Università degli
Studi di Milano-Bicocca

# Introduction to Pytorch

Prof. Flavio Piccoli

a.a. 2022-2023

# R&D process

Analysis of the state of art
↓
Data collection / analysis
↓
**Architectural Design**
↓
Parameter Search
↓
Benchmarking
↓
Deployment

Google Scholar

pandas

scikit learn

PyTorch

PyTorch Lightning

RAY

tune

Streamlit

Flask

ONNX

# Introduction to Pytorch

- machine learning framework based on the Torch library
- originally developed by Meta AI and now part of the Linux Foundation umbrella
- It is free and open-source software released under the modified BSD license
- Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.[12]

# Basic operations

- Convert a list in tensor

    mat = torch.tensor([1,2,3,4,5])

- Get the shape of a tensor

    mat.size() or mat.shape

```
torch.Size([5])
```

- Add a dimension to the tensor

    mat = mat.unsqueeze(0)

```
torch.Size([1, 5])
```

- Remove singleton dimensions

    mat = mat.squeeze()

# Autograd

- Pytorch automatically performs derivatives for you
- Example: evaluate the derivative of $y = x^2$ in the point x=3

```
import torch
import torch.nn as nn

# define a number
x = torch.tensor([3.0])

# ask to compute gradients for that variable
x.requires_grad = True

# do an operation on the number
y = x**2

# perform backpropagation
y.backward()

# obtain gradients
print(x.grad)
```

It will print 6, because $y' = 2x$ and $x = 3$

# Training procedure

The pseudocode corresponding to the full training procedure is:

1. initialize dataset (load filenames and divide in batches)
2. define predictive model
    a. initialize weights
3. define optimizer

Initial setup

4. for each epoch (or until convergence)
    a. for each batch in the dataset:
        i. compute forward propagation to get output
        ii. compute loss
        iii. compute gradients (back-propagation)
        iv. update weights of model

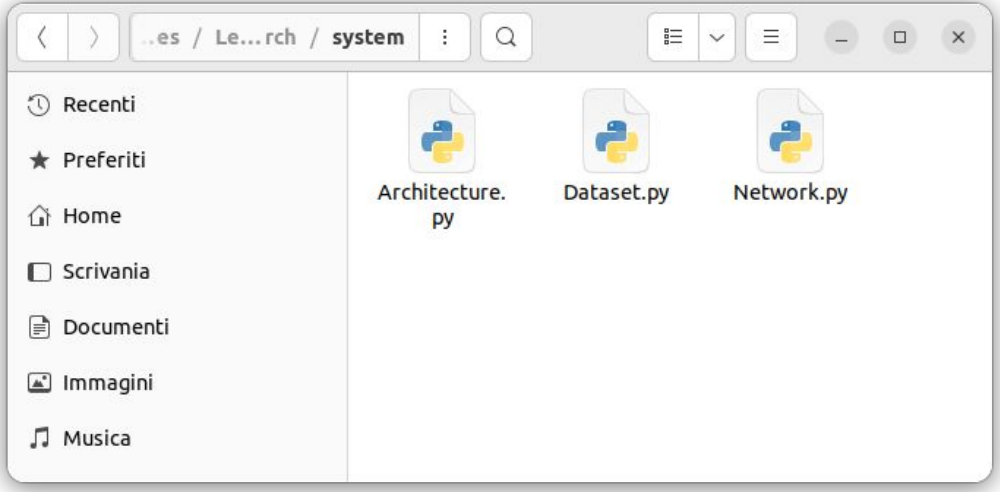
1-epoch training

    b. validate model
    c. if best model, save it

testing and checkpointing

# File structure

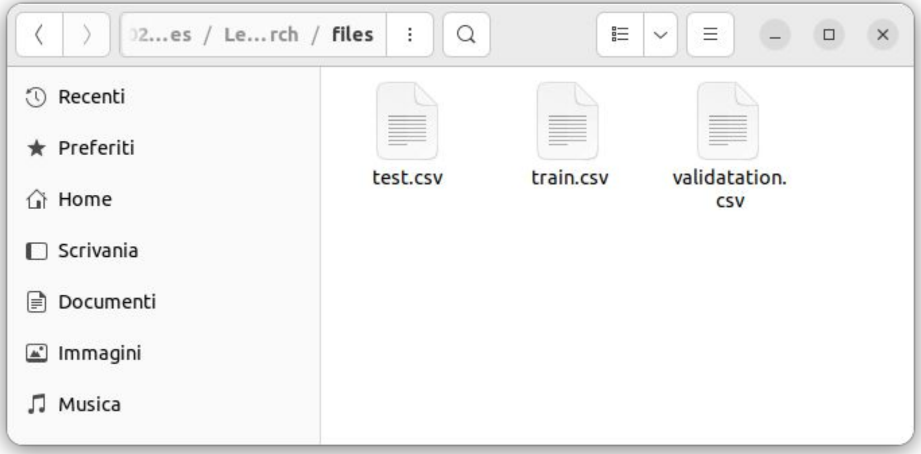
- It is a good practice to split in separate files:
  - data loading
  - network definition
  - architecture
- By doing so, each components can be tested separately without starting the whole system



# Data loading for CSV files

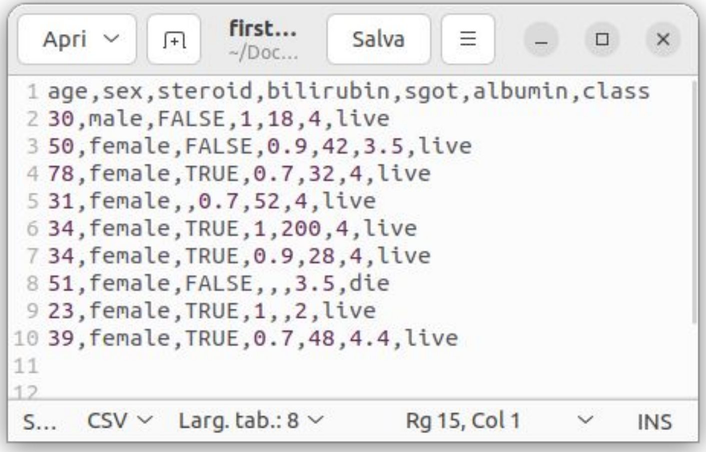
# Data structure for CSV files

Data is splitted in three files, train, validation and test




Each CSV file is has:

- the first row containing the headers (opt.)
- the other rows are samples of the dataset



# Data structure for CSV files

```
class Dataset(torch.utils.data.Dataset):

    def __init__(self, csv):
        # here i will read my CSV file
        self.df = pd.read_csv(csv)
```

Initialization of the dataset

- read the csv file

```
    def __len__(self):
        # here i will return the number of samples in the dataset
        return len(self.df)
```

Counting of the samples

- return the number of samples

```
    def __getitem__(self, idx):
        # here i will load the file in position idx
        cur_sample = self.df.iloc[idx]
        # split in input / ground-truth
        cur_sample_x, cur_sample_y = split_input_output(cur_sample)
        # return values
        return cur_sample_x, cur_sample_y
```

Load and return item

- read csv line
- split input and ground-truth variables

# Dataset usage

For a fast checking that everything is correct you can load a sample:

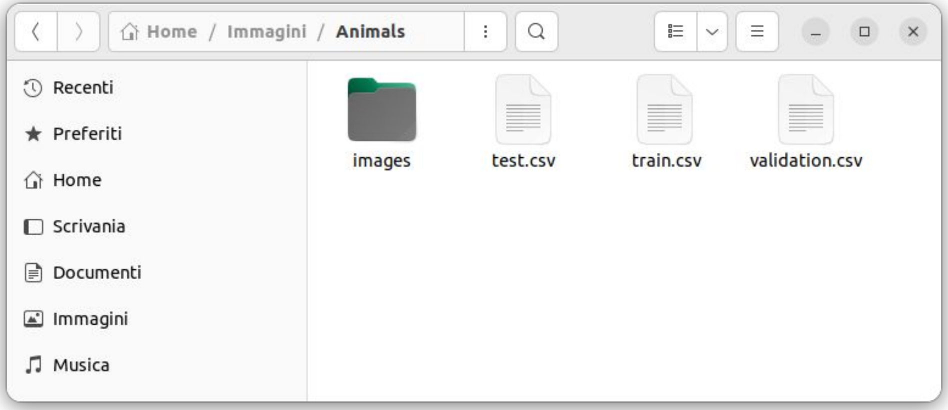
```
# init the dataset
ds = Dataset('/path/to/csv_file.csv')

# print number of samples
print(ds.__len__())

# load a sample to check that everything is fine
sample = ds.__getitem__(3)

# print it
print(sample)
```

# Data loading for image datasets

# Data structure for images

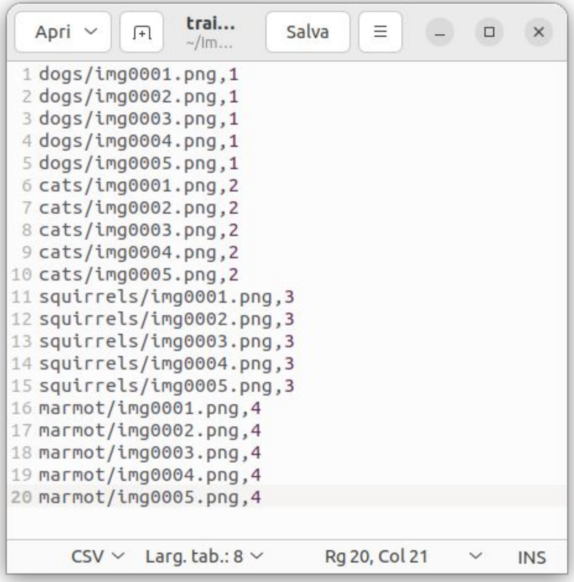
Generally, the dataset folder contains:

- a folder containing images
- three csv files containing the list of files
  - train
  - validation
  - test




Every line of the CSV contains:

- the path of the image
- the belonging class (ground truth)



# Dataset for images

```
import os
import torch

class Dataset(torch.utils.data.Dataset):

    def __init__(self, csv):
        # here i will read my file list
        self.fns = ...


    def __len__(self):
        # here i will return the number of samples in the dataset
        return len(self.fns)


    def __getitem__(self, idx):
        # get current sample
        cur_sample = self.fns[idx]
        # split filename from class
        cus_fn, cur_class = cur_sample.split(',')
        # load the image
        loaded_file = load_file(cur_fn)
        # return image and ground-truth
        return loaded_file, cur_class
```

# Dataset for images

```
import os
import torch

class Dataset(torch.utils.data.Dataset):
```

```
    def __init__(self, csv):
        # here i will read my file list
        self.fns = ...
```

Initialization of the dataset

- read file list

```
    def __len__(self):
        # here i will return the number of samples in the dataset
        return len(self.fns)
```

Counting of the samples

- return the number of samples

```
    def __getitem__(self, idx):
        # get current sample
        cur_sample = self.fns[idx]
        # split filename from class
        cus_fn, cur_class = cur_sample.split(',')
        # load the image
        loaded_file = load_file(cur_fn)
        # return image and ground-truth
        return loaded_file, cur_class
```

Load and return item

- get string containing filename and gt
- separate filename from gt
- load image
- return image and ground-truth

# A simple example

Create a dataloader with 2 inputs:

- a CSV file with filenames
- a directory containing images

The dataloader must:

- read the csv in __init__
- in __getitem__:
  - split filename from gt
  - load image

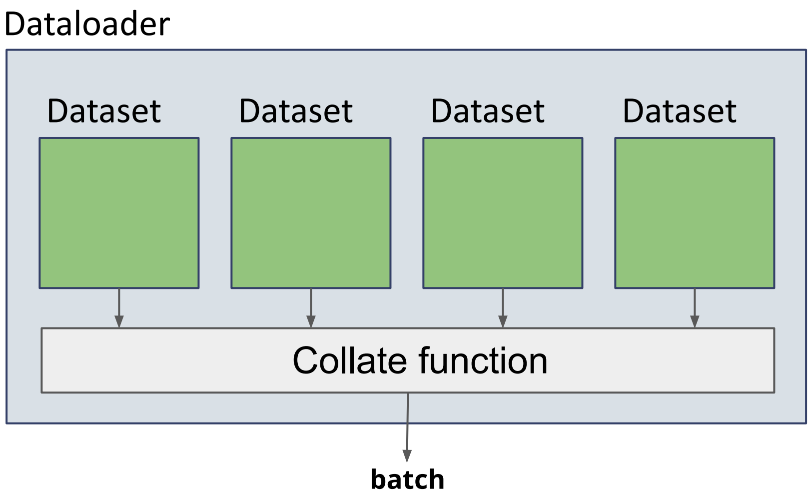
```python
import os
import torch
from torchvision import transforms
import numpy as np
from PIL import Image


class AnimalDataset(torch.utils.data.Dataset):

    def __init__(self, root_dir, csv_file):
        # save root directory
        self.root_dir = root_dir
        # read file
        f = open(csv_file, "r")
        txt = f.read()
        f.close()
        # get filenames
        self.fns = txt.split('\n')


    def __len__(self):
        return len(self.fns)


    def __getitem__(self, idx):
        # get current sample
        cur_sam = self.fns[idx]
        # split filename from class
        cur_fn, cur_gt =  cur_sam.split(',')
        # appen root path
        cur_fn = os.path.join(self.root_dir, cur_fn)
        # open image
        cur_img = Image.open(cur_fn)
        # convert to pytorch
        cur_img = transforms.ToTensor()(cur_img)
        # return
        return cur_img, cur_gt
```

# Parallel data loading

Parallelize your dataset to load multiple instances at once

- create n copies of your dataset
- uses each copy to read a sample
- joins the loaded samples in collate function

```
# init dataset
dataset = AnimalDataset('images','train.csv')

# define dataloader
dataloader = torch.utils.data.DataLoader(
                dataset,
                batch_size=80,
                drop_last=False,
                shuffle=False,
                num_workers=8
            )

# use dataloader
for cur_images, cur_gts in dataLoader:
    # for each batch do something
    print(cur_images.shape)
```

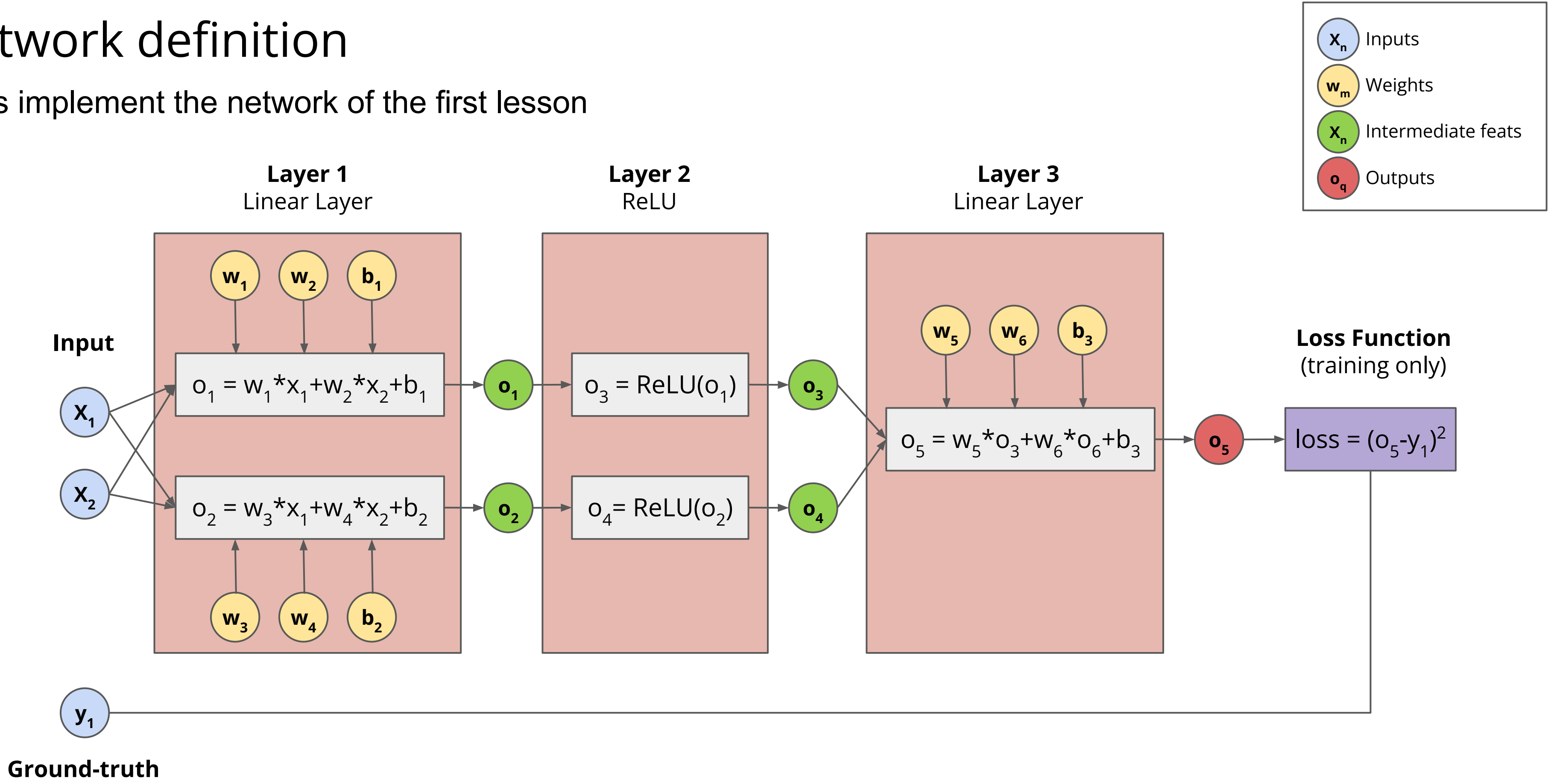



- defined with torch.utils.data.DataLoader
- you need to specify:
  - *batch_size*: dimension of the batch
  - *num_workers*: number of threads
  - *drop_last*: last batch must be dropped if uncomplete
  - *shuffle*: shuffle instances at each epoch

# Network definition

# Network definition

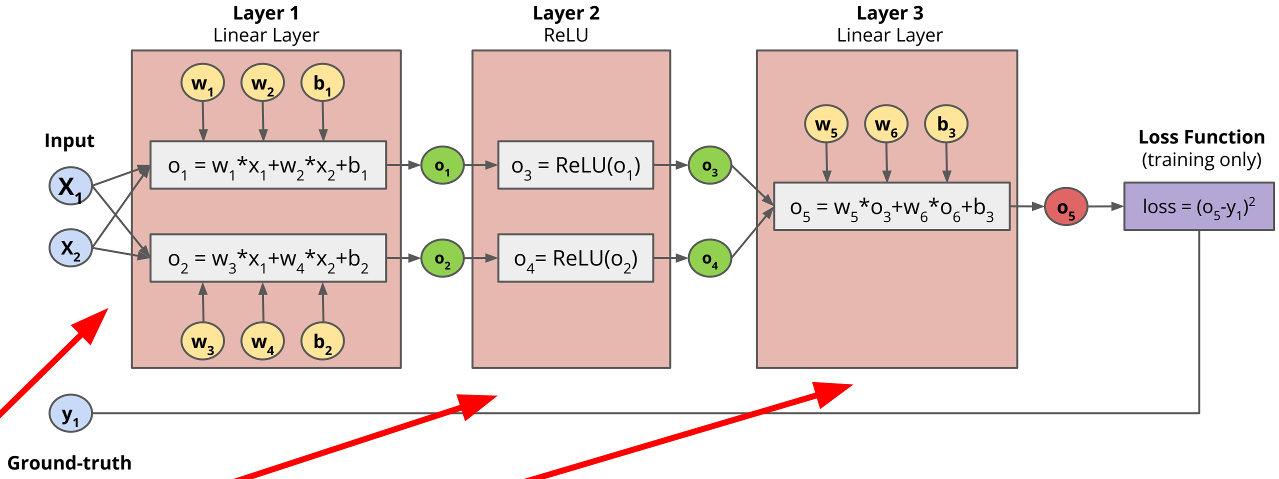
Let's implement the network of the first lesson



# Network definition

Let's implement the network of the first lesson

```python
import os
import torch
import torch.nn as nn


class Network(nn.Module):

  def __init__(self):
    # initialize super class
    super(Network, self).__init__()

    # create first layer
    self.layer1 = nn.Linear(2,2)
    # create second layer
    self.layer2 = nn.ReLU()
    # create third layer
    self.layer3 = nn.Linear(2,1)


  def forward(self, x):
    # apply layers in cascade
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    # return output
    return x
```




The network definition is performed by extending the nn.Module class. Must have two methods:

- __init__
- forward

nn.Linear(**n_inp**, **n_out**)

# Network definition

```
import os
import torch
import torch.nn as nn


class Network(nn.Module):

 def __init__(self):
    # initialize super class
    super(Network, self).__init__()

    # create first layer
    self.layer1 = nn.Linear(2,2)
    # create second layer
    self.layer2 = nn.ReLU()
    # create third layer
    self.layer3 = nn.Linear(2,1)

 def forward(self, x):
    # apply layers in cascade
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    # return output
    return x
```

Initialization of the network

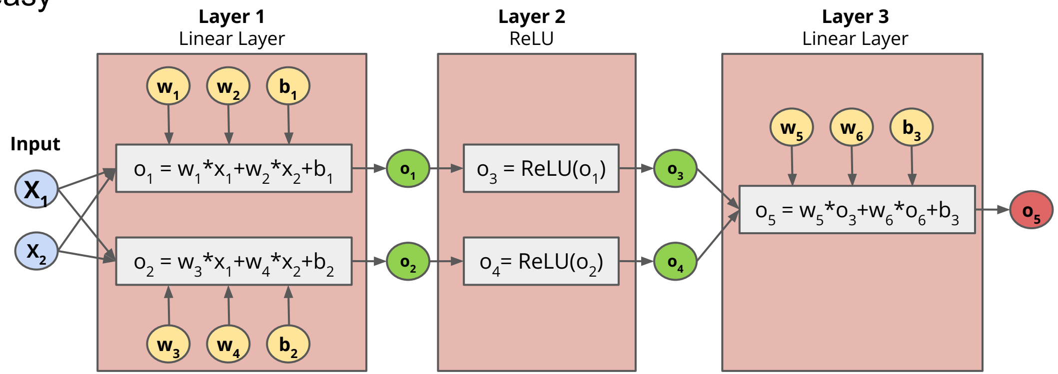
- define layers

Use of the network

- define how to use the layers
- from input to output

# Example of network usage

Instantiating and using a network in Pytorch is extremely easy




```
# initialize network
net = Network()

# create fake input
inp = torch.rand(20, 2)

# apply network
out = net(inp)

# print output size
print(out.size())
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                [-1, 20, 2]               6
              ReLU-2                [-1, 20, 2]               0
            Linear-3                [-1, 20, 1]               3
================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

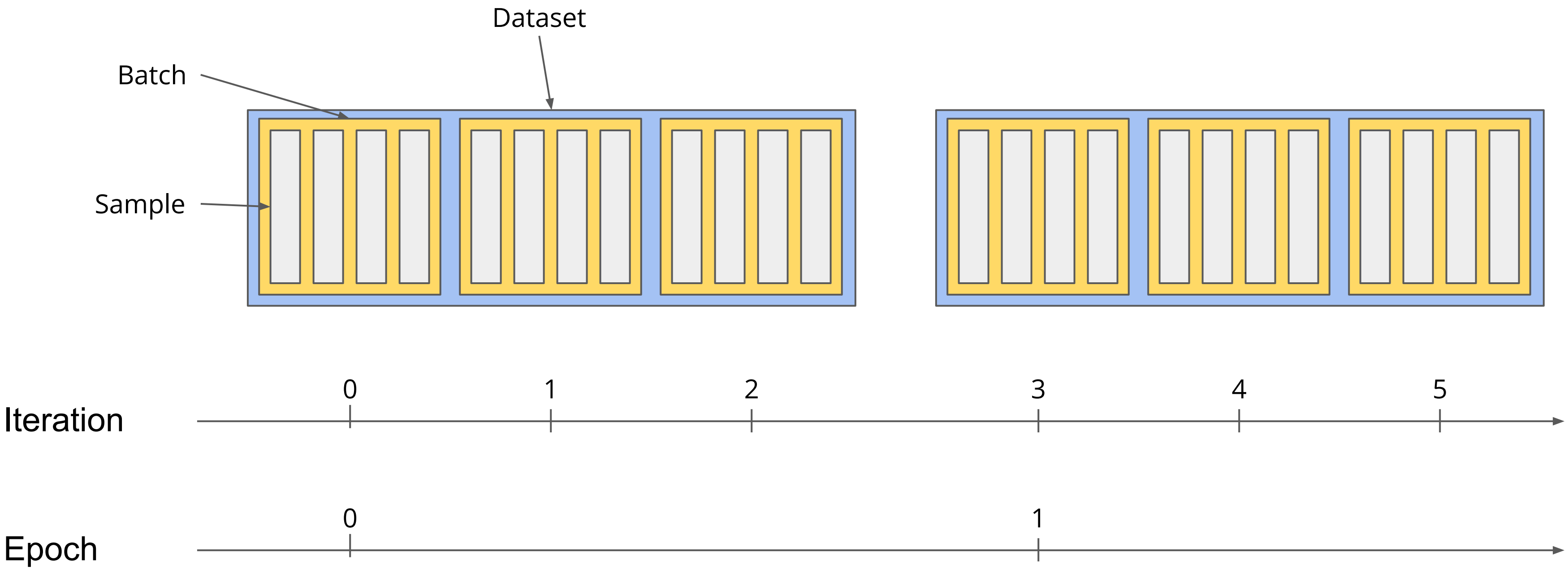
**Applying the network automatically calls forward() function**

# System definition

# Terminology

**Iteration** = processing of one batch

**Epoch** = processing of the entire training set



# Initial setup

```
import os
import torch
from Network import Network
from Dataset import Dataset
```

```
# initialize datasets
train_dataset = Dataset('./dataset/train.csv')
val_dataset = Dataset('./dataset/validation.csv')

# create training dataloader
train_dataloader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size = 20,
    drop_last = True,
    shuffle = True,
    num_workers = 8
)

# create validation dataloader
val_dataloader = torch.utils.data.DataLoader(
    val_dataset,
    batch_size = 20,
    drop_last = False,
    shuffle = False,
    num_workers = 8
)
```

Definition of the datasets and dataloaders

```
# create network
net = Network()

# define optimizer
optimizer = torch.optim.SGD(params=net.parameters(), lr=0.001)
```

Definition of the network and the optimizer

# Training cycle

```
# for each epoch
for cur_epoch in range(1000):

    # for each batch
    for inp, gt in train_dataloader:

        # reset gradients
        optimizer.zero_grads()
```

1. Reset grads

```
        # forward
        out = net(inp)
```


2. Forward

```
        # compute loss
        loss = F.mse_loss(out, gt)
```


3. Loss

```
        # backward
        loss.backward()
```


4. Backward

```
        # update weights
        optimizer.step()
```

5. Update weights

# Saving objects on disk (single object)

- With Pytorch is very easy to save stuff on disk
    - torch.save(tensor or dictionary, filename)
    - torch.load(filename)

```
# define a tensor
mat = torch.tensor([1,2,3,4,5])
# save it to disk
torch.save(mat, 'mat.pth')
# reload it
new_mat = torch.load('mat.pth')
```

# Saving objects on disk (dictionary of objects)

- We can also save dictionaries with Pytorch
    - good for saving multiple tensors in a single file

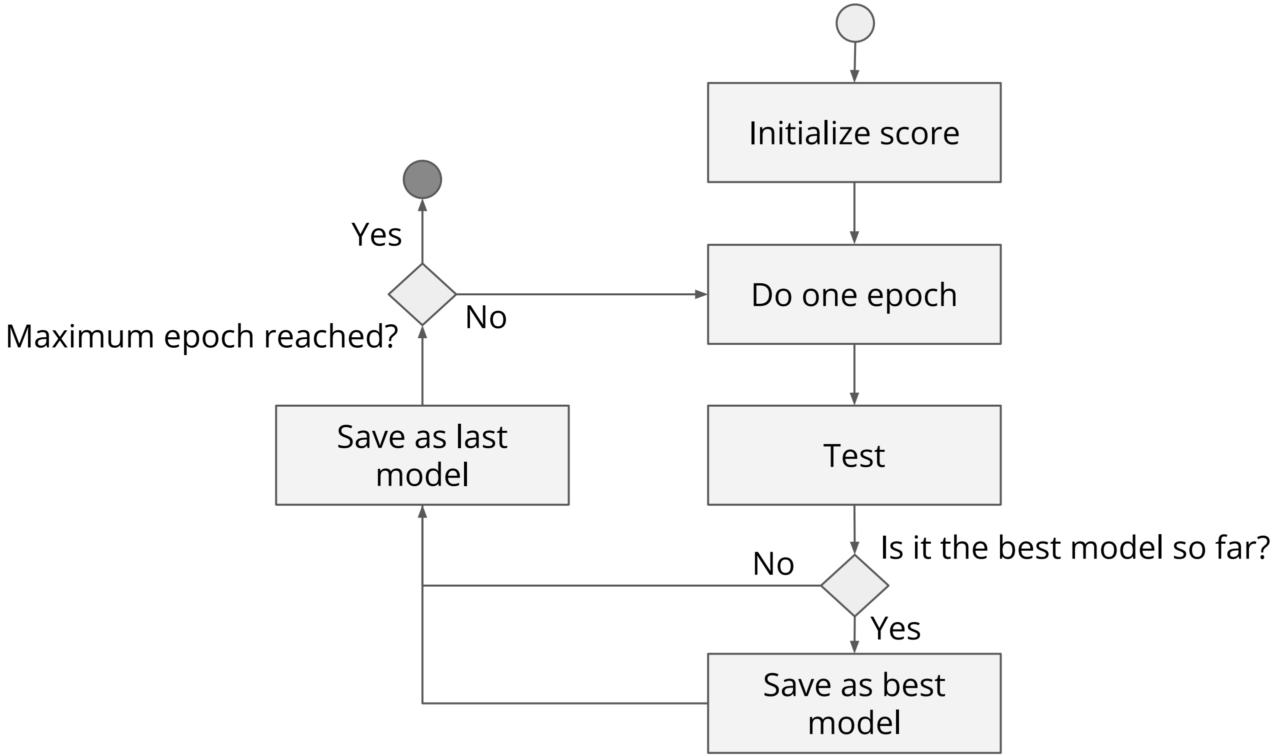
```
# define two tensors
mat1 = torch.tensor([1,2,3,4,5])
mat2 = torch.tensor([6,7,8,9,10])

# create dictionary
d = {
    'mat1': mat1,
    'mat2': mat2,
    'int1': 3,
    'string1': 'ciao',
}

# save it to disk
torch.save(d, 'dict.pth')

# reload it
new_d = torch.load('dict.pth')

# use a variable
new_mat1 = new_d['mat1']
```

# Saving network and optimizer state on disk

- We will use the saving function to save the state of the network and the optimizer

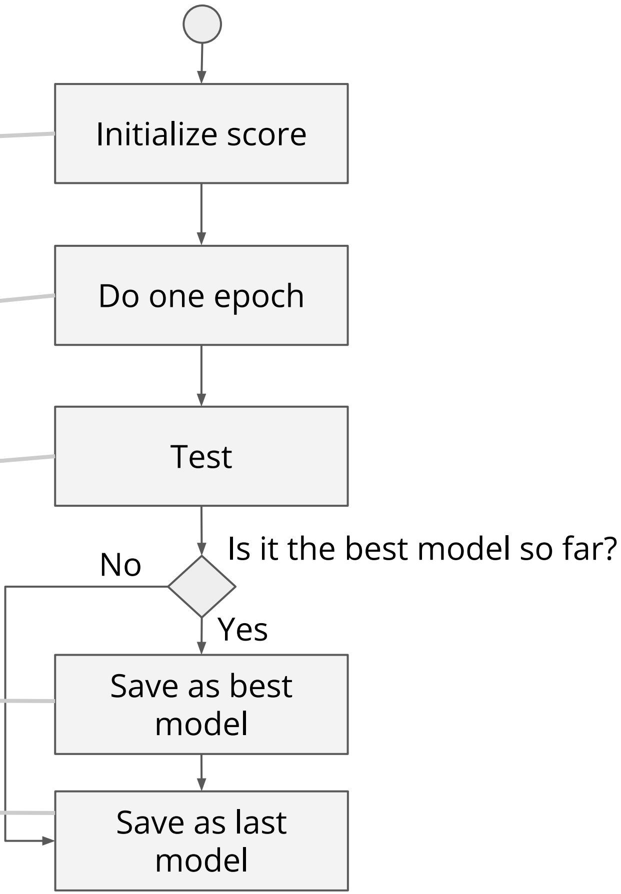
```
# initialize network
net = Network()

# initialize optimizer
opt = torch.optim.SGD(params=net.parameters(), lr=0.001)

# create dictionary containing states
d = {
    'net': net.state_dict(),
    'opt': opt.state_dict()
}

# save state
torch.save(d, 'last.pth')

# load state from disk
new_d = torch.load('last.pth')

# load parameters of network and optimizer
net.load_state_dict(new_d['net'])
opt.load_state_dict(new_d['opt'])
```

# Checkpointing



# Checkpointing (example)

```
# initialize best accuracy
best_accuracy = 0

# for each epoch
for cur_epoch in range(1000):
    # for each batch
    for inp, gt in train_dataloader:
        # … process batch and update weights

    # at the end of the epoch, test the model
    cur_accuracy = test_the_model(model, val_data_loader)

    # check if it is the best model
    if cur_accuracy > best_accuracy:
        # if yes, set new best accuracy and save best model
        best_accuracy = cur_accuracy
        save_model(model, best=True)

    # save last model
    save_model(model, best=False)
```



# Moving tensors and networks in GPU

You can easily move tensors and networks on device with:

```
# specify device
device = 'cuda'

# move tensor to device
mat = mat.to(device)

# move network to device
net.to(device)
```

Usually, you:

- move the network to GPU after network definition
- move the batch in GPU at the beginning of each iteration

# Quiz Time

# Quiz time!

Go to the website:

PollEv.com/flaviopiccoli014

# How do you convert a list into a tensor?

myTensor = torch.convert_to_tensor(myList)

myTensor = tensor(myList)

myTensor = torch.tensor(myList)

myTensor = myList.to_tensor()

# How do you get the shape of a tensor?

tensor.size()

tensor.shape()

tensor.shape

tensor.size

# Which methods must implement a class that extends torch.utils.data.Dataset?

__getitem__

__len__ and __getitem__

__init__, __len__ and __getitem__

__init__, __len__ , __collate__ and __getitem__

# Why do you need to convert a dataset into a dataloader?

To make it torch compatible

For fast testing

For parallel dataloading

For integrating the code in the pipeline

# What does the parameter num_workers of torch.utils.data.DataLoader specify?

The number of threads used for parellel loading

The number of items in a batch

The number of parameters in the network

The dimension of the batch

# What does the parameter drop_last of torch.utils.data.DataLoader mean?

Use last batch only in validation

Drop the last batch to keep it for testing

If the last batch is not big enough to cover the batch size, drop it

Put spurious data in the last batch and don't consider it during training

# How do you specify a linear layer of 20 neurons that takes 10 inputs? What will be the size of its output? (B=batch size)

nn.Linear(20, 10) and its output will be (20)

nn.Linear(10, 20) and its output will be (B, 20)

nn.Linear(20, 10) and its output will be (B, 20)

nn.Linear(10, 10) and its output will be (B, 10)

# How do you move a tensor in GPU?

myTensor = myTensor.to('cuda')

myTensor.to('cuda')

myTensor.to('gpu')

myTensor = myTensor.to('gpu')

# How do you move a neural network in GPU?

myNet = myNet.to('cuda')

myNet.to('cuda')

myNet.to('gpu')

myNet = myTensor.to('gpu')

Powered by Poll Everywhere

# What is an epoch?

The training process

The processing of a batch

The processing of a sample

The processing of the entire training set

# Regression exercises

# Google Colab

- University's computers do not have hardware acceleration (GPU)
- We need to use Google Colab

https://colab.research.google.com/

- It's the same as using a Notebook, but it is possible to add a GPU for parallel processing
- To enable:
  - Edit > Notebook settings as the following: Click on "Notebook settings" and select "GPU"

# Exercise 1 - regression

Given the Boston Housing Dataset, predict the variable MDEV from the variables

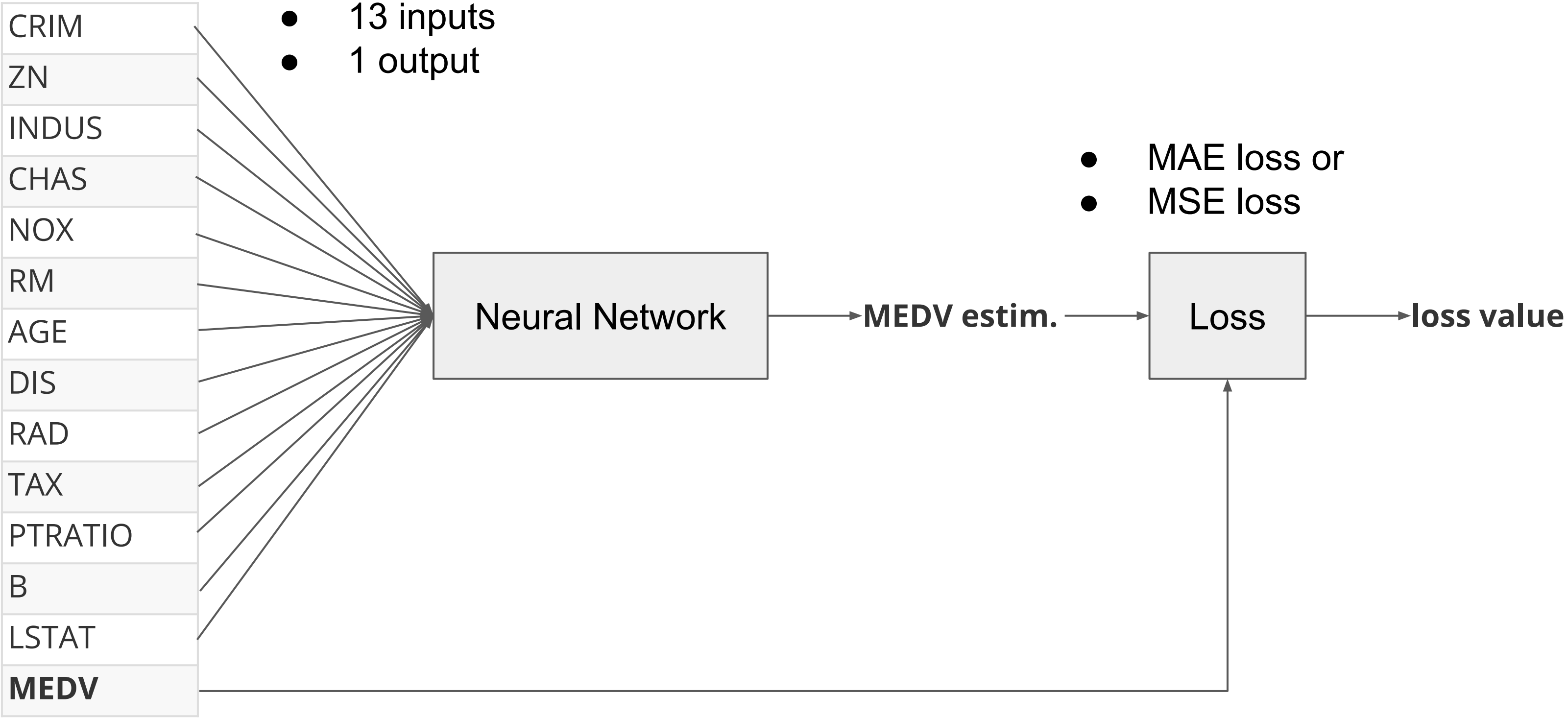
| **Variable** | **Description** |
|---|---|
| CRIM | per capita crime rate by town |
| ZN | proportion of residential land zoned for lots over 25,000 sq.ft. |
| INDUS | proportion of non-retail business acres per town |
| CHAS | Charles River dummy variable (= 1 if tract bounds river; 0 otherwise) |
| NOX | nitric oxides concentration (parts per 10 million) |
| RM | average number of rooms per dwelling |
| AGE | proportion of owner-occupied units built prior to 1940 |
| DIS | weighted distances to five Boston employment centres |
| RAD | index of accessibility to radial highways |
| TAX | full-value property-tax rate per $10,000 |
| PTRATIO | pupil-teacher ratio by town |
| B | 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town |
| LSTAT | % lower status of the population |
| **MEDV** | Median value of owner-occupied homes in $1000's |

# Exercise 1 - regression

System configuration

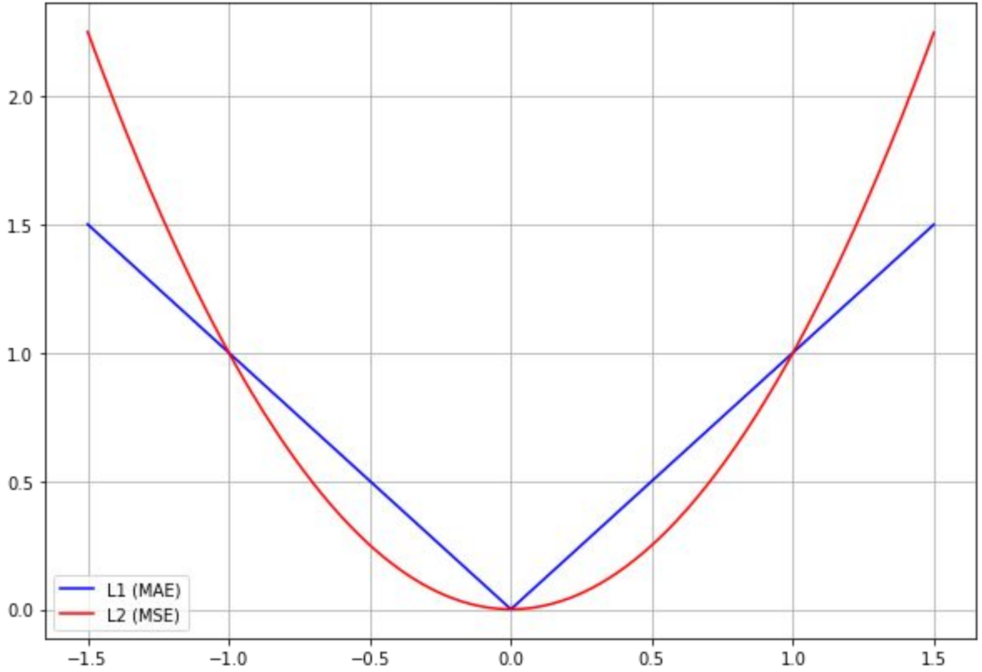
- 13 inputs
- 1 output

- MAE loss or
- MSE loss



# Exercise 1 - regression (together)

Losses for regression




$$\mathrm{MAE}(y,\hat{y}) = \frac{1}{N}\sum_{i=0}^{N-1} |y_i - \hat{y}_i|$$

$$\mathrm{MSE}(y,\hat{y}) = \frac{1}{N}\sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2$$

$$y_i = ground\ truth$$

$$\hat{y}_i = estimated\ value$$

# Exercise 1 - regression (together)

Operations:

1. create proper datasets and dataloaders
2. create a network
3. initialize optimizer
4. create training loop
5. use MAE (L1) both as training loss and metric for choosing the best model

# Exercise 2 - regression

Predict Uber fare (var. **far_amount**). Replicate the same steps we performed in the previous exercise.

N.B.: data has already been pre-processed.

| | fare_amount | pickup_longitude | pickup_latitude | dropoff_longitude | dropoff_latitude | passenger_count | year | Distance | month_1 | month_2 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 7.5 | -73.999817 | 40.738354 | -73.999512 | 40.723217 | 1 | 2015 | 1681.11 | 0 | 0 | ... |
| **1** | 7.7 | -73.994355 | 40.728225 | -73.994710 | 40.750325 | 1 | 2009 | 2454.36 | 0 | 0 | ... |
| **2** | 12.9 | -74.005043 | 40.740770 | -73.962565 | 40.772647 | 1 | 2009 | 5039.60 | 0 | 0 | ... |
| **3** | 5.3 | -73.976124 | 40.790844 | -73.965316 | 40.803349 | 3 | 2009 | 1661.44 | 0 | 0 | ... |
| **4** | 16.0 | -73.925023 | 40.744085 | -73.973082 | 40.761247 | 5 | 2014 | 4483.73 | 0 | 0 | ... |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **199995** | 3.0 | -73.987042 | 40.739367 | -73.986525 | 40.740297 | 1 | 2012 | 112.13 | 0 | 0 | ... |
| **199996** | 7.5 | -73.984722 | 40.736837 | -74.006672 | 40.739620 | 1 | 2014 | 1879.64 | 0 | 0 | ... |
| **199997** | 30.9 | -73.986017 | 40.756487 | -73.858957 | 40.692588 | 2 | 2009 | 12867.92 | 0 | 0 | ... |
| **199998** | 14.5 | -73.997124 | 40.725452 | -73.983215 | 40.695415 | 1 | 2015 | 3536.55 | 0 | 0 | ... |
| **199999** | 14.1 | -73.984395 | 40.720077 | -73.985508 | 40.768793 | 1 | 2010 | 5410.68 | 0 | 0 | ... |