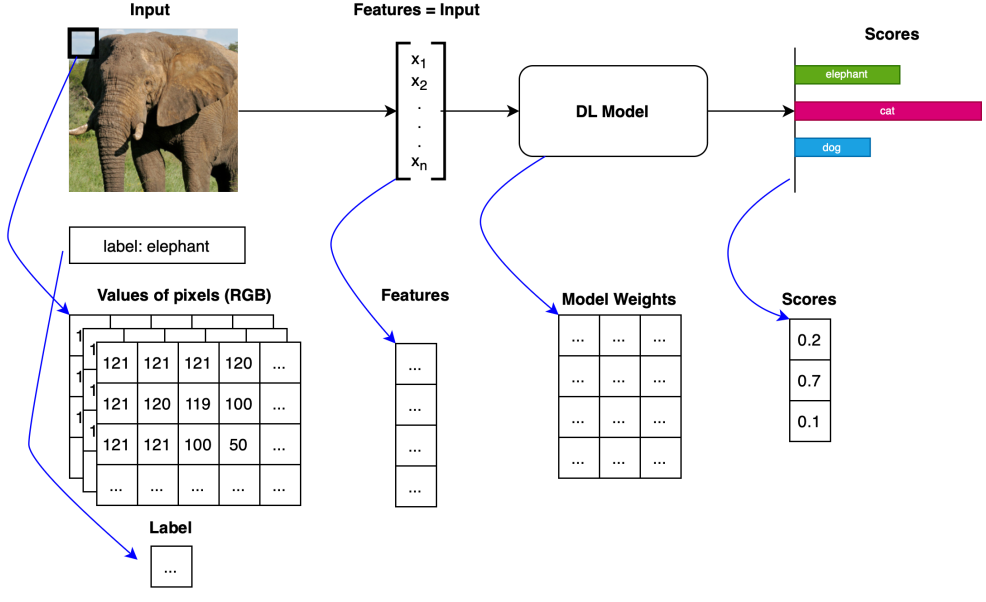
Part 03 - Data Representation with Tensors

Maura Pintor (maura.pintor@unica.it)

Representing data

Deep-learning systems have to be able to map input data to the outputs. To do so, they usually represent the input data with a specific format.

Deep Learning as Floating Point Numbers



We will understand soon what are these "boxes".

Deep Learning as Floating Point Numbers

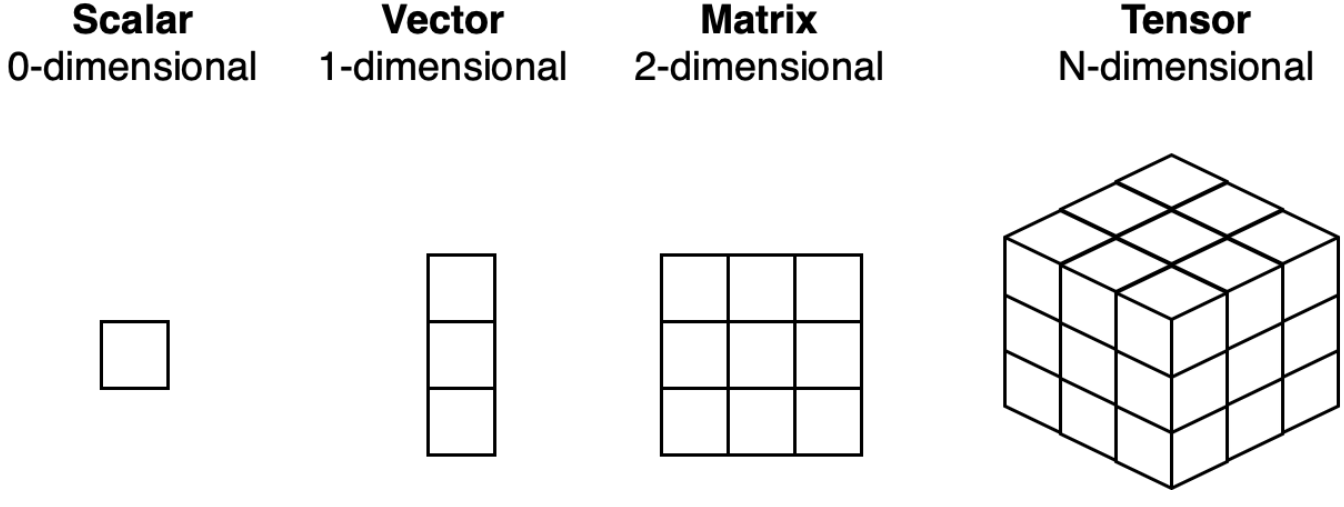
- **Pixels**: representations of the input images
- **Features**: internal representations of the model, similar inputs should have close representations
- **Model Weights**: matrices used to prioritize the inputs and extract the deep features
- **Scores**: output representations

How are these data units represented and stored?

Tensors of Floating Point Numbers

In the context of deep learning, a tensor is simply the extension of a vector that has an arbitrary numbers of dimensions.

tensors == multi-dimensional arrays



Tensors in PyTorch

`PyTorch` is not the only library that deals with n-dimensional arrays. `NumPy`, `SciPy`, `Scikit-learn`, `Pandas`, and other deep-learning libraries such as `Tensorflow` also support n-dimensional arrays.

Tensors in PyTorch

However, in `PyTorch`, the `Tensor` class is more powerful than standard numeric libraries.

- GPU support
- Parallel operations on multiple devices or machines
- Keep track of graph of computations that created them

All these features, especially the last one, are of utmost importance when dealing with deep learning!

Accessing Tensors and their Elements

Tensors are arrays, i.e., **data structures** that store a collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices (at most, one index for each dimension).

Practical session incoming

... prepare your interpreter ...

Environment manager: https://docs.conda.io/en/latest/miniconda.html

```
conda create -n deep-learning
conda activate deep-learning
conda install pip
pip install torch torchvision torchaudio
```

- Find the right line for installing `PyTorch` here: https://pytorch.org/get-started/locally/
- Other options are available (e.g., if you don't want to use `pip`)

Tensors vs Python lists

Creating a list and accessing one element.

```
l = [0, 1, 1]
print(l[0])
```

```
> 0
```

Creating a nested list and accessing one element.

```
nested_list = [[0, 1, 2], [1, 2, 3]]
print(l[0][1])
```

```
> 1
```

Tensors vs Python lists

```
import torch

t = torch.tensor([[0, 1, 2], [1, 2, 3]])

print(t[0, 1])  # indexing tensors (we will see more indexing tricks later)
```

```
> 1
```

Although on the surface this example doesn't differ much from a list of number objects, under the hood things are completely different.

How Tensors are Stored in Memory

- Python lists are collections of objects (also of different types) allocated and stored *individually* in memory
- PyTorch tensors are allocated *contiguously* in memory blocks containing C numeric types of 32-bit floats.

Indexing Tensors

```
import torch

t = torch.tensor([[0, 1, 2], [1, 2, 3]])

print(t[0, 1])  # indexing tensors (we will see more indexing tricks later)
```

The indexing operation does not create a *new* tensor by allocating memory and storing the values in it. That would be very inefficient, especially if we had millions of points.

PyTorch indexing directly references the original tensor.

Fancy Indexing

With tensors, we can use fancy indexing (like Numpy idexing).

```
x = torch.tensor([0, 1, 2, 3, 4])  # 1-d tensor
element = x[0]  # i-th element
first_elements = x[:1]  # from start to element 1
last_elements = x[1:]  # from element 1 to the end
some_elements = x[1:3]  # from element 1 to element 3
```

Works similarly with 2-d tensors (row and columns):

```
x = torch.tensor([[0, 1, 2], [1, 2, 3]])  # 2-d tensor
element = x[0, 0]
row = x[0, :]  # works also with x[0] in this case
column = x [:, 0]
some_rows = x[1:, :]  # from row 1 to the end, all columns
some_elements = x[1:2, :1]  # from row 1 to 2, from column 0 to 1
```

Of course, it works with n-d tensors as well!

Tensor element types

Why not using lists or Python numbers?

- The Python interpreter is slow compared to optimized, compiled code.
- PyTorch tensors provide low-level implementations of the data structures and high-level APIs for the operations.
- PyTorch Tensors keep track of the data type in their attribute `dtype`.
- Possible values of `dtype` are: `torch.float32, torch.float64, torch.int8, torch.uint8, torch.bool`[*], `...`

[*] particularly useful for indexing!

Handling (and changing) tensor dtypes

```
double_precision = torch.tensor([0, 1] dtype=torch.double)
print(double_precision.dtype)
short_tensor = double_precision.short()
print(short_tensor.dtype)
```

Tensor API

Your first source of information should be the PyTorch documentation.

- more complete
- more updated
- (it might also say something different than these slides!).

Basic Tensor operations

Creation operations and mutations

```
a = torch.ones(3, 2)  # 3x2 tensor of only ones
b = torch.zeros(3, 1)  # 3x1 tensor of only zeros
c = torch.zeros_like(a)  # same shape and type as a
a_t = a.t()  # 2x3 tensor (transpose of a)
print(a.shape)  # prints the shape (i.e., all the sizes of the dimensions)
```

Math operations

```
absolute_values = torch.abs(a)  # pointwise operations
mean_value = torch.mean(a)  # reduction operations
s = a + c  # element-wise sum
p = a * c  # element-wise product
z = torch.mm(a, c.t())  # matrix multiplication (careful with shapes!)
broadcasting = a + torch.tensor([1, 2])  # torch tries to match shapes
```

Tensor storage

We said that PyTorch tensors are stored in contiguous chuncks of memory. A `torch.Tensor` is an view of the storage instance that is capable of indexing into the storage through *offsets* and *strides*.

Note that multiple tensors can index the same storage but indexing into the data in a different way.

Tensor storage

```
a = torch.tensor([1, 2, 3, 4])
b = a[1]  # different Tensor, same storage (points to the same location)
c = a.reshape([2, 2])  # same storage, different stride
print(a.storage())
print(c.storage())
print(id(a.storage()) == id(c.storage()))  # same storage
print(c.stride())  # how many storage items to skip for incrementing each dimens
```

Remember: the underlying memory is allocated only once, which makes the view operation very lightweigth even for large storages.

Modifying stored values: in-place operations

In-place operations are used to modify directly stored values. The most used one is the `zero_`, that sets to zero all values. They can be recognized by the trailing underscore _ in their name.

```
a = torch.ones(3, 2)
a.zero_()  # in-place operation, does not create a new tensor
```

The methods that are not in-place, always return a new tensor.

Moving tensors to the GPU

Moving tensors to the GPU can make computations massively parallel and fast. Then, all the operations will be performed with GPU operations, while the API remains the same.

PyTorch supports all GPUs that have support for CUDA (Compute Unified Device Architecture), a software layer created by Nvidia.

An accelerated version of PyTorch is also available for Apple Silicon, but it is still not very stable.

Moving tensors to the GPU

Every PyTorch tensor has the attribute `device`, which says where the tensor data is placed in storage. Tensors can be "moved" (rather, copied) to another device by using the method `to`.

```
gpu_tensor = torch.zeros(1, device='cuda')  # created on the GPU
cpu_tensor = torch.zeros(1)
to_gpu = cpu_tensor.to(device='cuda')  # this creates a copy of the tensor!
to_gpu_another = cpu_tensor.cuda()  # shorthand for the previous command
again_to_cpu = to_gpu.cpu()  # shorthand for copying the tensor to cpu
```

If your machine has more GPUs, you can also specify which one to use, e.g., `cuda:0`. Note that operations can be performed only between tensors located on the same device.

Serializing tensors

Until now, we created tensors only in RAM. At some point, we will want to store a tensor in the persistent memory. PyTorch uses `pickle` to serialize the tensors. Here is how to store a tensor in memory.

```
torch.save(a, 'tensor.pth')  # note that the extension is arbitrary
```

And to load back the tensor, a similar API is available.

```
b = torch.load('tensor.pth')  # note that the extension is arbitrary
```

End of part 3

Summary:

- Data Representation
- N-dimensional Tensors
- Operations with tensors
- Tensor storage
- GPUs
- Serialization

End of part 3

In the next chapter:

- Data representation with PyTorch
- Building the first DNN
- Training the first DNN

Maura Pintor (maura.pintor@unica.it)