Introduction to Numpy

Lab 2

Introduction to Numpy

What is NumPy?

- NumPy name by itself stands for Numerical Python.
- NumPy is a Python library used for working with arrays.
- NumPy library has a huge number of functions and utilities. It has functions for working in domain of linear algebra, fourier transform, and matrices.

Introduction to Numpy

Why NumPy?

- Python has lists which we can use to implement arrays, but they are slow to process.
- NumPy on the other hand aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are frequently used in machine learning tasks, where speed and optimal resource allocation are very important.
- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. (Locality of Reference)

Introduction to Numpy

Installation of NumPy

- If you have Python and PIP already installed on a system, then installation of NumPy is very easy.
- Install it using this command: pip install numpy
- Once NumPy is installed, import it in your application: import numpy
- Now NumPy is imported and ready to use.

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

Introduction to Numpy

NumPy as np

- NumPy is usually imported under the np alias.
- **alias**: In Python alias are an alternate name for referring to the same thing.
- Create an alias with the as keyword while importing: import numpy as np

```
import numpy as np

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

print(arr)
```

NumPy Basics

Arrays

- A numpy array is a grid of values, same type, and is indexed by a tuple of nonnegative integers.
- The number of dimensions is the rank of the array.
- The shape of an array is a tuple of integers giving the size of the array along each dimension.

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

NumPy Basics

- Numpy also provides many functions to create arrays:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7) # Create a constant array
print(c)              # Prints "[[ 7.  7.]
                      #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

NumPy Basics

Tensors

- Tensors also known as multi dimensional arrays.
- A Scalar is a 0D Tensor
- A Vector is a 1D Tensor
- A Matrix is a 2D Tensor etc etc
- The understanding of Tensors will be essential when we move to pytorch!!

```
import numpy as np

T = np.random.random((2,3,4))
```

NumPy Basics

Tensors

- Let's use numpy random function to create a 3D Tensor!

```
T = np.random.random((2,3,4))
print("Our 3D Tensor:\n", T)
```

- This is what our 3D tensor looks like, we can pretend (for this Tensor) that it is just 2, 3x4 matrices stacked together.

```
# Our 3D Tensor:
#  [[[0.64268418 0.6355261  0.98267682 0.82583937]
#  [0.26318569 0.13252364 0.11492685 0.58669684]
#  [0.12021839 0.40946573 0.02935382 0.64855154]]

# [[0.69871537 0.92177547 0.43660625 0.81078584]
#  [0.21851072 0.72177234 0.33881577 0.44209828]
#  [0.02541905 0.23297449 0.55655358 0.76485194]]]
```

NumPy Basics

Basic Element-wise Operations

- Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
print(np.sqrt(x))
```

NumPy Basics

Matrix Operations

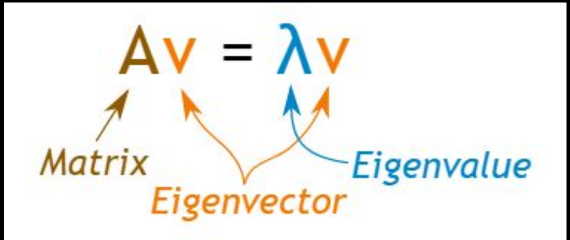
- Now let's see some Numpy functions to perform some real matrix operations:

```python
matrix_1 = np.random.random((3,3))
matrix_2 = np.random.random((3,3))

# Addition
print(np.add(matrix_1,matrix_2))

# Subtraction
print(np.subtract(matrix_1,matrix_2))

# Multiplication
print(np.matmul(matrix_1,matrix_2))

# Inverse of Matrix
print(np.linalg.inv(matrix_1))

# Transpose of matrix
print(matrix_1.T)

# Determinant of matrix
print(np.linalg.det(matrix_1))

# Print the Trace
print(matrix_1.trace())

# Rank
print(np.linalg.matrix_rank(matrix_1))
```

NumPy Basics

Matrix Operations

- For a square matrix A, an Eigenvector and Eigenvalue make this equation true:



- The Eigenvalues and Eigenvectors can be calculated using NumPy:

```
# Calculate the Eigenvalues and Eigenvectors of that Matrix
eigenvalues ,eigenvectors=np.linalg.eig(matrix_1)

print("The Eigenvalues are:\n", eigenvalues)
print("The Eigenvectors are:\n", eigenvectors)
```

NumPy Basics

Tensor Manipulation

- ***Indexing***
  - Indexing is the same as accessing an element in a vector or matrix.
  - You can access an array element by referring to its index number.
  - The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
vector = np.array([1, 2, 3, 4])

print("All elements:\n", vector[:])
print("Within a range:\n", vector[:3])
print("Another range:\n",vector[3:])
print("The last element:\n",vector[-1])
```

```
matrix = np.array([[1,2,3],[4,5,6],[7,8,9]])

print("A single element in 2nd row 2nd column:\n", matrix[1,1])
print("Elements in the first 2 rows and all the columns:\n", matrix[:2,:])
print("Elements in all rows and the 2nd column:\n", matrix[:,:2])
print("Elements in rows 1 and 2 and column 2 and 3:\n", matrix[:2,1:])
```

NumPy Basics

Tensor Manipulation

- ***Reshaping***
  - Reshaping means changing the shape of an array.
  - By reshaping we can add or remove dimensions or change number of elements in each dimension.

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

print("Reshape to 4x3:\n", arr.reshape(4, 3))
print("Reshape to 1D again:\n", arr.flatten())
print("Reshape to 3xwhatever:\n", arr.reshape(3,-1))
```

NumPy Basics

Tensor Manipulation

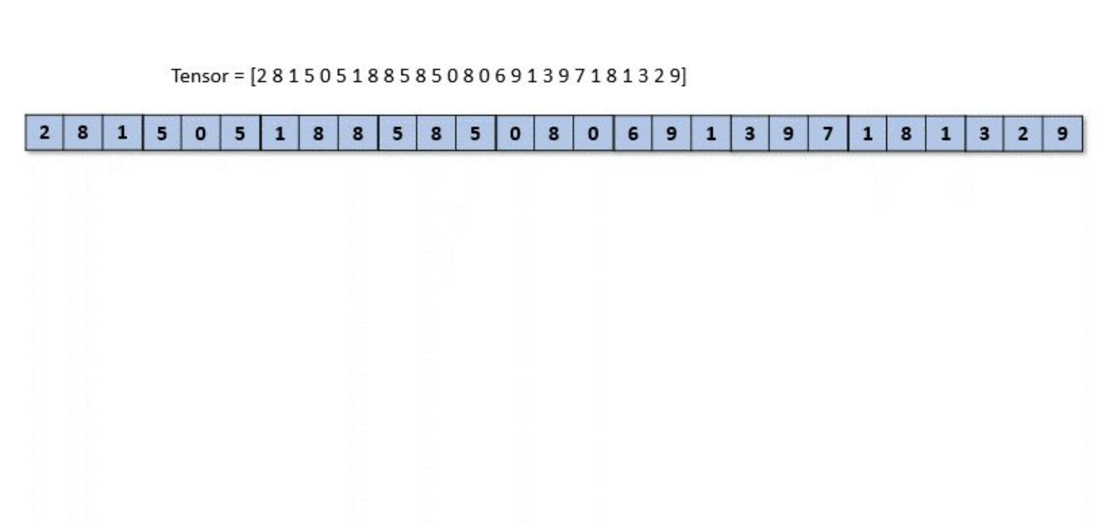
- ***Reshaping Example - flatten function***

3	9	7
1	8	1
3	2	9

NumPy Basics

Tensor Manipulation

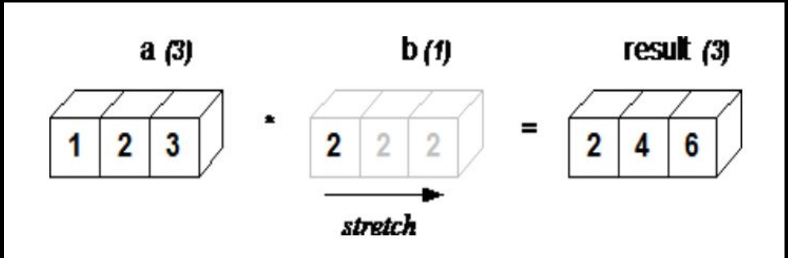
- ***Reshaping Example - Reshaping to 2D***

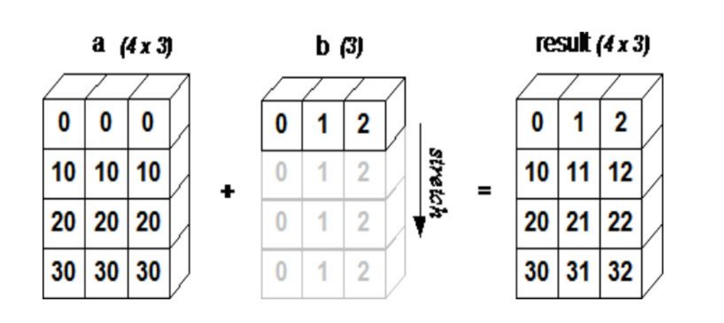


NumPy Basics

Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.
- It is the process of extending two arrays of different shapes and figuring out how to perform a vectorized calculation between them.
- Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.





NumPy Basics

Broadcasting

Broadcasting is possible if the following rules are satisfied:

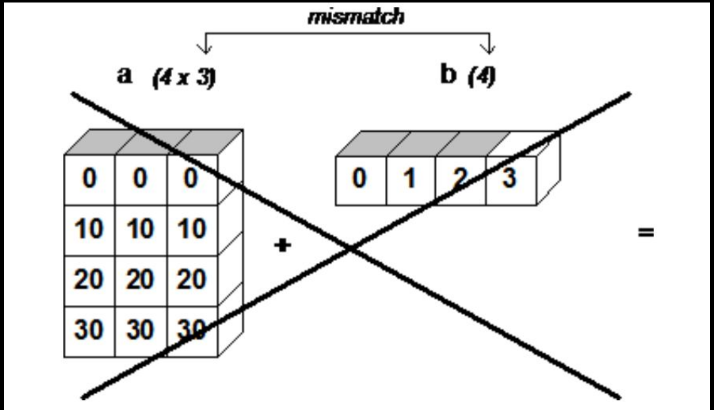
- Array with smaller ndim than the other is prepended with '1' in its shape.
- Size in each dimension of the output shape is maximum of the input sizes in that dimension.
- An input can be used in calculation, if its size in a particular dimension matches the output size or its value is exactly 1.
- If an input has a dimension size of 1, the first data entry in that dimension is used for all calculations along that dimension.

NumPy Basics

Broadcasting

A set of arrays is said to be ***broadcastable*** if the above rules produce a valid result and one of the following is true:

- Arrays have exactly the same shape.
- Arrays have the same number of dimensions and the length of each dimension is either a common length or 1.
- Array having too few dimensions can have its shape prepended with a dimension of length 1, so that the above stated property is true.



NumPy Basics

Broadcasting

```python
a = np.array([[0.0,0.0,0.0],
              [10.0,10.0,10.0],
              [20.0,20.0,20.0],
              [30.0,30.0,30.0]])

b = np.array([1.0,2.0,3.0])

print('First array: \n', a , '\n')

print('Second array:\n', b, '\n')

print('First Array + Second Array\n', a + b)
```

```
First array:
 [[ 0.  0.  0.]
 [10. 10. 10.]
 [20. 20. 20.]
 [30. 30. 30.]]

Second array:
 [1. 2. 3.]

First Array + Second Array
 [[ 1.  2.  3.]
 [11. 12. 13.]
 [21. 22. 23.]
 [31. 32. 33.]]
```

- We can see the resulting shape of the Tensor addition seems to come from the larger dimensions of the multiplication. i.e., 4x**4** + **4**x1 = **4x4**

NumPy Basics

Why Use Broadcasting?

- By using broadcasting we don’t have to use loops and indexing to perform operations!
- And you will end up with a clean quality code.

Dimensions

Confused?

Tensors -

When we talk about the number of dimensions of a Tensor we are referring to the **SHAPE** of the Tensor, aka how many numbers does it take to describe the shape e.g. NxM - 2D, CxNxM - 3D, BxCxNxM - 4D, etc….

Datapoints -

When we talk about the number of dimensions of a datapoint we are referring to the SIZE of the datapoint , aka how many numbers/attributes are needed to represent a single datapoint e.g. [4, 6] - 2D, [2, 5, 1, 8, 4] - 5D, etc…...

Dimensions

Combining the two:

Consider the datapoint x = [2, 5, 4, 6, 7, 8, 4, 3, 0]

This datapoint has 9 dimensions (9 “ways” it can vary)

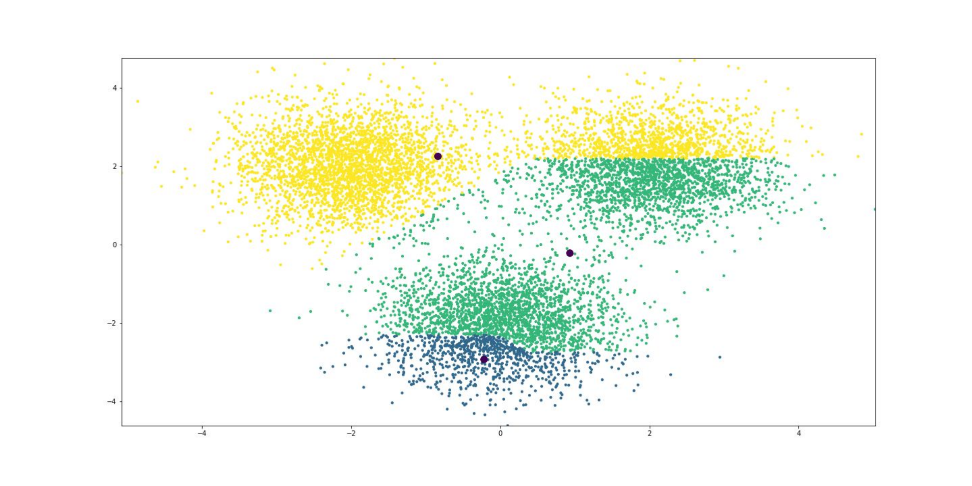
We can represent this 9D datapoint as a 2D Tensor:

$$X = \begin{matrix} [[2, 5, 4] \\ [6, 7, 8] \\ [4, 3, 0]] \end{matrix}$$

Today's Task! K-Means Clustering ...

What is K-means Clustering Algorithm ?

- A K-means clustering algorithm tries to group similar items in the form of clusters. The number of clusters/groups is represented by K. (the number of clusters usually defined by the user)
- It finds the similarity between the items and groups them into the clusters.



Today's Task! K-Means Clustering ...

How does the K-means Clustering Algorithm work ?

- K-Means clustering works by iteratively updating a pre-defined number of cluster centers (known as centroids).
- It does this by finding the distance between each datapoint and every cluster center.
- Datapoints are then assigned to the cluster center they are closest to and each cluster center is updated to be the mean of the new cluster.
- These steps are repeated for some number of steps or until the cluster centers converge (they stop moving so much).

Today's Task! K-Means Clustering ...

The steps of K-means clustering

1. Define the number of clusters "k" you want to group your data into.
2. Randomly initialise k vectors with the same size as each datapoint, this is the initialisation of our cluster centers (centroids).
3. Calculate the distance between each datapoint and each cluster center (using MSE or equivalent).
4. For every datapoint find the cluster center they are closest to.
5. Re-calculate the cluster centers by finding the mean of every new cluster.
6. Repeat steps 3-5 for n steps or until convergence.