Introduction to Pytorch Lightning

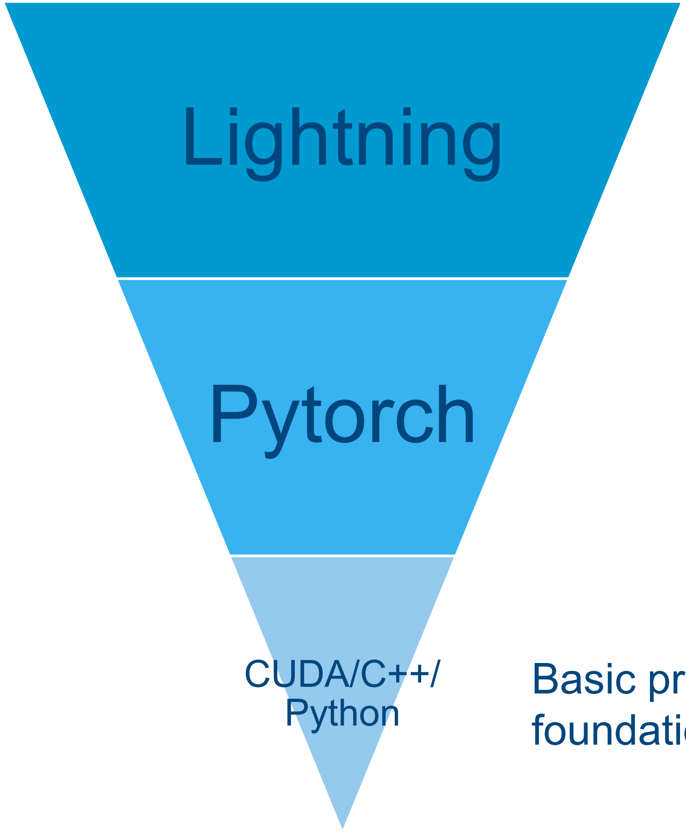
Deep Learning Course

Caroline Arnold

11.03.2026

Helmholtz-Zentrum hereon

What is Lightning and why are we using it?



- Deep learning framework, based on Pytorch
- Focus: Practitioners, experiments
- Handles a lot of boilerplate code for you
- Flexible and performant

- Deep Learning framework
- Production ready
- Code development required

Basic programming languages that are the foundation of what we can do with Pytorch

https://lightning.ai/pages/blog/pl-tutorial-and-overview/

Object oriented programming

Objects combine data and behaviour

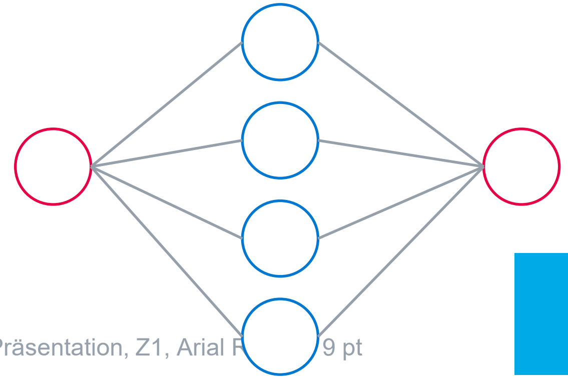
- Attributes
- Methods

Example: nn.Conv2D

- Attributes: weight / bias
- Method: forward function

Example: Deep Learning Model

- Attributes: Neural network
- Methods: forward function, training, validation, …



Class: Blueprint for objects

```
class MyFancyModel():
    def __init__(self, neurons=4):
        self.model = nn.Sequential(
            nn.Linear(1, neurons),
            nn.Linear(neurons, 1))

    def train_step(self, batch):
        x, y = batch
        y_pred = self.model(x)
        return F.mse_loss(y_pred, y)
```

Instances of the class

```
model1 = MyFancyModel(neurons=4)
model2 = MyFancyModel(neurons=16)

loss = model1.train_step( (x, y) )
```

Clean code / Re-use as much as possible

Core Components: LightningModule

Organises your code in 6 sections

1. Initialization (__init__ and setup()).
2. Train Loop ([training_step()]())
3. Validation Loop ([validation_step()]())
4. Test Loop ([test_step()]())
5. Prediction Loop ([predict_step()]())
6. Optimizers and Learning Rate Schedulers ([configure_optimizers()]())

```python
import lightning as L
import torch

from lightning.pytorch.demos import Transformer


class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def forward(self, inputs, target):
        return self.model(inputs, target)

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.model.parameters(), lr=0.1)
```


Helmholtz-Zentrum hereon

Core Components: Trainer

Handles the training for you, while remaining fully configurable if you need more control

BEFORE

```
# enable grads
torch.set_grad_enabled(True)

losses = []
for batch in train_dataloader:
    # calls hooks like this one
    on_train_batch_start()

    # train step
    loss = training_step(batch)

    # clear gradients
    optimizer.zero_grad()

    # backward
    loss.backward()

    # update parameters
    optimizer.step()

    losses.append(loss)
```

Simple interface to the training / validation loops

AFTER

```
model = MyLightningModule()

trainer = Trainer()
trainer.fit(model, train_dataloader, val_dataloader)
```

Yet flexible with > 20 flags, for example

- Device (CPU/GPU)
- Number of devices to use → scaling!
- Epoch length and number of epochs
- Callback that can for example stop training or save the best model