5CS037 - Concepts and Technologies of AI.

Worksheet - 1: A coding exercises on Numpy.

Prepared By: Siman Giri {Module Leader - 5CS037}

Completed By: Kamal Dhital {Group - L5CG5, Student ID - 2407046}

Date: December 2, 2024

3 Getting Started with Numpy

1. Importing Numpy:

Importing Numpy and Array Type

+ Code + Text

```
import numpy as np
# Create and display zero, one, and n-dimensional arrays
zero_dim_array = np.array(5)
one_dim_array = np.array([1,2,3])
n_dim_array = np.array([[1,2], [3,4]])
for arr in [zero_dim_array, one_dim_array, n_dim_array]:
  print(f"Array:\n{arr}\nDimension: {arr.ndim}\nData type: {arr.dtype}\n")
```

```
Array:
5
Dimension: 0
Data type: int64

Array:
[1 2 3]
Dimension: 1
Data type: int64

Array:
[[1 2]
 [3 4]]
Dimension: 2
Data type: int64
```

2. Array Dimensions: Shape and Reshape of Array:

Shape of an Array

```
import numpy as np
# Create arrays of different dimensions
array_0d = np.array(5)
array_1d = np.array([1, 2, 3, 4, 5])
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
# Print arrays with shapes
for i, arr in enumerate([array_0d, array_1d, array_2d, array_3d]):
  print(f"{i}D Array:\n{arr}\nShape: {arr.shape}\n")
```

```
0D Array:
5
Shape: ()

1D Array:
[1 2 3 4 5]
Shape: (5,)

2D Array:
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)

3D Array:
```

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

 [[5 6]
  [7 8]]]
Shape: (2, 2, 2)
```

Reshaping an Array

```
import numpy as np
array = np.array([[1, 2, 3], [4, 5, 6]]) # Shape (2, 3)
reshaped_array = array.reshape(3, 2) # Reshape to (3, 2), keeping 6 elements
print("Original Shape:", array.shape, "\nReshaped Shape:", reshaped_array.shape)
```

```
Original Shape: (2, 3)
Reshaped Shape: (3, 2)
```

1. Using In-Built function:

Arrays with evenly Spaced values - arange

```
import numpy as np
a = np.arange(1, 10)
print(a)
x = range(1, 10)
print(x) # x is an iterator
print(list(x))
# further arange examples:
x = np.arange(10.4)
print(x)
x = np.arange(0.5, 10.4, 0.8)
print(x)
```

```
[1 2 3 4 5 6 7 8 9]
range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 0.5  1.3  2.1  2.9  3.7  4.5  5.3  6.1  6.9  7.7  8.5  9.3 10.1]
```

Arrays with evenly Spaced values - linspace.

```
import numpy as np
# 50 values between 1 and 10:
print(np.linspace(1, 10))
# 7 values between 1 and 10:
print(np.linspace(1, 10, 7))
# excluding the endpoint:
print(np.linspace(1, 10, 7, endpoint=False))
```

```
[ 1.         1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
  2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
  3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12244898
  4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.2244898
  5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32653061
  6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42857143
  7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53061224
  8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
  9.81632653 10.        ]
[ 1.   2.5  4.   5.5  7.   8.5 10. ]
[1.         2.28571429 3.57142857 4.85714286 6.14285714 7.42857143
 8.71428571]
```

2. Initializing Arrays with Ones, Zeros and Empty:

Initializing an Array

```
import numpy as np
# Create arrays of specified shapes
ones_array = np.ones((2, 3)) # Shape: (2, 3)
```

```
zeros_array = np.zeros((3, 2)) # Shape: (3, 2)
empty_array = np.empty((2, 2)) # Shape: (2, 2)
identity_matrix = np.eye(3) # Shape: (3, 3)
print(ones_array, zeros_array, empty_array, identity_matrix, sep='\n\n')
```

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

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

[[4.9e-324 9.9e-324]
 [1.5e-323 2.0e-323]]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

3. By Manipulating Existing Array:

3.1 Using np.array:

Converting list to array using np.array.

```
import numpy as np
array_from_list = np.array([1, 2, 3]) # [1 2 3]
array_from_tuple = np.array((4, 5, 6)) # [4 5 6]
array_from_nested_list = np.array([[1, 2, 3], [4, 5, 6]]) # [[1 2 3] [4 5 6]]
print(array_from_list, array_from_tuple, array_from_nested_list, sep='\n')
```

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

3.2 Using shape of an existing array:

From shape of an existing array.

```
import numpy as np
# Existing Array of Shape:(2,3)
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Creating Array with Shape of existing array:
zeros, ones, empty = np.zeros(arr.shape), np.ones(arr.shape), np.empty(arr.shape) # Shape: (2,3)
print(arr, zeros, ones, empty, sep='\n')
```

```
[[1 2 3]
 [4 5 6]]
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
[[4.99829508e-310 0.00000000e+000 6.94592882e-310]
 [4.99930287e-310 6.94595963e-310 6.94595963e-310]]
```

3.3 With Math, Copying or Slicing and Array:

Manipulation existing array.

```
import numpy as np
array = np.array([1, 2, 3])
# Multiplies each element by 2, result: [2, 4, 6]
new_array = array * 2
# Creates a new array with the same elements
copied_array = np.copy(array)
# Slices elements from index 1 to 3, result:[2]
sliced_array = array[1:2]
```

With Concatenating Operation.

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]]) # Shape: (2, 2)
```

```
arr2 = np.array([[5, 6], [7, 8]]) # Shape: (2, 2)
concat_axis0 = np.concatenate((arr1, arr2), axis=0)
# Concatenate along axis 0 (vertical) Shape: (4, 2)
concat_axis1 = np.concatenate((arr1, arr2), axis=1)
# Concatenate along axis 1 (horizontal) Shape: (2, 4)
print(concat_axis0, concat_axis1, sep='\n')
```

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

3.4 Array: Indexing and Slicing:

Indexing and Slicing.

```
import numpy as np
# Arrays
arr1d = np.array([0, 1, 2, 3, 4, 5])
arr2d = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
arr3d = np.array([[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]])
# Slicing examples
print("Basic:", arr1d[1:4], arr2d[1:3, 0:2], arr3d[1:2, 0:2, 1:2])
# Output Basic: arr1d:[1 2 3] arr2d:[[3 4] [6 7]] arr3d:[[[5] [7]]]
print("Step:", arr1d[1:5:2], arr2d[::2, 1::2], arr3d[::2, ::2, 1::2])
# Output: Step: [1 3] [[1] [7]] [[[1] [9]]]
```

```
Basic: [1 2 3] [[3 4]
 [6 7]] [[[5]
  [7]]]
Step: [1 3] [[1]
 [7]] [[[1]]

 [[9]]]
```

3.3 Array Mathematics:

1. Elementary Mathematical Operation with universal functions.

Introduction to "ufuncs".

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
print("Add:", np.add(arr1, arr2)) # Output Add: [5 7 9]
print("Subtract:", np.subtract(arr1, arr2))
# Output Subtract: [-3 -3 -3]
print("Divide:", np.divide(arr1, arr2))
# Output Divide: [0.25 0.4 0.5 ]
print("Multiply:", np.multiply(arr1, arr2))
# Output Multiply: [ 4 10 18]
print("Power:", np.power(arr1, arr2))
# Output Power: [ 1 32 729]
print("Exp:", np.exp(arr1))
# Output Exp: [ 2.71828183 7.3890561 20.08553692]
```

```
Add: [5 7 9]
Subtract: [-3 -3 -3]
Divide: [0.25 0.4  0.5 ]
Multiply: [ 4 10 18]
Power: [  1  32 729]
Exp: [ 2.71828183  7.3890561  20.08553692]
```

2. Matrix Multiplications.

Using "np.dot".

```
import numpy as np
# Define two matrices
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8], [9, 10], [11, 12]])
# Matrix multiplication using np.dot
result_dot = np.dot(A, B)
print("Result with np.dot:\n", result_dot) # Output shape: (2, 2)
```

```
Result with np.dot:
 [[ 58  64]
 [139 154]]
```

Using "np.matmul".

```
import numpy as np
# Define two matrices
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8], [9, 10], [11, 12]])
# Matrix multiplication using @ operator
result_at = A @ B
print("Result with @ operator:\n", result_at) # Output shape: (2, 2)
# Matrix multiplication using np.matmul
result_matmul = np.matmul(A, B)
print("Result with np.matmul:\n", result_matmul) # Output shape: (2, 2)
```

```
Result with @ operator:
 [[ 58  64]
 [139 154]]
Result with np.matmul:
 [[ 58  64]
 [139 154]]
```

3.4 Linear Algebra with Numpy - np.linalg:

Common linear algebra operation with np.linalg.

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([5, 6])
print("Inverse:\n", np.linalg.inv(A))
# Output: Inverse: [[-2. 1. ], [ 1.5 -0.5]]
print("Determinant:", np.linalg.det(A))
# Output: Determinant: -2.0
print("Frobenius Norm:", np.linalg.norm(A,'fro'))
# Output: Frobenius Norm: 5.4772
print("2-Norm (Euclidean):", np.linalg.norm(A))
# Outptut: 2-Norm(Euclidean): 5.4772
print("Solution x:", np.linalg.solve(A, B))
#Output: Solution x: [-4. 4.5]
```

```
Inverse:
 [[-2.   1. ]
 [ 1.5 -0.5]]
Determinant: -2.0000000000000004
Frobenius Norm: 5.477225575051661
2-Norm (Euclidean): 5.477225575051661
Solution x: [-4.   4.5]
```

4. TO - DO - Task

Please complete all the problem listed below.

4.1 Warming Up Exercise: Basic Vector and Matrix Operation with Numpy.

Problem - 1: Array Creation:

Complete the following Tasks:

1. Initialize an empty array with size 2X2.

2. Initialize an all one array with size 4X2.
3. Return a new array of given shape and type, filled with fill value.{Hint: np.full}
4. Return a new array of zeros with same shape and type as a given array.{Hint: np.zeros like}
5. Return a new array of ones with same shape and type as a given array.{Hint: np.ones like}
6. For an existing list new_list = [1,2,3,4] convert to an numpy array.{Hint: np.array()}

```
import numpy as np


def empty_array(arr):
  return np.empty(arr)

empty_array = empty_array((2,2))
print(empty_array)
```

```
[[1. 2.]
 [3. 4.]]
```

```
def one_array(arr):
  return np.ones(arr)

one_array = one_array((4,2))
print(one_array)
```

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

```
def filled_array(arr, fill_value):
  return np.full(arr, fill_value)

filled_array = filled_array((2,2), 3.14)
print(filled_array)
```

```
[[3.14 3.14]
 [3.14 3.14]]
```

```
def zeros_array(arr):
  return np.zeros_like(arr)

zeros_array = zeros_array(filled_array)
print(zeros_array)
```

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

```
def ones_array(arr):
  return np.ones_like(arr)

ones_array = ones_array([[1,2],[3,4]])
print(ones_array)
```

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

```
def new_array(arr):
  return np.array(arr)

new_list = [1,2,3,4]
new_array = new_array(new_list)
print(new_array)
```

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

4.1.1 Problem - 2: Array Manipulation: Numerical Ranges and Array indexing:

Complete the following tasks:

1. Create an array with values ranging from 10 to 49. {Hint:np.arrange()}.
2. Create a 3X3 matrix with values ranging from 0 to 8. {Hint:look for np.reshape()}

3. Create a 3X3 identity matrix.{Hint:np.eye()}
4. Create a random array of size 30 and find the mean of the array. {Hint:check for np.random.random() and array.mean() function}
5. Create a 10X10 array with random values and find the minimum and maximum values.
6. Create a zero array of size 10 and replace 5th element with 1.
7. Reverse an array arr = [1,2,0,0,4,0].
8. Create a 2d array with 1 on border and 0 inside.
9. Create a 8X8 matrix and fill it with a checkerboard pattern.

```
range_array = np.arange(10,49)
print(range_array)
```

```
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48]
```

```
matrix_array = np.arange(9).reshape((3,3))
print(matrix_array)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

```
new_identity_matrix = np.eye(3)
print(new_identity_matrix)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
random_array = np.random.random(30)
mean_of_array = random_array.mean()
print(mean_of_array)
```

```
0.4422104326925898
```

```
random_array = np.random.random(100).reshape((10,10))
max_of_array = random_array.max()
min_of_array = random_array.min()
print(random_array)
print(max_of_array)
print(min_of_array)
```

```
[[0.89071179 0.81152162 0.72445061 0.3123124  0.53650995 0.05542641
  0.41658478 0.26375355 0.07968741 0.99338438]
 [0.0658038  0.07331661 0.89065653 0.76000925 0.90726251 0.96607585
  0.44762471 0.8843236  0.84257832 0.18800393]
 [0.3281709  0.28910419 0.96540768 0.27085345 0.17914269 0.76972482
  0.81414529 0.63073552 0.21065739 0.40315779]
 [0.47091095 0.22083378 0.86466441 0.68220788 0.84276014 0.28457973
  0.16563061 0.05878887 0.20992406 0.93546138]
 [0.00749941 0.276519   0.69054723 0.13545384 0.16777275 0.57200751
  0.18642539 0.19017215 0.38908088 0.33576541]
 [0.23356345 0.70675673 0.92996567 0.59131477 0.45254834 0.55862841
  0.45476834 0.92473354 0.22996387 0.12185509]
 [0.93979946 0.29476323 0.47976144 0.49741987 0.40367141 0.60881324
  0.98487863 0.77078886 0.14295762 0.3740599 ]
 [0.50189107 0.26610392 0.81859784 0.32065804 0.82657201 0.98024193
  0.6477825  0.46018145 0.03157767 0.22423169]
 [0.20971254 0.49155351 0.57740268 0.40272499 0.39488816 0.21564092
  0.63639377 0.25895488 0.61031351 0.44687558]
 [0.32670538 0.38294218 0.61617502 0.35698822 0.77618625 0.78499702
  0.18648882 0.53142233 0.44866763 0.24126912]]
0.9933843842276454
0.007499408357306336
```

```
zeros_array = np.zeros(10)
zeros_array[4] = 1
print(zeros_array)
```

```
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

```
arr = [1,2,0,0,4,0]
arr.reverse()
print(arr)
```

```
[0, 4, 0, 0, 2, 1]
```

```
arr = np.zeros((5, 5))
arr[0, :] = 1
arr[-1, :] = 1
arr[:, 0] = 1
arr[:, -1] = 1
print(arr)
```

```
[[1. 1. 1. 1. 1.]
 [1. 0. 0. 0. 1.]
 [1. 0. 0. 0. 1.]
 [1. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1.]]
```

```
checkerboard = np.zeros((8, 8))
checkerboard[::2, ::2] = 1
checkerboard[::2, 1::2] = 1
checkerboard[1::2, ::2] = 1
checkerboard[1::2, 1::2] = 0
print(checkerboard)
```

```
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]]
```

Problem - 3: Array Operations:

For the following arrays:

x = np.array([[1,2],[3,5]]) and y = np.array([[5,6],[7,8]]);

Complete all the task using numpy:

1. Add the two array.
2. Subtract the two array.
3. Multiply the array with any integers of you choice.
4. Find the square of each element of the array.
5. Find the dot product between: v(and)w; x(and)v; x(and)y.
6. Concatenate x(and)y along row and Concatenate v(and)w along column.

   {Hint: try np.concatenate() or np.vstack() functions.
7. Concatenate x(and)v; if you get an error, observe and explain why did you get the error?

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
```

```
addition_of_array = x+y
print(f"Addition of {x} array and {y} array is: {addition_of_array}")
```

```
Addition of [[1 2]
 [3 4]] array and [[5 6]
 [7 8]] array is: [[ 6  8]
 [10 12]]
```

```
subtraction_of_array = x-y
print(f"Subtraction of {x} array and {y} array is: {subtraction_of_array}")
```

```
Subtraction of [[1 2]
 [3 4]] array and [[5 6]
 [7 8]] array is: [[-4 -4]
```

```
      [-4 -4]]


multiplication_of_array = x*y
print(f"Multiplication of {x} array and {y} array is: {multiplication_of_array}")
```

```
Multiplication of [[1 2]
 [3 4]] array and [[5 6]
 [7 8]] array is: [[ 5 12]
 [21 32]]
```

```
square_of_array = x**2
print(f"Square of {x} array is: {square_of_array}")
```

```
Square of [[1 2]
 [3 4]] array is: [[ 1  4]
 [ 9 16]]
```

```
v = np.array([9,10])
w = np.array([11, 12])
dot_product_v_w = np.dot(v,w)
dot_product_x_v = np.dot(x,v)
dot_product_x_y = np.dot(x,y)

print(f"Dot product of {v} array and {w} array is: {dot_product_v_w}\n")
print(f"Dot product of {x} array and {v} array is: {dot_product_x_v}\n")
print(f"Dot product of {x} array and {y} array is: {dot_product_x_y}\n")
```

```
Dot product of [ 9 10] array and [11 12] array is: 219

Dot product of [[1 2]
 [3 4]] array and [ 9 10] array is: [29 67]

Dot product of [[1 2]
 [3 4]] array and [[5 6]
 [7 8]] array is: [[19 22]
 [43 50]]
```

```
xy_row = np.concatenate((x,y), axis=0)
vw_column = np.concatenate((v.reshape(2,1),w.reshape(2,1)), axis=1)

print("Concatenate x(and)y along row:\n", xy_row, "\n")
print("Concatenate v(and)w along column:\n", vw_column, "\n")
```

```
Concatenate x(and)y along row:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]

Concatenate v(and)w along column:
 [[ 9 11]
 [10 12]]
```

```
# xv_concatenate = np.concatenate((x,v),axis = 0)
# print("Concatenate x(and)v along row: \n", xv_concatenate)
```

Here, in the above concatenate we get the error as we have two dimension value for x where v has the value of one dimension. We must need to have same dimension to concatenation two variables.

Problem - 4: Matrix Operations:

For the following arrays:

A = np.array([[3,4],[7,8]]) and **B = np.array([[5,3],[2,1]])**

Prove following with Numpy:

1. Prove A.A−1 = I.

2. Prove AB ≠ BA.
3. Prove (AB)T = BT AT.

```
A = np.array([[3,4],[7,8]])
B = np.array([[5,3],[2,1]])

C = np.linalg.inv(A)
D = np.linalg.inv(B)

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

```
[[1.00000000e+00 0.00000000e+00]
 [1.77635684e-15 1.00000000e+00]]
```

```
AB = np.dot(A,B)
BA = np.dot(B,A)

print(AB)
print(BA)

print(AB == BA)
```

```
[[23 13]
 [51 29]]
[[36 44]
 [13 16]]
[[False False]
 [False False]]
```

```
AB = np.dot(A,B)
C = np.transpose(B)
D = np.transpose(A)
E = np.dot(C,D)

ABT = np.transpose(AB)

print(f"Transpose of AB is: {ABT}")
print(f"Transpose of B is: {C}")
print(f"Transpose of A is: {D}")
print(f"Transpose of CD is: {E}")

print(ABT == E)
```

```
Transpose of AB is: [[23 51]
 [13 29]]
Transpose of B is: [[5 2]
 [3 1]]
Transpose of A is: [[3 7]
 [4 8]]
Transpose of CD is: [[23 51]
 [13 29]]
[[ True  True]
 [ True  True]]
```

- Solve the following system of Linear equation using Inverse Methods.

$$2x - 3y + z = -1$$
$$x - y + 2z = -3$$
$$3x + y - z = 9$$

{Hint: First use Numpy array to represent the equation in Matrix form. Then Solve for: AX = B}

```
A = ([[2,-3,1],[1,-1,2],[3,1,-1]])
B = ([-1,-3,9])

X = np.linalg.solve(A,B)
print(X)
```

```
[ 2.  1. -2.]
```

- Now: Solve the above equation using np.linalg.inv function. {Explore more about "linalg" function of Numpy}

```
A = [[2,-3,1],[1,-1,2],[3,1,-1]]
B = [-1,-3,9]
```

```
X = np.dot(np.linalg.inv(A),B)
print(X)
```

```
[ 2.  1. -2.]
```