Deep Learning: Introduction to Numpy

Gurgen Hovakimyan

Yereva State University

Agenda

1. Introduction to numpy

Agenda

1. Introduction to numpy
2. Simple computational graph

Agenda

1. Introduction to numpy
2. Simple computational graph
3. Example of gradient descent

Agenda

1. Introduction to numpy
2. Simple computational graph
3. Example of gradient descent
4. Logistic Regression from scratch

Agenda

1. Introduction to numpy
2. Simple computational graph
3. Example of gradient descent
4. Logistic Regression from scratch
5. Introduction to deep learning frameworks (PyTorch)

Section 1

Introduction to numpy

What is NumPy?

- NumPy is a Python library used for working with arrays

What is NumPy?

- NumPy is a Python library used for working with arrays
- Very often used for linear algebra, fourier transformations and matrices

What is NumPy?

- NumPy is a Python library used for working with arrays
- Very often used for linear algebra, fourier transformations and matrices
- NumPy stands for Numerical Python

Benefits of using NumPy

- In Python we have lists that serve the purpose of arrays, but they are slow to process.

Benefits of using NumPy

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

Importing numpy

NumPy is usually imported under the np alias.

```
import numpy as np

print(np.__version__)
```

```
## 1.21.5
```

Arrays in numpy

```
x = np.array([10, 1, 1, 2, 3])
print(x)
```

```
## [10  1  1  2  3]
```

```
print(x.shape)
```

```
## (5,)
```

The shape of array

- For 1D array, return a shape tuple with only 1 element (i.e. (n,))

The shape of array

- For 1D array, return a shape tuple with only 1 element (i.e. (n,))
- For 2D array, return a shape tuple with only 2 elements (i.e. (n,m))

The shape of array

- For 1D array, return a shape tuple with only 1 element (i.e. (n,))
- For 2D array, return a shape tuple with only 2 elements (i.e. (n,m))
- For 3D array, return a shape tuple with only 3 elements (i.e. (n,m,k))

The shape of array

- For 1D array, return a shape tuple with only 1 element (i.e. (n,))
- For 2D array, return a shape tuple with only 2 elements (i.e. (n,m))
- For 3D array, return a shape tuple with only 3 elements (i.e. (n,m,k))
- For 4D array, return a shape tuple with only 4 elements (i.e. (n,m,k,j))

Creating a column vector

```
x = np.ones(5)
print(x)
```

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

```
print(x.shape)
```

```
## (5,)
```

```
print(x.shape)
```

```
## (5,)
```

```
x = x.reshape(5, 1)
print(x)
```

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

Creating a column vector

Same as

```
x = np.ones(5).reshape(-1,1)
print(x.shape)
```

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

Creating a row vector

```
x = x.reshape(1,5)
print(x)
```

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

n-d arrays

Create 2d array

```
x = np.array([[1,5,2,3], [5,6,3,3]])
print(x)
```

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

n-d arrays

Is this 2d or 3d array?

```
x = np.array([[[1,5,3,2],[2,2,2,2],[3,5,3,3]]])
```

n-d arrays

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

```
## [[[1 5 3 2]
##   [2 2 2 2]
##   [3 5 3 3]]
##
##  [[1 5 7 2]
##   [1 2 6 2]
##   [1 2 6 2]]]
```

Indexing

```
x = np.array([10,1,1,2,3])
# the fourth element from the array
print(x[3])
```

```
## 2
```

```
# Everything up to the second element
print(x[:2])
```

```
## [10  1]
```

Indexing n-d arrays

```
x = np.arange(21)
print(x)
```

```
## [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```

```
x = x.reshape(3, 7)
print(x)
```

```
## [[ 0  1  2  3  4  5  6]
##  [ 7  8  9 10 11 12 13]
##  [14 15 16 17 18 19 20]]
```

```
# Second element of the first array
print(x[0,1])
```

```
## 1
```

```
# First two rows of the array
print(x[:2,:])
```

```
## [[ 0  1  2  3  4  5  6]
##  [ 7  8  9 10 11 12 13]]
```

Indexing n-d arrays

```
# First two columns of the array
print(x[:,:2])
```

```
## [[ 0  1]
##  [ 7  8]
##  [14 15]]
```

```
# first two rows and first two columns
print(x[:2,:2])
```

```
## [[0 1]
##  [7 8]]
```

```
# Slicing with the range
print(x[:2,2:5])
```

```
## [[ 2  3  4]
##  [ 9 10 11]]
```

Numpy methods

```
x = np.array([10, 1, 1, 2, 3])
print(x.mean())
```

```
## 3.4
```

```
print(x.sum())
```

```
## 17
```

```
print(x.min())
```

```
## 1
```

```
print(x.max())
```

```
## 10
```

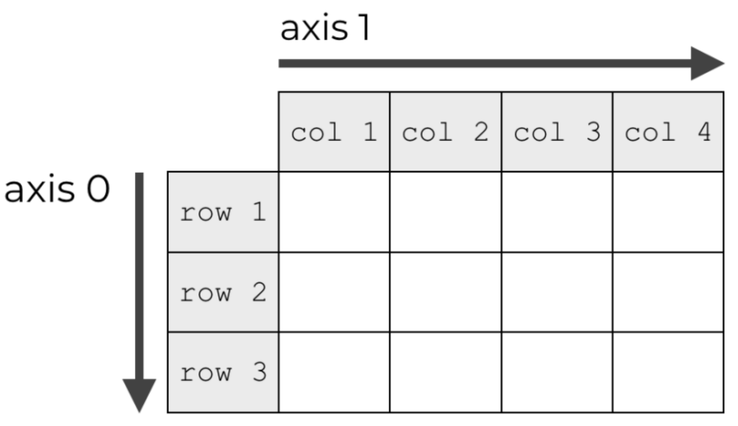
Numpy methods

```python
l1 = [5, 2, 3, 4]
np.mean(l1)
```

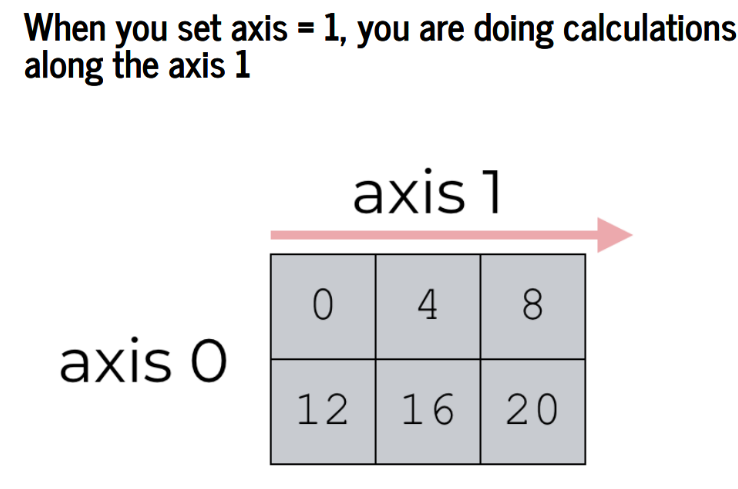
```
## 3.5
```

```python
# l1.mean() this will rais an error
```

Axes in numpy

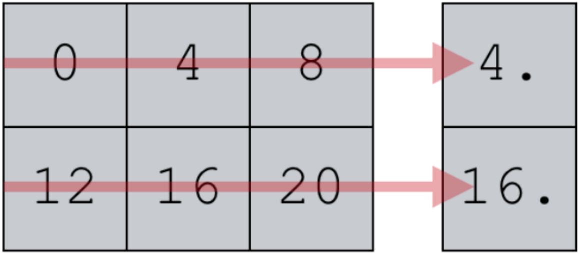


Axes in numpy



Axes in numpy

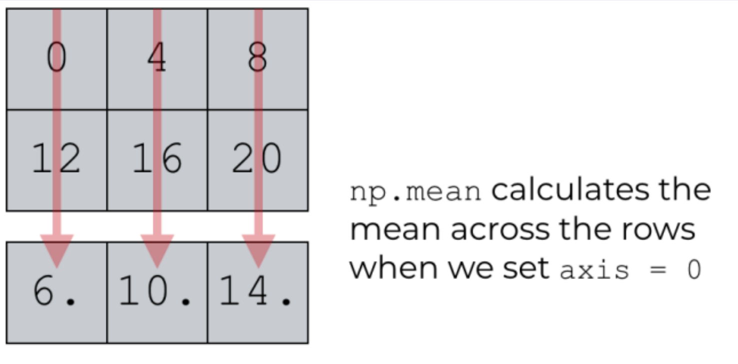
`np.mean` calculates in the column direction when we set `axis = 1`



Axes in numpy

```
print(x.mean(axis = 0))
```

```
## 3.4
```



NumPy basic operations

Generate two random vectors

```
np.random.seed(10)
x = np.random.random(size = 10)
print(x)
```

```
## [0.77132064 0.02075195 0.63364823 0.74880388 0.49850701 0.22479665
##  0.19806286 0.76053071 0.16911084 0.08833981]
```

```
np.random.seed(10)
y = np.random.random(size = 10)
print(y)
```

```
## [0.77132064 0.02075195 0.63364823 0.74880388 0.49850701 0.22479665
##  0.19806286 0.76053071 0.16911084 0.08833981]
```

NumPy basic operations: Element wise operations

```
print(x + y)
```

```
## [1.54264129 0.0415039  1.26729647 1.49760777 0.99701402 0.44959329
##  0.39612573 1.52106142 0.33822167 0.17667963]
```

```
print(2*x)
```

```
## [1.54264129 0.0415039  1.26729647 1.49760777 0.99701402 0.44959329
##  0.39612573 1.52106142 0.33822167 0.17667963]
```

```
print(x*y)
```

```
## [5.94935535e-01 4.30643402e-04 4.01510086e-01 5.60707255e-01
##  2.48509241e-01 5.05335318e-02 3.92288984e-02 5.78406964e-01
##  2.85984750e-02 7.80392277e-03]
```

NumPy basic operations: Dot product

$$\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \bullet \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = a_x.b_x + a_y.b_y + a_z.b_z$$

```
print(y.dot(x))
```

```
## 2.510664551824326
```

```
print(np.dot(x, y))
```

```
## 2.510664551824326
```

NumPy basic operations: Matrix multiplication

```
m1 = np.ones(16).reshape(4,4)
np.random.seed(10)
m2 = np.random.rand(16).reshape(4,4)
```

This is not matrix multiplication

```
print(m1*m2)
```

```
## [[0.77132064 0.02075195 0.63364823 0.74880388]
##  [0.49850701 0.22479665 0.19806286 0.76053071]
##  [0.16911084 0.08833981 0.68535982 0.95339335]
##  [0.00394827 0.51219226 0.81262096 0.61252607]]
```

This is the matrix multiplication

```
print(m1.dot(m2))
```

```
## [[1.44288676 0.84608067 2.32969188 3.07525401]
##  [1.44288676 0.84608067 2.32969188 3.07525401]
##  [1.44288676 0.84608067 2.32969188 3.07525401]
##  [1.44288676 0.84608067 2.32969188 3.07525401]]
```

Concatenating numpy arrays

- np.concatenate()

```
np.random.seed(10)
ar1 = np.random.randn(20)
ones = np.ones(20)
```

```
ar2 = np.array([ar1, ones])
print(ar2)
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]]
```

Concatenating numpy arrays

- np.concatenate()
- np.stack()

```
np.random.seed(10)
ar1 = np.random.randn(20)
ones = np.ones(20)
```

```
ar2 = np.array([ar1, ones])
print(ar2)
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]]
```

Concatenating numpy arrays

- np.concatenate()
- np.stack()
- np.vstack()

```
np.random.seed(10)
ar1 = np.random.randn(20)
ones = np.ones(20)
```

```
ar2 = np.array([ar1, ones])
print(ar2)
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]]
```

Concatenating numpy arrays

- np.concatenate()
- np.stack()
- np.vstack()
- np.hstack()

```
np.random.seed(10)
ar1 = np.random.randn(20)
ones = np.ones(20)
```

```
ar2 = np.array([ar1, ones])
print(ar2)
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]]
```

Concatenating numpy arrays

- np.concatenate()
- np.stack()
- np.vstack()
- np.hstack()
- etc. . .

```
np.random.seed(10)
ar1 = np.random.randn(20)
ones = np.ones(20)
```

```
ar2 = np.array([ar1, ones])
print(ar2)
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]]
```

Vertical stacking

```
print(np.vstack([ar1, ones]))
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]]
```

```
ar3 = np.vstack([ar1, ones, np.zeros(20)])
print(ar3)
```

```
## [[ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##    0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##   -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##    1.484537   -1.07980489]
##  [ 1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.          1.          1.          1.          1.
##    1.          1.        ]
##  [ 0.          0.          0.          0.          0.          0.
```

Horizontal stacking

```
ar4 = np.hstack([ar1, ones])
print(ar4)
```

```
## [ 1.3315865   0.71527897 -1.54540029 -0.00838385  0.62133597 -0.72008556
##   0.26551159  0.10854853  0.00429143 -0.17460021  0.43302619  1.20303737
##  -0.96506567  1.02827408  0.22863013  0.44513761 -1.13660221  0.13513688
##   1.484537   -1.07980489  1.          1.          1.          1.
##   1.          1.          1.          1.          1.          1.
##   1.          1.          1.          1.          1.          1.
##   1.          1.          1.          1.        ]
```

np.concatenate()

This one is similar to np.vstack() and np.hstack()

```
ar1 = ar1.reshape(-1,1)
print(ar1.shape)
```

```
## (20, 1)
```

```
ones = ones.reshape(-1,1)
print(ones.shape)
```

```
## (20, 1)
```

```
print(np.concatenate([ar1, ones], axis = 0).shape)
```

```
## (40, 1)
```

```
print(np.concatenate([ar1, ones], axis = 1).shape)
```

```
## (20, 2)
```

```
print(np.vstack([ar1, ones]).shape)
```

```
## (40, 1)
```