IMPLEMENTING AND TRAINING NEURAL NETWORKS

STFANO MELACCI

UNVERSITY OF SIENA

BEFORE GOING INTO FURTHER DETAILS

- We will focus on the **Python** language
  - *Do you have any experiences with Python?*
- Let's recall some basic concepts about Python and Python programming
  - Get it here: https://www.python.org/ (latest: 3.10.0)
  - See this tutorial: https://docs.python.org/3/tutorial/index.html
    - Let's have a quick look at it…
    - …and let's run some basic examples on-the-fly

ENVIRONMENTS

- It is very useful to create a **virtual environment** (VENV) in which we will install all the packages we need without making any changes to the system-level Python installation
  - We can create a VENV in a folder named *my_local_folder* (customizable) by running ONE of the following commands (not both of them)

```
python3 –m venv my_local_folder
```

```
virtualenv –p python3 my_local_folder
```

  - Then, we have to jump into the just-created folder and activate the VENV (Linux/Mac)

```
cd my_local_folder
source bin/activate
```

ENVIRONMENTS

- All the packages that will be installed using pip, will be physically installed inside the VENV folder
- Removing the VENV folder will clean the whole thing
- Let's install some basic packages that we will use during this course

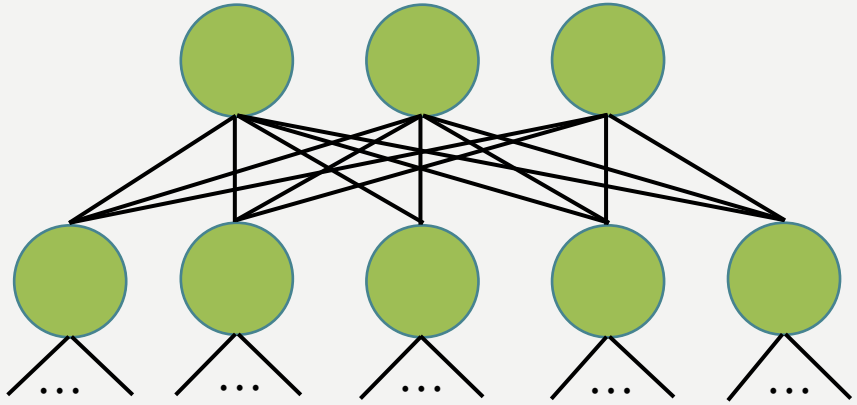
pip install numpy

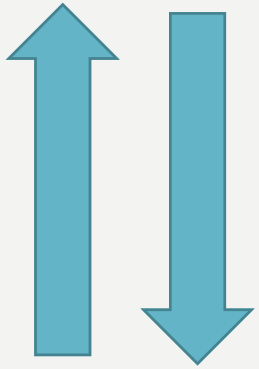
pip install torch torchvision

pip install tensorflow

BACKPROPAGATION REVISITED

- Once we have fully acquired the details of the BackPropagation (BP) algorithm, we are ready to implement it…
- …**are we really ready**?

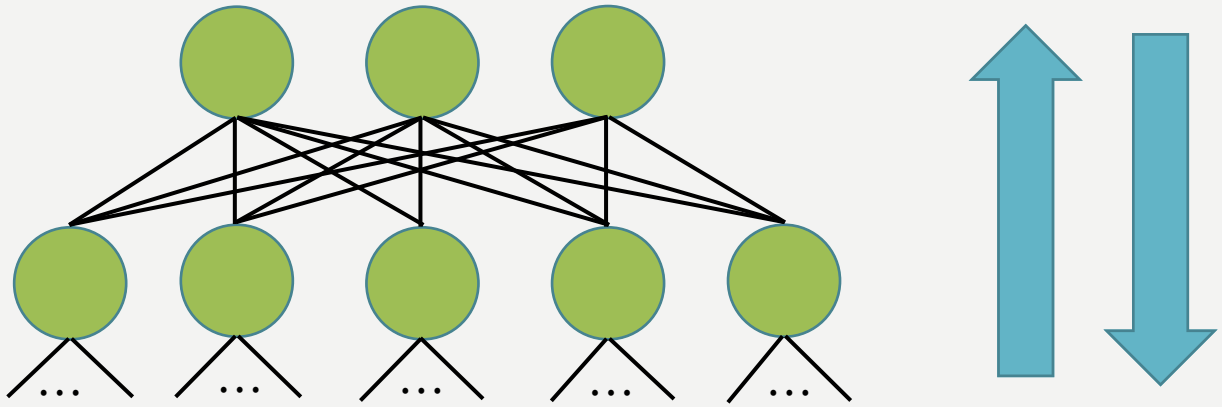




```
for (int l=0; l<layers; l++) {
        out[l] = forward(l, net, …)
}
for (int l=layers-1; l>=0; l--) {
        grad[l] = backward(l, out[l], net, …)
}
for (int l=0; l<layers; l++) {
        update_weights(l, grad[l], net, …)
}
```

BACKPROPAGATION REVISITED

- Once we have fully acquired the details of the BackPropagation (BP) algorithm, we are ready to implement it…
- …**are we really ready**?



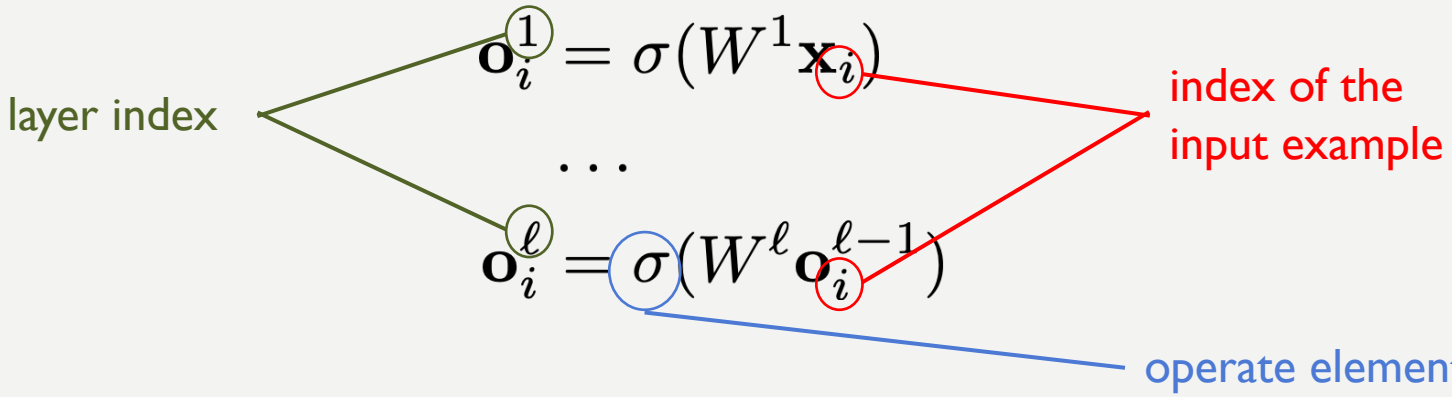
```
forward(l, net, …) {
        for (int j=0; j <num_examples; j++) {
        for (int i=0; i <num_neurons; i++) {
                …
        }
        }
}

backward(l, net, …) {
        for (int j=0; j <num_examples; j++) {
        for (int i=0; i <num_neurons; i++) {
                …
        }
        }
}
```

FORWARD

- We are given an example **x** and we have to compute the outputs of the net
  - For each layer
  - For each neuron of the layer
- If we have a **batch** of examples $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ then the same operations must be repated N times

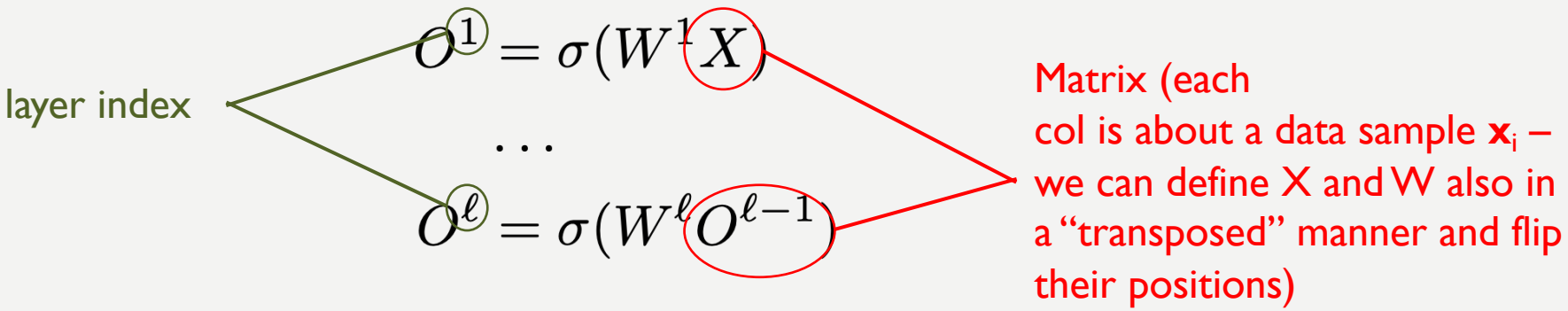
FORWARD

- We are given an example $\mathbf{x}_i$ belonging to batch X and we have to compute the output values of the net
  - We already discussed the matrix-notation of the layer output computation
  - Storing weights as contiguous data in the host machine allows us to exploit the benefits from data (spatial) locality, caching, SIMD instructions, …



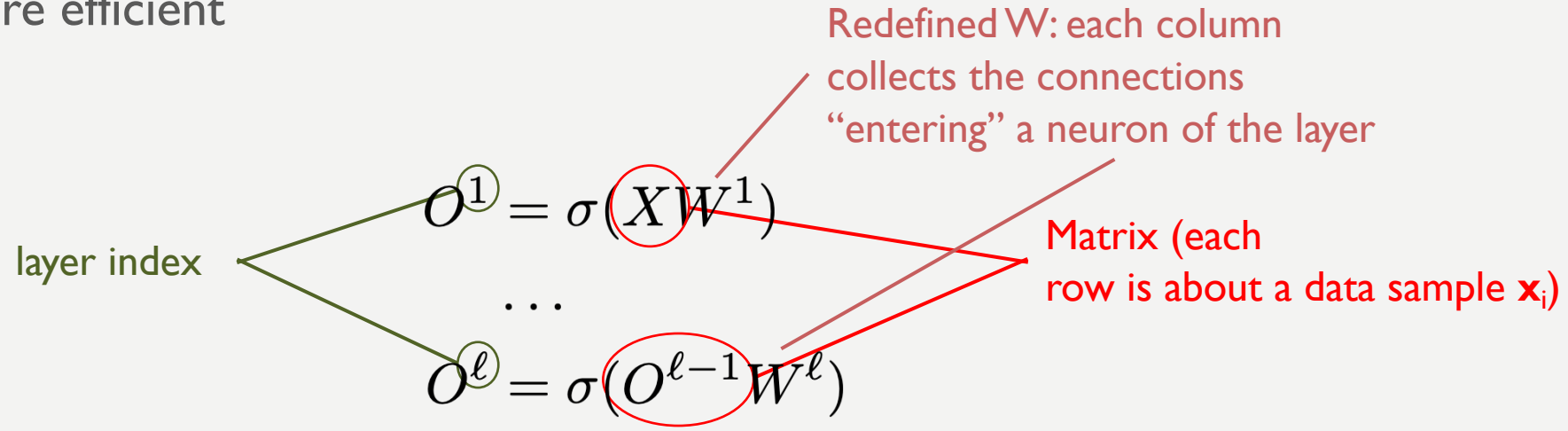
FORWARD

- We are given **all the examples** belonging to batch X and we have to compute the output values of the net
  - We can introduce a matrix-like representation of the batch of data to make things even more efficient



FORWARD

- We are given **all the examples** belonging to batch X and we have to compute the output values of the net
  - We can introduce a matrix-like representation of the batch of data to make things even more efficient



BACKWARD

- The backward phase must be reviewed considering the just introduced matrix notation
- Once we do this, we will discover that the backward-related operations can be efficiently implemented in modular manner
- Before doing this, we must introduce and discuss several notions that are bout
  - Tensors
  - Matrix Calculus (matrix derivatives)
  - Computational Graphs

DATA & VIEWS

- **Tensors**: for the purpose of this course, it is enough to consider a "tensor" as a generic d-dimensional structure that indexes data (informal/practical definition)
  - Basically, a **multi-dimensional array**
- When defining tensors, we also have to define the way **operations** are applied to tensors
  - Scalar: no dimensions at all (it's a number!)
  - Vector: one-dimensional tensor
  - Matrix: two-dimensional tensor
  - 3D structure (Cube) : three dimensional tensors
  - …

DATA & VIEWS

- **Vectors** (mono-dimensional) are usually assumed to be organized as columns

$$\mathbf{v} = [v_1, v_2, \ldots, v_n]^T = \begin{bmatrix} v_1 \\ v_2 \\ \ldots \\ v_n \end{bmatrix}$$

- **From tensors to vectors, matrices, scalars**
  - Example: Tensor whose dimensions are *1*x*1*, or *1*x*1*x*1*x*1*, or just *1*: it is a scalar?
    - It is a tensor with 1 element, but it is still preserving its dimensions
  - Example: Tensor whose dimensions are *1*x*m*x*n*, or *m*x*n*x*1*: is it a matrix?
    - It is a tensor with *m*x*n* elements, and it has 3 dimensions
  - Why do we care about the number of dimensions?
    - Because they are important when we will define/apply **operations** among multiple tensors!

STORING TENSORS

- Let's consider the implementation of a tensor in a target machine, using a certain PL
- We have to keep track of:
  1. Number of **dimensions**
  2. Number of elements in each dimensions (**size** of each dimension): overall, the **shape** of the tensor is the collection of the (ordered) number of elements in each dimensions: *m*x*n*x*c*
  3. The raw **data** that is stored in the tensor (i.e., the numbers ☺), sometimes called **storage**
  4. The **type** of the data (single precision floating points, double precision floating points, integers on 8-16-32-64 bits, …)
  5. The **stride** (we will take about it later)
  6. …

STORING TENSORS

- The memory of the host machine is a 1-dimensional structure, so we have to define how the raw elements are stored, in which order
  - Common assumption: the last dimension of the tensor is the one on which elements are "contiguous" in the memory of the target machine
    - You can also find the exact opposite assumption (contiguous elements are the ones on the first dimension)

Tensor

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \end{bmatrix}$$

Mono-dim View

$$[X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{23}]$$

Stored Data

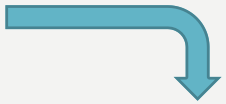
$$[D_1, D_2, D_3, D_4, D_5, D_6]$$

STORING TENSORS

- Common assumption: the last dimension of the tensor is the one on which elements are ”contiguous” in the memory of the target machine
  - The dimensions with the ”closest” elements are the ones right next to the last dim, while elements on the first dimension are the farthest ones

2x3x2 Tensor

$$X = \begin{bmatrix} [X_{111}, X_{112}] & [X_{121}, X_{122}] & [X_{131}, X_{132}] \\ [X_{211}, X_{212}] & [X_{221}, X_{222}] & [X_{231}, X_{232}] \end{bmatrix}$$



Mono-dim View

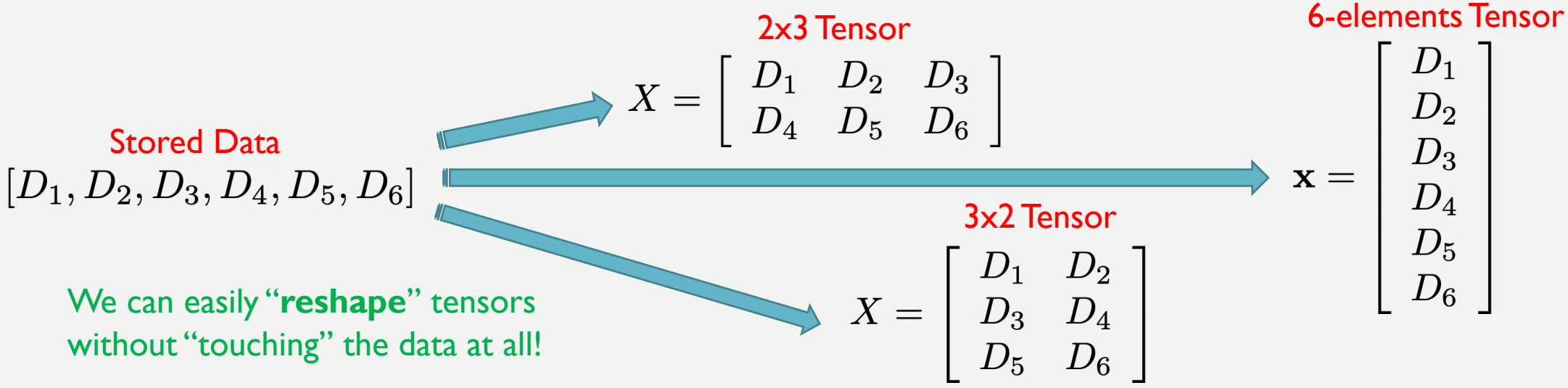
$$[X_{111}, X_{112}, X_{121}, X_{122}, X_{131}, X_{132}, X_{211}, X_{212}, X_{221}, X_{222}, X_{231}, X_{232}]$$

Stored Data

$$[D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11}, D_{12}]$$

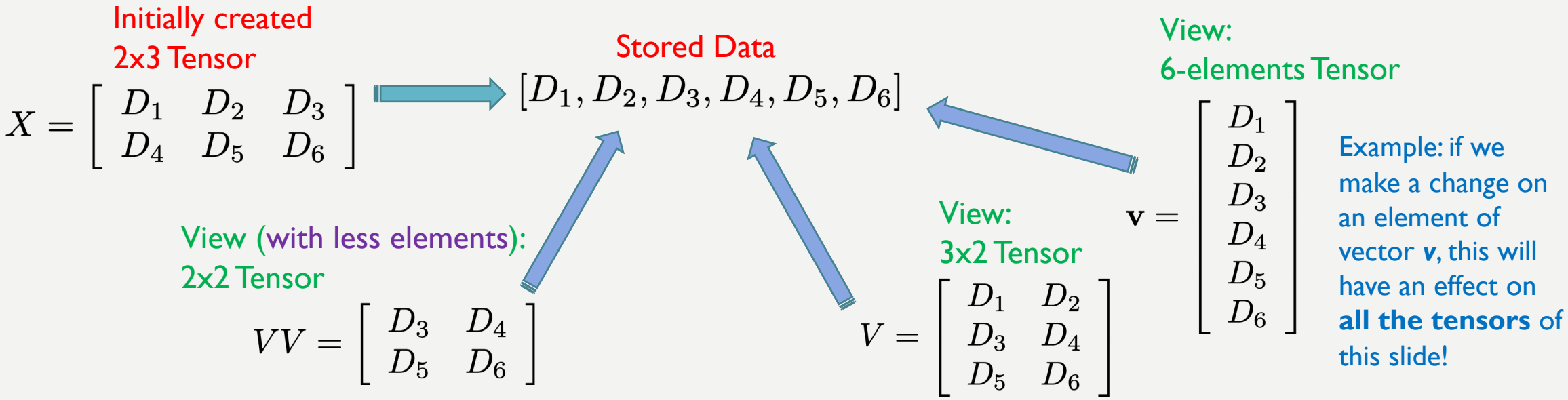
RESHAPING TENSORS

- Once we agree on the fact that the tensor data are always stored in a mono-dim fashion, then it is straightforward to understand that the **shape** of the tensor is just a way to formalize how the data are organized



VIEWS

- A **view** is a tensor that shares the data memory with another tensor, but it has a **different shape** and eventually also a **different number of elements**



STRIDE

- Can we create a view that involves non-contiguous data elements?
  - We need to consider a new piece of information in the tensor structure
  - **Stride: distance among elements on each dimension**
    - In what we described so far, the stride is ”one” on the last dimension (it must be multiplied by the number of bytes of the tensor data type to get the real offset)
    - In the case of a matrix, it is “one” on the second dimension and it is “number of columns” on the first dimension
  - The stride can be artificially altered to create custom views (see next slide)

STRIDE

– For example, a view that is composed by the odd columns of a matrix requires a stride that is ”two” in the last dimension!

Initially created
2x3 Tensor

$$X = \begin{bmatrix} D_1 & D_2 & D_3 \\ D_4 & D_5 & D_6 \end{bmatrix}$$

Dimension 1, stride 3
Dimension 2, stride 1

Stored Data

$$[D_1, D_2, D_3, D_4, D_5, D_6]$$

View (with less elements):
2x2 Tensor

$$VVV = \begin{bmatrix} D_1 & D_3 \\ D_4 & D_6 \end{bmatrix}$$

Dimension 1, stride 3
Dimension 2, stride 2

Suppose that X stores 4-byte floats:
Dimension 1: stride * 4 bytes = 3*4 bytes = 12 bytes between consecutve elements
Dimension 2: stride * 4 bytes = 1*4 bytes = 4 bytes between consecutve elements

Dimension 1: 3*4 bytes = 12 bytes between consecutve elements
Dimension 2: 2 * 4 bytes = 8 bytes between consecutve elements

STRIDE

– For example, a view that is composed by the odd columns of a matrix requires a stride that is ”two” in the last dimension!

Initially created
2x3 Tensor

$$X = \begin{bmatrix} D_1 & D_2 & D_3 \\ D_4 & D_5 & D_6 \end{bmatrix}$$

Dimension 1, stride 3
Dimension 2, stride 1

Stored Data

$$[D_1, D_2, D_3, D_4, D_5, D_6]$$

View (with less elements):
2x2 Tensor

$$VVV = \begin{bmatrix} D_1 & D_3 \\ D_4 & D_6 \end{bmatrix}$$

Dimension 1, stride 3
Dimension 2, stride 2

A 2x2 tensor created from scratch (not a view) would have:
Dimension 1, stride 2
Dimension 2, stride 1

Dimension 1: 3*4 bytes = 12 bytes between consecutve elements
Dimension 2: 2 * 4 bytes = 8 bytes between consecutve elements

STRIDE

- **Just a quick note**: in order to exploit some hardware-related facilities (SIMD operations, some GPU operations, …) it is required to allocate tensors such that the stride (in bytes) is a multiple of a given number
  - This means that we must store some ”dummy”/”unused” elements in the tensor data, and set the **stride** accordingly
  - The memory allocated to store the tensor data will be larger that what it is needed to store the real tensor contents
  - Sometimes the dummy data is called “**extra padding**” in tensor allocation
  - This means that it is not always possible to simply reshape a tensor without copying the data

CODE BREAK: TENSORS

- Let's have a look at the code!
  - C
    - Entry point: multi-dimensional C arrays
  - Matlab
    - Multi-dimensional arrays in Matlab, indexing
  - Python (using Numpy)
    - Multi-dimensional NumPy arrays, indexing, views

OPERATIONS

- The basic rules of linear algebra tell us how to perform operations involving scalars, vectors and matrices
- Once we move to generic tensor, things are not different, we must take care of having clear ideas on which **dimensions** are involved in the operation(s)
- For the purpose of this course, we will consider operations as they defined in Python libraries (Numpy, PyTorch, TensorFlow, …)

OPERATIONS

- Given a tensor *X* with *d* dimensions, some common operations are usually provided in most of the existing libraries
  - Take the sum of the elements in *X*
    - Result: scalar
  - Sum/mean all the elements along a certain dimension

    (for example: for each column of a matrix, take the sum)
    - Result. Tensor with *d-1* dimensions
    - These operations "**reduce**" the dimensionality
  - Multiply the whole tensor by a scalar
    - Result: tensor with the same size of X
  - Take the sum/difference/**element-wise**-product of two tensors (same size)
    - Result: tensor with the same size of X

  …

$$a = \sum_{i=1}^{n} \sum_{j=1}^{m} X_{ij}$$

$$\mathbf{v} = \left[\sum_{j=1}^{m} X_{ij}, \; i = 1, \ldots, n \right]$$

$$A = s \cdot X$$

In these examples X is a n-by-m matrix

$$A = X + Y$$

$$A = X \cdot Y$$

OPERATIONS - BROADCASTING

- Other common operations are element-wise exponentiation, dot-product, matrix-by-matrix multiplication, or matrix-by-vector multiplication
- An important feature that is also (not always) usually provided is **broadcasting**

$$X \in \mathbb{R}^{n,m}$$

$$\mathbf{v} \in \mathbb{R}^{n,1}$$

$$X + \mathbf{v} = ?$$

OPERATIONS - BROADCASTING

- Other common operations are element-wise exponentiation, dot-product, matrix-by-matrix multiplication, or matrix-by-vector multiplication
- An important feature that is also (not always) usually provided is **broadcasting**

$$X \in \mathbb{R}^{n,m}$$

$$\mathbf{v} \in \mathbb{R}^{n,1}$$

$$X + \mathbf{v} = ?$$

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$X + \mathbf{v} = \begin{bmatrix} X_{11} + v_1 & X_{12} + v_1 & X_{13} + v_1 \\ X_{21} + v_2 & X_{22} + v_2 & X_{23} + v_3 \end{bmatrix}$$

OPERATIONS - BROADCASTING

- Broadcasting is a powerful "tool" that allows us to avoid wasting memory to replicate a tensor along certain dimensions
  - It is basically like creating a **for loop** in C to perform the operation between the given tensors, with no further memory allocation
  - Manually creating such for loop in Python would result in very slow execution times

- See the exampels at: https://numpy.org/doc/stable/user/basics.broadcasting.html

- **Are two tensors A and B compatible for broadcasting a certain operation?**
  - Compare the shapes of A and B reading them from right to left (compare last dimension of A with last dimension of B and so on …)
    - If the compared dimensions are (1) the same or (2) one of them is "one", then A and B are compatible with broadcasting
    - Of course, if A has more dimensions than B (or vice versa) you do not have to consider these "**extra** dimensions"

OPERATIONS - BROADCASTING

- **What will happen then?**
  - The singleton ("one") dimensions are virtually replicated in order to match the size of the corresponding non-singleton dimensions in the other tensor
  - The tensor with the smallest number of dimensions is also virtually replicated along the extra dimensions
  - Given the virtually replicated tensors, we can perform an **element-wise** operation

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$X + \mathbf{v} = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \end{bmatrix} + \begin{bmatrix} v_1 & v_1 & v_1 \\ v_2 & v_2 & v_2 \end{bmatrix}$$

Element-wise sum (virtual operation, no new memory is allocated!)

CODE BREAK: OPERATIONS

- Let's have a look at the code!
  - NumPy
    - Operations on multi-dimensional NumPy arrays
    - Matrix-by-matrix multiplication, exp, sums and means over certain dimensions