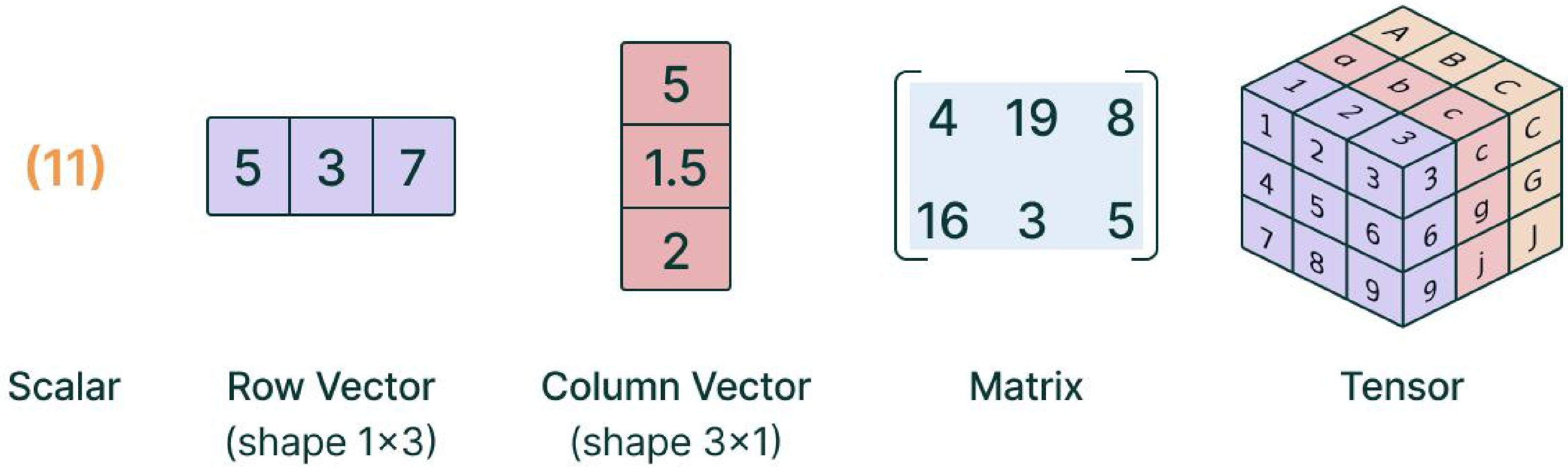
PyTorch

Tensor Algebra

Tensors are **multi-dimensional arrays** of numbers that generalize scalars, vectors, and matrices to higher dimensions.

Tensor algebra is a **branch** of **mathematics** that deals with **tensors** and their mathematical **operations**.



Tensor Algebra: Addition and multiplication with scalars

To perform and **addition** or **multiplication** between a **scalar** and a **tensor** simply add or multiply each value by it

$$a + \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a+m_{00} & a+m_{01} & a+m_{02} \\ a+m_{10} & a+m_{11} & a+m_{12} \\ a+m_{20} & a+m_{21} & a+m_{22} \end{pmatrix}$$

$$a \bullet \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a \bullet m_{00} & a \bullet m_{01} & a \bullet m_{02} \\ a \bullet m_{10} & a \bullet m_{11} & a \bullet m_{12} \\ a \bullet m_{20} & a \bullet m_{21} & a \bullet m_{22} \end{pmatrix}$$

Tensor Algebra: Addition

To **add** two **tensor** simply **add** each **respective element**. For $C = A + B$:

$$C_{ij} = A_{ij} + B_{ij}$$

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} + \begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{pmatrix} = \begin{pmatrix} m_{00}+h_{00} & m_{01}+h_{01} & m_{02}+h_{02} \\ m_{10}+h_{10} & m_{11}+h_{11} & m_{12}+h_{12} \\ m_{20}+h_{20} & m_{21}+h_{21} & m_{22}+h_{22} \end{pmatrix}$$

Only tensors with the same shape can be added

Tensor Algebra: Broadcasting

Only tensors with the same shape can be added? that is not 100% true.
Different tensors can be added exploiting the **broadcast** operation.

Broadcasting is an implicit operation where a **tensor** is **automatically replicated** to **match** the **dimension** of the other operand **tensor**.

```
import torch

a = torch.tensor([[1,2,3],[4,5,6]])
b = torch.tensor([[1,2,3]])

print("shape a =", a.shape)
print("shape b =", b.shape)

c = a + b
print("c =",c)
print("shape c =",c.shape)
```

```
shape a = torch.Size([2, 3])
shape b = torch.Size([1, 3])
c = tensor([[2, 4, 6],
        [5, 7, 9]])
shape c = torch.Size([2, 3])
```

```
import torch

a = torch.tensor([[1,2,3],[4,5,6]])
b = torch.tensor([[1,2,3], [1,2,3]])

print("shape a =", a.shape)
print("shape b =", b.shape)

c = a + b
print("c =",c)
print("shape c =",c.shape)
```

```
shape a = torch.Size([2, 3])
shape b = torch.Size([2, 3])
c = tensor([[2, 4, 6],
        [5, 7, 9]])
shape c = torch.Size([2, 3])
```

Tensor Algebra: Multiplication

To **multiply** two **tensor** simply **multiply** each **respective element**. For $C = A \bullet B$:

$$C_{ij} = A_{ij} \bullet B_{ij}$$

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \bullet \begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{pmatrix} = \begin{pmatrix} m_{00} \bullet h_{00} & m_{01} \bullet h_{01} & m_{02} \bullet h_{02} \\ m_{10} \bullet h_{10} & m_{11} \bullet h_{11} & m_{12} \bullet h_{12} \\ m_{20} \bullet h_{20} & m_{21} \bullet h_{21} & m_{22} \bullet h_{22} \end{pmatrix}$$

Tensor Algebra: Multiplication with broadcasting

The broadcast operator also works for multiplication

```python
import torch

a = torch.tensor([[1,2,3],[4,5,6]])
b = torch.tensor([[1,2,3]])

print("shape a =", a.shape)
print("shape b =", b.shape)

c = a * b
print("c =",c)
print("shape c =",c.shape)
```

```
shape a = torch.Size([2, 3])
shape b = torch.Size([1, 3])
c = tensor([[ 1,  4,  9],
        [ 4, 10, 18]])
shape c = torch.Size([2, 3])
```

```python
import torch

a = torch.tensor([[1,2,3],[4,5,6]])
b = torch.tensor([[1,2,3], [1,2,3]])

print("shape a =", a.shape)
print("shape b =", b.shape)

c = a * b
print("c =",c)
print("shape c =",c.shape)
```

```
shape a = torch.Size([2, 3])
shape b = torch.Size([2, 3])
c = tensor([[ 1,  4,  9],
        [ 4, 10, 18]])
shape c = torch.Size([2, 3])
```

Tensor Algebra: Multiplication with broadcasting (2)

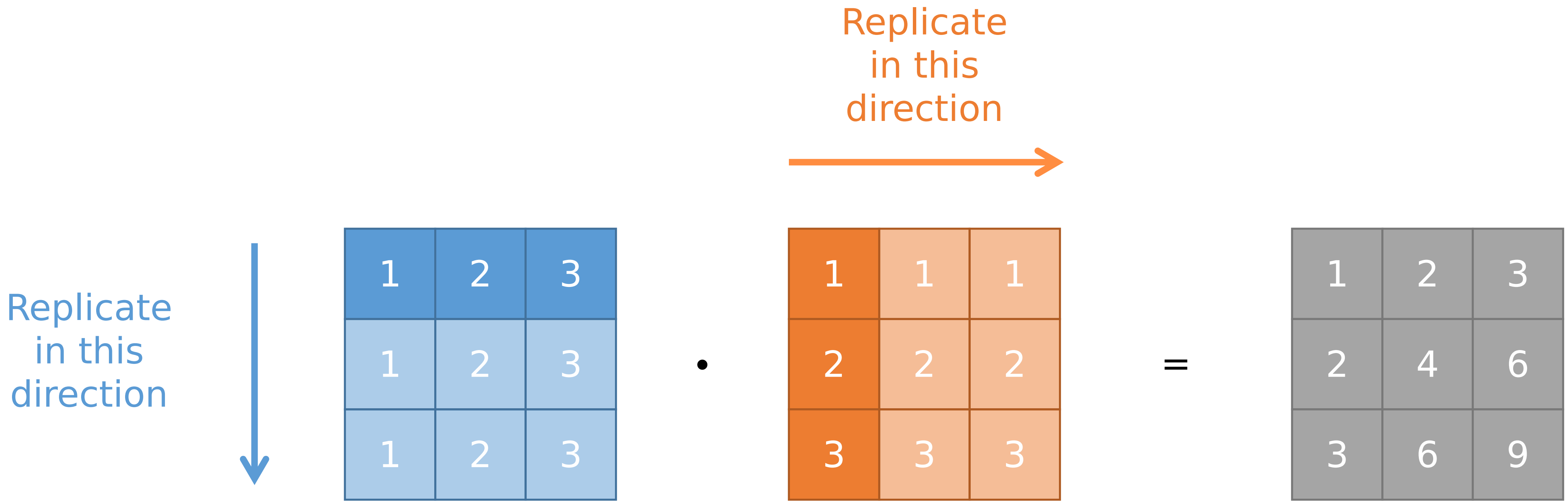
```python
import torch

a = torch.tensor([[1,2,3]])

print("shape a   =", a.shape)
print("shape a.T =", a.T.shape)

c = a * a.T
print("c =",c)
print("shape c =",c.shape)
```

```
shape a   = torch.Size([1, 3])
shape a.T = torch.Size([3, 1])
c = tensor([[1, 2, 3],
        [2, 4, 6],
        [3, 6, 9]])
shape c = torch.Size([3, 3])
```

```python
import torch

a = torch.tensor([[1,2,3], [1,2,3], [1,2,3]])
b = torch.tensor([[1,1,1], [2,2,2], [3,3,3]])

print("shape a =", a.shape)
print("shape b =", b.shape)

c = a * b
print("c =",c)
print("shape c =",c.shape)
```

```
shape a = torch.Size([3, 3])
shape b = torch.Size([3, 3])
c = tensor([[1, 2, 3],
        [2, 4, 6],
        [3, 6, 9]])
shape c = torch.Size([3, 3])
```

Tensor Algebra: Multiplication with broadcasting (3)

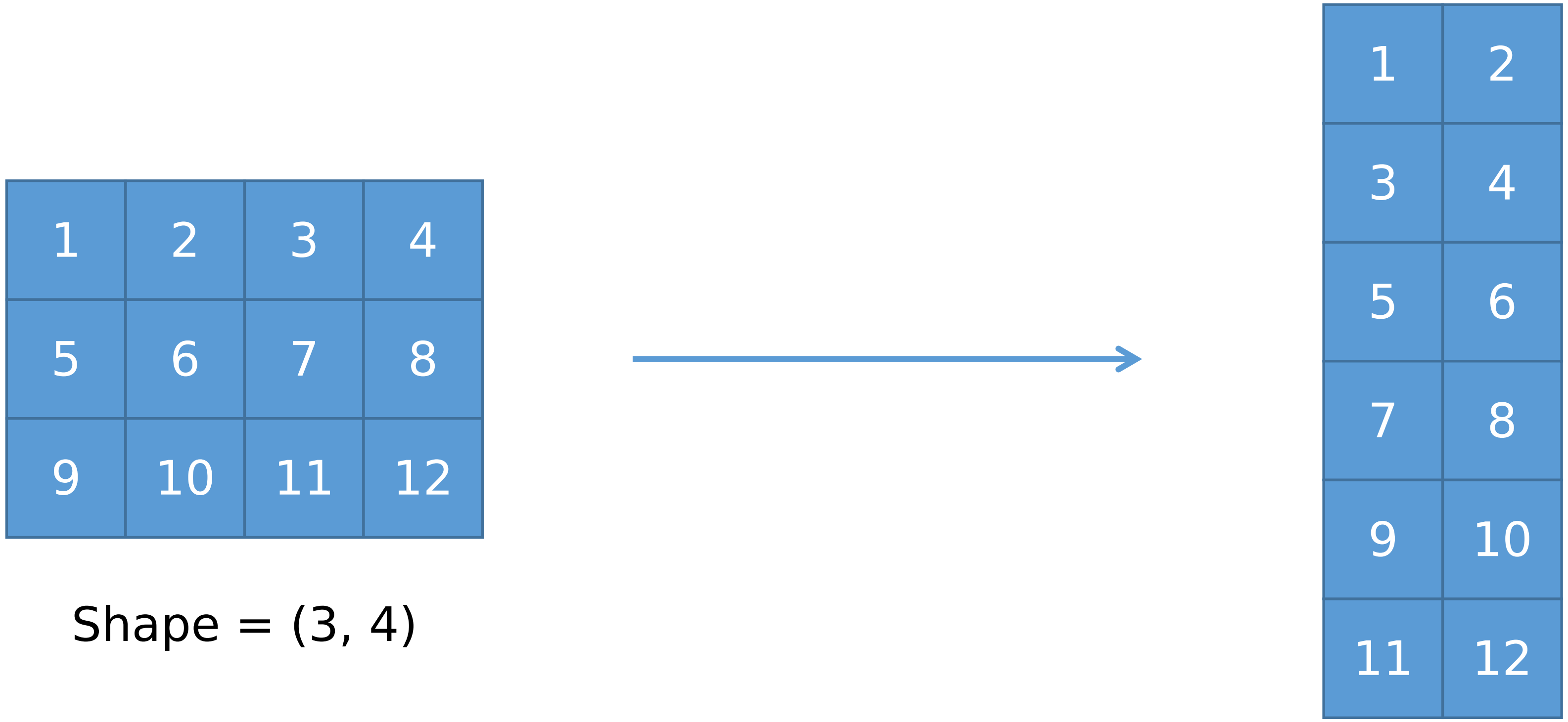


Tensor Algebra: Rearrange

Manipulating the **shape** of a **tensor** is called a rearrange operation.

When **rearranging** a tensor the **number** of **elements** remains the **same.**

The **product** of the **dimensions** is then **constant**.

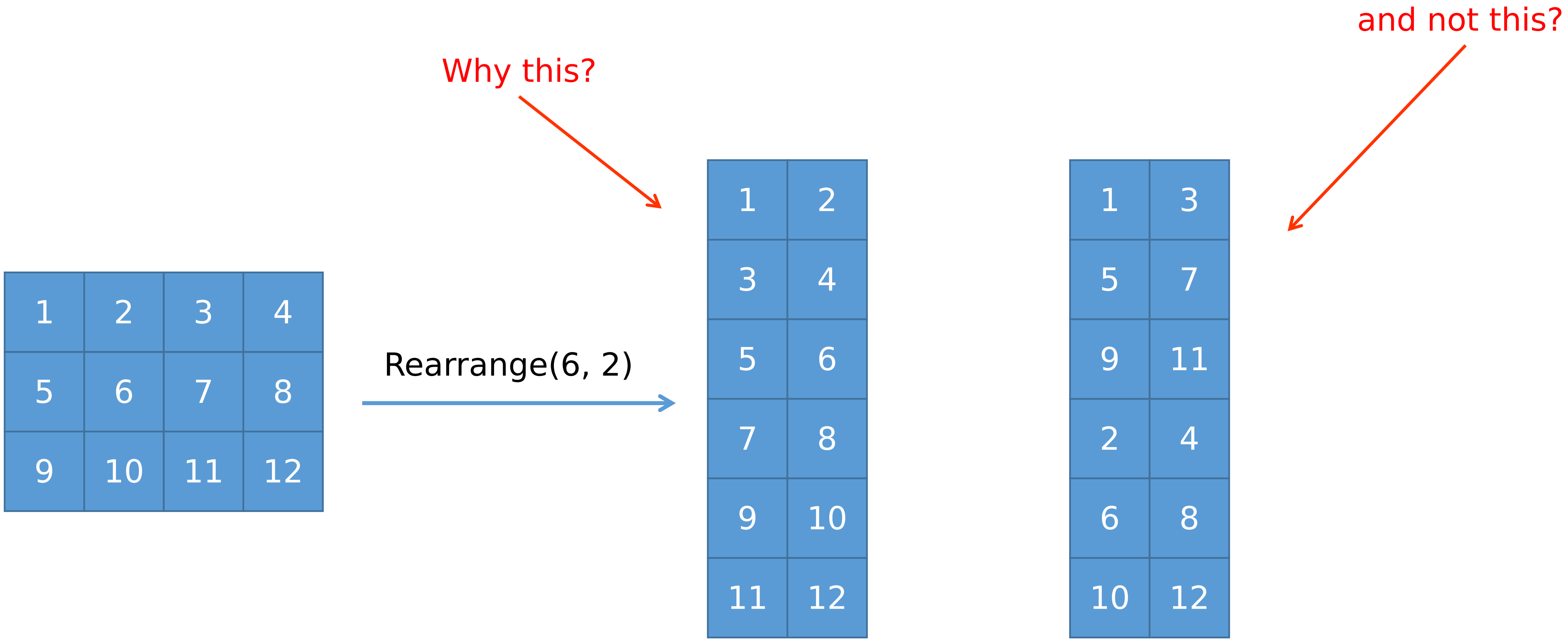


Shape = (3, 4)

Shape = (6, 2)

Tensor Algebra: Rearrange (2)

Rearranging a tensor **implicitly** defines an **order.**



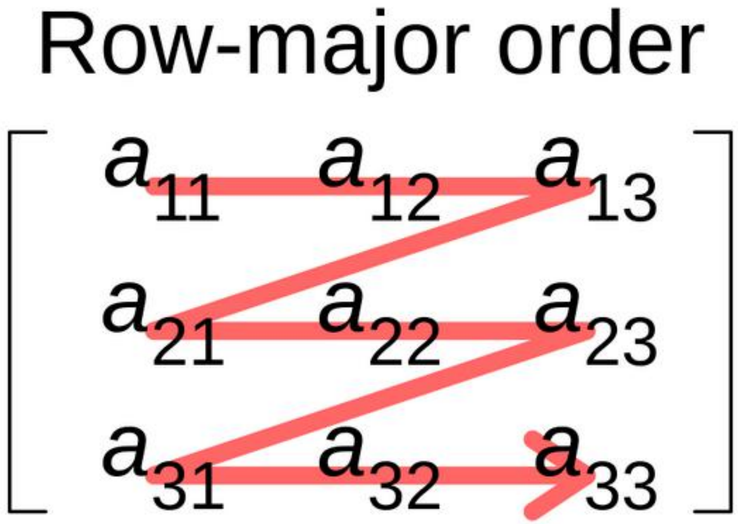
Tensor Algebra: Rearrange (3)

There is **no right answer**.

PyTorch uses **row-major** tensor algebra.

When **rearranging** elements are **ordered** from **left to right starting** from the **first** (**row**) **dimension**.

Elements **order** is **maintained** during the **rearrange** operation.



Tensor Algebra: Rearrange (4)

In PyTorch there are **two ways** to **rearrange** a tensor:

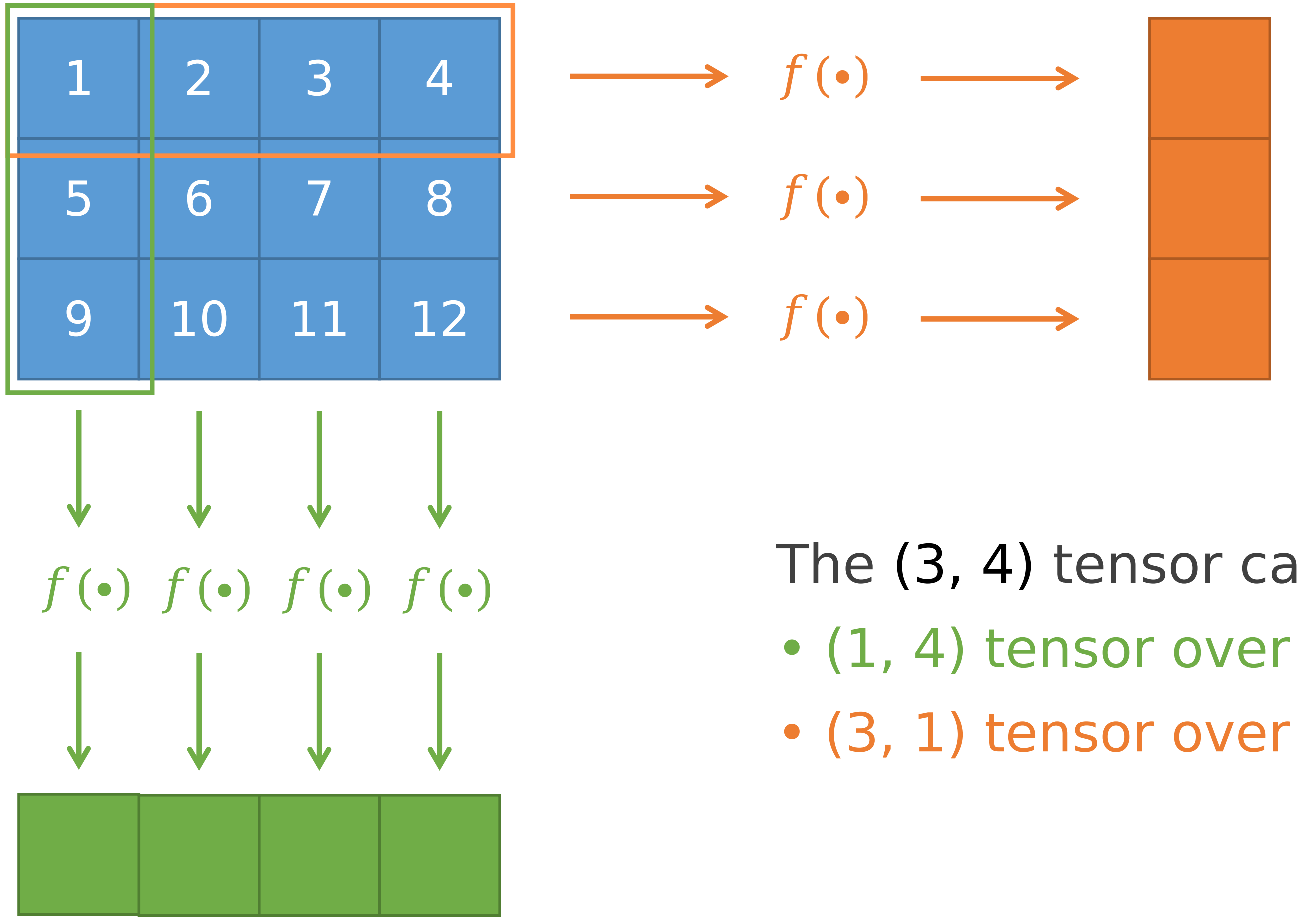
- **reshape**
  - **Actually rearrange** the tensor in the **memory**
  - Slow rearrange, fast access
- **view**
  - Only **modifies** the **indexing**. The **memory is not modified**
  - Fast rearrange, slightly slower access

```python
import torch

a = torch.tensor([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a.reshape((6, 2))
c = a.view((6, 2))

print("b =", b)
print("c =", c)
```

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

Tensor Algebra: Reduce

In a **reduction** the **elements** of a **dimension** are **combined** together and **reduced** into a **new element.**



The (3, 4) tensor can be reduced to a:

- (1, 4) tensor over the first dimension (rows)
- (3, 1) tensor over the second dimension (columns)

Tensor Algebra: Reduce (2)

There are **several reduction** operations.

For example:

- Max
- Min
- Sum
- Mean
- Std
- ...

dim is the keyword
to specify the reduction
dimension

```
import torch

a = torch.tensor([[1,2,3,4], [5,6,7,8], [9,10,11,12]]).float()
b = a.mean(dim=0)
c = a.mean(dim=1)
d = a.max(dim=0).values

print("b =", b)
print("c =", c)
print("d =", d)
```

```
b = tensor([5., 6., 7., 8.])
c = tensor([ 2.5000,  6.5000, 10.5000])
d = tensor([ 9., 10., 11., 12.])
```

Tensor Algebra: Matrix Multiplication

Matrix Multiplication is a binary operation that **produces** a **matrix** from **two matrices**.

The number of **columns** in the **first matrix** must be **equal** to the number of **rows** in the **second matrix**.

$$C = A \times B$$

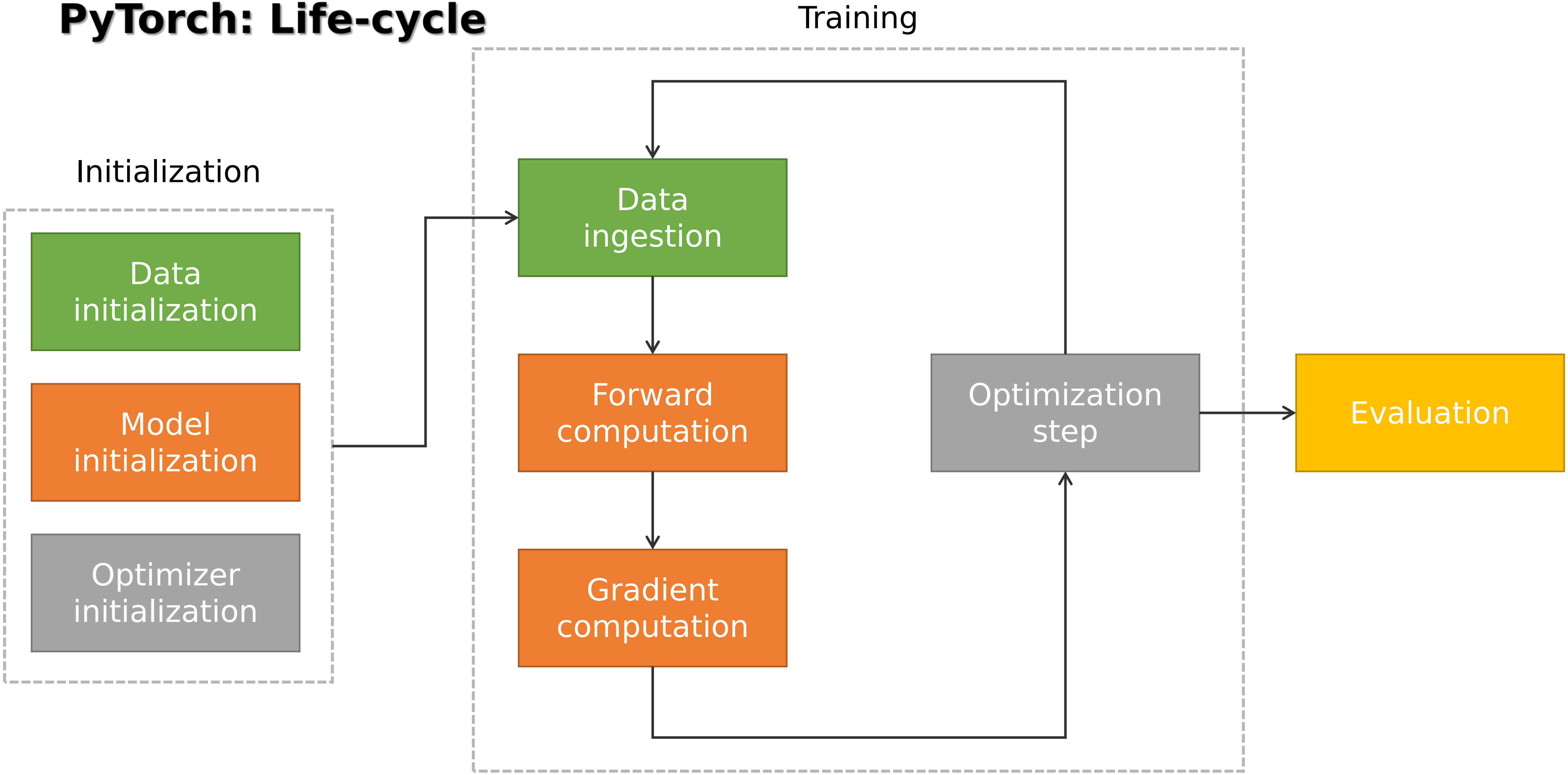
$$C_{ij} = \sum_{k=0}^{n-1} a_{ik} \bullet b_{kj}$$

Tensor Algebra: Matrix Multiplication (2)

```python
import torch

a = torch.tensor([[1,2,3], [4,5,6]])
b = torch.tensor([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

c = torch.matmul(a,b)

print("c = ", c)

print("shape a =", a.shape)
print("shape b =", b.shape)
print("shape c =", c.shape)
```

```
c =  tensor([[ 38,  44,  50,  56],
        [ 83,  98, 113, 128]])
shape a = torch.Size([2, 3])
shape b = torch.Size([3, 4])
shape c = torch.Size([2, 4])
```

PyTorch: Life-cycle



PyTorch: Tensors

There are several ways to initialize a tensor in PyTorch:

- From a **list** or a **numpy array**
- Using torch.ones(shape) to initialize it with **ones**
- Using torch.zeros(shape) to initialize it with **zeros**
- Using the **random module**
- etc..

```python
import torch

a = torch.tensor([[1,2,3], [4,5,6]])
b = torch.ones((3,3))
c = torch.rand((2, 3))

print("a =", a)
print("b =", b)
print("c =", c)
```

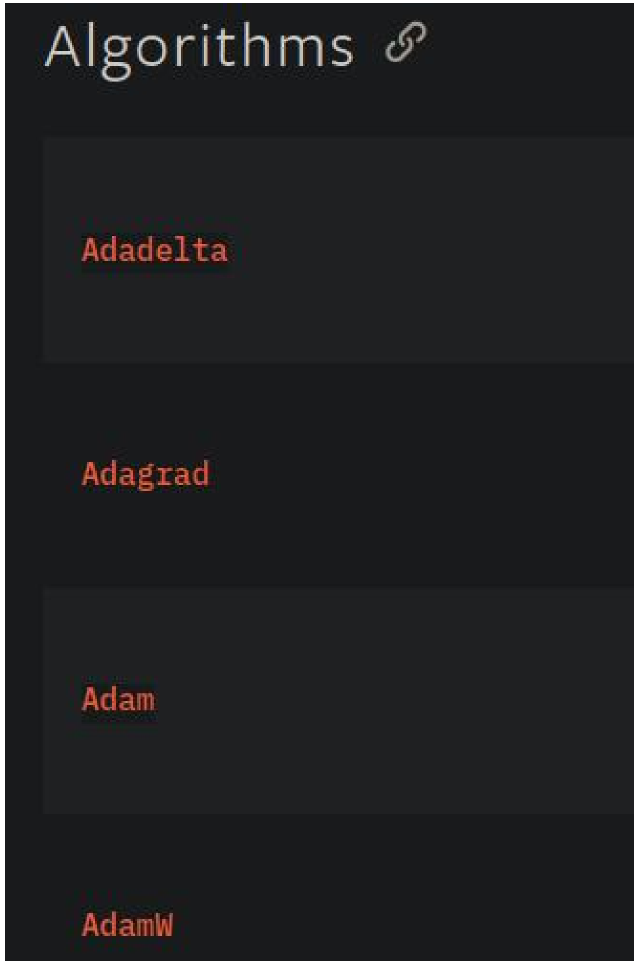
```
a = tensor([[1, 2, 3],
        [4, 5, 6]])
b = tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
c = tensor([[0.7326, 0.1586, 0.8283],
        [0.3195, 0.4380, 0.5967]])
```

PyTorch: Optimizers

There are **several SGD** optimizers implemented in the **torch.optim package**

For example:

- Vanilla SGD
- Adam
- Adagrad
- RMSProp
- etc…



PyTorch: Functional Interface

The **functional interface** is located in the torch.nn.functional **package.**

There are several functions available:

- Activation functions
- Linear functions
- Loss functions
- etc…

PyTorch: Dataset

In PyTorch there are two type of dataset abstractions:

- torch.utils.data.Dataset
- torch.utils.data.IterableDataset

torch.utils.data.Dataset follows the **map** python data model.

You have to implement the methods:

- **__len__(self)**: returns the number of data points
- **__getitem__(self, i)**: returns the **i-th** data point

torch.utils.data.IterableDataset follows the **iterable** python data model.

You have to implement the method:

- **__iter__(self)**: returns an iterable over the data points

PyTorch: Dataset (2)

```python
class SinDataset(torch.utils.data.Dataset):
    def __init__(self, points):
      self.points = points

    def __len__(self):
        return self.points

    def __getitem__(self, idx):
      norm = (idx / self.points) * (2*torch.pi)
      return math.sin(norm)


ds = SinDataset(100)
print("len:", len(ds))
print("0:", ds[0])
print("1:", ds[25])
print("2", ds[75])
```

```
len: 100
0: 0.0
1: 1.0
2 -1.0
```

PyTorch: Dataset (3)

For **torch.utils.data.IterableDataset** its easier and less error prone to make the dataset class itself an **iterator**. This can be achieved by implementing the methods:

- **__iter__(self)** that returns **self**
- **__next__(self)** that returns the **next data point**

```
class MyIterableDataset(torch.utils.data.IterableDataset):
    def __init__(self, points):
      ...

    def __iter__(self):
      return self

    def __next__(self):
      ...
```

PyTorch: Dataset (4)

```
class UnitCircleDataset(torch.utils.data.IterableDataset):
    def __init__(self, points):
      self.total_points = points
      self.generated_points = 0

    def __iter__(self):
      return self

    def __next__(self):
      if self.generated_points >= self.total_points:
        raise StopIteration

      x = torch.rand(1)
      y = torch.rand(1)
      inside = torch.sqrt(x**2 + y**2) < 1

      self.generated_points += 1
      return x, y, inside

ds = UnitCircleDataset(5)
for elem in ds:
  print(elem)
```

```
(tensor([0.6505]), tensor([0.1626]), tensor([True]))
(tensor([0.3634]), tensor([0.9226]), tensor([True]))
(tensor([0.0676]), tensor([0.9742]), tensor([True]))
(tensor([0.6160]), tensor([0.9532]), tensor([False]))
(tensor([0.5621]), tensor([0.5406]), tensor([True]))
```

PyTorch: DataLoader

the class **torch.utils.data.DataLoader**
Combines a dataset and a sampler, and provides an iterable over the given dataset.

The **DataLoader** supports **both map-style** and **iterable-style** datasets with single- or multi-process loading, customizing loading order and automatic **batching**.

```python
ds = UnitCircleDataset(100)

dl = torch.utils.data.DataLoader(
    dataset=ds,
    batch_size=4,
)

for batch in dl:
  x, y, inside = batch
  print(type(batch), len(batch))
  print(x.shape)
  print(y.shape)
  print(inside.shape)
  break
```

```
<class 'list'> 3
torch.Size([4, 1])
torch.Size([4, 1])
torch.Size([4, 1])
```

PyTorch: DataLoader (2)

the class **torch.utils.data.DataLoader** has various arguments, the most important ones are:

- dataset
  - The dataset which to load the data from
- batch_size
  - The batch_size to use
- shuffle
  - Use a random order when accessing the dataset, works only for map-style datasets
- num_workers
  - How many parallel processes to use, each process loads one data point
- sampler
  - The sampler to use

PyTorch: Module

The fundamental building block of PyTorch models is the **torch.nn.Module** class.

- **Parameters**
  - Each model may encompass parameters, represented by tensors wrapped with **torch.nn.Parameter**.
  - Any **attribute** of a Module that **is** a **torch.nn.Parameter** is automatically **included** in the **parameters list** of that module.

- **Sub-modules**
  - Models can be **composed** of **sub-modules**.
  - All **parameters within** these **sub-modules** are seamlessly **aggregated** into the **parameters list** of the **containing module**.

PyTorch: Module (1)

PyTorch **provides** a comprehensive **set of modules** that serve as the **foundational building blocks** for constructing **neural networks**.

Among these, some of the **most frequently utilized** ones include:

- torch.nn.Linear:
  - Implements a linear transformation, commonly used for fully connected layers.
- torch.nn.Sequential
  - A container module that allows for the orderly arrangement of other modules in a sequential manner.
- torch.nn.ReLU
  - Introduces non-linearity through the Rectified Linear Unit (ReLU) activation function.
- torch.nn.Conv2d
  - The 2d convolution operation.

next lecture

PyTorch: Module (2)

The most important methods of a **torch.nn.Module** are:

- **__init__(self)**
  - Constructor method where you define the attributes (parameters and sub-modules) of your module.
  - Remember to call the parent constructor with super().__init__()
- **forward(self, input)**
  - Crucial method where the actual computation of the module occurs.
  - You specify the forward pass of your neural network in this function.
  - Receives input data and returns the output of the module.
- **parameters(self)**
  - Returns an iterator over module parameters.
  - Parameters are tensors that are automatically optimized during training.
- **to(self, device)**
  - Moves the module to the specified device (CPU or GPU).
  - Useful for managing device allocation.

PyTorch: Module (3)

```
class QuadraticModel(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.a = torch.nn.Parameter(torch.rand(1))
    self.b = torch.nn.Parameter(torch.rand(1))
    self.c = torch.nn.Parameter(torch.rand(1))

  def forward(self, x):
    return self.a*x.pow(2) + self.b*x + self.c

model = QuadraticModel()
print(list(model.parameters()))
print(model(torch.tensor(2)))
```

```
[Parameter containing:
tensor([0.0897], requires_grad=True), Parameter containing:
tensor([0.6999], requires_grad=True), Parameter containing:
tensor([0.1056], requires_grad=True)]
tensor([1.8640], grad_fn=<AddBackward0>)
```

Training a Module

In PyTorch you can use **variants** of the **Stochastic Gradient Descent** algorithm to **train** a module from the **torch.optim** package.

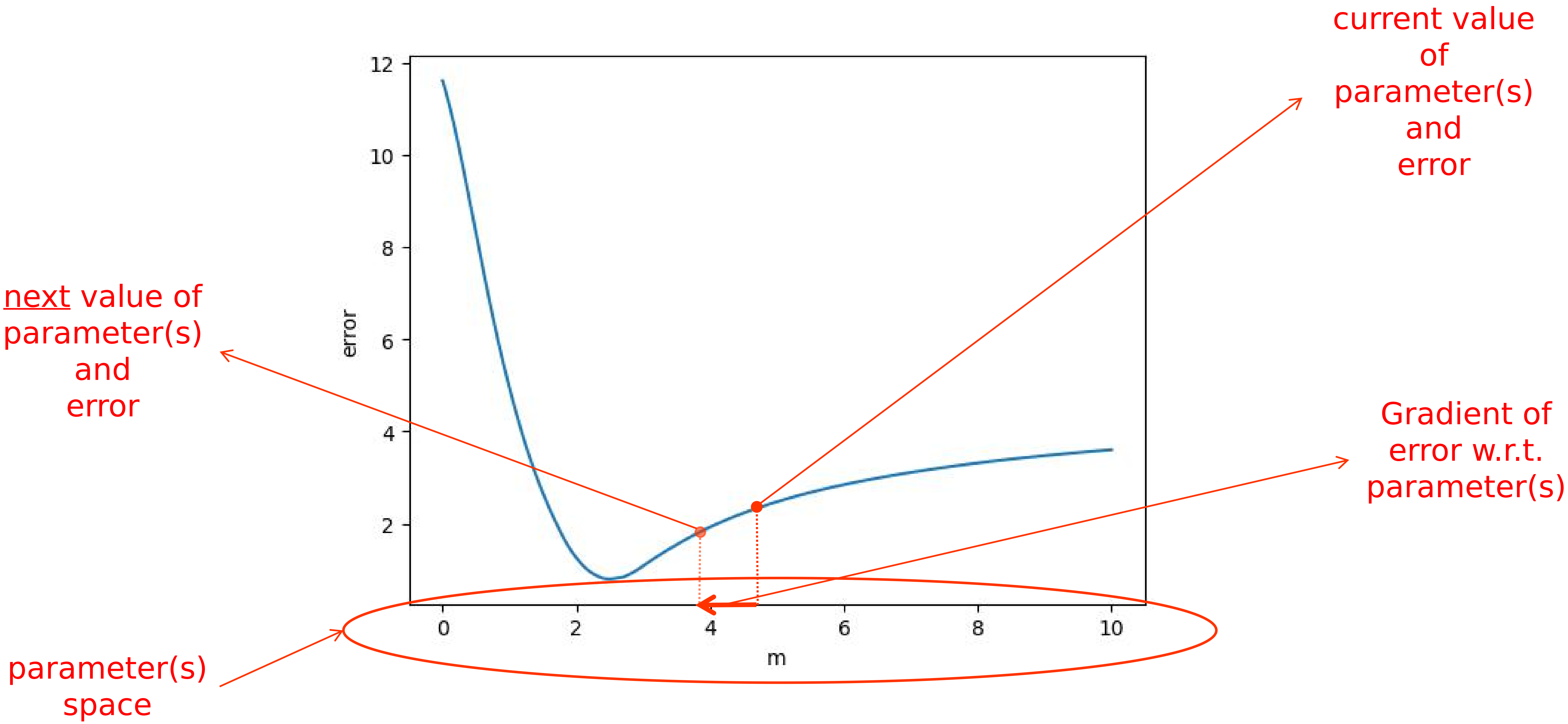
The **main training loop** should (most of the times) **perform** these operations:

- Initialize dataloader, model and optimizer
- for epoch in range(0, epochs):
  - for batch in dataloader
    - output = model(batch.input)
    - error = error_fn(output, batch.target)
    - optimizer.zero_grad()
    - error.backward()
    - optimizer.step()

An "epoch" is training the model over the whole dataset one time.

To train a model you need to perform several epochs

Training a Module (2)



Training a Module (3)

Training a model means **computing an error** (a scalar function), computing the **gradient** of the **error w.r.t.** the model **parameters** and **following** the **direction** that **minimize** the **error**.

PyTorch has some **commonly used** error functions (also called loss functions) already implemented, for example:

- Mean Absolute Error (MAE): **torch.nn.L1Loss**
  - Measures the average absolute difference between predicted and actual values.
  - Less sensitive to outliers compared to MSE.
- Mean Squared Error (MSE): **torch.nn.MSELoss**
  - Measures the average squared difference between predicted and actual values.
  - Suitable for tasks like predicting numerical values.
- Cross-Entropy: **torch.nn.CrossEntropyLoss**
  - Ideal for problems where each sample belongs to one class.
  - Measures the cross-entropy between predicted and target distributions

Exercise 1: (1)

Write the **PointsDataset** class that implements the **torch.utils.data.Dataset** interface:

- Reads a txt file in which each line represents a bidimensional data point, with each dimension separated by a space.
- Saves the content of the file in a data structure of your choice
- The **__len__(self)** method should return the number of data points
- The **__getitem__(self, i)** method should return the i-th data point as a tuple.

dataset.txt

```
-2.067504630593728 -4.3940160659490966
-9.220342264640013 -26.54222567962877
-3.354104007299002 -9.130317428586983
2.190182077823536 6.24970978934776
8.800144432229736 25.615330535203725
1.2955720622286666 4.457587948472233
-7.368560590563282 -24.042007327266063
-7.885062936720201 -21.902512966035843
```

```python
ds = PointsDataset("dataset.txt")
print(ds[0])
```

✓ 0.0s

```
(-2.067504630593728, -4.3940160659490966)
```

Exercise 1: (2)

Write the **LineModule** class that implements the torch.nn.Module interface implementing the function $f(x) = wx$.

- LineModule has 1 parameter w
- The **forward(self, x)** method should return wx

```
model = LineModule()
print(list(model.parameters()))
print(model(torch.tensor([1.])))
```

✓ 0.0s

```
[Parameter containing:
tensor([-0.0583], requires_grad=True)]
tensor([-0.0583], grad_fn=<MulBackward0>)
```

Exercise 1: (3)

Write a complete python script that trains **LineModule** to approximate the data in **dataset1.txt**

- Use the **SGD** optimizer from **torch.optim**
- Use the **MSELoss** from **torch.nn**
- Use a batch_size of 8
- Use a learning rate of 0.001
- Train for 1000 epochs

```
Epoch 0: loss 167.87667012832455
Epoch 0: loss 101.41640371214211
Epoch 0: loss 173.47983306483772
Epoch 0: loss 124.89566487061 7
Epoch 0: loss 99.13100804449708
Epoch 0: loss 147.6078938833903
Epoch 0: loss 108.22731560725815
Epoch 0: loss 75.81846755449054
Epoch 0: loss 96.93804414637638
Epoch 0: loss 77.51460574750205
Epoch 0: loss 20.28168295408831 4
Epoch 0: loss 133.66231409443913
Epoch 0: loss 62.41936717002218
Epoch 1: loss 38.00877930294306
Epoch 1: loss 37.03041242639919
```

Exercise 2 (hard)

Train a polynomial model in this form:

$$ax^4 + bx^3 + cx^2 + dx + e$$

To divide the points of **dataset2.txt** (in the form “x y class” each line)

dataset2.txt

```
0.6694837762958694 0.8945464835833724 0
-0.025934533700649937 1.1107433076159958 0
0.9953308180872537 0.2506698694973196 0
-0.9322951409309891 0.5909418115823066 0
0.16912470429691373 1.0438085765110712 0
0.6182140008161767 -0.6014423583073267 1
0.5906876196569684 -0.31555239872653296 1
-0.6571244845423326 0.7645023530709449 0
0.5672735172344342 -0.36143767441115565 1
```

Exercise 2 (hard)

- The polynomial model takes x as input and gives $\overline{y}$ as output
- Given an (x, y) pair from the dataset you can compute $\overline{y}$ = model(x)
- If y is **above** the line (so y > $\overline{y}$) and **it should be** (class 0) than everything is **ok**
- if y is **above** the line and **it should be below it** (class 1) than you should compute an **error**
- And so on...