Numpy

[Numpy](http://www.numpy.org/) is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. To use Numpy, we first need to import the numpy package as `import numpy as np`

```
import numpy as np
```

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

```
[1 2]
```

Arrays

A numpy array is a grid of values, **all of the 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. We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

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

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

```
b[0][0]
```

```
1
```

```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)
```

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

```
import numpy as np
b = np.ones((1,2))   # Create an array of all ones
print(b)
```

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

```
b.shape
```

```
(1, 2)
```

```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)
```

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

```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

```
[[0.93581555 0.68974007]
 [0.25245327 0.18313774]]
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. **Numpy tries to guess a datatype when you create an array**, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

You can read all about numpy datatypes in the documentation.

```
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
```

```
print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

Array math

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))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

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

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

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

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Important Note

`*` is elementwise multiplication, not matrix multiplication. **We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.** dot is available both as a function in the numpy module and as an instance method of array objects:

```
import numpy as np
```

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

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

You can also use the @ operator which is equivalent to numpy's dot operator.

```
print(v @ w)
```

```
219
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
[29 67]
[29 67]
[29 67]
```

```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

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

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the [documentation](https://numpy.org/doc/stable/reference/routines.math.html).

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

```
print(x)
print("transpose\n", x.T)
```

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

```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

```
import numpy as np

# example of numpy array
x = np.array([1, 2, 3])
print(x)
```

```
[1 2 3]
```

If x is a vector, then a Python operation such as $s = x + 3$ or $s = \frac{1}{x}$ will output s as a vector of the same size as x.

```
# example of vector operation
x = np.array([1, 2, 3])
print (x + 3)
```

```
[4 5 6]
```

In fact, if $x = (x_1, x_2, \ldots, x_n)$ is a row vector then $np.exp(x)$ will apply the exponential function to every element of x. The output will thus be: $np.exp(x) = (e^{x_1}, e^{x_2}, \ldots, e^{x_n})$

```
import numpy as np

# example of np.exp
x = np.array([1, 2, 3])
print(np.exp(x)) # result is (exp(1), exp(2), exp(3))
```

```
[ 2.71828183  7.3890561  20.08553692]
```

Any time you need more info on a numpy function, we encourage you to look at the official documentation.

What is Pytorch?

PyTorch is a python package built by **Facebook AI Research (FAIR)** that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autograd (*Automatic Gradient Calculation*) system

Why Pytorch?

- **More Pythonic**
  - Flexible
  - Intuitive and cleaner code
  - Easy to learn & debug
  - Dynamic Computation Graph (*network behavior can be changed programmatically at runtime*)
- **More Neural Networkic**
  - Write code as the network works
  - forward/backward

Checking PyTorch version

```
import torch

print(torch.__version__)
```

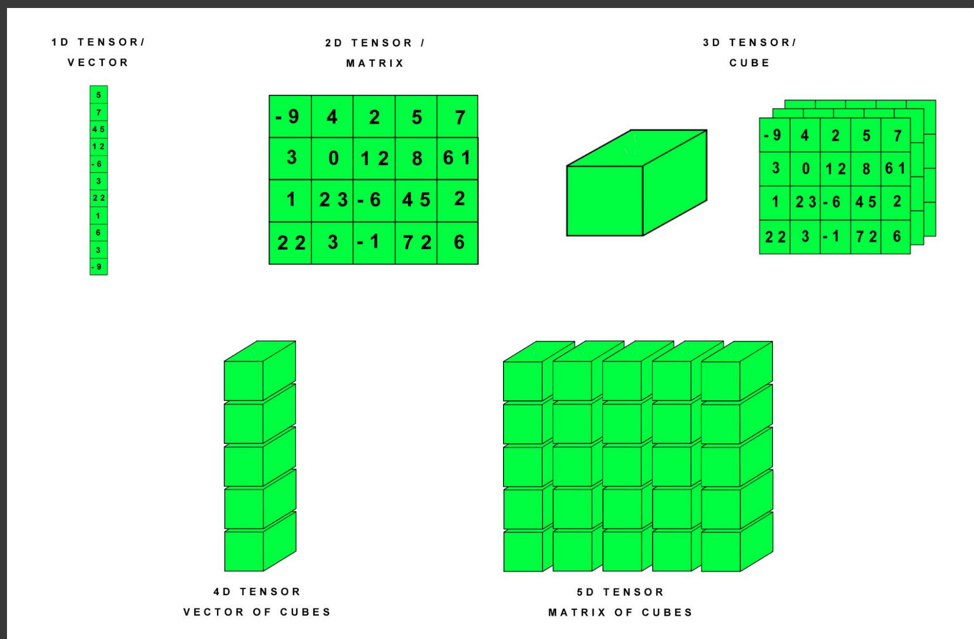
```
2.0.0+cu118
```

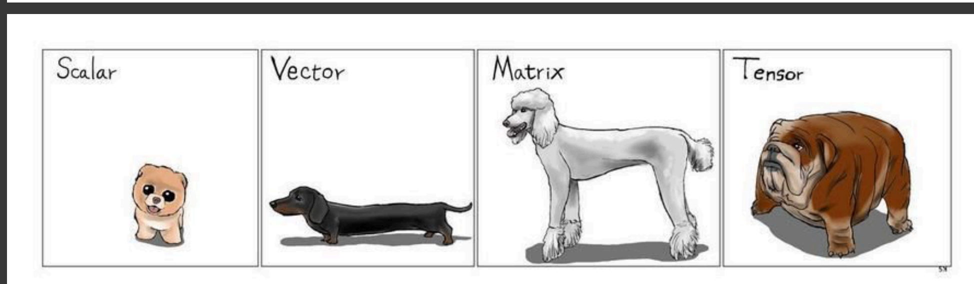
Introduction to Tensors

A **PyTorch Tensor** is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic **n-dimensional array** to be used for arbitrary **numeric computation**.

The biggest difference between a numpy array and a PyTorch Tensor is that a **PyTorch Tensor can run on either CPU or GPU**. To run operations on the GPU, **just cast the Tensor to a cuda datatype**.

A scalar is **zero-order tensor** or rank zero tensor. A vector is a **one-dimensional** or first order tensor, and a matrix is a **two-dimensional** or second order tensor.





A torch.Tensor is a **multi-dimensional matrix** containing elements of a **single data type**.

`torch.Tensor` is an alias for the default tensor type (`torch.FloatTensor`).

```
np.array([1,2,3,4])
torch.tensor([1,2,3,4])
```

```
torch.tensor([[1., -1.], [1., -1.]])
```

```
tensor([[ 1., -1.],
        [ 1., -1.]])
```

```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.7878, 0.7632, 0.5334],
        [0.3148, 0.8141, 0.5708],
        [0.8645, 0.3849, 0.7457],
        [0.5847, 0.7187, 0.6906],
        [0.1597, 0.6442, 0.0510]])
```

```
# Converting numpy arrays to tensors
import numpy as np
torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
# Converting numpy arrays to tensors
np_values = np.array([[1, 2, 3], [4, 5, 6]])

tensor_values = torch.from_numpy(np_values)

print (tensor_values)
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
# A tensor of specific data type can be constructed by passing a torch.dtype

torch.zeros([2, 4], dtype=torch.int32)
```

```
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=torch.int32)
```

```
# The contents of a tensor can be accessed and modified using Python's indexing and slicing notation:
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(x[1][2])

# Modify a certain element
x[0][1] = 8
print(x)
```

```
tensor(6)
tensor([[1, 8, 3],
        [4, 5, 6]])
```

```
# Use torch.Tensor.item() to get a Python number from a tensor containing a single value

x = torch.tensor([[1]])
print (x)

print(x.item())

x = torch.tensor(2.5)

print(x.item())
```

```
tensor([[1]])
1
2.5
```

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(x.size())
```

```
torch.Size([2, 3])
```

```
x.shape
```

```
torch.Size([2, 3])
```

```
import torch
```

```
# Tensor addition & subtraction
x = torch.randint(1, 10, (2, 2))
y = torch.randint(1, 10, (2, 2))
print(x)
print(y)

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

```
tensor([[4, 2],
        [7, 1]])
tensor([[7, 6],
        [1, 5]])
tensor([[11,  8],
        [ 8,  6]])
tensor([[-3, -4],
        [ 6, -4]])
```

```
# Syntax 2 for Tensor addition & subtraction in PyTorch
print(torch.add(x, y))
print(torch.sub(x, y))
```

```
tensor([[1.6467, 1.5640, 1.0048],
        [0.7332, 1.2294, 1.2699],
        [1.1228, 0.8972, 1.4218],
        [0.9923, 0.1450, 1.2395],
        [1.0253, 1.3472, 1.0611]])
tensor([[ 0.3302, -0.2585, -0.0466],
        [ 0.3623,  0.7502, -0.5925],
        [ 0.3077,  0.3040, -0.2434],
        [-0.5304, -0.0994, -0.1986],
        [ 0.5035, -0.0153, -0.1597]])
```

```python
# Tensor Product & Transpose

mat1 = torch.randn(2, 3)
mat2 = torch.randn(3, 3)

print(mat1)
print(mat2)

print(torch.mm(mat1, mat2))

print(mat1.t())
```

```
tensor([[-0.8031,  0.2446,  0.7940],
        [-0.3707,  0.0465,  1.4219]])
tensor([[ 0.3405, -0.5077,  0.0098],
        [ 2.4161,  0.2791, -1.2381],
        [ 0.3947,  0.1022, -0.7730]])
tensor([[ 0.6309,  0.5572, -0.9245],
        [ 0.5475,  0.3465, -1.1604]])
tensor([[-0.8031, -0.3707],
        [ 0.2446,  0.0465],
        [ 0.7940,  1.4219]])
```

```python
# Elementwise multiplication
t = torch.Tensor([[1, 2], [3, 4]])
t.mul(t)
```

```
tensor([[ 1.,  4.],
        [ 9., 16.]])
```

```python
# Shape, dimensions, and datatype of a tensor object

x = torch.rand(5, 3)

print('Tensor shape:', x.shape)   # t.size() gives the same
print('Number of dimensions:', x.dim())
print('Tensor type:', x.type())   # there are other types
```

```
Tensor shape: torch.Size([5, 3])
Number of dimensions: 2
Tensor type: torch.FloatTensor
```

```python
# Slicing
t = torch.Tensor([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],[[1, 2, 3], [4, 5, 6], [7, 8, 9]]])

# Every row, only the last column
print(t[:, -1])

# First 2 rows, all columns
print(t[:2, :])

# Lower right most corner
print(t[-1:, -1:])
```

```
tensor([[7., 8., 9.],
        [7., 8., 9.]])
tensor([[[1., 2., 3.],
         [4., 5., 6.],
```

```
          [7., 8., 9.]],

         [[1., 2., 3.],
          [4., 5., 6.],
          [7., 8., 9.]]]])
tensor([[[[7., 8., 9.]]]])
```

```
print(t[0,-2:-1, :1])
```

```
tensor([[4.]])
```

```
Start coding or generate with AI.
```