PyTorch

Laboratory Lecture for the “**Intelligent Systems for Pattern Recognition**” course of the Master’s Degree in Computer Science (2024–2025)

Riccardo **Massidda**
riccardo.massidda@di.unipi.it

Huge thanks to Valerio *De Caro* and Antonio *Carta* for previous versions of this material.

Why PyTorch?

Tensor Manipulation.

Tensor operations on a MATLAB/NumPy-like API.

Accelerator Support.

Seamless execution on CPU, GPU, and TPU devices.

Automatic Differentiation.

Only need to define forward computation → chain rule! ⛓️

High-Level API.

Readily available neural networks layers, losses, optimizers, ...

Getting Started

For **this lecture**:

1. Clone the repository di-unipi/ispr-lab from GitHub,
2. Install PyTorch, either using an environment manager (conda, pipenv, poetry, etc.) or using Docker/**Podman** 🐳.

In a hurry? Just open the repository in Google Colab!

Up-to-date instructions to install **PyTorch** here: Start Locally | PyTorch

Basics of Tensor Operations and Manipulation

Tensors

Tensors are the main data structure and represent **multidimensional arrays**.

As for NumPy arrays, they support advanced **indexing** and **broadcasting**.

Attributes:

- **dtype**: determine the type of the tensor elements (float{16, 32, 64}, int{8, 16, 32, 64}, uint8). Can be specified during the initialization.
- **device**: memory location, as in CPU or GPU
- **layout**: dense tensors (`strided`) or sparse (`sparse_coo`)

Tensor Initialization

- **Existing Array:** `torch.tensor(list)`
- **Constants:** `torch.zeros(*dims)`, `torch.ones(*dims)`
- **Random:** `torch.randn(*dims)`, `torch.rand(*dims)`
- **Range:** `torch.linspace(start, end, steps=100)`
- **NumPy:** `torch.from_numpy(arr)`

Tensor Operations

Some operators are **overloaded**:

- `+`, `-` for addition and subtraction (support broadcasting)
- `*` is the elementwise multiplication (not the matrix product, supports broadcasting)
- `@` for matrix multiplication (`torch.matmul`)

In-place operations are defined with a suffix underscore:

- `add_`, `sub_`, `matmul_` are the in-place equivalent for the previous operators, and also support broadcasting.

Check the documentation: http://pytorch.org/docs/stable/tensors.html

Broadcasting Rules

PyTorch broadcasting semantics follows NumPy own semantics.

Two tensors are “**broadcastable**” if the following rules hold:

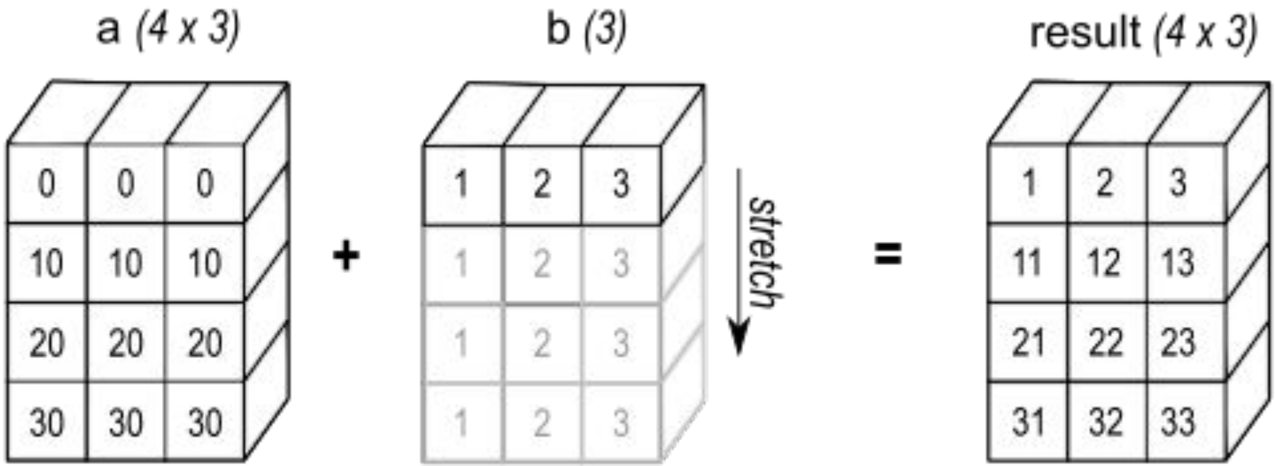
1. Each tensor has **at least one dimension**.
2. When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be **equal**, one of them is **1**, or one of them **does not exist**.

Broadcasting Rules

```
>>> x=torch.empty(5,7,3)
>>> y=torch.empty(5,7,3)
# same shapes are always broadcastable (i.e. the above rules always hold)

>>> x=torch.empty((0,))
>>> y=torch.empty(2,2)
# x and y are not broadcastable, because x does not have at least 1 dimension

# can line up trailing dimensions
>>> x=torch.empty(5,3,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist

# but:
>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3
```

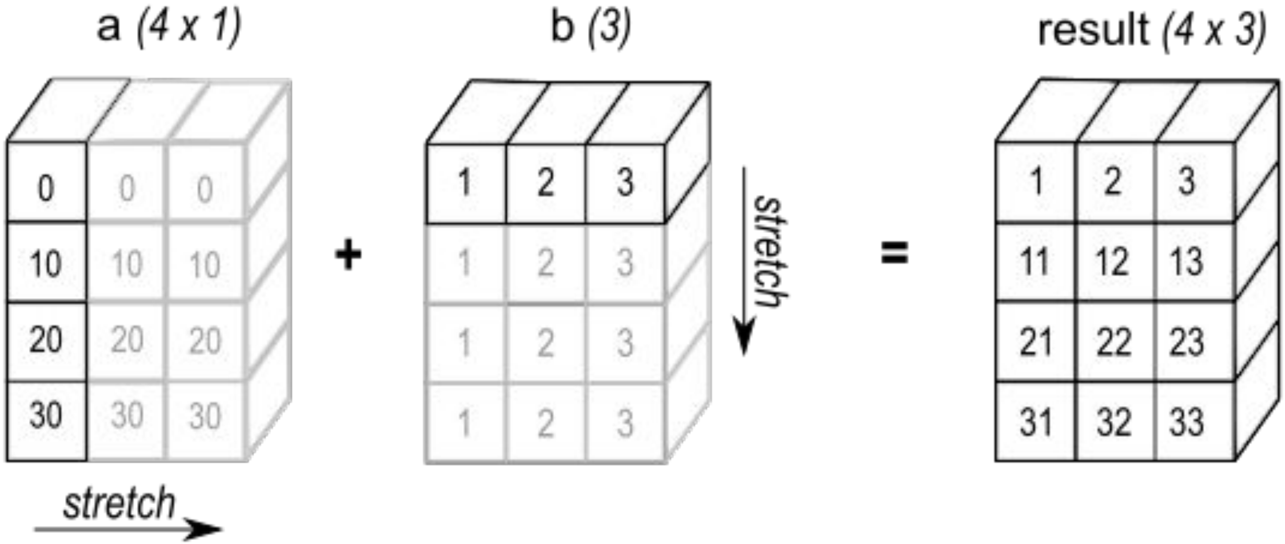
https://pytorch.org/docs/stable/notes/broadcasting.html

Broadcasting Rules



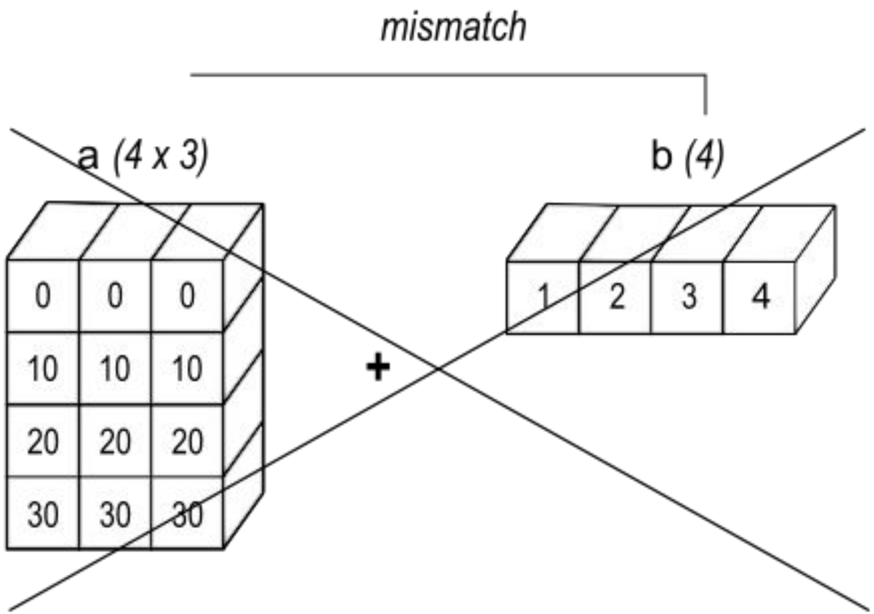
https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting Rules



https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting Rules



https://numpy.org/doc/stable/user/basics.broadcasting.html

Tensors in GPU

The **submodule** `torch.cuda` provides the API for **GPU management**.

Check availability of the GPU:

```
torch.cuda.is_available
```

Create or move to GPU:

```
torch.tensor([2., -1.], device="cuda")
tensor.to("cuda")
```

In all operations, all the tensor must **reside on the same device** and result on the same device.

You can move tensors back to the CPU with the `tensor.cpu()` method.

Tensors in GPU

On a server, you typically have access to **multiple shared GPU** and you must select one:

1. **Manually** selecting with the device argument (‘`cuda:0`’, ‘`cuda:1`’...), or
2. Using the **context manager** `torch.cuda.device`

Changing the shell environment variable `CUDA_VISIBLE_DEVICES` to limit the visible GPUs

```
export CUDA_VISIBLE_DEVICES=0
```

Note that the indices of the GPU IDs will always start from 0.

⚠ Remember to de-allocate tensors from the GPU if you’re not using it!

Tensor Indexing

Basic tensor **indexing** is similar to list indexing, but with multiple dimensions.

Boolean arrays can be used to filter elements that satisfy some condition.

If the indices are less than the number of dimensions, the **missing indices** are considered **complete slices**.

```
# first k elements
x = arr[:k]
# all but the first k
x = arr[k:]
# negative indexing
x = arr[-k:]
# mixed indexing
arr[:t_max, b:b+k, :]

# indexing with Boolean condition
def relu(x):
    x[x < 0] = 0
    return x
```

Tensor Reshaping

Reshaping is fundamental to combine tensors.

`tensor.squeeze()` removes all singleton dimensions

`tensor.unsqueeze(dim)` add a singleton dimension at the provided dimension

`tensor.transpose(dim1, dim2)` transposes the two dimensions of the tensor

`tensor.permute(*dims)` re-arranges the dimensions as in *dims

```
x = torch.randn(5,1,5)

# squeeze
x.squeeze() → [5,5]

# unsqueeze
x.unsqueeze(3) → [5,1,5,1]

# transpose
x.transpose(1, 2) → [5,5,1]

# indexing with Boolean condition
x.permute(1,0,2) → [1,5,5]
```

Tensor Reduce

Reduction operations collapse the tensor dimensionality.

```
tensor.sum(dim)
tensor.mean(dim)
tensor.prod(dim)
tensor.amin(dim)
tensor.amax(dim)
```

The `keepdim` parameter keeps an empty dimension in place.

```
x = torch.randn(5,1,5)

x.sum(0) → [1, 5]
x.mean(1) → [5, 5]
x.amin(2) → [5,1]
```

Your Turn!

The **Kaiming** uniform initialization scheme provides a standard baseline to train Neural Networks with rectified activation functions.

$$\mathcal{U}\left(-\sqrt{\frac{6}{\text{in\_size}}}, \sqrt{\frac{6}{\text{in\_size}}}\right)$$

Write the following functions:

`_relu_kaiming_init_(weights: torch.Tensor)` that modifies **in-place** the provided tensor,

`_relu_kaiming_init(in_size, out_size)` that **returns** a new tensor with shape `(out_size × in_size)`

Autograd
Automatic Differentiation in PyTorch

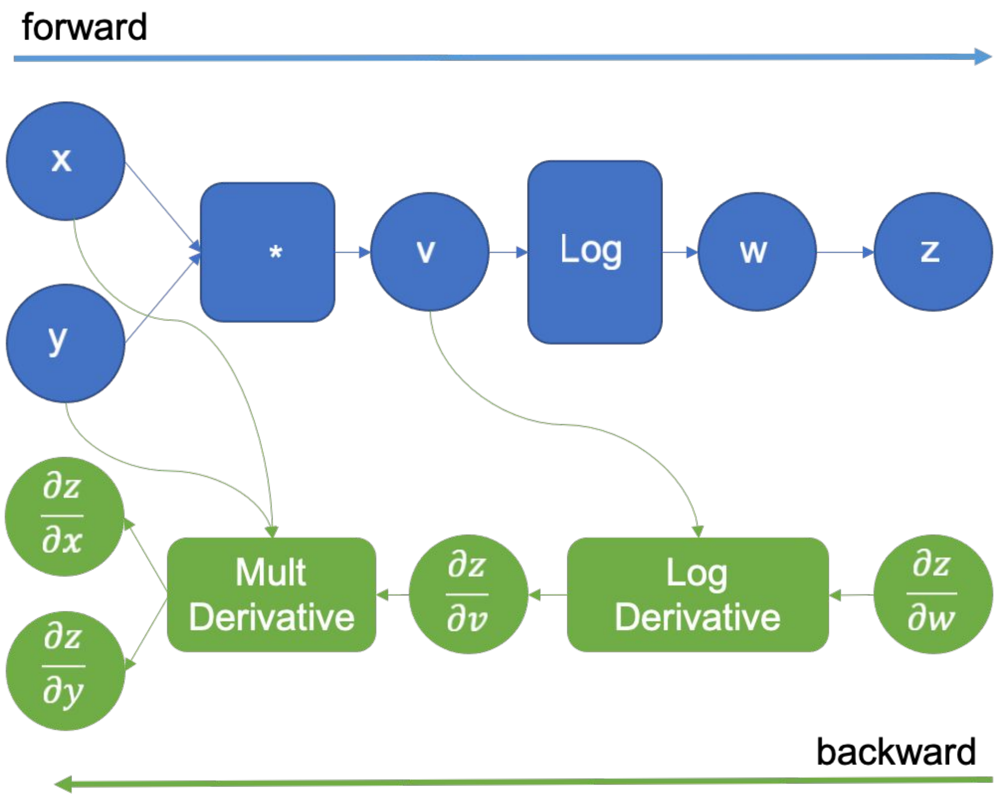
Autograd

The **submodule** `torch.autograd` is responsible for **automatic differentiation**.

Each operation creates a `Function` node in a dynamic computational graph, connected to its `Tensor` arguments.

The gradient is computed on each tensor by calling the `backward()` method.

Autograd



[Overview of PyTorch Autograd Engine]

Autograd

The main **Tensor attributes** related to the graph structure are:

- **data**: Tensor containing the data itself
- **grad**: Tensor containing the gradient (initially set to None)
- **grad_fn**: the function used to compute the gradient

Each **Function** implements two methods:

- **forward**: function application
- **backward**: gradient computation

Autograd

The **requires_grad** attribute is used to specify if the gradient computation should propagate into the Tensor or not, which also stops the **backpropagation**.

For optimizable model parameters ⇒ `requires_grad=True`

For input data or constant values ⇒ `requires_grad=False`

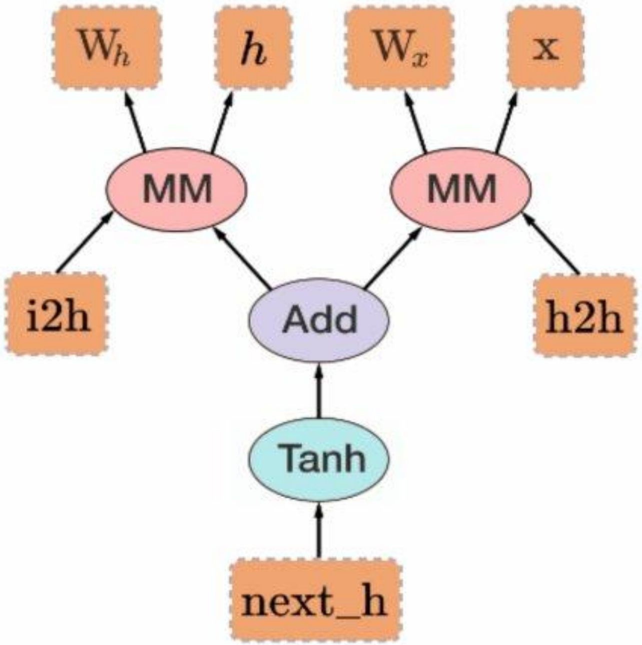
The method detach removes the tensor from the graph, truncating the gradient.

In-place modification is not allowed, as it breaks the automatic differentiation.

At inference time, the **context manager** `torch.no_grad` speeds up computation.

Autograd documentation: http://pytorch.org/docs/stable/autograd.html

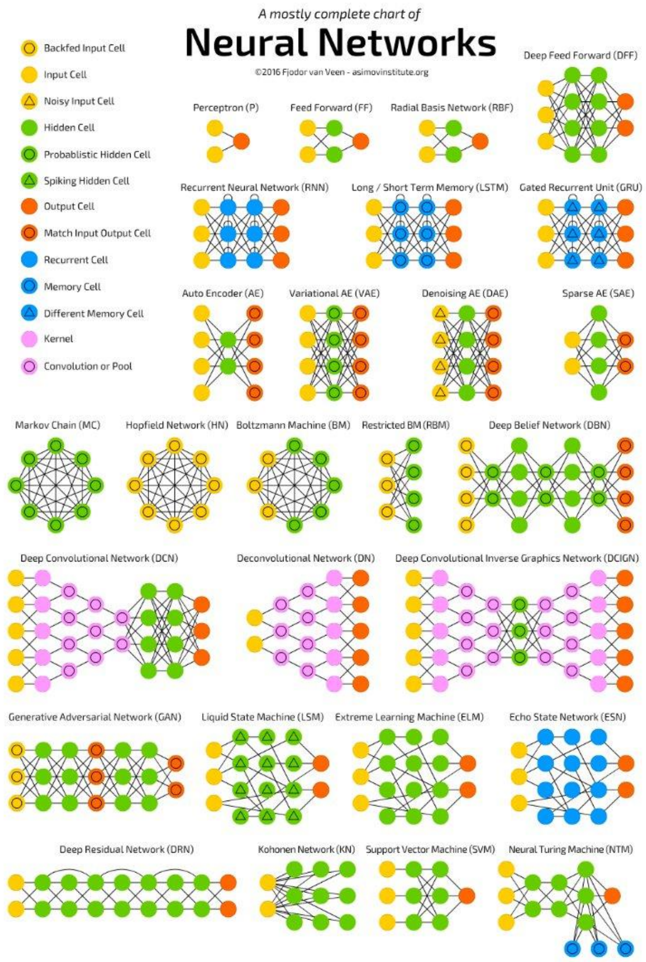
Autograd



Building Models and Pipelines

Model Interface

torch.nn contains the basic components to define your neural networks, loss functions, regularization techniques and optimizers.



Module and Parameters

Module is the base class for all the neural network components: Linear, Convolutional, Recurrent Layers...

Each Module contains a set of **Parameter** objects, i.e., a “tensor with a name and `requires_grad=True`”

The **`parameters()`** method returns an iterator over model parameters.

If you want to add a list of parameters or sub-modules, you can use the **ParameterList** and **ModuleList** objects.

⚠ If you use a regular list, the parameters will <u>not</u> be registered!

Forward and Backward

The logic of the module is defined in the **`forward()`** method, which you can call either as `net(in_tensor)` or `net.forward(in_tensor)`.

The **`backward()`** step is automatically defined by Autograd, but can be overridden!

It is possible to define forward and backward hooks to debug your model!

Modules can operate in **train** or **eval mode**: `net.train()` or `net.eval()`

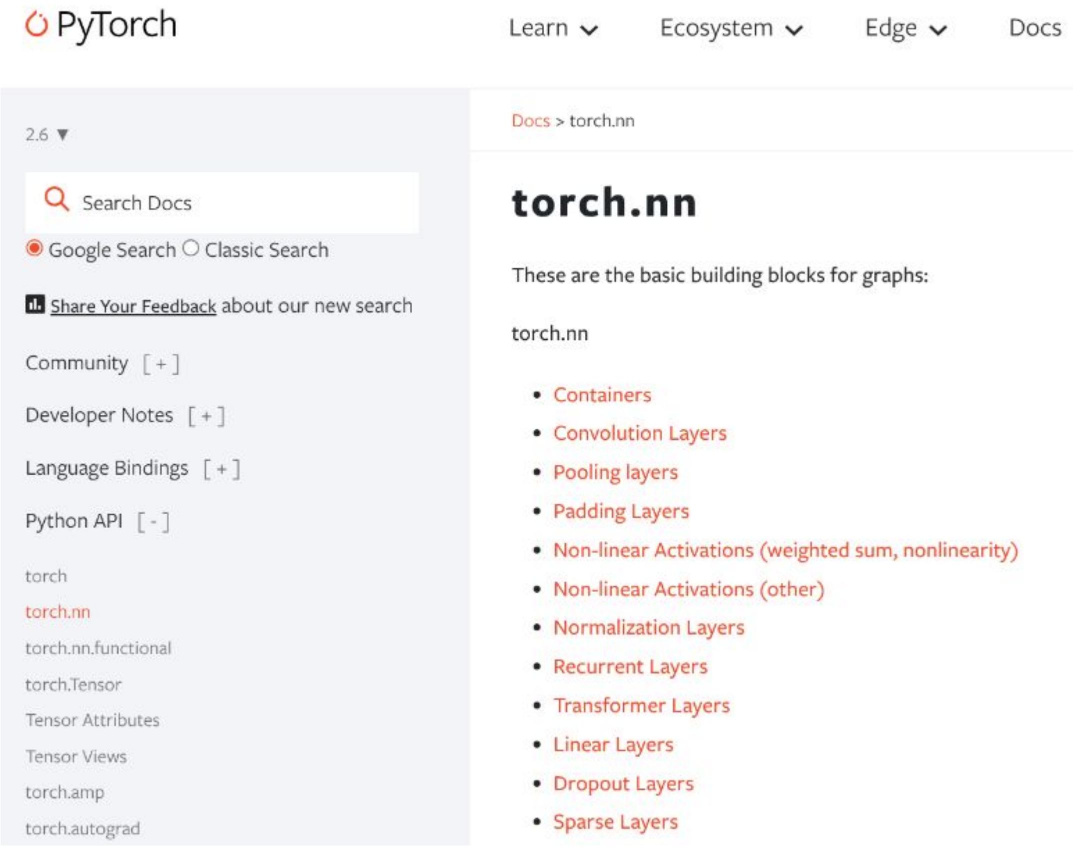
This is useful for layers that define a different behavior during train and test, e.g. Dropout, BatchNormalization...

Existing Modules

There's no need to reinvent the wheel!

(in most cases, but sometimes you really do: good luck)

PyTorch provides lots of **common modules** that can be easily glued together.

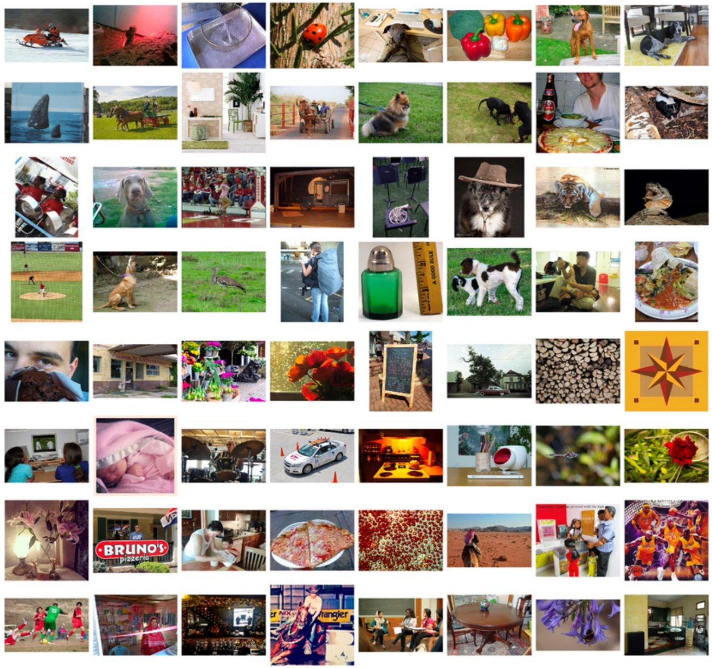


https://pytorch.org/docs/stable/nn.html

Datasets and Data Loaders

The module **`torch.data.utils`** defines classes to handle datasets and load them from data.

DataLoader automatizes mini-batching, shuffling of the dataset, sampling techniques and any pre-processing, and allows parallel loading.



Training Loop

To define a training loop, we need a **loss function** and an **optimizer**.

Always check the **documentation** for the correct **shape** and **input** arguments (does the loss need logits or probabilities? Which dimension should be the last? Is the average for each element or for each sample?)

⚠ Remember to reset gradients using the `zero_grad()` method!

(*less talk, more code*)

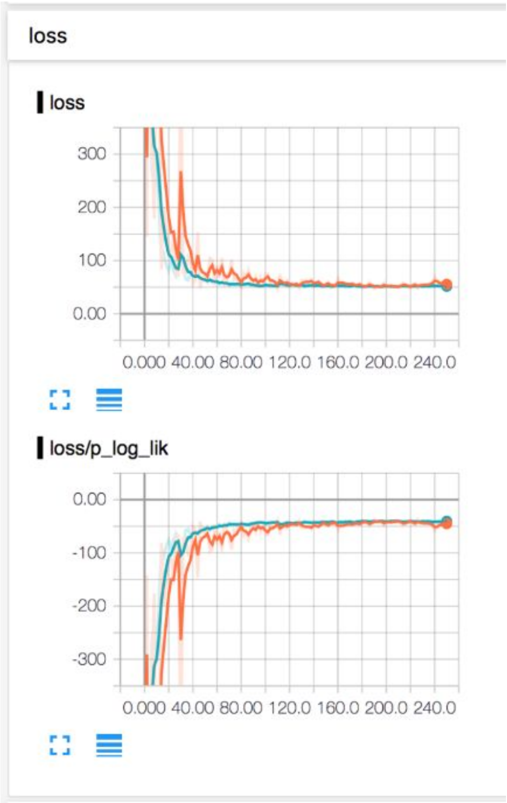
Logging

Several **metrics** can help to understand your model.

Logging them, it's always a good idea!

TensorBoard works great for PyTorch as well.

Otherwise, there are cloud-based commercial products (Weights & Biases, neptune.ai, ...)



Model Serialization

Last, but not least, how do I store my model?

The **state dictionary** stores the value of all model **parameters**.

```
torch.save(the_model.state_dict(), PATH)
```

Then, instantiate the object and reload the state dictionary.

```
net = MyModelClass(*args, **kwargs)
net.load_state_dict(torch.load(PATH))
```

PyTorch Ecosystem

To know how things work **under-the-hood** is worth the effort.

... but in practice, most “routine” operations can be abstracted away.

Both PyTorch **Lightning** and **Transformers** by HuggingFace 🤗 provide APIs for common practices such as training, logging, evaluating and performing inference on Machine Learning models.

Also, tons of libraries in the **PyTorch Ecosystem**: graph neural networks, interpretability, continual learning, federated learning, quantum ML...

Your Turn!

Implement and train a **Convolutional Neural Network** to perform image classification on the **MNIST dataset**.

📜 Side Quests:

1. Monitor the performance with a logger,
2. Play around with `dropout`, `batch_norm`, etc.
   (remember of train vs eval!)
3. Try a PyTorch Lightning implementation