represents a straight line. The solution is where these lines meet.

```
# Define matrix A and vector B
A = np.array([[3, 1], [1, 2]])
B = np.array([9, 8])
# Solve the system of linear equations Ax = B
x = np.linalg.solve(A, B)
print("Solution to Ax = B:", x)
```

✓**Output**:

```
Solution to Ax = B: [2. 3.]
```

Least Squares Fitting

- The least squares fitting finds the best match for data points. It lowers the squared differences between actual and predicted values.

```
# Define matrix A and vector B
A = np.array([[1, 1], [1, 2], [1, 3]])
B = np.array([1, 2, 2])
# Solve the linear least-squares problem
x, residuals, rank, s = np.linalg.lstsq(A, B, rcond=None)
print("Least Squares Solution:", x)
print("Residuals:", residuals)
print("Rank of the matrix:", rank)
print("Singular values:", s)
```

✓**Output**:

Least Squares Solution: [0.66666667 0.5]

Residuals: [0.16666667]

Rank of the matrix: 2

Singular values: [4.07914333 0.60049122]

Condition Number

- The condition number of a matrix measures sensitivity to input changes. A high condition number means the solution could be unstable.

```
# Define a matrix
A = np.array([[1, 2], [3, 4]])
# Compute the condition number of the matrix
condition_number = np.linalg.cond(A)
print("Condition Number:", condition_number)
```

✓**Output**:

Condition Number: 14.933034373659265