Deep Learning tutorial with PyTorch

AIDD workshop – Frei Universität Berlin – 14.03.2024

AIChemist / Advanced machine learning for Innovative Drug Discovery

Adrien Bitton – Machine Learning Research

A bit on PyTorch (and me …)

PyTorch is one of the **major deep learning frameworks** and one of the **backends** for “applied” libraries like e.g. HuggingFace for “NLP”

You probably heard of JAX, Keras >=3.0 or TensorFlow>=2.0 … check them out too!

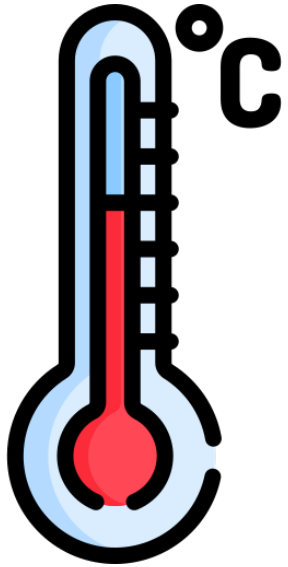
My personal take: PyTorch is **great for research**, **documentation is well done** (both technical and scientifical), has a great community/integration, is **“active”** and continuously **expanding** (e.g. production, optimization, hardware compatibility, …)

… but source code is hard to use/navigate :’(

I am using it for protein modelling, mRNA optimization, and some predictions in genomes. A lot is based on “language models” for bio-molecules (enzymes, antibodies, cDNA, …) and structure prediction / inverse folding.

… neither I am a biologist nor a chemist by training
(PhD in deep learning for signal processing in audio and music @ ircam)

Deep Learning thermometer



Who has ≈ zero experience in deep learning?

Who has practical experience in deep learning
and which framework?

Who has advanced experience and/or fundamental training?

Feel free to ask any questions at any time!

Tutorial's outline

Introduction

Part 1. Getting started with PyTorch

- Data
- Neural Network Modules
- Core functionalities

Part 2. Training with PyTorch-Lightning

- The lightning module template
- Trainer and utilities

All **codes** and **presentation slides**
are accessible on GitHub

https://github.com/adrienchaton/hello_pt/

Tutorial's scope

a.k.a. some things I wish I knew at the start of my PhD

✓ getting started basics
✓ core concepts to build-up on
✓ practical tips and tricks
✓ minimal project template with PyTorch Lightning

✗ theoretical ML and DL course (i.e. we will work through code examples)
✗ exhaustive / advanced PyTorch techniques
✗ production / software engineering

Related resources

PyTorch doc. https://pytorch.org/docs/stable/index.html

+ tutorials https://pytorch.org/tutorials/
e.g. https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

PyTorch Lightning doc. + tutorials https://lightning.ai/docs/pytorch/stable/

e.g. https://lightning.ai/docs/pytorch/stable/advanced/training_tricks.html

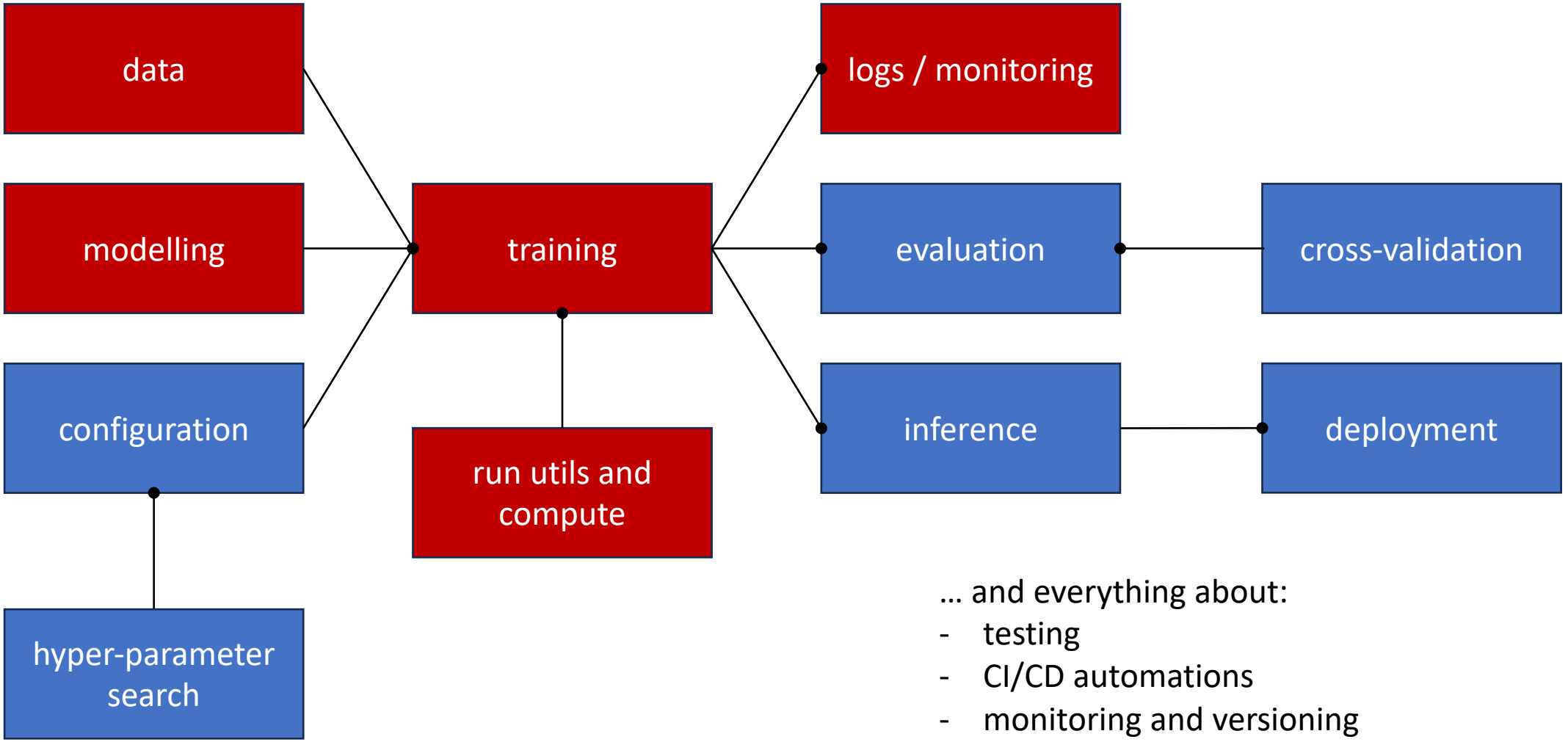
University of Amsterdam tutorials https://uvadlc.github.io/ (also for JAX!)

Python packaging https://python-packaging.readthedocs.io/en/latest/index.html

The overall picture (one of …)

Today's scope

Next steps



… and everything about:

- testing
- CI/CD automations
- monitoring and versioning of deployed models

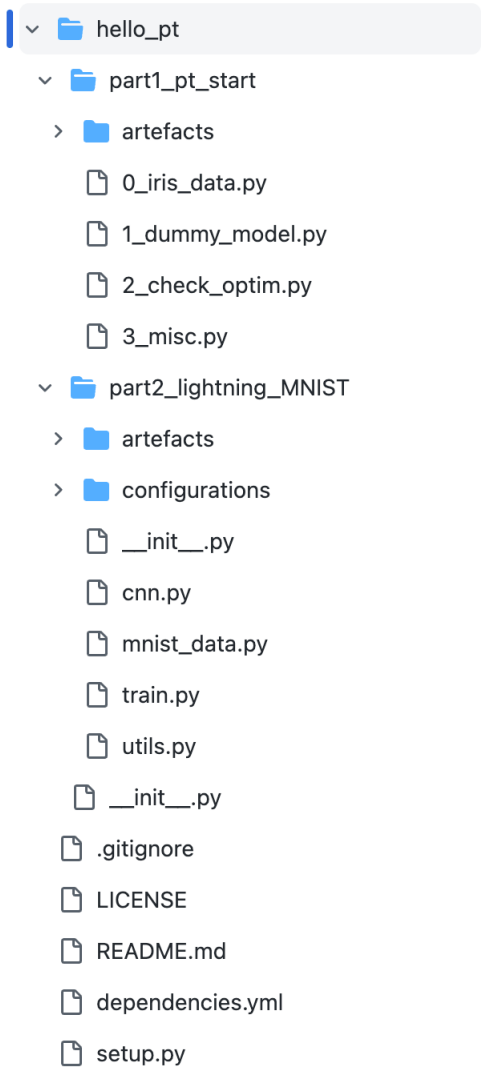
PyTorch core & “ecosystem” → sort https://pytorch.org/ecosystem/ by GitHub stars

- torch.Tensor = array with dtype, device, grads. and ops (e.g. algebra)
- torch.utils.data = dataset and minibatch loading
- torch.nn and torch.nn.functional = built-in layers and ops for modelling
- torch.distributions = built-in probabilistic ops
- torch.optim and torch.autograd = backprop. and gradient descent
- torch.distributed and torch.amp = low-level utils for parallel and mixed precision

  (… and a lot more!)
- pl.LightningModule = wrapper around a torch model organized by initialization, optim and l.r. configuration, train/validation/test/inference
- pl.Trainer = execute, monitor with callbacks, built-in utils for distributed, mixed precision training (… and more)

& friends: torchmetrics, torchvision, torchaudio, PyTorch Geometric, BoTorch, HuggingFace, Optuna …

Setting things up



Getting started

This is one of the many ways to setup the project. For managing the python environments, I use mamba and install via https://github.com/conda-forge/miniforge. In this case, it is tested on a Linux server with NVIDIA GPUs. Please, adjust when setting up without GPU, on Windows or for Apple Silicon ...

```
git clone https://github.com/adrienchaton/hello_pt.git
cd hello_pt
conda update -n base -c conda-forge mamba conda
conda create -n hello_pt python=3.9
conda activate hello_pt
mamba env update -n hello_pt -f dependencies.yml
```

If all went well, you should be able to use PyTorch and the GPUs, e.g.

```
python
import torch
assert torch.cuda.is_available()  # should return True
```

And you should be able to import from our own package, e.g.

```
python
from hello_pt import *
# otherwise, you can run "pip install -e ." which will execute "setup.py" to install the package in your current env
```

Getting started with PyTorch

From data to mini-batches

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/0_iris_data.py

```
from functools import partial
import numpy as np
from sklearn import datasets
import torch
from torch.utils.data import Dataset, DataLoader


iris_dataset = datasets.load_iris()
batch_size = 8
num_workers = 2

#########################################################################################################
## 0) load data as torch.tensor and cast to usual dtypes for classification task

x_features = torch.from_numpy(iris_dataset["data"]).float()
x_min, x_max = torch.min(x_features), torch.max(x_features)
print(f"\nloaded IRIS features with shape {x_features.shape} and type {x_features.dtype}, in range {[x_min, x_max]}")
# loaded IRIS features with shape torch.Size([150, 4]) and type torch.float32, in range [tensor(0.1000), tensor(7.9000)]
# TODO: we may want to e.g. scale the input values in [0, 1]

y_labels = torch.from_numpy(iris_dataset["target"]).long()
y_classes, y_counts = np.unique(y_labels.numpy(), return_counts=True)
print(f"\nloaded IRIS labels with shape {y_labels.shape} and type {y_labels.dtype}")
print(f"classes and counts are {dict(zip(y_classes, y_counts))}")
# loaded IRIS labels with shape torch.Size([150]) and type torch.int64
# classes and counts are {0: 50, 1: 50, 2: 50}
# TODO: the "ideal" dataset, but watch out e.g. for class imbalance
#   all usual data explorations, statistical checks and pre-processing shall be used for "real-world" data
```

From data to mini-batches

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/0_iris_data.py

```
#########################################################################################################################
## 1) the "simplest" dataloader, since our inputs are already stacked with fixed dimension

dataset1 = torch.utils.data.TensorDataset(x_features/x_max, y_labels)
dataloader1 = DataLoader(dataset1, batch_size=batch_size, num_workers=num_workers,
                         shuffle=True, drop_last=True, pin_memory=True)
# the default collate fn. just does stacking the random minibatch
x1, y1 = next(iter(dataloader1))
print(f"\ndataloader1: sampling minibatch with shape {x1.shape} {y1.shape}")
# dataloader1: sampling minibatch with shape torch.Size([8, 4]) torch.Size([8])

#########################################################################################################################
## 2) a dataloader with custom collate fn. applied on-the-fly

def some_collate_fn(data, scale_value=1.):
    x, y = zip(*data)
    minibatch = {"x": torch.stack(x)/scale_value, "y": torch.stack(y)}
    return minibatch
# TODO: can be used to handle inputs of varying shapes, e.g. padding to the largest element when needed

dataset2 = torch.utils.data.TensorDataset(x_features, y_labels)  # here we will scale values on-the-fly
dataloader2 = DataLoader(dataset2, batch_size=batch_size, num_workers=num_workers,
                         shuffle=True, drop_last=True, pin_memory=True,
                         collate_fn=partial(some_collate_fn, scale_value=x_max))
minibatch2 = next(iter(dataloader2))
print(f"\ndataloader2: sampling minibatch with shape {minibatch2['x'].shape} {minibatch2['y'].shape}")
# dataloader2: sampling minibatch with shape torch.Size([8, 4]) torch.Size([8])
```

From data to mini-batches

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/0_iris_data.py

```
#########################################################################################################################
## 3) a custom dataset to handle loading the data and applying transformations

class IrisDataset(Dataset):
    def __init__(self, x, y, scale_value=1.):
        self.x = x
        self.y = y
        self.scale_value = scale_value
    def __len__(self):
        return len(self.y)
    def __getitem__(self, idx):
        return self.x[idx]/self.scale_value, self.y[idx]
# TODO: can be used to handle input of various formats
#   e.g. a pandas dataframe, a long text which isn't chunked, streaming data from local files or from the web

dataset3 = IrisDataset(x_features, y_labels, scale_value=x_max)
dataloader3 = DataLoader(dataset3, batch_size=batch_size, num_workers=num_workers,
                         shuffle=True, drop_last=True, pin_memory=True)
x3, y3 = next(iter(dataloader3))
print(f"\ndataloader3: sampling minibatch with shape {x3.shape} {y3.shape}")
# dataloader3: sampling minibatch with shape torch.Size([8, 4]) torch.Size([8])
```

note: Dataset.__getitem__ only processes individual samples

DataLoader.collate_fn can process all minibatch for e.g. padding

The torch.nn.Module

A “basic” neural network is a stack of **layers**, e.g.

- Feature **transformation** → most of the **training parameters**
- Feature **normalization** → few to no training parameters
- **Non-linear activation** function → usually fixed

→ a **block** that is repeated x-times to **map inputs to outputs**

torch.nn.Module handles e.g.

- **model state**, i.e. trainable/fixed parameters, saving/restoring checkpoints, …
- training/evaluation modes (e.g. for dropout, batch-normalization, …)
- interfacing with compute device(s), optimizer(s), …

… Pytorch layers are nn.Module themselves … so **everything is nested nn.Module**
(beware of the pitfalls!)

The torch.nn.Module

```
import os
from pathlib import Path
import torch
import torch.nn as nn
from torch.nn import functional as F
```

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/1_dummy_model.py

```
batch_size, n_features, n_classes, x_max = 8, 4, 3, 7.9  # as in the previous iris_dataset
n_hidden_layers, hidden_size = 2, 16  # your first model configuration! ;)
cuda_device = 0  # compute on 1st visible GPU if possible
device = torch.device(f"cuda:{cuda_device}" if (torch.cuda.is_available() and cuda_device>=0) else "cpu")
# TODO: you can set (relative) visible devices through the environment variable CUDA_VISIBLE_DEVICES
#   but using os.environ["CUDA_VISIBLE_DEVICES"] doesn't work after the python codes have been called

#########################################################################################################################
## 0.A) NN block with a linear layer

class LinearBlock_A(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearBlock_A, self).__init__()
        self.linear = nn.Linear(input_size, output_size, bias=False)  # since it is followed with batch-norm
        self.norm = nn.BatchNorm1d(output_size)
        self.act = nn.LeakyReLU()
        self.apply(self._init_p)  # TODO: customize init, or comment to use defaults
    def _init_p(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight.data)  # TODO: note that usually, operations ending with "_" are in-place
            if module.bias is not None:
                module.bias.data.zero_()
    def forward(self, x):
        x = self.linear(x)
        x = self.norm(x)
        x = self.act(x)
        return x
# TODO: further customize with e.g. dropout "regularization"
```

The torch.nn.Module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/1_dummy_model.py

```
model1 = LinearBlock_A(n_features, hidden_size).to(device)  # instantiate model on device
input1 = torch.randn((batch_size, n_features), device=device, dtype=torch.float32)  # instantiate random data on device
output1 = model1(input1)
print(f"\nmodel1: forwarded input1 of shape {input1.shape} to output1 of shape {output1.shape}")
# model1: forwarded input1 of shape torch.Size([8, 4]) to output1 of shape torch.Size([8, 16])
# TODO: here the output is some intermediate/hidden features

def print_trainable_parameters(model, verbose=False):
    trainable_params = 0
    all_param = 0
    for pname, param in model.named_parameters():
        all_param += param.numel()
        if param.requires_grad:
            trainable_params += param.numel()
        if verbose:
            print(f"{'trainable' if param.requires_grad else 'freeze'}\t{pname} of size {param.numel()}")
    print(f"trainable params: {trainable_params} || "
          f"all params: {all_param} || "
          f"trainable%: {100 * trainable_params / all_param:.2f}")
# it can be a one-liner, e.g. sum(p.numel() for p in model.parameters() if p.requires_grad)

print_trainable_parameters(model1, verbose=True)
# trainable       linear.weight of size 64
# trainable       norm.weight of size 16
# trainable       norm.bias of size 16
# trainable params: 96 || all params: 96 || trainable%: 100.00
# TODO: notice that all parameters are trainable and the activation doesn't have any learned parameters

print(model1.state_dict().keys())  # TODO: notice the added "buffers" for tracking norm statistics
# ['linear.weight', 'norm.weight', 'norm.bias', 'norm.running_mean', 'norm.running_var', 'norm.num_batches_tracked']
torch.save(model1.state_dict(), os.path.join(Path(__file__).parent, "artefacts", "model1.pt"))
model1.apply(model1._init_p)
print(f"same output after random init., {torch.equal(output1, model1(input1))}")  # False
model1.load_state_dict(torch.load(os.path.join(Path(__file__).parent, "artefacts", "model1.pt"), map_location=device))
print(f"same output after restoring ckpt, {torch.equal(output1, model1(input1))}")  # True
```

The torch.nn.Module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/1_dummy_model.py

```
#########################################################################################################################
## 0.B) linear block with a fixed scaling factor

class LinearBlock_B(LinearBlock_A):
    def __init__(self, input_size, output_size, scale_value=1.):
        super(LinearBlock_B, self).__init__(input_size, output_size)
        self.register_buffer("scale_value", torch.tensor(scale_value, dtype=torch.float32), persistent=True)
        # buffers are persistent by default but not considered as model parameters
    def forward(self, x):
        x = x/self.scale_value
        x = self.linear(x)
        x = self.norm(x)
        x = self.act(x)
        return x

model2 = LinearBlock_B(n_features, hidden_size, scale_value=x_max).to(device)
output2 = model2(input1)
assert all(d1==d2 for d1, d2 in zip(output1.shape, output2.shape))
print("\nmodel2: state dict after registering buffer for scale_value")
print(model2.state_dict().keys())  # TODO: notice the added buffer for scale_value
print_trainable_parameters(model2, verbose=False)  # unchanged, i.e. trainable params: 96 || all params: 96 || trainable%: 100.00
```

The torch.nn.Module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/1_dummy_model.py

```
#########################################################################################################
## 0.C) sequential block with learned scaling factors for each input dimension (for the example ...)

class LinearBlock_C(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearBlock_C, self).__init__()
        self.block = nn.Sequential(nn.Linear(input_size, output_size, bias=False), nn.BatchNorm1d(output_size), nn.LeakyReLU())
        # since we don't need to modify intermediate values, let's pack all layers into a sequential container
        self.learned_scaling = nn.Parameter(torch.ones(input_size, dtype=torch.float32), requires_grad=True)
        # the parameter is persistent and by default the gradients will be tracked for training
        # there are other ways to add parameters, e.g. register_parameter, register_module, add_module
    def forward(self, x):
        scale_value = F.softplus(self.learned_scaling)  # TODO: take care of e.g. zero division when training params
        x = x/scale_value  # TODO: note that scale_value is automatically broadcast to the batch size of x
        x = self.block(x)
        return x

model3 = LinearBlock_C(n_features, hidden_size).to(device)
output3 = model3(input1)
assert all(d1==d3 for d1, d3 in zip(output1.shape, output3.shape))
print("\nmodel3: state dict after adding parameter for learned_scaling")
print(model3.state_dict().keys())  # TODO: notice the added parameter for learned_scaling
print_trainable_parameters(model3, verbose=False)  # trainable params: 100 || all params: 100 || trainable%: 100.00
```

The torch.nn.Module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/1_dummy_model.py

```
#########################################################################################################################
## 1) MLP classifier

class MLP_classifier(nn.Module):
    def __init__(self, input_size, output_size, n_hiddens, d_hiddens):
        super(MLP_classifier, self).__init__()
        layers = [LinearBlock_A(input_size, d_hiddens)]  # the input layer
        layers += [LinearBlock_A(d_hiddens, d_hiddens) for _ in range(n_hiddens)]  # some hidden layers
        layers.append(nn.Linear(d_hiddens, output_size))
        # TODO: beware of last layer, often it isn't normalized
        #   AND one must take care of the output range to match that of the target data (e.g. regression task)
        self.classifier = nn.Sequential(*layers)
    def forward(self, x):
        return self.classifier(x)

model4 = MLP_classifier(n_features, n_classes, n_hidden_layers, hidden_size).to(device)
output4 = model4(input1)
print(f"\nmodel4: forwarded input1 of shape {input1.shape} to output4 of shape {output4.shape}")
# model4: forwarded input1 of shape torch.Size([8, 4]) to output4 of shape torch.Size([8, 3])
# TODO: here the output is some "weights" for each of the n_classes
#   you can convert logits to probas. with torch.softmax
print_trainable_parameters(model4, verbose=False)
# trainable params: 723 || all params: 723 || trainable%: 100.00
```

The torch.nn.Module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/1_dummy_model.py

```
#########################################################################################################################
## !!! Pitfalls !!!

class buggy_model(nn.Module):
    def __init__(self, input_size, output_size, d_hiddens):
        super(buggy_model, self).__init__()
        # correct way to add a list of modules (e.g. as opposed to a fixed ordering in Sequential)
        self.layers1 = nn.ModuleList([LinearBlock_A(input_size, d_hiddens), LinearBlock_A(d_hiddens, d_hiddens),
                                      nn.Linear(d_hiddens, output_size)])
        # FIXME: below, the plain list of nn.Module won't be registered to the state_dict and not trained with the optimizer
        self.layers2 = [LinearBlock_A(input_size, d_hiddens), LinearBlock_A(d_hiddens, d_hiddens),
                        nn.Linear(d_hiddens, output_size)]
        # FIXME: below, the torch.tensor won't be registered to the state_dict and not trained with the optimizer
        self.learned_scaling = torch.ones(input_size, dtype=torch.float32, requires_grad=True)
        self.fixed_scaling = torch.ones(input_size, dtype=torch.float32, requires_grad=False)
    def forward(self, x):
        for i, l in enumerate(self.layers1):
            x = l(x)
        return x

model5 = buggy_model(n_features, n_classes, hidden_size).to(device)
output5 = model5(input1)
print(f"\nmodel5: forwarded input1 of shape {input1.shape} to output5 of shape {output5.shape}")
# model5: forwarded input1 of shape torch.Size([8, 4]) to output5 of shape torch.Size([8, 3])
print(model5.state_dict().keys())  # only params from layers1
print_trainable_parameters(model5, verbose=False)
# trainable params: 435 || all params: 435 || trainable%: 100.00 (all from layers1 and trainable)
```

Model optimization

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/2_check_optim.py

```
import torch
import torch.nn as nn
from torch.nn import functional as F

batch_size, n_features, n_classes = 8, 4, 3
cuda_device = 0  # compute on 1st visible GPU if possible
device = torch.device(f"cuda:{cuda_device}" if (torch.cuda.is_available() and cuda_device>=0) else "cpu")

dummy_x = torch.randn((batch_size, n_features), device=device, dtype=torch.float32)  # instantiate random data on device
dummy_y = torch.randint(low=0, high=n_classes, size=(batch_size, ), device=device, dtype=torch.long)

#########################################################################################################################
## setting up a simple model, optimizer, loss and perform one update

class LinearModel(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearModel, self).__init__()
        self.layer = nn.Linear(input_size, output_size)
        self.act = nn.Softmax(dim=1)  # to compute probabilities over the classes dimension
        self.loss_fn = nn.CrossEntropyLoss(reduction='mean', weight=None, ignore_index=-100)
        # TODO: weight can used to e.g. compensate class imbalance
        #   and ignore_index can be used to fill-up missing values to be ignored, or as padding value
    def forward(self, x):
        return self.layer(x)
    def compute_loss(self, x, y):
        y_hat = self.forward(x)
        y_loss = self.loss_fn(y_hat, y) # here, CrossEntropyLoss already handles scaling the logits before computing NLL
        return {"y_hat": self.act(y_hat), "y_loss": y_loss}
    @torch.inference_mode()  # this automatically disable gradients and reduce calculations to speed-up inference
    def predict(self, x):
        y_hat = self.forward(x)
        return {"y_hat": self.act(y_hat)}

model1 = LinearModel(n_features, n_classes).to(device)
optim1 = torch.optim.AdamW(model1.parameters(), lr=1e-4, weight_decay=0.01)
print(f"\nmodel1: with {sum(p.numel() for p in model1.parameters() if p.requires_grad)} trainable parameters and "
      f"{sum(p.numel() for p in model1.parameters() if not p.requires_grad)} fixed parameters")
# model1: with 15 trainable parameters and 0 fixed parameters
```

Model optimization

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/2_check_optim.py

```
def try_optim_step(model, optim, x, y):
    try:
        loss = model.compute_loss(x, y)["y_loss"]
        optim.zero_grad()  # ensure all gradients are cleared before doing an update step
        loss.backward()
        print("forward loss", loss)  # you can see the trace of grad_fn=<NllLossBackward0>
        print(f"gradients with norm={torch.sum(torch.abs(model.layer.weight.grad)).item()}")
        optim.step()  # this will act on the provided model.parameters() which have gradients
        print(f"optim state = {optim.state_dict()}")  # here you see e.g. moving average values due to SGD with momentum
        # TODO: usually, both the model and the optimizer states are saved, to allow e.g. resuming training
        optim.zero_grad()
        print(f"clearing gradients after optimization step, gradients={model.layer.weight.grad}")  # gradients=None
    except RuntimeError:
        # we get "RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn"
        optim.zero_grad()
        print(f"could not perform optimization step")

model1.train()
print("\nmodel1: step in train mode")
try_optim_step(model1, optim1, dummy_x, dummy_y)

model1.eval()  # this doesn't mean gradients are disabled!
print("\nmodel1: step in eval mode")
try_optim_step(model1, optim1, dummy_x, dummy_y)

print("\nmodel1: step in inference mode")
with torch.inference_mode():  # this effectively disable everything related to autograd
    # preferred over torch.no_grad()
    try_optim_step(model1, optim1, dummy_x, dummy_y)  # --> could not perform optimization step
```

Model optimization

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/2_check_optim.py

```
#########################################################################################################################
## more "custom" model, checking where gradients are properly computed

class FrankensteinModel(LinearModel):
    def __init__(self, input_size, output_size):
        super(FrankensteinModel, self).__init__(input_size, output_size)
        self.register_buffer("a_buffer", torch.randn(input_size, dtype=torch.float32), persistent=True)
        self.learned_p = nn.Parameter(torch.randn(input_size, dtype=torch.float32), requires_grad=True)
        self.fixed_p = nn.Parameter(torch.randn(input_size, dtype=torch.float32), requires_grad=False)
    def forward(self, x):
        x = x+self.a_buffer
        x = x+self.learned_p
        x = x+self.fixed_p
        return self.layer(x)
    def gradient_check(self, optim, x, y):
        loss = self.compute_loss(x, y)["y_loss"]
        optim.zero_grad()
        loss.backward()
        for pname, param in self.named_parameters():
            if param.grad is None:
                print(f"{pname} has no gradients")
            else:
                print(f"{pname} has gradients with norm {torch.sum(torch.abs(param.grad)).item()}")
        optim.zero_grad()
# TODO: "gradient_check" can be a good sanity check to make sure all init layers are training
#   and check their respective gradient magnitudes

model2 = FrankensteinModel(n_features, n_classes).to(device)
optim2 = torch.optim.Adam(model2.parameters(), lr=1e-4, weight_decay=0.)
print(f"\nmodel2: with {sum(p.numel() for p in model2.parameters() if p.requires_grad)} trainable parameters and "
      f"{sum(p.numel() for p in model2.parameters() if not p.requires_grad)} fixed parameters")
# model2: with 19 trainable parameters and 4 fixed parameters

print("\ngradient check")
model2.gradient_check(optim2, dummy_x, dummy_y)  # --> fixed_p has no gradients
```

Model optimization

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/2_check_optim.py

```
print("\ngradient check after freezing learned_p")
model2.learned_p.requires_grad = False
# it can be useful for e.g. fine-tuning to dynamically freeze/un-freeze some parameters
model2.gradient_check(optim2, dummy_x, dummy_y)  # --> learned_p has no gradients (too)

print("\ngradient check after adding fixed_p2 to model")
model2.fixed_p2 = nn.Parameter(torch.randn(1, device=device, dtype=torch.float32), requires_grad=False)
# alternatively, model2.add_module("another_layer", nn.Linear(n_features, n_features))
model2.to(device)  # make sure all params. are on the same device
model2.gradient_check(optim2, dummy_x, dummy_y)  # --> fixed_p2 has no gradients (too)
# TODO: if we wished to add new trainable parameters, we would need e.g.
#   - optim.add_param_group({'params': the_new_parameters})
#   - modify the forward method to apply the parameters in the computation
# TODO: in some cases we don't want an optimizer to update all parameters, e.g. GANs
#   and we also need to detach some variables from the autograd graph, e.g. generator outputs in the discriminator loss
y_loss = model2.compute_loss(dummy_x, dummy_y)["y_loss"]
print(y_loss)  # tensor(0.7223, device='cuda:0', grad_fn=<NllLossBackward0>)
print(y_loss.detach())  # tensor(0.7223, device='cuda:0')
print(y_loss.detach().cpu())  # tensor(0.7223)
print(y_loss.detach().cpu().numpy())  # 0.7223418
```

Misc. tips and tricks!

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/3_misc.py

```
# here are some misc. details and utils to optimise your codes
# this isn't intended to be run as the other scripts 0/1/2

# further reads at:
# - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
# - https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html
# - https://pytorch.org/docs/stable/notes/randomness.html
# - https://pytorch.org/tutorials/beginner/ptcheat.html
# - https://pytorch.org/blog/accelerating-training-on-nvidia-gpus-with-pytorch-automatic-mixed-precision/


torch.backends.cudnn.benchmark = True  # will let CuDNN optimize your computation depending on hardware,
# favour parameters with dimensions as multiple of 8 for maximising tensor core usage
# and adapt float precision to your needs and hardware (will be handled by pt-lightning in the part2), for TF32
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True

torch.randn((100,), device="cuda")  # instantiate on GPU memory when possible, instead of transferring with .cuda()
torch.randn((100,)).to(device, non_blocking=True)  # or transfer asynchronously

torch.utils.data.DataLoader(dataset, num_workers=3, shuffle=True, drop_last=True, pin_memory=True)
# use multiple CPU workers and pin_memory if GPU doesn't go OOM, drop_last=True along with ...cudnn.benchmark=True

# within e.g. the training loop, keep all data/processing on GPU and asynchronous
# no print(), no .cpu(), no .numpy(), no .item() BUT .detach() should be used to e.g. log loss values through an epoch

# watch-out for non-differentiable operations which prevent backprop. (e.g. in model forward)
x_notdiff = torch.argmax(x_diff)  # instead, look e.g. for Gumbel-Softmax (here x_notdiff.grad_fn=None)
```

Misc. tips and tricks!

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part1_pt_start/3_misc.py

```python
def seed_everything(seed=1234):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
# but avoid using torch.use_deterministic_algorithms(True) except if explicitly required for reproducibility

# do not use int to represent categorical data (when there is no relation/ranking), instead use
nn.Embedding(num_embeddings=num_classes, embedding_dim=hidden_dim)
torch.nn.functional.one_hot(int_labels, num_classes=num_classes)

# init your nn.Module on cpu and only transfer to GPU the whole model once it is instantiated
self.layer = nn.Linear(in_dim, out_dim).to(device)  # DON'T
model = MyModel().to(device)  # DO this instead

torch.optim.AdamW(model1.parameters(), weight_decay=0.01)  # use AdamW instead of Adam is using weight_decay

nn.utils.clip_grad_norm_(model.parameters(), 1.0)  # after loss.backward() and before optimizer.step(), to stabilise training

# if OOM (GPU), use gradient accumulation, i.e. summing losses over several forward passes before calling .backward()
# if OOM (RAM), look into iterable datasets and sharding
# https://pytorch.org/data/main/torchdata.datapipes.iter.html
# https://pytorch.org/docs/stable/notes/multiprocessing.html

torch.distributed.get_rank() == 0  # can be used along with multiprocessing to only do an operation on the main process

# monitoring resources within python, use psutil (disk, RAM, CPU) or pynvml
# https://suzyahyah.github.io/code/pytorch/2024/01/25/GPUTrainingHacks.html
```

Training with PyTorch-Lightning

Scalable deep learning without boilerplate

LightningModule wraps N.N. with all common steps of a ML experiment

e.g. init., configure optim. and data, train/validation/test/inference calls

→ help with code **structure**, **readability** and template **re-use**

Trainer automates these steps and interfaces with backends and compute

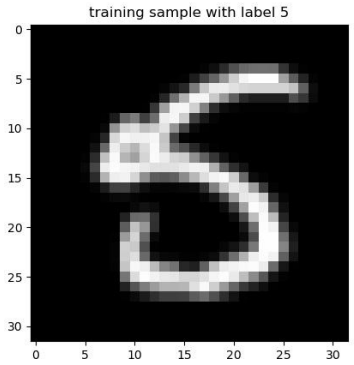
- ensures **best practices** (e.g. proper handling of modes/gradients, …)
- **scalability** (takes care of GPUs, distributed modes, multi-node …)
- “standard” tasks can be run with **little code**
- automations can be **override** for “less standard” problems (e.g. GANs)
- can be customized with callbacks for e.g. logging and validation

MNIST classification example

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/mnist_data.py

```python
import os
from pathlib import Path
import numpy as np
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import Compose, ToTensor, Resize
import matplotlib.pyplot as plt

from utils import MNIST_CST
```

```python
class MNISTdataset(Dataset):
    def __init__(self, path_to_mnist, img_hw, train=True):
        MNISThw = MNIST_CST().MNISThw
        self.data = datasets.MNIST(root=path_to_mnist, train=train, download=True)
        print(f"\nloading MNIST {'train' if train else 'test'} set of size {len(self)}")
        img_transforms = [ToTensor()]
        if img_hw!=MNISThw:
            print(f"rescaling MNIST images from size {MNISThw} to {img_hw}")
            img_transforms.append(Resize((img_hw, img_hw), antialias=True))
        self.img_transforms = Compose(img_transforms)
        # TODO: could add e.g. data augmentations for the training
        # TODO: could automatically check for counts per digit
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        image = self.img_transforms(self.data[idx][0])
        label = self.data[idx][1]
        return image, label

if __name__ == "__main__":
    img_hw = 32  # e.g. we set a %8 size for input data, original MNIST data is (28, 28)

    train_data = MNISTdataset(MNIST_CST().path_to_mnist, img_hw, train=True)  # MNIST train set of size 60000
    test_data = MNISTdataset(MNIST_CST().path_to_mnist, img_hw, train=False)  # MNIST test set of size 10000
    train_sample, train_label = train_data[np.random.choice(len(train_data))]
    test_sample, test_label = test_data[np.random.choice(len(test_data))]

    print(f"train sample of shape {train_sample.shape}, "
          f"values in range {[torch.min(train_sample), torch.max(train_sample)]} and label {train_label}")
    print(f"test sample of shape {test_sample.shape}, "
          f"values in range {[torch.min(test_sample), torch.max(test_sample)]} and label {test_label}")
    # train sample of shape torch.Size([1, 32, 32]), values in range [tensor(0.), tensor(0.9978)] and label 0
    # --> minibatches will then be as (batch, 1, 32, 32) and (batch)
```



MNIST classification example

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/utils.py

```python
class MNIST_CST:
    def __init__(self):
        self.path_to_mnist = os.path.join(Path(__file__).parent, "artefacts")
        self.MNISThw = 28
        self.n_classes = 10

def load_config(path_to_yaml):
    with open(path_to_yaml, 'r') as f:
        config = yaml.safe_load(f)
    return config

def try_checking_process_rank0():
    try:
        return torch.distributed.get_rank() == 0
    except:  # fallback in case we are not using distributed
        return True

class PrintingCallback(Callback):
    def on_train_start(self, trainer, pl_module):
        print("\nTraining is starting (╯°□°)╯︵ ┻━┻") if try_checking_process_rank0() else None
    def on_train_end(self, trainer, pl_module):
        print("\nTraining is ending ♩̤̮(ง ˙˘˙ )ว♩̤̮") if try_checking_process_rank0() else None
```

```python
class GradientCheckCallback(Callback):
    def on_train_start(self, trainer, pl_module):
        if try_checking_process_rank0():
            x, y = next(iter(pl_module.train_data))
            logits = pl_module.model.forward(x.to(pl_module.device, non_blocking=True))
            loss = pl_module.loss_fn(logits, y.to(pl_module.device, non_blocking=True))
            loss.backward()
            tot_grad = 0
            for pname, param in pl_module.model.named_parameters():
                if param.grad is None:
                    print(f"{pname} has no gradients")
                else:
                    pgrad = torch.sum(torch.abs(param.grad)).item()
                    tot_grad += pgrad
                    print(f"{pname} has gradients with norm {pgrad}")
            print(f"total sum of gradients = {tot_grad}")
    def on_train_end(self, trainer, pl_module):
        self.on_train_start(trainer, pl_module)
```

CNN classifier

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/cnn.py

```
import os
import torch
import torch.nn as nn
from torch.nn import functional as F
from torch.utils.data import Dataset, DataLoader
import lightning.pytorch as pl
import torchmetrics

from utils import MNIST_CST
from mnist_data import MNISTdataset
```

```
class ConvBlock(nn.Module):
    def __init__(self, c_in, c_out):
        super(ConvBlock, self).__init__()
        self.block = nn.Sequential(nn.Conv2d(c_in, c_out, kernel_size=3, padding=1, stride=2, bias=False),
                                   nn.BatchNorm2d(c_out), nn.LeakyReLU())
        # H,W will be halved because of kernel_size=3, padding=1, stride=2
    def forward(self, x):
        return self.block(x)

class CNN(nn.Module):
    def __init__(self, img_hw, n_classes, n_layers, n_channels):
        super(CNN, self).__init__()
        hidden_size = int(img_hw/2**n_layers * n_channels * 2)
        print(f"\nbuilding CNN with hidden size = {hidden_size}")
        layers = [ConvBlock(1, n_channels)]+[ConvBlock(n_channels, n_channels) for _ in range(n_layers-1)]
        self.cnn = nn.Sequential(*layers)
        self.proj = nn.Linear(hidden_size, n_classes)
        print(f"model with {sum(p.numel() for p in self.parameters() if p.requires_grad)} params.")
    def forward(self, x):
        x = self.cnn(x)
        y = self.proj(x.reshape(x.shape[0], -1))
        return y
    @torch.inference_mode()  # although pytorch_lightning will automatically disable gradients for eval/test/inference
    def predict(self, x):
        return torch.softmax(self.forward(x), dim=-1)
    @torch.inference_mode()
    def classify(self, x):
        return torch.argmax(self.predict(x), dim=-1)
```

```
cnn = CNN(32, 10, 4, 64)
cnn.forward(torch.randn(3, 1, 32, 32))
# building CNN with hidden size = 256
# model with 114250 params. (MNIST is ~47M pixels ... most of which are black)
```

MNIST module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/cnn.py

```
class MNISTmodule(pl.LightningModule):
    def __init__(self, config, seed=1234, num_workers=3):
        super().__init__()
        pl.seed_everything(seed, workers=True)
        self.save_hyperparameters()
        self.cst = MNIST_CST()
        self.create_dataloaders()
        print(f"\ninstantiating MNIST classifier with config. {self.hparams}")  # built-in attribute from lightning
        self.model = CNN(config["img_hw"], self.cst.n_classes, config["n_layers"], config["n_channels"])
        self.loss_fn = nn.CrossEntropyLoss()
        self.metrics_acc = torchmetrics.classification.MulticlassAccuracy(num_classes=self.cst.n_classes)
        self.metrics_f1 = torchmetrics.classification.MulticlassF1Score(num_classes=self.cst.n_classes)
    def create_dataloaders(self):
        self.train_data = DataLoader(MNISTdataset(self.cst.path_to_mnist, self.hparams.config["img_hw"], train=True),
                                     batch_size=self.hparams.config["batch_size"], num_workers=self.hparams.num_workers,
                                     shuffle=True, drop_last=True, pin_memory=True)
        self.test_data = DataLoader(MNISTdataset(self.cst.path_to_mnist, self.hparams.config["img_hw"], train=False),
                                    batch_size=self.hparams.config["batch_size"], num_workers=self.hparams.num_workers,
                                    shuffle=False, drop_last=False, pin_memory=False)
    def train_dataloader(self):
        return self.train_data
    def test_dataloader(self):
        return self.test_data
    def forward(self, x):
        return self.model.forward(x)
```

MNIST module

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/cnn.py

```
def configure_optimizers(self):
    optimizer = torch.optim.AdamW(self.parameters(), lr=self.hparams.config["lr"],
                                  weight_decay=self.hparams.config["weight_decay"])
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=self.hparams.config["n_iters"],
                                                           eta_min=self.hparams.config["end_lr"])
    return [optimizer], [scheduler]
def training_step(self, batch, batch_idx):
    x, y = batch
    logits = self.model.forward(x)
    loss = self.loss_fn(logits, y)
    self.log('train_loss', loss, on_step=True, on_epoch=True, sync_dist=True)
    return loss  # lightning will automatically take care of the optimization step
def validation_step(self, batch, batch_idx):  # lightning will automatically switch between train/eval modes
    # TODO: stratified split for train/validation and e.g. early stopping callback on validation metrics
    x, y = batch
    preds = self.model.classify(x)
    self.log('val_acc', self.metrics_acc(preds, y), on_step=False, on_epoch=True, sync_dist=True)
    self.log('val_f1', self.metrics_f1(preds, y), on_step=False, on_epoch=True, sync_dist=True)
def test_step(self, batch, batch_idx):
    x, y = batch
    preds = self.model.classify(x)
    self.log('test_acc', self.metrics_acc(preds, y), on_step=False, on_epoch=True, sync_dist=True)
    self.log('test_f1', self.metrics_f1(preds, y), on_step=False, on_epoch=True, sync_dist=True)
```

Training

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/train.py

```
import os
from pathlib import Path
import argparse
import torch
import lightning.pytorch as pl
from lightning.pytorch.callbacks import LearningRateMonitor

from utils import load_config, PrintingCallback, GradientCheckCallback
from cnn import MNISTmodule
```

```
def configure():
    parser = argparse.ArgumentParser()
    parser.add_argument("--seed",               default=1234, type=int)
    parser.add_argument("--devices",            default=-1)
    parser.add_argument("--exp_path",           default="", type=str)
    parser.add_argument("--config_path",        default="", type=str)
    parser.add_argument("--num_workers",        default=3, type=int)
    parser.add_argument("--freq_val",           default=1., type=float)
    parser.add_argument('--test_on_start',      action='store_true')
    parser.add_argument('--deterministic',      action='store_true')
    parser.add_argument('--mixed_precision',    action='store_true')
    parser.add_argument('--ddp',                action='store_true')
    parser.add_argument('--profiler',           action='store_true')
    args = parser.parse_args()
    return args

def set_defaults(args):
    if args.devices == -1:
        args.devices = torch.cuda.device_count()
    if args.devices <= 1:
        args.ddp = False
    if args.exp_path == "":
        args.exp_path = os.path.join(Path(__file__).parent, "artefacts", "MNIST_CNN")
    if args.config_path == "":
        args.config_path = os.path.join(Path(__file__).parent, "configurations", "mnist_cnn.yaml")
    return args
```

Training

https://github.com/adrienchaton/hello_pt/blob/main/hello_pt/part2_lightning_MNIST/train.py

```
def main():
    args = configure()
    args = set_defaults(args)
    config = load_config(args.config_path)
    pl_module = MNISTmodule(config, seed=args.seed, num_workers=args.num_workers)
    callbacks = [PrintingCallback(), LearningRateMonitor(logging_interval='epoch'), GradientCheckCallback()]
    trainer = pl.Trainer(max_steps=config["n_iters"], val_check_interval=args.freq_val, devices=args.devices,
                         precision="16-mixed" if args.mixed_precision else "32-true", benchmark=True,
                         default_root_dir=args.exp_path, callbacks=callbacks,
                         deterministic=args.deterministic, accelerator="cpu" if args.devices=="cpu" else "gpu",
                         enable_progress_bar=True, profiler=args.profiler, strategy="ddp" if args.ddp else "auto")
    if args.devices <=1 and args.test_on_start:  # check the "random" performance
        trainer.test(model=pl_module, dataloaders=pl_module.test_data, verbose=True)
    trainer.fit(model=pl_module, train_dataloaders=pl_module.train_data, val_dataloaders=pl_module.test_data)
    if args.devices <=1:  # if distributed training, it is recommended to run testing separately on single GPU
        trainer.test(model=pl_module, dataloaders=pl_module.test_data, verbose=True)

if __name__ == "__main__":
    # can be run with e.g.  (best to debug without distributed!)
    # CUDA_VISIBLE_DEVICES=0 python train.py --test_on_start --freq_val 0.2 --exp_path ./artefacts/MNIST_CNN_1
    # CUDA_VISIBLE_DEVICES=1,2 torchrun --standalone --nnodes=1 --nproc_per_node=2 train.py --ddp --exp_path ./artefacts/MNIST_CNN_2
    # then check logs with e.g. tensorboard --logdir
    # or use within python with e.g. from tensorboard.backend.event_processing.event_accumulator import EventAccumulator
    main()
```

Good luck for your future research projects

And now let’s run it! 🚀