Working with tensors

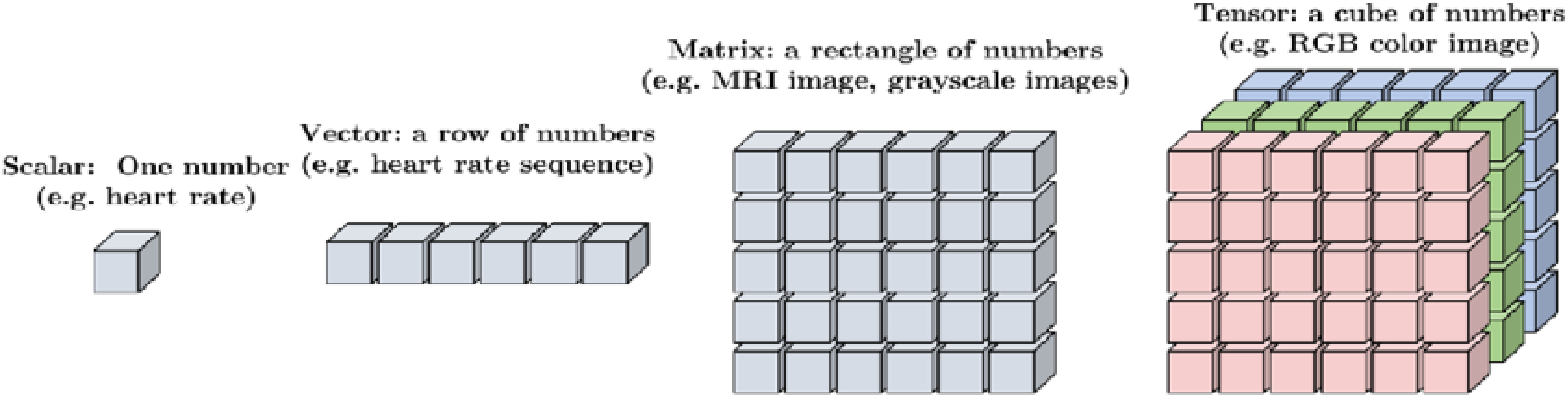
PyTorch

הטכניון
היחידה ללימודי המשך
ולימודי חוץ ע״ש עזריאלי

What is a tensor

- A **tensor** is a multi-dimensional array that stores data. It can represent data of various dimensions, enabling efficient computations in machine learning models.
- Tensors are the core data structure in frameworks like TensorFlow and PyTorch.

- **Rank 0:** Scalar – single value
- **Rank 1:** Vector – list of values
- **Rank 2:** Matrix – grid of values
- **Rank n:** Tensor – multi-dimensional array



Tensor attributes:

shape, size(), ndim, dtype, item(), numel().

```
t = torch.tensor([[1,2],[3,4],[5,6],
                  [10,20],[30,40],[50,60]])
print(f"dtype: {t.dtype}")
print(f"shape {t.shape}")
print(f"ndim {t.ndim}")
print (f"numel {t.numel()}")
```

```
dtype: torch.int64
shape torch.Size([6, 2])
ndim 2
numel 12
```

```
t[0][0]
```

```
tensor(1)
```

Slicing: Access tensor elements using square brackets ([]).

```
t[0]
```

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

Tensor Creation

Create tensors manually or with the following functions:

torch.rand(),

torch.randint(),

torch.ones(),

torch.zeros(),

torch.arange().

Use random values, range, or specific

initialization when creating tensors.

```
t2 = torch.arange(0,100,10)
t2
```

#(start end step)

```
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

```
tensor0 = torch.zeros(3,2)
print(f"tensor0 {tensor0}")

tensor1 = torch.ones(3,2)
print(f"tensor1 {tensor1}")

tensor_rand = torch.rand(3,2)
print(f"tensor_rand {tensor_rand}")

tensor_randint = torch.randint(1,10,(3,2))
print(f"tensor_randint {tensor_randint}")
```

Tensor Shape

Number Range

```
tensor0 tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
tensor1 tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
tensor_rand tensor([[0.5915, 0.6311],
        [0.4835, 0.1594],
        [0.4164, 0.2283]])
tensor_randint tensor([[6, 5],
        [6, 3],
        [3, 2]])
```

Tensor Operations

Stacking and Repeating:

Stack()

- Must be with equal shaped tensors
- dim = 0/ default will stack them as "lines" on top of the other
- dim =1 will stack them as "columns, one next to the other

```
t3= torch.stack([t2]*4)
t3
```

```
tensor([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90]])
```

```
t22 = torch.arange(100,0,-10)
t3 = torch.stack([t2,t22])
t3
```

```
tensor([[  0,  10,  20,  30,  40,  50,  60,  70,  80,  90],
        [100,  90,  80,  70,  60,  50,  40,  30,  20,  10]])
```

```
t22 = torch.arange(100,0,-10)
t3 = torch.stack([t2,t22], dim = 1)
t3
```

```
tensor([[  0, 100],
        [ 10,  90],
        [ 20,  80],
        [ 30,  70],
        [ 40,  60],
        [ 50,  50],
        [ 60,  40],
```

Tensor Operations

Stacking and Repeating:

repeat().

It is possible to use a variety of shapes to determine the dimension of the repetition

Repeate 3 times along the first and only dimension

```
t3 = t2.repeat(3)
t3
```

```
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90,  0, 10, 20, 30, 40, 50, 60, 70,
        80, 90,  0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Repeated 3 sperate times, "pack" the 3 different times in 1 new dimension

```
t3 = t2.repeat(3,1)
t3
```

```
tensor([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90]])
```

Repeated 2 times along the original dimension, make that happen 3 times

```
t3 = t2.repeat(3,2)
t3
```

```
tensor([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90,  0, 10, 20, 30, 40, 50, 60, 70,
         80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90,  0, 10, 20, 30, 40, 50, 60, 70,
         80, 90],
        [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90,  0, 10, 20, 30, 40, 50, 60, 70,
         80, 90]])
```

Shape manipulation: Will only change the shape/dimensions

Will not change/add/remove data

squeeze()

- reduces singleton dimensions.
- by default, will reduce all singleton dimensions
- can accept dim =.. To chose what singleton dimension to reduce
- if there aren't any singletons or if the requested dimension is not a singleton will do nothing, there will be no error returned

```
t.shape
torch.Size([3, 1, 2, 1])
t.squeeze().shape
torch.Size([3, 2])
```

```
t.shape
torch.Size([3, 1, 2, 1])
t.squeeze(dim = 3).shape
torch.Size([3, 1, 2])
```

unsqueeze()

- adds a singleton dimension
- must receive a position to add the dimension to.

```
t.shape
torch.Size([3, 2])
t.unsqueeze(dim=0).shape
torch.Size([1, 3, 2])
```

TECHNION Azrieli Continuing Education and External Studies Division

Shape manipulation:

Both will change the shape and the data of the tensor.

view requires contagious data.

reshape(),

view(),

```
t
```

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

```
t.reshape(6,2)
```

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

```
t.view(6,2)
```

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

Shape manipulation:

permute() – changes the dimension order by dimension index. It does not create new data- just changes the view of it.

```
t.shape
```

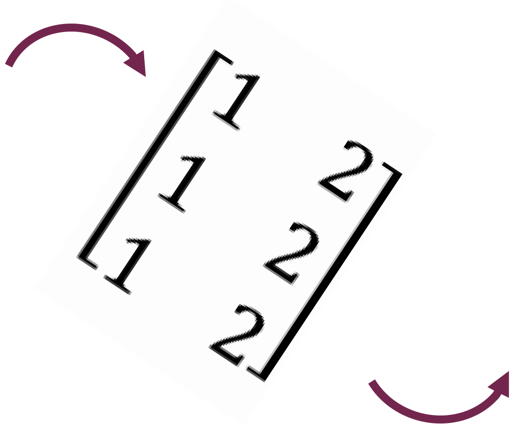
```
torch.Size([3, 4])
```

```
t.permute(1,0).shape
```

```
torch.Size([4, 3])
```

.T (transpose) – it's python thing… we already know it.

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \end{bmatrix}$$



$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

Always create a new tensor	←Could go ether way→	Always “look” on the existing data in a different way, but not recreating data
Stack()	reshape	Squeeze()
Repeat()	permute	Unsqueeze()
		View()
		.T

Arithmetic Operations

Basic arithmetic operations on tensors (+, -, *, /).

Common tensor operations:

mean(), max(), min(), sum(), std().

Convert data type to float using .to(torch.float).

Use the dim parameter in operations like sum(dim=...) to specify which axis to reduce.

Matrix multiplication:

matmul() (or @ operator for matrix operations).

Gradient Calculation

Working with gradients for optimization:

Use requires_grad=True to enable gradient tracking on a tensor.

Define a function, apply backward() to calculate gradients, and access them using .grad.

```
x = torch.tensor(2.0, requires_grad=True)
y = x**2 + 3 * x + 5
y.backward()  # Compute gradients
print(x.grad)  # Output: gradient of y with respect to x
```