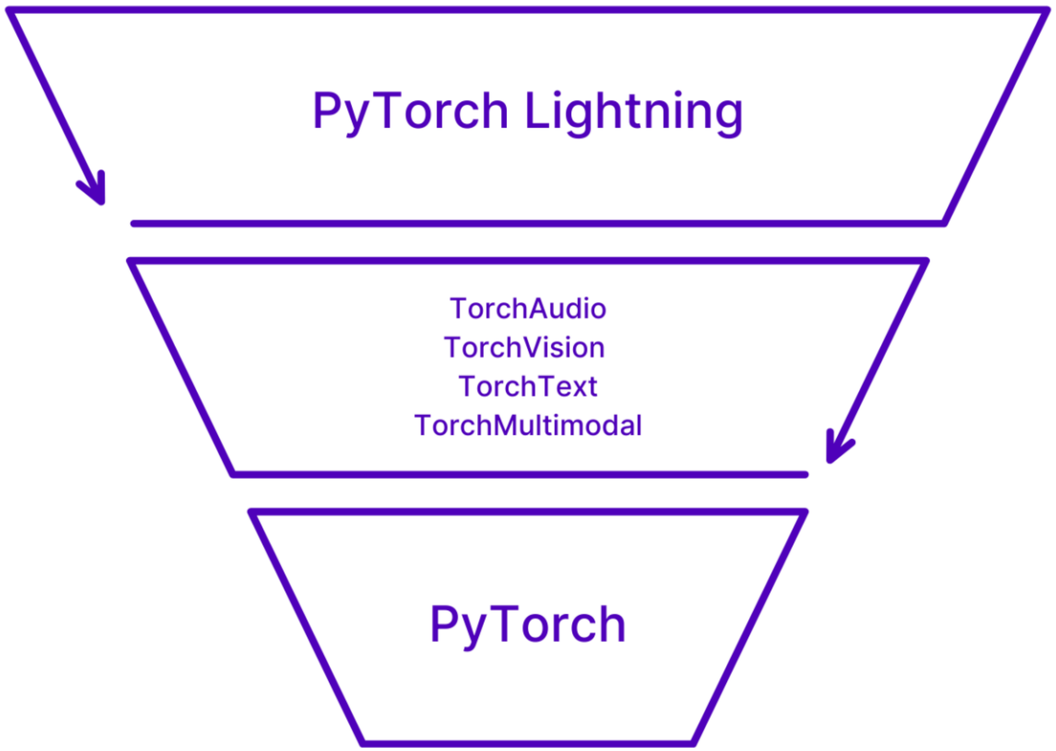
Elegantly Configured ML Training Frameworks with Lightning & Hydra

Ceanga (Rădulescu) Teodora - 20.11.2025

PyTorch Lightning

https://lightning.ai/docs/pytorch/stable/

- A Deep learning framework built on top of PyTorch for researchers & engineers
- Faster iterations than vanilla PyTorch
- Key advantages
  - Organization (standardization , decoupling)
  - Scalability (multi GPU, TPU)
  - Reproducibility



Hydra

https://hydra.cc/

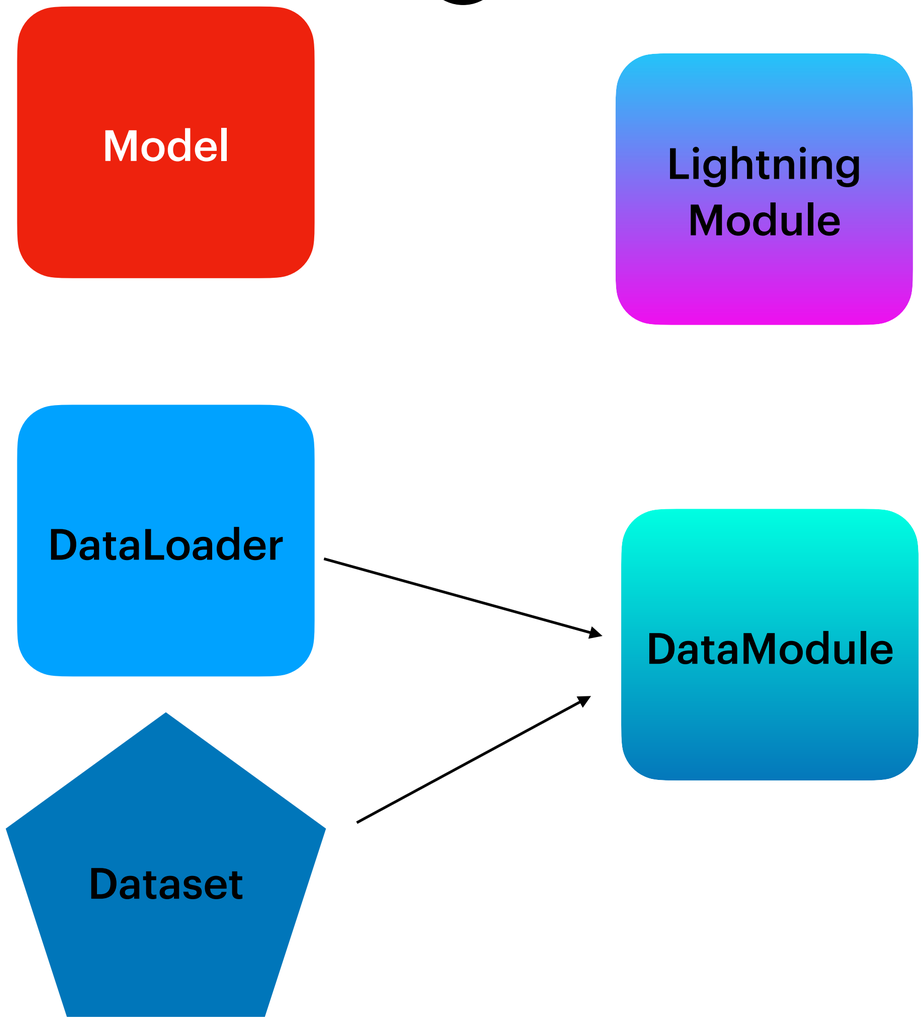
- A framework for elegantly configuring complex applications
- Key advantages:
  - No boilerplate
  - Powerful configuration
  - Pluggable architecture

Lightning + Hydra

https://github.com/ashleve/lightning-hydra-template

- **Rapid Experimentation**: thanks to hydra command line superpowers
- **Minimal Boilerplate**: thanks to automating pipelines with config instantiation
- **Main Configs**: allow you to specify default training configuration
- **Experiment Configs**: allow you to override chosen hyperparameters and version control experiments
- **Workflow**: comes down to 4 simple steps
- **Experiment Tracking**: Tensorboard, W&B, Neptune, Comet, MLFlow and CSVLogger
- **Logs**: all logs (checkpoints, configs, etc.) are stored in a dynamically generated folder structure
- **Hyperparameter Search**: simple search is effortless with Hydra plugins like Optuna Sweeper
- **Tests**: generic, easy-to-adapt smoke tests for speeding up the development
- **Continuous Integration**: automatically test and lint your repo with Github Actions
- **Best Practices**: a couple of recommended tools, practices and standards

PyTorch -> Lightning




```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are computed during test mo
    # also serves to reduce unnecessary gradient computations and memory usage for tensors with requi
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

torch.optim.SGD.step

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

Lightning DataModule

https://lightning.ai/docs/pytorch/latest/data/datamodule.html

A LightningDataModule implements 7 key methods:

- **def prepare_data(self):**
  - Things to do on 1 GPU/TPU (not on every GPU/TPU in DDP).
  - Download data, pre-process, split, save to disk, etc...
- **def setup(self, stage):**
  - Things to do on every process in DDP.
  - Load data, set variables, etc...

- **def train_dataloader(self):**
- **def val_dataloader(self):**
- **def test_dataloader(self):**
- **def predict_dataloader(self):**

- **def teardown(self, stage):**
  - Called on every process in DDP.
  - Clean up after fit or test

Lightning Module

https://lightning.ai/docs/pytorch/latest/common/lightning_module.html

A `LightningModule` implements 8 key methods:

- **def __init__(self):**
  - Define initialization code here.

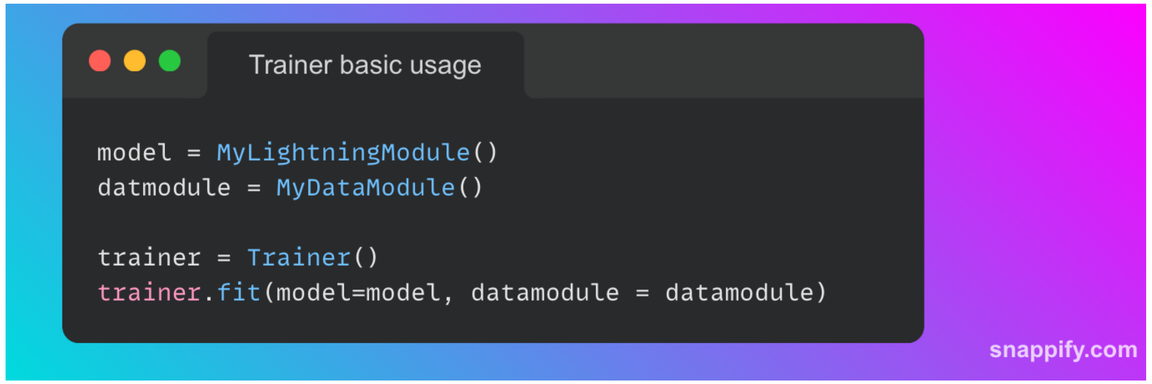
- **def setup(self, stage):**
  - Things to setup before each stage, 'fit', 'validate', 'test', 'predict'.
  - This hook is called on every process when using DDP.

- **def training_step(self, batch, batch_idx):**
  - The complete training step.
- **def validation_step(self, batch, batch_idx):**
  - The complete validation step.
- **def test_step(self, batch, batch_idx):**
  - The complete test step.
- **def predict_step(self, batch, batch_idx):**
  - The complete predict step.

- **def configure_optimizers(self):**
  - Define and configure optimizers and LR schedulers.

Lightning Trainer

https://lightning.ai/docs/pytorch/latest/common/trainer.html



- Under the hood the Trainer handles:
  - Enabling / disabling gradients
  - Iterating through the data loaders
  - Putting data & computations on correct devices
  - Backpropagation & parameter updates
  - Optimizer and LR Scheduler calls

Lightning Callbacks

https://lightning.ai/docs/pytorch/latest/extensions/callbacks.html

- The training loop automated by the Trainer can be customized using Callbacks.
- Callbacks are arbitrary self-contained programs that can be added to the training (for customizing things like checkpointing, model summary, progress bars, etc) using **hooks**
- Common hooks:
  - on_load_checkpoint
  - on_validation_epoch_start, on_training_epoch_end
  - on_before_backward, on_after_backward

Hydra - getting started

```
pip install hydra-core --upgrade
```

src > train copy.py > ...

```python
import hydra
from lightning import LightningDataModule, LightningModule, Trainer
from omegaconf import DictConfig



def train(cfg: DictConfig):

    datamodule: LightningDataModule = hydra.utils.instantiate(cfg.data)

    model: LightningModule = hydra.utils.instantiate(cfg.model)

    trainer: Trainer = hydra.utils.instantiate(cfg.trainer)

    if cfg.get("train"):
        trainer.fit(model=model, datamodule=datamodule, ckpt_path=cfg.get("ckpt_path"))

    if cfg.get("test"):
        ckpt_path = trainer.checkpoint_callback.best_model_path
        trainer.test(model=model, datamodule=datamodule, ckpt_path=ckpt_path)


@hydra.main(version_base="1.3", config_path="../configs", config_name="train.yaml")
def main(cfg: DictConfig):
    """Main entry point for training.

    :param cfg: DictConfig configuration composed by Hydra.
    """
    train(cfg)


if __name__ == "__main__":
    main()

```

Basic folder structure:

- my_project/
  - configs/
    - train.yaml
  - src/
    - data/
    - models/
    - train.py

Hydra - example config

configs > ! train copy.yaml

```yaml
# specify here default configuration
# order of defaults determines the order in which configs override each other
defaults:
  - _self_
  - data: mnist
  - model: mnist
  - trainer: default

  # experiment configs allow for version control of specific hyperparameters
  # e.g. best hyperparameters for given model and datamodule
  - experiment: null

# set False to skip model training
train: True

# evaluate on test set, using best model weights achieved during training
# lightning chooses best weights based on the metric specified in checkpoint callback
test: True

# simply provide checkpoint path to resume training
ckpt_path: null

```

configs > trainer > ! default.yaml

```yaml
_target_: lightning.pytorch.trainer.Trainer

default_root_dir: ${paths.output_dir}

min_epochs: 1 # prevents early stopping
max_epochs: 10

accelerator: cpu
devices: 1

# mixed precision for extra speed-up
# precision: 16

# perform a validation loop every N training epochs
check_val_every_n_epoch: 1

# set True to to ensure deterministic results
# makes training slower but gives more reproducibility than just setting seeds
deterministic: False
```

configs > data > ! mnist.yaml

```yaml
_target_: src.data.mnist_datamodule.MNISTDataModule
data_dir: ${paths.data_dir}
batch_size: 128 # Needs to be divisible by the number of devices (e.g., if in a distributed setup)
train_val_test_split: [55_000, 5_000, 10_000]
num_workers: 0
pin_memory: False

```

Value interpolation

configs > model > ! mnist.yaml

```yaml
_target_: src.models.mnist_module.MNISTLitModule

optimizer:
  _target_: torch.optim.Adam
  _partial_: true
  lr: 0.001

scheduler:
  _target_: torch.optim.lr_scheduler.ReduceLROnPlateau
  _partial_: true
  mode: min
  factor: 0.1
  patience: 10

net:
  _target_: src.models.components.simple_dense_net.SimpleDenseNet
  input_size: 784
  lin1_size: 64
  lin2_size: 128
  lin3_size: 64
  output_size: 10

# compile model for faster training with pytorch 2.0
compile: false

```

Hydra - advanced features

- **Value interpolation**
  - https://hydra.cc/docs/upgrades/1.0_to_1.1/defaults_list_interpolation/
- **Custom resolvers**
  - https://omegaconf.readthedocs.io/en/2.1_branch/custom_resolvers.html
- **Partial instantiation**
  - https://hydra.cc/docs/advanced/instantiate_objects/overview/

Lightning-hydra-template

Code walkthrough

https://github.com/ashleve/lightning-hydra-template

Cool Tricks

mnist.yaml

configs > model > mnist.yaml

```yaml
_target_: src.models.mnist_module.MNISTLitModule

optimizer:
  _target_: torch.optim.Adam
  _partial_: true
  lr: 0.001

scheduler:
  _target_: torch.optim.lr_scheduler.ReduceLROnPlateau
  _partial_: true
  mode: min
  factor: 0.1
  patience: 10
```

Override any config

```
python train.py trainer.max_epochs=20 model.optimizer.lr=1e-4
```

Add new parameters with + sign

```
python train.py +model.optimizer.weight_decay=0.1
```

Accelerators and distributed training

```
# train on CPU
python train.py trainer.accelerator=cpu

# train on 1 GPU
python train.py trainer.accelerator=gpu

# train on TPU
python train.py +trainer.tpu_cores=8

# train with DDP (Distributed Data Parallel) (4 GPUs)
python train.py trainer.accelerator=gpu trainer.strategy=ddp trainer.devices=4

# train with DDP (Distributed Data Parallel) (8 GPUs, 2 nodes)
python train.py trainer.accelerator=gpu trainer.strategy=ddp trainer.devices=4 trainer.num_nodes=2

# simulate DDP on CPU processes
python train.py trainer.accelerator=cpu trainer.strategy=ddp_spawn trainer.devices=2

# accelerate training on mac
python train.py trainer.accelerator=mps
```

Other tricks

Training / Optimization

- `max_epochs`
- `min_epochs`
- `max_steps`
- `limit_train_batches`
- `limit_val_batches`
- `limit_test_batches`
- `limit_predict_batches`
- `gradient_clip_val`
- `gradient_clip_algorithm`
- `accumulate_grad_batches`
- `deterministic`
- `benchmark`

Devices & Distributed Training

- `devices` (e.g., `"auto"`, `1`, `4`, `[0,1]`)
- `accelerator` (`"cpu"`, `"gpu"`, `"cuda"`, `"mps"`, `"tpu"`, `"auto"`)
- `strategy` (`"ddp"`, `"deepspeed"`, `"fsdp"`, `"dp"`, `"auto"`)
- `num_nodes`
- `precision` (`16`, `32`, `bf16`, `"16-mixed"`, etc.)
- `sync_batchnorm`

.

Validation & Early Stopping

- `val_check_interval`
- `check_val_every_n_epoch`
- `enable_validation`
- `reload_dataloaders_every_n_epochs`

Logging / Checkpointing

- `logger`
- `enable_checkpointing`
- `default_root_dir`
- `log_every_n_steps`
- `enable_progress_bar`
- `enable_model_summary`
- `callbacks`