L2 Tensors – Multidimensional Arrays

Tensors

- PyTorch is a library for Python programs that facilitates building deep learning projects
- PyTorch is an open-source machine learning library based on the Torch library, developed by Facebook's AI Research lab
- Tensors, the basic data structure in PyTorch
- The PyTorch library primarily supports NVIDIA CUDA-based GPUs

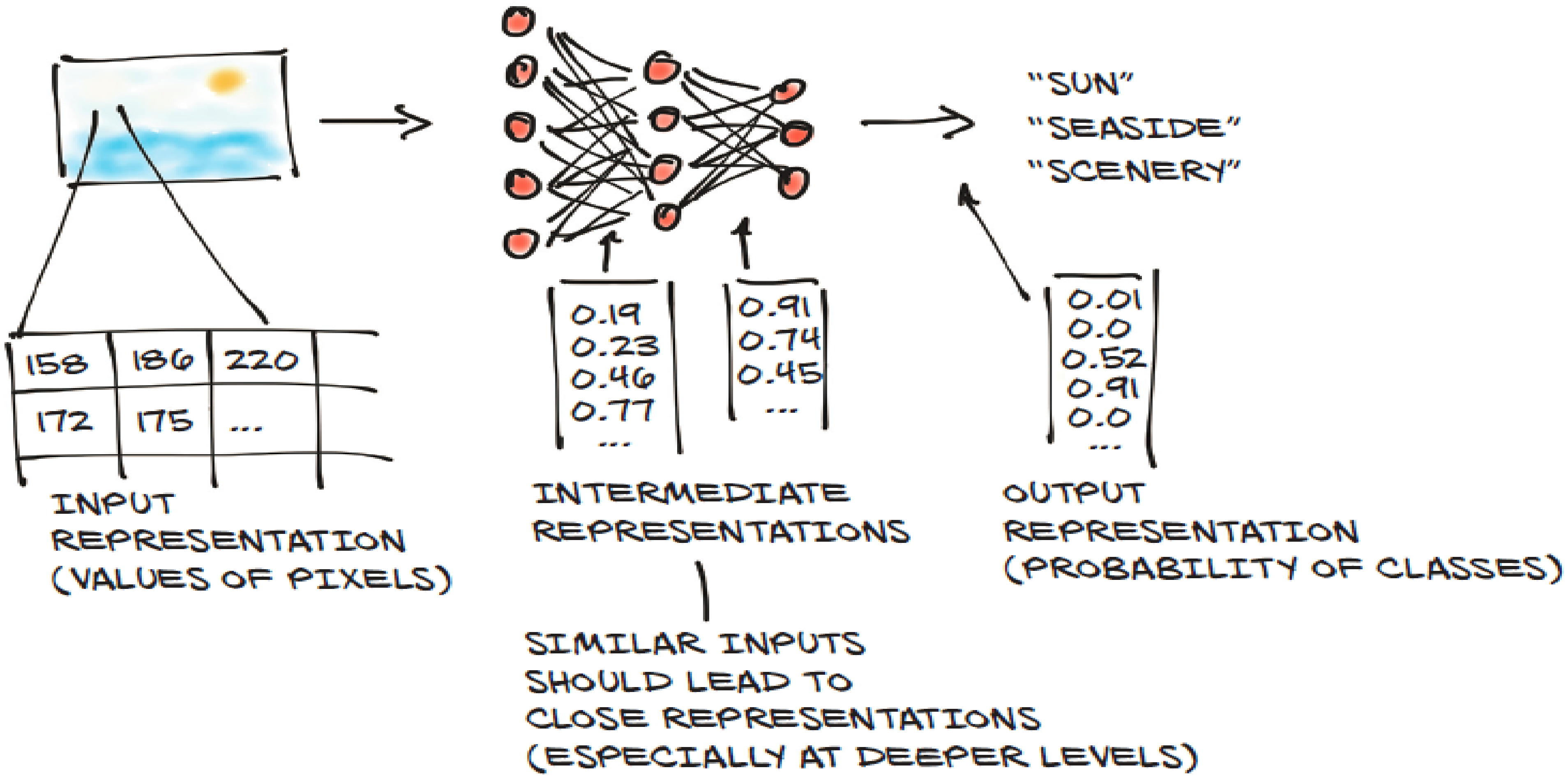
Transforming an input representation to an output representation

- Deep learning consists of building a system that can transform data from one representation to another.
- This transformation is driven by extracting commonalities from a series of examples that demonstrate the desired mapping
- Floating-point numbers are the way a network deals with information
- we need a way to encode real-world data and then decode the output back and use for our purpose
- Hence, we need to deal with all the floating-point numbers in PyTorch by using tensors

Transforming an input representation to an output representation

- A deep neural network typically learns the transformation from one form of data to another in stages
- This means the partially transformed data between each stage can be thought of as a sequence of intermediate representations (2nd step in figure).
- Intermediate representations are the results of combining the input with the weights of the previous layer of neurons.
- Each intermediate representation is unique to the inputs that preceded it
- For image recognition, early representations can be things such as edge detection or certain textures like fur.
- Deeper representations can capture more complex structures like ears, noses, or eyes.
- In general, such intermediate representations are collections of floating-point numbers that characterize the input
- Such characterization is specific to the task at hand and is learned from relevant examples.
- These collections of floating-point numbers and their manipulation are at the heart of modern AI

Transforming an input representation to an output representation

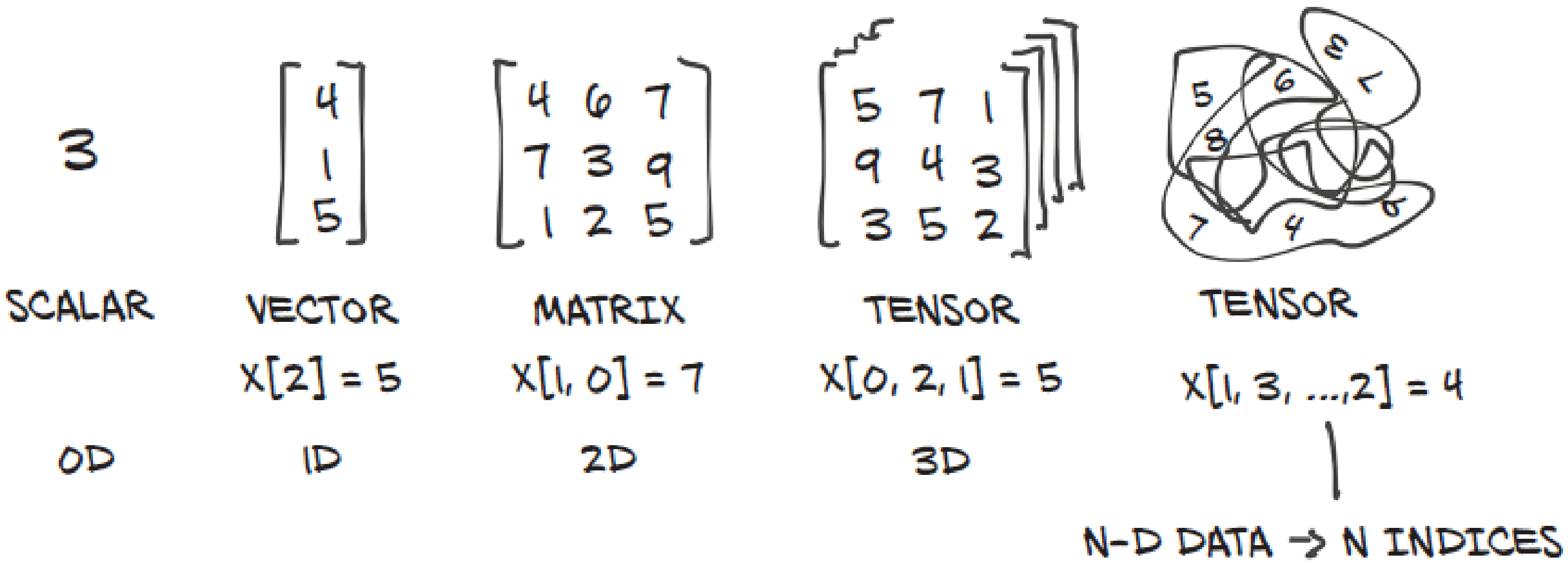


Converting image pixel into floating-point numbers

Tensors

- Before we can begin the process of converting our data to floating-point input
  - We must need to deal with how PyTorch handles and stores data—
  - as input, as intermediate representations, and as output
- PyTorch introduces a fundamental data structure: the tensor
- In the context of deep learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions
- Another name for the same concept is multidimensional array.
- The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor
- NumPy is also popular for multidimensional array library
- Compared to NumPy arrays, PyTorch tensors have a few superpowers
  - ability to perform very fast operations on graphical processing units (GPUs),
  - distribute operations on multiple devices or machines, and
  - keep track of the graph of computations that created them
- PyTorch features seamless interoperability with NumPy, SciPy, Scikit-learn and Pandas

Tensors are the building blocks for representing data in PyTorch



Tensors vs Numpy and Tensors vs multidimensional Arrays of C, C++, Java

- Tensors vs Numpy
- Tensors are multidimensional arrays like n-dimensional NumPy array.
- However, tensors can be used in GPUs as well, which is not in the case of NumPy array
- Tensors vs multidimensional array used in C, C++, and Java
- tensors should have the same size of columns in all dimensions
- Also, the tensors can contain only numeric data types

How to manipulate tensors using the PyTorch tensor library

Creating Tensor in PyTorch

- A tensor can contain elements of a single data type.
- We can create a tensor using a python list or NumPy array.
- The torch has 10 variants of tensors for both GPU and CPU.
- Different ways of defining a tensor.
- torch.tensor() : It copies the data to create a tensor; however, it infers the data type automatically.
- torch.Tensor() : It copies the data and creates its tensor. It is an alias for torch.FloatTensor.
- torch.as_tensor() : The data is shared and not copied in this case while creating the data and accepts any type of array for tensor creation.
- torch.from_numpy() : It is similar to tensor.as_tensor() however it accepts only numpy array.

Creating Tensor in PyTorch

```
import torch
import numpy as np
data1 = [1, 2, 3, 4, 5, 6]
data2 = np.array([1.5, 3.4, 6.8, 9.3, 7.0, 2.8])
 # creating tensors and printing
t1 = torch.tensor(data1)
t2 = torch.Tensor(data1)
t3 = torch.as_tensor(data2)
t4 = torch.from_numpy(data2)
print("Tensor: ",t1, "Data type: ", t1.dtype,"\n")
print("Tensor: ",t2, "Data type: ", t2.dtype,"\n")
print("Tensor: ",t3, "Data type: ", t3.dtype,"\n")
print("Tensor: ",t4, "Data type: ", t4.dtype,"\n")
print("numpy array", data2, "Data type:", data2. dtype)
```

```
Tensor:  tensor([1, 2, 3, 4, 5, 6]) Data type: torch.int64

Tensor:  tensor([1., 2., 3., 4., 5., 6.]) Data type: torch.float32

Tensor:  tensor([1.5000, 3.4000, 6.8000, 9.3000, 7.0000, 2.8000], dtype=torch.float64) Data type: torch.float64

Tensor:  tensor([1.5000, 3.4000, 6.8000, 9.3000, 7.0000, 2.8000], dtype=torch.float64) Data type: torch.float64

numpy array [1.5 3.4 6.8 9.3 7.  2.8] Data type: float64
```

Verify that the numpy array and torch tensor have similar data types which is float64 as shown above

Methods for NumPy to PyTorch and PyTorch to NumPy

- The two main methods for NumPy to PyTorch (and back again) are:

- torch.from_numpy(ndarray) - NumPy array -> PyTorch tensor.
- torch.Tensor.numpy() - PyTorch tensor -> NumPy array

Convert tensor to numpy array - .numpy()

Convert a PyTorch tensor to a Numpy array using the .numpy method of a tensor.

```
# Convert a torch tensor to a numpy array
import numpy as np
npx = np.array([[1, 2],
                [3, 4.]])
ty = torch.from_numpy(npx)
print("npx=", npx)
print("ty=", ty)
print("npx.dtype=",npx.dtype)
print("ty.dtype=", ty.dtype)
npz = ty.numpy()
print("npz=",npz)
```

```
npx= [[1. 2.]
 [3. 4.]]
ty= tensor([[1., 2.],
        [3., 4.]],
dtype=torch.float64)
npx.dtype= float64
ty.dtype= torch.float64
npz= [[1. 2.]
 [3. 4.]]
```

Tensor Attributes

The two fundamental attributes of a tensor are:

- Shape: refers to the dimensionality of array or matrix
- Rank: refers to the number of dimensions present in tensor
  - TENSOR.ndim: Check number of dimensions for TENSOR or
  - len(TENSOR.shape)
- dtype - what datatype are the elements within the tensor stored in?
- device - what device is the tensor stored on? (usually GPU or CPU)

Create a random tensor and find details

```
# Create a tensor
some_tensor = torch.rand(3, 4)

# Find out details about it
print(some_tensor)
print(f"Shape of tensor: {some_tensor.shape}")
print(f"Datatype of tensor: {some_tensor.dtype}")
print(f"Device tensor is stored on: {some_tensor.device}")
```

```
tensor([[0.8549, 0.5509, 0.2868, 0.2063],
        [0.4451, 0.3593, 0.7204, 0.0731],
        [0.9699, 0.1078, 0.8829, 0.4132]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Tensor Attribute - .shape

```
t1 = torch.tensor(4.)
t2 = torch.tensor([1., 2, 3, 4])
t3 = torch.tensor([[5., 6],
              [7, 8],
              [9, 10]])
t4 = torch.tensor([
   [[11, 12, 13],
    [13, 14, 15]],
   [[15, 16, 17],
    [17, 18, 19.]]])
```

```
print(t1)
print(t1.shape)
print(t2)
print(t2.shape)
print(t3)
print(t3.shape)
print(t4)
print(t4.shape)
```

```
tensor(4.)
torch.Size([])
tensor([1., 2., 3., 4.])
torch.Size([4])
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
torch.Size([3, 2])
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
torch.Size([2, 2, 3])
```

Tensor Attribute - .shape

```
import torch
 # creating a tensors
t1=torch.tensor([1, 2, 3, 4])
t2=torch.tensor([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 10, 11, 12]])
 # printing the tensors:
print("Tensor t1: \n", t1)
print("\nTensor t2: \n", t2)
# shape of tensors
print("\nShape of t1: ", t1.shape)
print("Shape of t2: ", t2.shape)
 # rank of tensors
print("\nRank of t1: ", len(t1.shape))
print("Rank of t2: ", len(t2.shape))
```

```
Tensor t1:
 tensor([1, 2, 3, 4])

Tensor t2:
 tensor([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])

Shape of t1:  torch.Size([4])
Shape of t2:  torch.Size([3, 4])

Rank of t1:  1
Rank of t2:  2
```

Restructuring Tensors in Pytorch

- We can modify the shape and size of a tensor as desired in PyTorch.
- We can also create a transpose of an n-d tensor.
- Three common ways to change the structure of tensor :
- .reshape(a, b) : returns a new tensor with size a,b
- .resize(a, b) : returns the same tensor with the size a,b
- .transpose(a, b) : returns a tensor transposed in a and b dimension
- transposes in PyTorch use either:
  - torch.transpose(input, dim0, dim1) - where input is the desired tensor to transpose and dim0 and dim1 are the dimensions to be swapped.
  - tensor.T - where tensor is the desired tensor to transpose.

Restructuring Tensors in Pytorch

Transpose Parameters

- input (Tensor) – the input tensor.
- dim0 (int) – the first dimension to be transposed
- dim1 (int) – the second dimension to be transposed
- transpose(0, 1) (or transpose(): defaults to swapping the first two dimensions)
- for a 2D tensor, transpose(0, 1) and transpose(1, 0) have the same effect—they transpose the matrix. The difference is with tensors of higher dimensions

```
x = torch.randn(2, 3)
print(x)
torch.transpose(x, 0, 1)
```

```
tensor([[ 0.0094,  0.3476,  0.5798],
        [ 0.5781,  0.6393, -1.6365]])
tensor([[ 0.0094,  0.5781],
        [ 0.3476,  0.6393],
        [ 0.5798, -1.6365]])
```

Restructuring Tensors in Pytorch

```
import torch
 # defining tensor
t = torch.tensor([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
 # reshaping the tensor
print("Reshaping")
print(t.reshape(6, 2))
 # resizing the tensor
print("\nResizing")
print(t.resize_(2, 6))
 # transposing the tensor
print("\nTransposing")
print(t.transpose(1, 0))
```

```
Reshaping
tensor([[ 1,  2],
        [ 3,  4],
        [ 5,  6],
        [ 7,  8],
        [ 9, 10],
        [11, 12]])

Resizing
tensor([[ 1,  2,  3,  4,  5,  6],
        [ 7,  8,  9, 10, 11, 12]])

Transposing
tensor([[ 1,  7],
        [ 2,  8],
        [ 3,  9],
        [ 4, 10],
        [ 5, 11],
        [ 6, 12]])
```

Restructuring Tensors in Pytorch - reshape vs resize

```
import torch
original_tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])
# Reshape example
new_shape = (3, 2)
reshaped_tensor = original_tensor.reshape(new_shape)
print("Original Tensor:")
print(original_tensor)
print("Reshaped Tensor:")
print(reshaped_tensor)
print()
# Resize example (inplace)
new_size = (3, 2)
original_tensor.resize_(new_size)
print("Resized Tensor (inplace):")
print(original_tensor)
print()
# Transpose example
transposed_tensor = original_tensor.transpose(0, 1)
print("Original Tensor:")
print(original_tensor)
print("Transposed Tensor:")
print(transposed_tensor)
```

```
Original Tensor:
tensor([[1, 2, 3],
        [4, 5, 6]])
Reshaped Tensor:
tensor([[1, 2],
        [3, 4],
        [5, 6]])

Resized Tensor (inplace):
tensor([[1, 2],
        [3, 4],
        [5, 6]])

Original Tensor:
tensor([[1, 2],
        [3, 4],
        [5, 6]])
Transposed Tensor:
tensor([[1, 3, 5],
        [2, 4, 6]])
```

Reshape vs resize

- reshape
- The reshape operation is used to change the shape of the tensor. In this example, the original tensor is reshaped from a 2x3 matrix to a 3x2 matrix.
- The reshape operation is used to change the shape of a tensor while keeping the same underlying data. It creates a new view of the original tensor with the specified shape.
- resize (Inplace):
- The resize_ operation is an inplace operation used to change the size of the tensor. Here, the original tensor is resized to a new size of 3x2.
- The resize_ method is the inplace version that modifies the original tensor. It's recommended to use inplace operations to avoid deprecated warnings
- transpose
- The transpose operation is used to swap dimensions of the tensor. There is no inplace version of transpose. It always returns a new tensor

Mathematical Operations on Tensors in PyTorch

We can perform various mathematical operations on tensors using Pytorch similar to NumPy arrays

Also works with operators - +, - *, /

```
import torch
# defining two tensors
t1 = torch.tensor([1, 2, 3, 4])
t2 = torch.tensor([5, 6, 7, 8])
 # adding two tensors
print("tensor2 + tensor1")
print(torch.add(t2, t1))
# subtracting two tensor
print("\ntensor2 - tensor1")
print(torch.sub(t2, t1))
 # multiplying two tensors
print("\ntensor2 * tensor1")
print(torch.mul(t2, t1))
 # diving two tensors
print("\ntensor2 / tensor1")
print(torch.div(t2, t1))
```

```
tensor2 + tensor1
tensor([ 6,  8, 10, 12])

tensor2 - tensor1
tensor([4, 4, 4, 4])

tensor2 * tensor1
tensor([ 5, 12, 21, 32])

tensor2 / tensor1
tensor([5.0000, 3.0000, 2.3333, 2.0000])
```

From Python lists to PyTorch tensors

- List indexing vs Tensor indexing
- List of three numbers in Python

```
[4]: a = [1.0, 2.0, 1.0]
     print(a[0])
     a[2] = 3.0
     a

     1.0

[4]: [1.0, 2.0, 3.0]
```

Import the torch module

Creates a one-dimensional tensor of size 3 filled with 1s

```
import torch
a = torch.ones(3)
print(a)
print(a[1])
print(float(a[1]))
a[2] = 2.0
print(a)
```

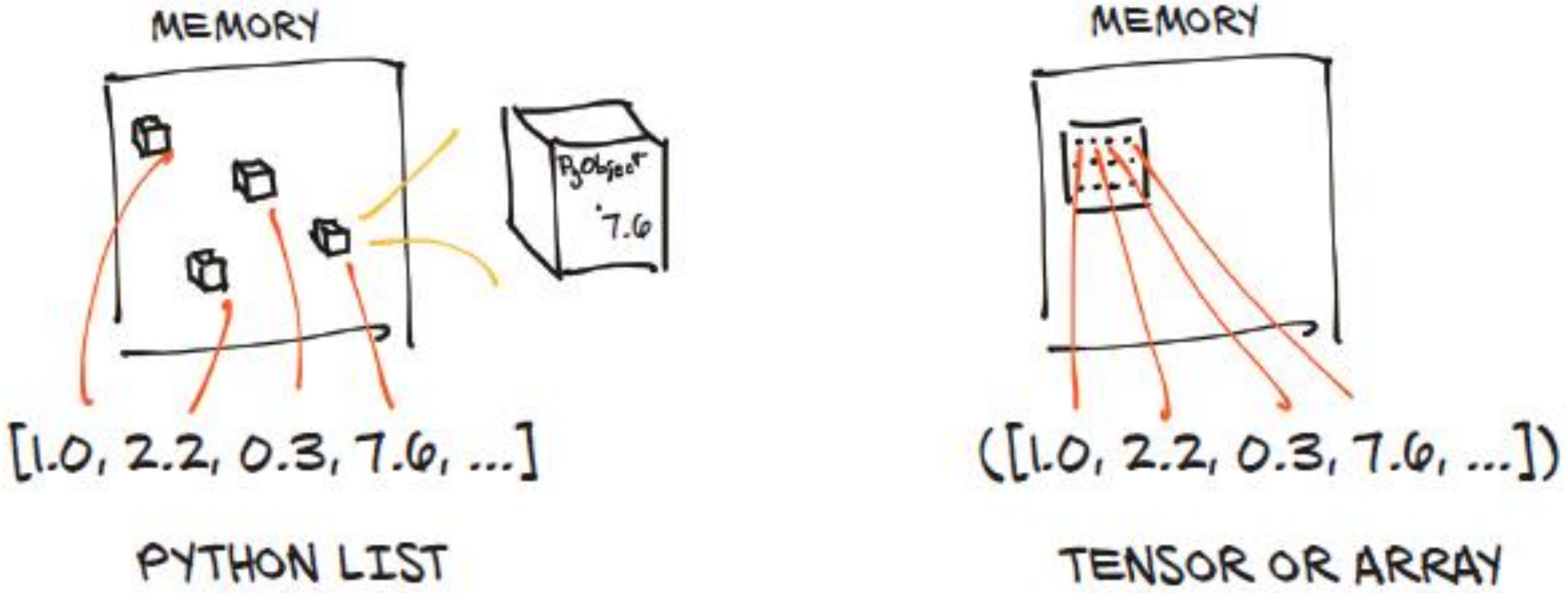
```
tensor([1., 1., 1.])
tensor(1.)
1.0
tensor([1., 1., 2.])
```

How tensors are different from a list ? The essence of tensors

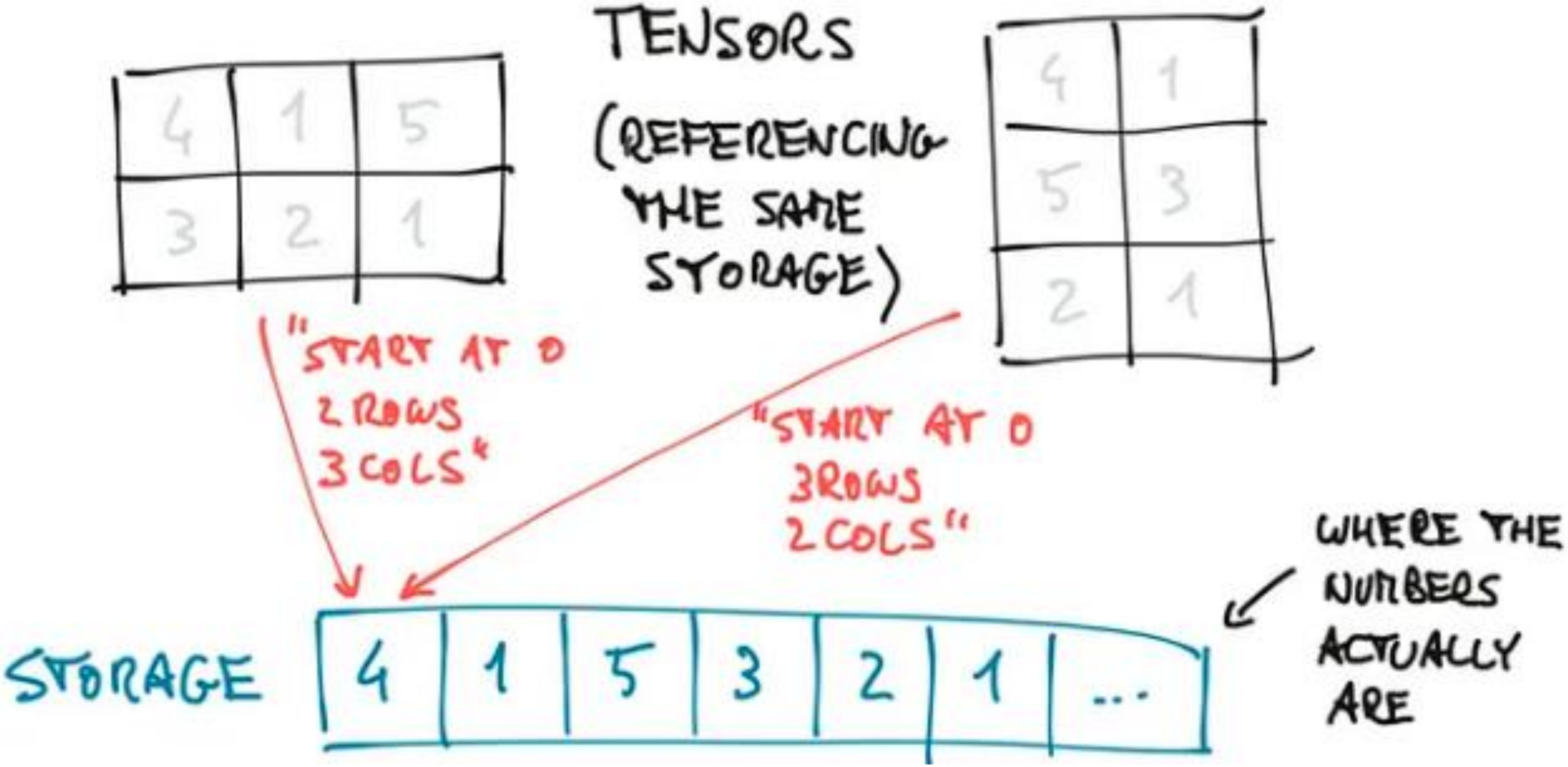
The essence of tensors

- Python lists or tuples of numbers are collections of Python objects that are individually allocated in memory (left side figure)
- PyTorch tensors or NumPy arrays, on the other hand, are views over (typically) contiguous memory blocks containing unboxed C numeric types rather than Python objects.
- Each element is a 32-bit (4-byte) float in this case (right side of figure)
- This means storing a 1D tensor of 1,000,000 float numbers will require exactly 4,000,000 contiguous bytes, plus a small overhead for the metadata (such as dimensions and numeric type).

The essence of tensors



Tensors are views over a Storage instance



Matrix Multiplication using PyTorch

- The methods in PyTorch expect the inputs to be a Tensor
- matrix multiplication methods:

1. torch.mm()
2. torch.matmul()
3. torch.bmm()
4. @ operator

torch.mm is a shortcut for matmul

Matrix multiplication using transpose

```
# Perform matmul on tensor_A and tensor_B
tensor_A = torch.rand(size=(2,3)).to(device)
tensor_B = torch.rand(size=(2,3)).to(device)
print("tensor_A=", tensor_A)
print("tensor_B=", tensor_B)
# tensor_C = torch.matmul(tensor_A, tensor_B) # won't work because of shape error
print("tensor_B.T=", tensor_B.T)
tensor_C = torch.matmul(tensor_A, tensor_B.T)
print("tensor_C=", tensor_C, tensor_C.shape)
```

Matrix multiplication using transpose

```
tensor_A= tensor([[0.0108, 0.9455, 0.7661],
        [0.2634, 0.1880, 0.5174]])
tensor_B= tensor([[0.7849, 0.1412, 0.3112],
        [0.7091, 0.1775, 0.4443]])
tensor_B.T= tensor([[0.7849, 0.7091],
        [0.1412, 0.1775],
        [0.3112, 0.4443]])
tensor_C= tensor([[0.3803, 0.5159],
        [0.3943, 0.4501]]) torch.Size([2, 2])
```

Matrix Multiplication using PyTorch - torch.mm()

- Computes matrix multiplication by taking an m×n Tensor and an n×p Tensor.
- It can deal with only two-dimensional matrices and not with single-dimensional ones.
- This function does not support broadcasting.
- Broadcasting is the way the Tensors are treated when their shapes are different.
- The smaller Tensor is broadcasted to suit the shape of the wider or larger Tensor for operations.
- torch.mm(Tensor_1, Tensor_2, out=None)
- The parameters are two Tensors and the third one is an optional argument. Another Tensor to hold the output values can be given there.

Matrix Multiplication using PyTorch - torch.mm()

Ex1: Same dimensions

```
import torch
mat_1 = torch.tensor([[1, 2, 3],
                      [4, 3, 8],
                      [1, 7, 2]])
mat_2 = torch.tensor([[2, 4, 1],
                      [1, 3, 6],
                      [2, 6, 5]])
torch.mm(mat_1, mat_2, out=None)
```

Ex2: tensor_1 is of 2×2 dimension, tensor_2 is of 2×3 dimension.
So the output will be of 2×3

```
import torch
mat_1 = torch.tensor([[1, 2], [4, 3]])
mat_2 = torch.tensor([[2, 4, 1],  [1, 3, 6]])
torch.mm(mat_1, mat_2, out=None)
```

Matrix Multiplication using PyTorch - torch.mm()

Ex1: Same dimensions

```
import torch
mat_1 = torch.tensor([[1, 2, 3],
                      [4, 3, 8],
                      [1, 7, 2]])
mat_2 = torch.tensor([[2, 4, 1],
                      [1, 3, 6],
                      [2, 6, 5]])
torch.mm(mat_1, mat_2, out=None)
```

```
tensor([[10, 28, 28],
        [27, 73, 62],
        [13, 37, 53]])
```

Ex2: tensor_1 is of 2×2 dimension, tensor_2 is of 2×3 dimension. So the output will be of 2×3

```
import torch
mat_1 = torch.tensor([[1, 2], [4, 3]])
mat_2 = torch.tensor([[2, 4, 1],  [1, 3, 6]])
torch.mm(mat_1, mat_2, out=None)
```

```
tensor([[ 4, 10, 13],
        [11, 25, 22]])
```

Matrix Multiplication using PyTorch - torch.matmul()

- Allows the computation of multiplication of single-dimensional matrices, 2D matrices and mixed ones also.
- This method also supports broadcasting and batch operations.
- Depending upon the input matrices dimensions, the operation to be done is decided.
- The general syntax is given below.
- torch.matmul(Tensor_1, Tensor_2, out=None)

Matrix Multiplication using PyTorch - torch.matmul()

Various possible dimensions of the arguments and the operations

argument_1	**argument_2**	**Action taken**
1-dimensional	1-dimensional	The dot product (scalar) is calculated
2-dimensional	2-dimensional	General matrix multiplication is done
1-dimensional	2-dimensional	The tensor-1 is prepended with a ‘1’ to match dimension of tensor-2 a 1 for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
2-dimensional	1-dimensional	Matrix-vector product is calculated
1/N-dimensional (N>2)	1/N-dimensional (N>2)	Batched matrix multiplication is done

Matrix Multiplication using PyTorch - torch.matmul()

Ex1: Arguments of the same dimension

```
import torch
vec_1 = torch.tensor([3, 6, 2])
vec_2 = torch.tensor([4, 1, 9])
print("Single dimensional tensors :", torch.matmul(vec_1, vec_2))
# both arguments 2D
mat_1 = torch.tensor([[1, 2, 3],
                      [4, 3, 8],
                      [1, 7, 2]])
  mat_2 = torch.tensor([[2, 4, 1],
                      [1, 3, 6],
                      [2, 6, 5]])
out = torch.matmul(mat_1, mat_2)
print("\n3x3 dimensional tensors :\n", out)
```

Matrix Multiplication using PyTorch - torch.matmul()

Ex1: Arguments of the same dimension

```
import torch
vec_1 = torch.tensor([3, 6, 2])
vec_2 = torch.tensor([4, 1, 9])
print("Single dimensional tensors :", torch.matmul(vec_1, vec_2))
# both arguments 2D
mat_1 = torch.tensor([[1, 2, 3],
                      [4, 3, 8],
                      [1, 7, 2]])
  mat_2 = torch.tensor([[2, 4, 1],
                      [1, 3, 6],
                      [2, 6, 5]])
out = torch.matmul(mat_1, mat_2)
print("\n3x3 dimensional tensors :\n", out)
```

3*4+6*1+2*9=36

Single dimensional tensors : tensor(36)

3x3 dimensional tensors :

```
 tensor([[10, 28, 28],
        [27, 73, 62],
        [13, 37, 53]])
```

Matrix Multiplication using PyTorch - torch.matmul()

Ex1: Arguments of different dimensions

```
import torch
# first argument 1D and second argument 2D
mat1_1 = torch.tensor([3, 6, 2])
mat1_2 = torch.tensor([[1, 2, 3],
                       [4, 3, 8],
                       [1, 7, 2]])
out_1 = torch.matmul(mat1_1, mat1_2)
print("\n1D-2D multiplication :\n", out_1)
# first argument 2D and second argument 1D
mat2_1 = torch.tensor([[2, 4, 1],
                       [1, 3, 6],
                       [2, 6, 5]])
mat2_2 = torch.tensor([4, 1, 9])
# assigning to output tensor
out_2 = torch.matmul(mat2_1, mat2_2)
print("\n2D-1D multiplication :\n", out_2)
```

Matrix Multiplication using PyTorch - torch.matmul()

Ex1: Arguments of different dimensions

```
import torch
# first argument 1D and second argument 2D
mat1_1 = torch.tensor([3, 6, 2])
mat1_2 = torch.tensor([[1, 2, 3],
                       [4, 3, 8],
                       [1, 7, 2]])
out_1 = torch.matmul(mat1_1, mat1_2)
print("\n1D-2D multiplication :\n", out_1)
# first argument 2D and second argument 1D
mat2_1 = torch.tensor([[2, 4, 1],
                       [1, 3, 6],
                       [2, 6, 5]])
mat2_2 = torch.tensor([4, 1, 9])
# assigning to output tensor
out_2 = torch.matmul(mat2_1, mat2_2)
print("\n2D-1D multiplication :\n", out_2)
```

```
1D-2D multiplication :
 tensor([29, 38, 61])

2D-1D multiplication :
 tensor([21, 61, 59])
```

Matrix Multiplication using PyTorch - torch.matmul()

Ex3: N-dimensional argument (N>2)

```
import torch
# creating Tensors using randn()
mat_1 = torch.randn(2, 3, 3)
mat_2 = torch.randn(3)

# printing the matrices
print("matrix A :\n", mat_1)
print("\nmatrix B :\n", mat_2)

# output
print("\nOutput :\n", torch.matmul(mat_1, mat_2))
```

Matrix Multiplication using PyTorch - torch.matmul()

Ex3: N-dimensional argument (N>2)

```
import torch
# creating Tensors using randn()
mat_1 = torch.randn(2, 3, 3)
mat_2 = torch.randn(3)

# printing the matrices
print("matrix A :\n", mat_1)
print("\nmatrix B :\n", mat_2)

# output
print("\nOutput :\n", torch.matmul(mat_1, mat_2))
```

```
matrix A :
 tensor([[[ 0.3437, -0.1045,  0.2069],
         [-0.7087,  3.0298,  1.8346],
         [ 1.0761, -0.2179, -0.0404]],

        [[ 0.9028,  0.6267, -0.6288],
         [-0.3468, -0.3376,  1.8786],
         [-0.9405, -0.8161,  0.2485]]])

matrix B :
 tensor([-0.6310, -0.3815, -0.3336])

Output :
 tensor([[-0.2460, -1.3208, -0.5824],
        [-0.5990, -0.2790,  0.8220]])
```

Matrix Multiplication using PyTorch - torch.bmm()

- provides batched matrix multiplication for the cases where both the matrices to be multiplied are of only 3-Dimensions (x×y×z) and the first dimension (x) of both the matrices must be same.
- This does not support broadcasting. The syntax is as given below.
- torch.bmm(Tensor_1, Tensor_2, deterministic=false, out=None)
- The “deterministic” parameter takes up boolean value. A ‘false‘ does a faster calculation which is non-deterministic.
- A ‘true‘ does a slower calculation however, it is deterministic.

Matrix Multiplication using PyTorch - torch.bmm()

Ex: the matrix_1 is of dimension 2×3×3. The second matrix is of dimension 2×3×4.

```
import torch
 # 3D matrices
mat_1 = torch.randn(2, 3, 3)
mat_2 = torch.randn(2, 3, 4)

print("matrix A :\n",mat_1)
print("\nmatrix B :\n",mat_2)

print("\nOutput :\n",torch.bmm(mat_1,mat_2))
```

Matrix Multiplication using PyTorch - torch.bmm()

matrix A :

```
tensor([[[ 0.8639,  1.6221,  0.1931],
         [ 2.3902,  0.3274, -1.7375],
         [ 0.6995, -0.2053, -0.5686]],

        [[-0.9331, -0.3916, -0.8546],
         [-0.5468, -1.8374, -0.3086],
         [-2.2238, -1.2308, -1.0526]]])
```

matrix B :

```
tensor([[[-7.7382e-02,  5.3086e-01, -1.6793e+00, -
2.2021e+00],
         [ 1.1075e+00, -6.5119e-01,  8.2038e-04,  1.1264e-01],
         [-4.5405e-01,  6.0790e-01, -4.1423e-01, -3.0507e-01]],

        [[ 1.1997e+00, -1.0194e+00,  4.8544e-02,  6.8989e-01],
         [ 3.3041e-01, -9.4842e-01, -1.0319e+00, -5.3241e-01],
         [-5.0360e-01,  4.0240e-01, -8.7856e-02,  1.1704e-01]]])
```

Output :

```
tensor([[[ 1.6419e+00, -4.8024e-01, -1.5295e+00, -1.7787e+00],
         [ 9.6655e-01, -5.3465e-04, -3.2939e+00, -4.6965e+00],
         [-2.3310e-02,  1.5936e-01, -9.3928e-01, -1.3900e+00]],

        [[-8.1845e-01,  9.7871e-01,  4.3389e-01, -5.3530e-01],
         [-1.1076e+00,  2.1758e+00,  1.8967e+00,  5.6492e-01],
         [-2.5444e+00,  3.0107e+00,  1.2547e+00, -1.0021e+00]]])
```

Matrix Multiplication using PyTorch - @ operator

- @ operator:
- The @ – Simon H operator, when applied on matrices performs multiplication element-wise on 1D matrices and
- normal matrix multiplication on 2D matrices.
- If both the matrices have the same dimension, then the matrix multiplication is carried out normally without any broadcasting/prepending.
- If any one of the matrices is of a different dimension, then appropriate broadcasting is carried out first and then the multiplication is carried out.
- This operator applies to N-Dimensional matrices also

Matrix Multiplication using PyTorch - @ operator

```
# single dimensional matrices
oneD_1 = torch.tensor([3, 6, 2])
oneD_2 = torch.tensor([4, 1, 9])
# two dimensional matrices
twoD_1 = torch.tensor([[1, 2, 3],
                       [4, 3, 8],
                       [1, 7, 2]])
twoD_2 = torch.tensor([[2, 4, 1],
                       [1, 3, 6],
                       [2, 6, 5]])
  # N-dimensional matrices (N>2)
  # 2x3x3 dimensional matrix
ND_1 = torch.tensor([[[-0.0135, -0.9197, -0.3395],
                      [-1.0369, -1.3242,  1.4799],
                      [-0.0182, -1.2917,  0.6575]],

                     [[-0.3585, -0.0478,  0.4674],
                      [-0.6688, -0.9217, -1.2612],
                      [1.6323, -0.0640,  0.4357]]])
```

```
# 2x3x4 dimensional matrix
ND_2 = torch.tensor([[[0.2431, -0.1044, -0.1437, -1.4982],
                      [-1.4318, -0.2510,  1.6247,  0.5623],
                      [1.5265, -0.8568, -2.1125, -0.9463]],

                     [[0.0182,  0.5207,  1.2890, -1.3232],
                      [-0.2275, -0.8006, -0.6909, -1.0108],
                      [1.3881, -0.0327, -1.4890, -0.5550]]])

print("1D matrices output :\n", oneD_1 @ oneD_2)
print("\n2D matrices output :\n", twoD_1 @ twoD_2)
print("\nN-D matrices output :\n", ND_1 @ ND_2)
print("\n Mixed matrices output :\n", oneD_1 @ twoD_1 @
twoD_2)
```

Matrix Multiplication using PyTorch - @ operator

```
1D matrices output :
 tensor(36)
2D matrices output :
 tensor([[10, 28, 28],
        [27, 73, 62],
        [13, 37, 53]])
N-D matrices output :
 tensor([[[ 0.7953,  0.5231, -0.7751, -0.1757],
         [ 3.9030, -0.8274, -5.1287, -0.5915],
         [ 2.8487, -0.2372, -3.4850, -1.3212]],

        [[ 0.6531, -0.1637, -1.1250,  0.2633],
         [-1.5532,  0.4309,  1.6526,  2.5166],
         [ 0.6491,  0.8869,  1.4995, -2.3370]]])
 Mixed matrices output :
 tensor([218, 596, 562])
```