The Modern PyTorch Workflow: From Boilerplate to Lightning.

Scaling PyTorch Experiments Everywhere with Lightning 2.x + Fabric.

Siman Giri

MADAN BHANDARI UNIVERSITY OF SCIENCE AND TECHNOLOGY
HERALD COLLEGE
CENTRE FOR AI & EMERGING TECHNOLOGIES
Super Computing Center Kirtipur, Kathmandu

Disclaimer!!!

- I am not a core **PyTorch developer** – I haven't been involved in designing its internal packages.
- Then Why am I here:
  - While I didn't design PyTorch, I've been living in its world for over four years doing AI research.
  - Today, I'll share my lessons learned,
    - plus, the latest Lightning 2.x and Fabric updates that
      - help you scale models without losing your sanity.

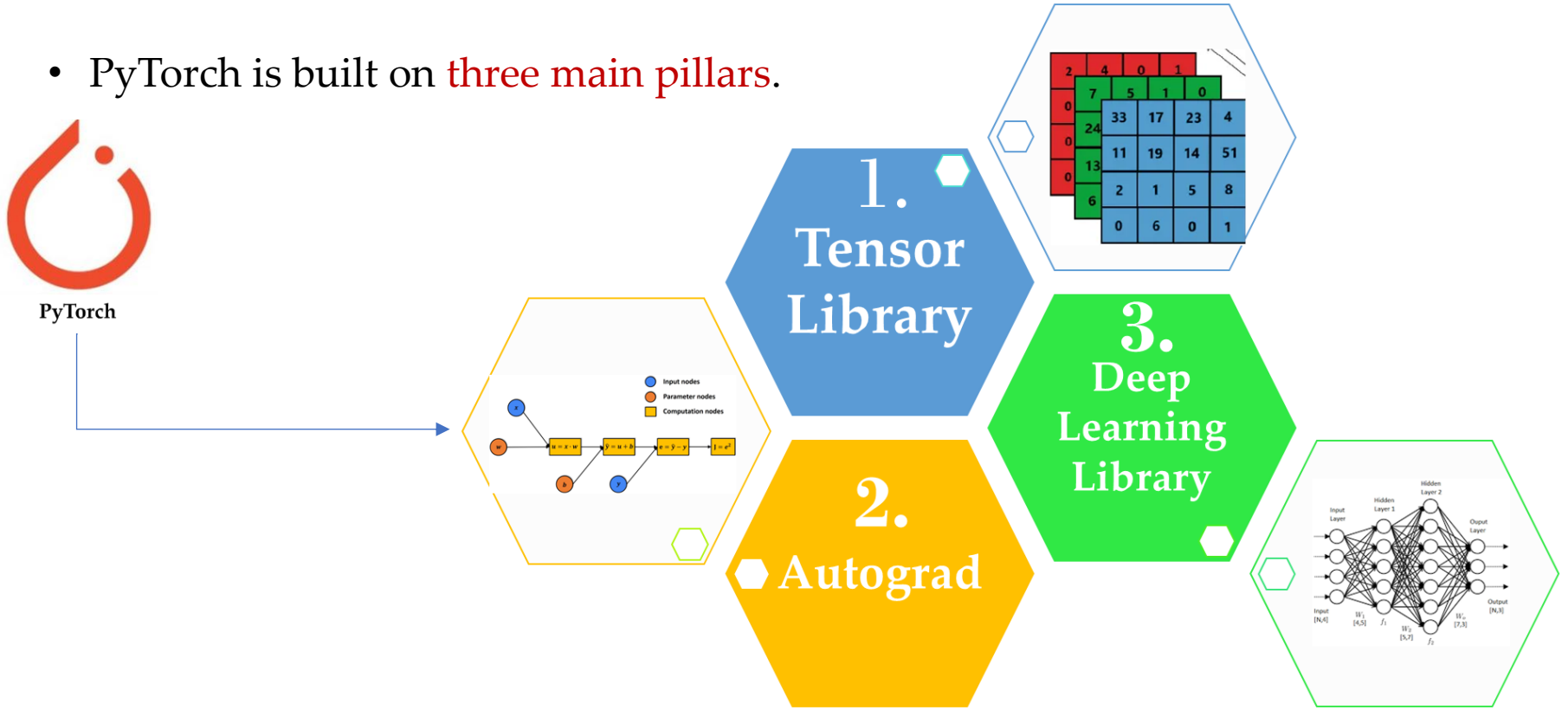
1. Introduction.

About Me!!!

- **Siman Giri**
  - **B.E. Biomedical, MS Applied Mathematics and Machine Learning, MSc Mathematics**
    - **Doctoral Researcher @ Madan Bhandari University of Science and Technology.**
      - **Advised By: Dr. Suresh Manandhar**
      - Research Focus: Theoretical and Mathematical Framework for Reasoning in Multimodal AI.
      - Current Work: Robust Object Tracking for small Video Language Model.
    - **Head @ Centre for AI and Emerging Technologies at Herald College Kathmandu.**
      - Research Unit: IoT & Robotics Unit, Vision & Language Research Unit
      - Objective: Prepare Undergrad for world of Modern-Day AI Research
    - **Researcher @ TU Supercomputing Center.**
      - Research Focus: HPC Infrastructure for AI and Quantum Research.
      - **Advised By: Dr. Madhav Pd. Ghimire (Head of the Center).**

1.1 Understanding PyTorch: The Three Pillars.

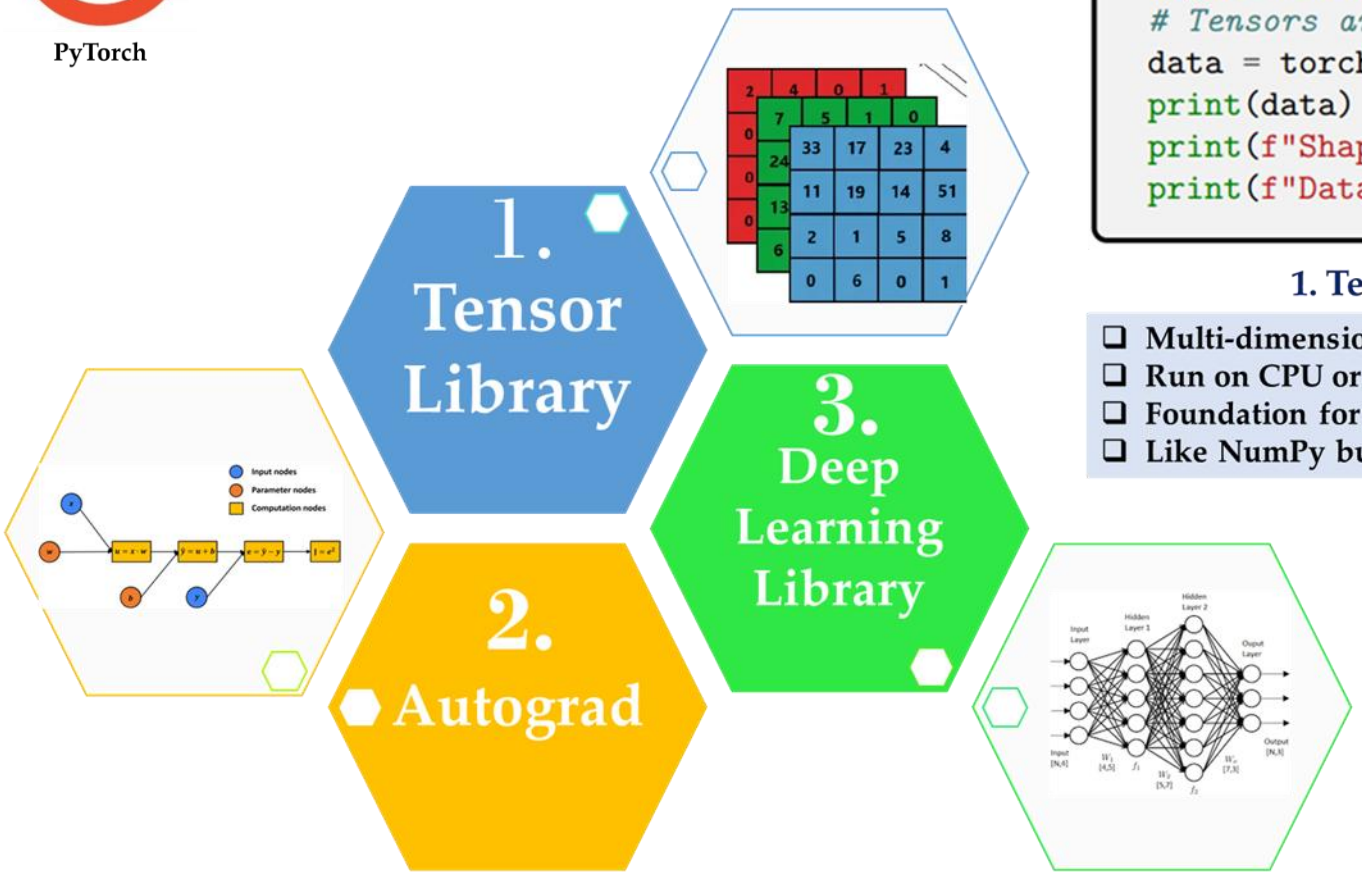
- PyTorch is built on three main pillars.



✓ At its core, PyTorch combines a **flexible tensor library**, a dynamic **automatic differentiation system**, and **practical deep learning utilities**.
✓ Together, these components enable researchers to **efficiently build, optimize, and train deep neural network models**.

1.1 Understanding PyTorch: The Three Pillars.

PyTorch



PyTorch Tensor Example

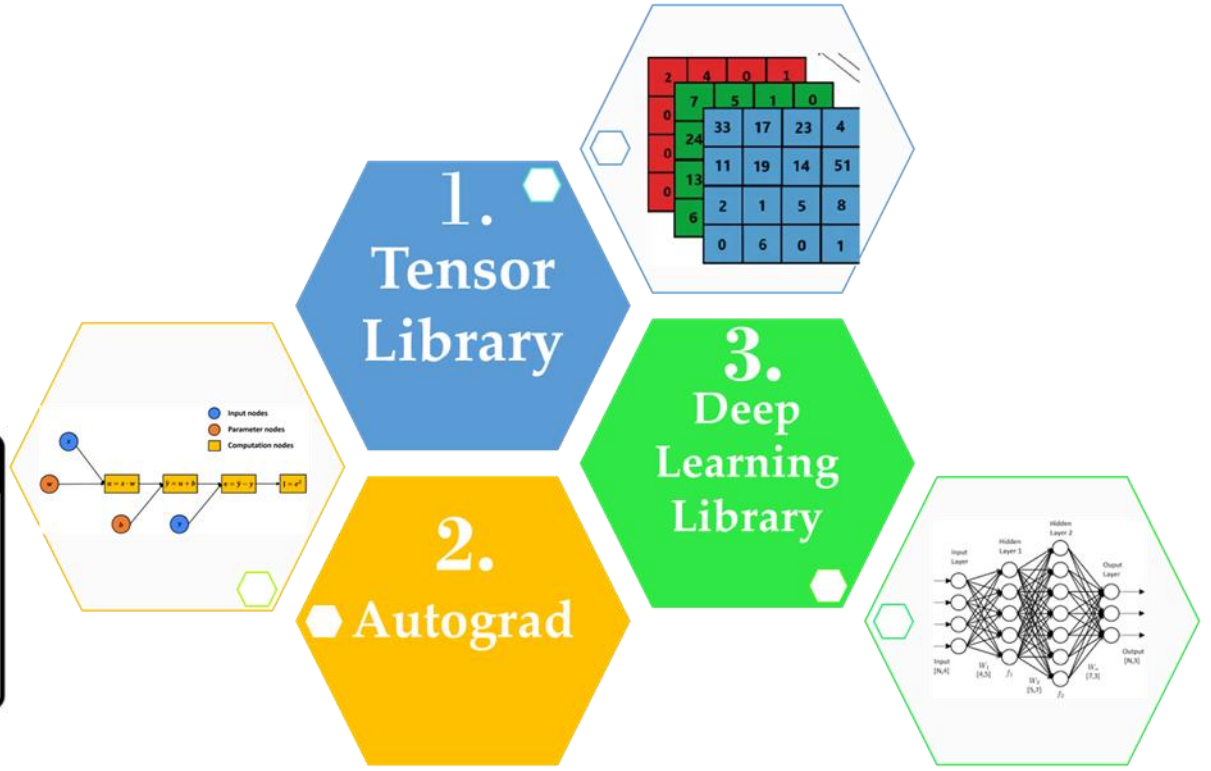
```
import torch
# Tensors are like NumPy arrays but with superpowers
data = torch.tensor([[1, 2], [3, 4]])
print(data)
print(f"Shape: {data.shape}")
print(f"Data type: {data.dtype}")
```

1. Tensors: The Universal Data Container

- **Multi-dimensional arrays** that can hold any type of data (images, text, audio).
- **Run on CPU or GPU** seamlessly → massive speedup for computations.
- **Foundation for everything**: All data in PyTorch flows as tensors through the network.
- **Like NumPy but better**: Similar interface but with GPU support and gradient tracking.

1.1 Understanding PyTorch: The Three Pillars.

PyTorch



Autograd Example

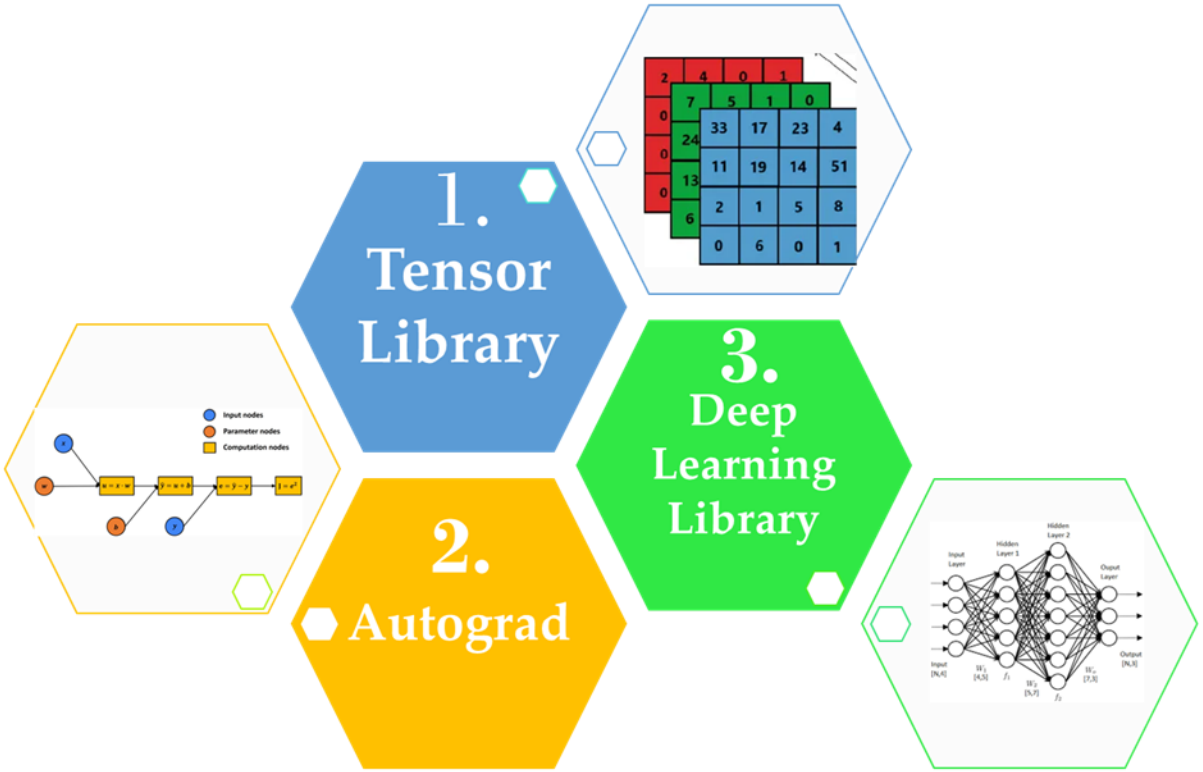
```
x = torch.tensor([2.0], requires_grad=True)
y = x ** 2  # y = x²
y.backward()  # Auto-computes dy/dx = 2x
print(x.grad)  # Output: tensor([4.]) ← Magic!
```

2. Autograd : Automatic Differentiation Engine

- **Automatically computes gradients** (derivatives) for optimization
- **Builds computation graphs** behind the scenes
- **Enables backpropagation** with just .backward() call
- **Why it matters**: Manually calculating derivatives for deep networks (millions of parameters) is impossible;
  - autograd does it automatically.

1.1 Understanding PyTorch: The Three Pillars.

PyTorch



Deep Learning Libraries

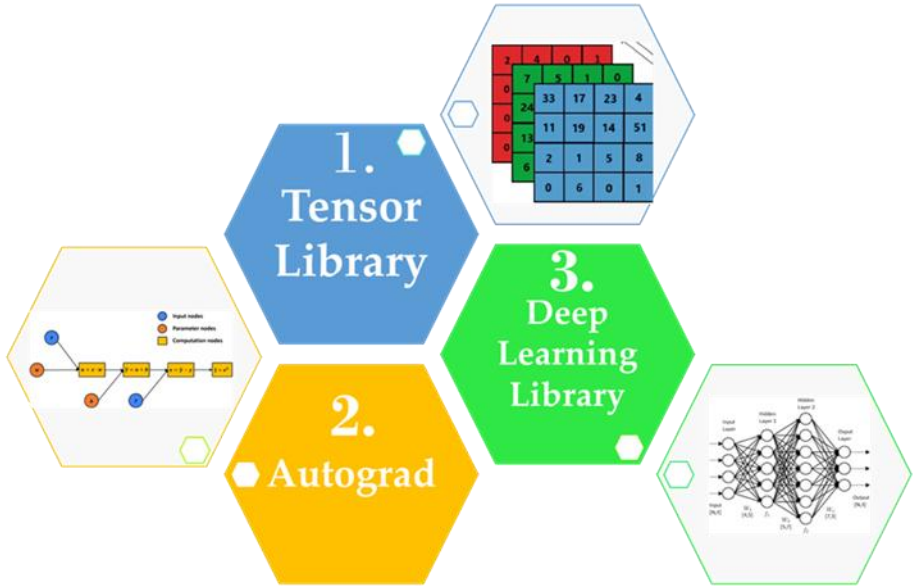
```python
import torch.nn as nn

# Pre-built layers
model = nn.Sequential(
    nn.Linear(100, 50),   # Fully connected layer
    nn.ReLU(),            # Activation function
    nn.Linear(50, 10)     # Output layer
)
```

3. Neural Network Library: Pre-built AI Components

- ❑ **Pre-implemented layers** (convolutional, recurrent, attention, etc.)
- ❑ **Loss functions** (cross-entropy, MSE, etc.)
- ❑ **Optimizers** (SGD, Adam, etc.)
- ❑ **Utilities** (data loaders, transformers).

1.2 Why These Matters?



PyTorch

How They Work Together: A Complete Picture

```
# 1. TENSORS: Create input data
inputs = torch.randn(32, 3, 224, 224)  # Batch of 32 images

# 2. NEURAL NETWORK: Define model
model = torchvision.models.resnet18()

# 3. AUTGRAD: Training loop
outputs = model(inputs)
loss = nn.CrossEntropyLoss()(outputs, labels)
loss.backward()  # Autograd computes all gradients!
optimizer.step()  # Update weights using those gradients
```

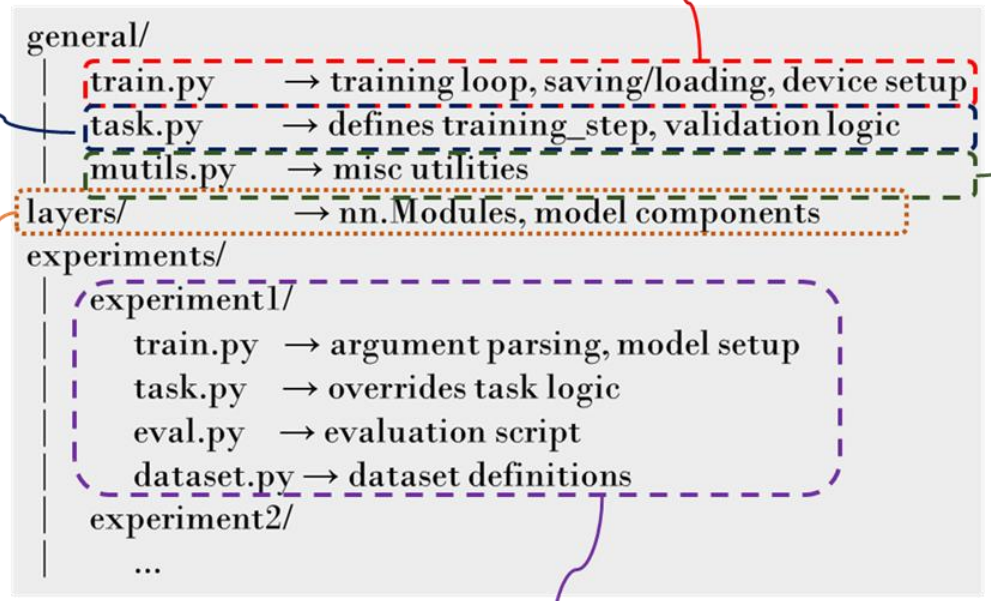
- ✓ Tensors → represent data.
- ✓ Autograd → computes gradients.
- ✓ Deep learning libs → build, train, and optimize models.

- ✓ Together they form a **flexible, low-level deep learning framework** that researchers love.

1.2 The PyTorch Paradox: Unlimited Freedom, Repeated Structure.

- One possible way to Build Projects using PyTorch – {**Suggestive**} Only.

A **shared script** that handles the universal tasks: `training loop`, `checkpointing`, `device setup`, and `experiment management` reused across all **experiments.**"



Provides a **template** for task-specific logic: defining the `training step`, `validation step`, and other `model behaviors`."

contains all **custom** "`nn.Module`" components used to build your **model components**.

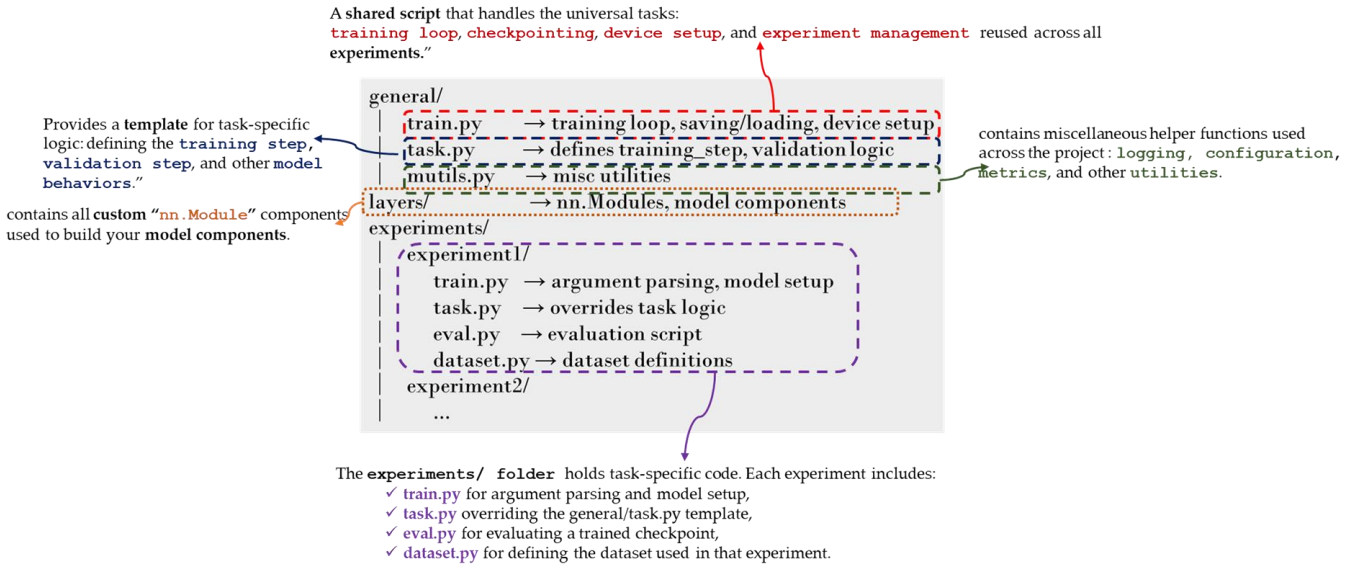
contains miscellaneous helper functions used across the project : `logging`, `configuration`, `metrics`, and other `utilities`.

The `experiments/ folder` holds task-specific code. Each experiment includes:

- ✓ train.py for argument parsing and model setup,
- ✓ task.py overriding the general/task.py template,
- ✓ eval.py for evaluating a trained checkpoint,
- ✓ dataset.py for defining the dataset used in that experiment.

1.2 The PyTorch Paradox: Unlimited Freedom, Repeated Structure.

- One possible way to Build Projects using PyTorch – {**Suggestive**} Only.

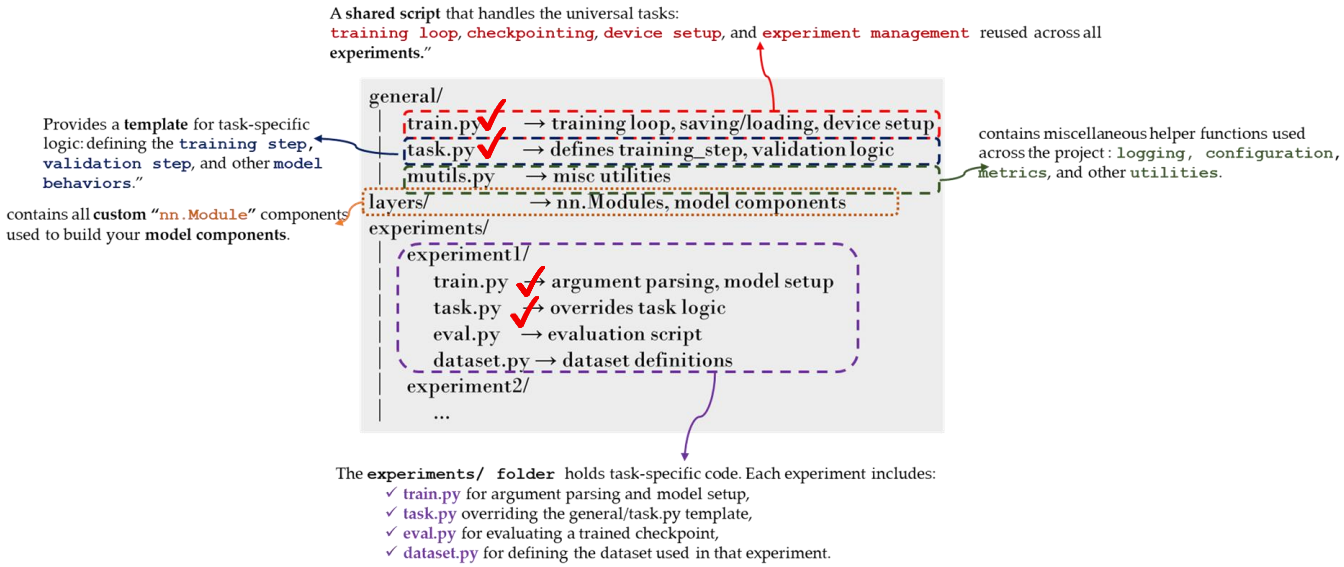


❑ **PyTorch's Philosophy:**
- ✓ **Explicit over implicit**:
  - ✓ You see and control every step
- ✓ **Pythonic**:
  - ✓ It feels like regular Python code
- ✓ **Dynamic**:
  - ✓ Computation graphs are built on-the-fly

If **PyTorch** is great, then **what** are the limitations?

1.2 The PyTorch Paradox: Unlimited Freedom, Repeated Structure.

- One possible way to Build Projects using PyTorch – {**Suggestive**} Only.



❑ **PyTorch's Philosophy:**
- ✓ **Explicit over implicit**:
  - ✓ You see and control every step
- ✓ **Pythonic**:
  - ✓ It feels like regular Python code
- ✓ **Dynamic**:
  - ✓ Computation graphs are built on-the-fly

If **PyTorch** is great, then **what** are the limitations?

1.3 PyTorch: Flexibility with a Side of Boilerplate

The Universal PyTorch Training Template

```python
# THE BOILERPLATE YOU WRITE 100 TIMES
def train_one_epoch(model, dataloader, optimizer, device):
    model.train()
    for batch in dataloader:
        x, y = batch
        x, y = x.to(device), y.to(device)    # ← Always this line

        optimizer.zero_grad()
        y_hat = model(x)
        loss = loss_fn(y_hat, y)
        loss.backward()
        optimizer.step()

        # Logging (different each time)
        writer.add_scalar('loss', loss, step)

        # Checkpoint logic (reinvented each project)
        if step % 1000 == 0:
            torch.save(model.state_dict(), f'ckpt_{step}.pt')

    return average_loss
```

Fig: This is what a Standard Training Loop in PyTorch Looks like, So what is the challenge?

1.3 PyTorch: Flexibility with a Side of Boilerplate

The Universal PyTorch Training Template

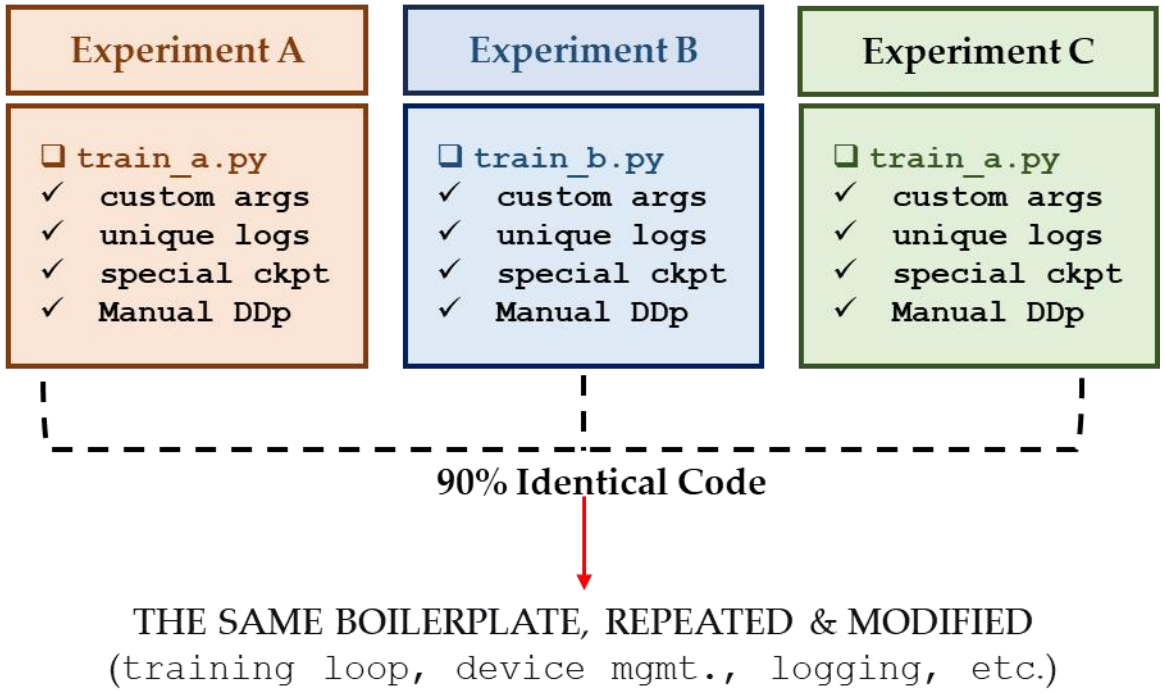
```python
# THE BOILERPLATE YOU WRITE 100 TIMES
def train_one_epoch(model, dataloader, optimizer, device):
    model.train()
    for batch in dataloader:
        x, y = batch
        x, y = x.to(device), y.to(device)    # ← Always this line

        optimizer.zero_grad()
        y_hat = model(x)
        loss = loss_fn(y_hat, y)
        loss.backward()
        optimizer.step()

        # Logging (different each time)
        writer.add_scalar('loss', loss, step)

        # Checkpoint logic (reinvented each project)
        if step % 1000 == 0:
            torch.save(model.state_dict(), f'ckpt_{step}.pt')

    return average_loss
```



The irony: We write the same **infrastructure code** repeatedly while trying to do novel research.

1.4 The Hidden Costs of Reinventing the Wheel.

1. **Repetitive Boilerplate**
   - **Problem:**
     - Duplicated code across experiments increases maintenance cost.
     - Small mistakes in copy – pasted loops lead to **silent** training bugs.
     - Researchers spend more time re – writing loops than innovating on models.
   - **Impact:**
     - Slows down experimentation, reduces productivity, and creates inconsistencies across projects.
2. **Manual Device Handling**
   - **Problem:**
     - Device placement `(.to(device))` scattered throughout code leads to errors.
     - Switching from `CPU → GPU → Multi-GPU → TPU` forces complex code rewrites.
     - Device logic **pollutes** model/training logic.
   - **Impact:**
     - Fragile workflows, increased debugging time, and **non – portable code** across environments.
3. **DIY Experiment Management**
   - **Problem:**
     - Researchers must implement `logging, checkpointing, reproducibility`, and `monitoring` on their own.
     - Each **implementation** is different, inconsistent, and prone to errors.
     - Reproducibility relies on **manually** setting seeds and managing randomness – often overlooked.
   - **Impact:**
     - Harder collaboration, poorer reproducibility, and duplicated engineering effort across teams.
4. **Scaling Complexity**
   - **Problem:**
     - Distributed training (**DDP, FSDP, TPUs, AMP**) requires deep engineering knowledge.
     - Wrong configuration **silently produces** incorrect gradients or broken synchronization.
     - Scaling even a simple model **requires rewriting** loops and setup code.
   - **Impact:**
     - High barrier to scaling, **greater likelihood** of distributed **bugs,** and slower research iteration.

1.4.1 The Hidden Costs of Reinventing the Wheel.

5. **Dataset Duplication**
   - **Problem:**
     - Slight changes in augmentation or data splits lead to almost identical `dataset.py` files.
     - Each experiment contains repeated `DataLoader` logic.
     - Hard to maintain consistency across experiments.
   - **Impact:**
     - Code bloat, inconsistent preprocessing, and increased chance of data-related bugs.
6. **Error-Prone Infrastructure**
   - **Problem:**
     - Custom training loops hide subtle bugs (gradient accumulation errors, `missing .eval()`, incorrect `zero_grad`, wrong validation frequency).
     - Distributed training bugs are extremely hard to detect.
     - Checkpointing and resuming training is fragile and usually incomplete.
   - **Impact:**
     - Unreliable experiments, wasted compute, and difficulty reproducing results.

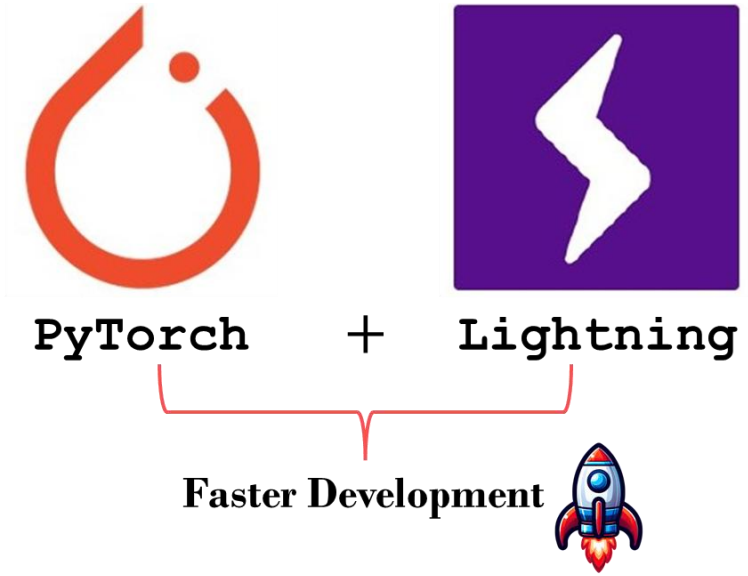
❑ **Raw PyTorch gives full control → but with control comes the burden of writing and maintaining training infrastructure.**

❑ "What if we could keep PyTorch's flexibility but eliminate this repetition?"

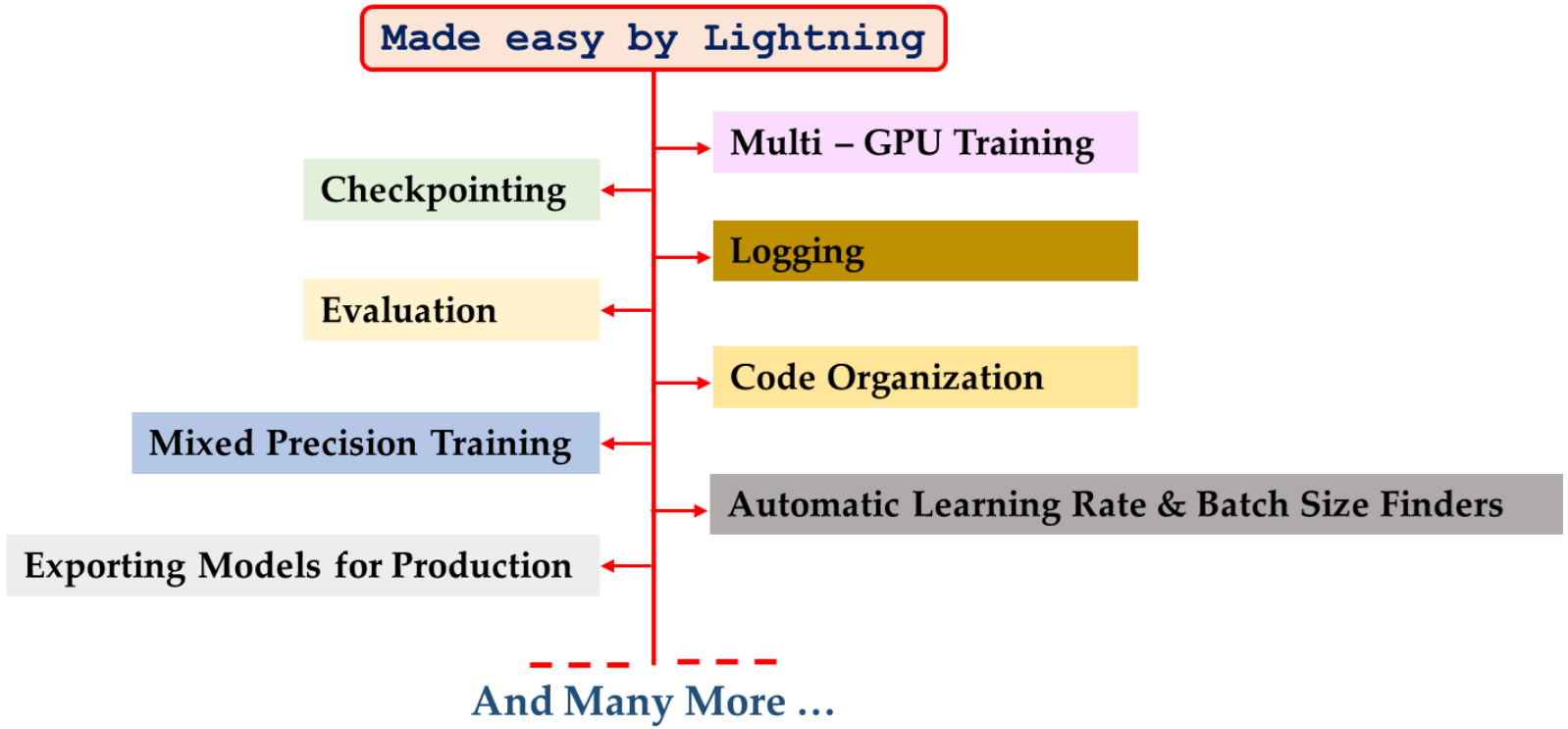
2. Scale, Don't Struggle: Modern Distributed PyTorch with Lightning & Fabric

2.1 Introducing {PyTorch} Lightning
Because I'd Rather Debug My Ideas, Not My Training Loop.

- PyTorch Lightning is a **high-level deep learning framework** built on top of **PyTorch.**
- Its primary goal is simple:
  - *"Let researchers **focus on the science**, not the **boilerplate**."*
- Lightning **separates research logic from engineering code**, giving you a structured, clean workflow for experiments
  - without forcing you to abandon **PyTorch's flexibility**.



2.2 Introducing {PyTorch} Lightning: Less Code, More Results: Lightning in Action.



Big Claim: *"By adopting PyTorch Lightning you could reduce Engineering Overhead in PyTorch Workflows."*

2.3 Lightning by Example: Clean Code, Same Science.

1. **Full Flexibility – Use Raw PyTorch where it Matters:**
   - You still write pure PyTorch inside the model.

Full Flexibility

```
import torch
import torch.nn as nn
import pytorch_lightning as pl
class LitClassifier(pl.LightningModule):
    def __init__(self, lr=1e-3):
        super().__init__()
        self.save_hyperparameters()
        self.model = nn.Sequential(
            nn.Linear(784, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, x):
        return self.model(x)
```

2.3 Lightning by Example: Clean Code, Same Science.

2. Reproducible + Readable :

- Decoupled Research & Engineering

Reproducible + Readable

```python
def training_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    loss = self.loss_fn(logits, y)
    self.log("train_loss", loss)
    return loss

def validation_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    acc = (logits.argmax(dim=1) == y).float().mean()
    self.log("val_acc", acc)
```

Reproducible + Readable

```python
def configure_optimizers(self):
    return torch.optim.Adam(self.parameters(), lr=self.hparams.lr)
```

❑ **Why this proves the point:**

- ✓ Research logic = `training_step, validation_step`
- ✓ Engineering logic (loops, devices, AMP, DDP) = **Trainer**
- ✓ `self.save_hyperparameters()` → full experiment reproducibility

2.3 Lightning by Example: Clean Code, Same Science.

3. **DataModule** – Dataset Handling Without Duplication:

DataModule

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
class MNISTDataModule(pl.LightningDataModule):
    def __init__(self, batch_size=64):
        super().__init__()
        self.batch_size = batch_size

    def setup(self, stage=None):
        transform = transforms.ToTensor()
        self.train_ds = datasets.MNIST(".", train=True, download=True,
        transform=transform)
        self.val_ds = datasets.MNIST(".", train=False, download=True,
        transform=transform)

    def train_dataloader(self):
        return DataLoader(self.train_ds, batch_size=self.batch_size)

    def val_dataloader(self):
        return DataLoader(self.val_ds, batch_size=self.batch_size)
```

Change **dataset**, **augmentation**, or **batch size** → **no change to training loop**.

2.3 Lightning by Example: Clean Code, Same Science.

4. **Simple Multi-GPU / TPU / Mixed Precision** – No Code Changes:

Simple Multi-GPU / TPU / Mixed Precision

```
trainer = pl.Trainer(
    accelerator="gpu",
    devices=4,       # 1 GPU, 4 GPUs, TPU, HPU | same code
    precision=16,    # mixed precision
    max_epochs=10,
    log_every_n_steps=50
)

model = LitClassifier()
data = MNISTDataModule()

trainer.fit(model, datamodule=data)
```

- ❑ **This replaces hundreds of lines of PyTorch boilerplate:**
  - ✓ No `.to(device)`, No DDP setup, No AMP context managers, No manual logging or checkpoint code

2.3 Lightning by Example: Clean Code, Same Science.

5. **Built – in Testing –** Battle –Tested Infrastructure:

Built-in Testing

```
trainer.test(model, datamodule=data)
trainer.predict(model, datamodule=data)
```

❑ **What Lightning already tests for you:**
- ✓ Distributed correctness
- ✓ Mixed precision stability
- ✓ Checkpoint restore
- ✓ Gradient accumulation
- ✓ Fault-tolerant training

*"**Lightning** doesn't change how you think about models — it changes **how much infrastructure** you have to write."*

3. Organizing Your PyTorch Code with Lightning.

{How Lightning organizes your code?}

3.1 The Lightning Mental Model

- *Lightning* organizes **PyTorch** code into 3 core components:

Component	Responsibility
LightningModule	What the model learns
DataModule	Where data comes from
Trainer	How training runs

"*Lightning* enforces separation of concerns → models, data, and training infrastructure never mix."

3.2 The LightningModule

- A **`LightningModule`** organizes your **PyTorch** code into 6 sections:

1.	INITIALIZATION → `__init__() & setup()`
2.	TRAIN LOOP → `training_step()`
3.	VALIDATION LOOP → `validation_step()`
4.	TEST LOOP → `test_step()`
5.	PREDICTION LOOP → `predict_step()`
6.	OPTIMIZERS → `configure_optimizers()`

3.2.1 The LightningModule – Initialization.

Initialization

```python
class LitClassifier(pl.LightningModule):
 def __init__(self, num_classes=10, hidden_dim=128):
     super().__init__()
     # Save hyperparameters (auto-logged!)
     self.save_hyperparameters()

     # Define your model architecture (pure PyTorch)
     self.layers = nn.Sequential(
         nn.Linear(28*28, hidden_dim),
         nn.ReLU(),
         nn.Linear(hidden_dim, num_classes)
     )

 def setup(self, stage=None):
     """Called at beginning of fit/test/predict"""
     # OPTIONAL: Prepare data, initialize weights, etc.
     if stage == "fit":
         # Load pretrained weights, etc.
         pass
```

- **What it does:**
  - ✓ `__init__`: Define your model architecture, layers, hyperparameters
  - ✓ `setup()`: Optional hook for data-dependent initialization
- **Key benefit**: `save_hyperparameters()` automatically logs everything

3.2.2 The LightningModule – Train Loop.

Train Loop (`training_step()`)

```python
def training_step(self, batch, batch_idx):
    # batch comes from your train DataLoader
    x, y = batch

    # 1. Forward pass (no .to(device) needed!)
    y_hat = self(x)  # Calls self.forward()

    # 2. Compute loss
    loss = F.cross_entropy(y_hat, y)

    # 3. Log metrics (auto-logged to all loggers)
    self.log("train_loss", loss,
             on_step=True,   # Log per batch
             on_epoch=True,  # Also log epoch average
             prog_bar=True)  # Show in progress bar

    # 4. Return loss (Lightning handles backward() automatically)
    return loss
```

- **What it does:**
  - ✓ Defines **what happens per training batch**
  - ✓ Lightning automatically calls:
    - `loss.backward()`
    - `optimizer.step()`
    - `optimizer.zero_grad()`
- **Key benefit**: Automatic logging with `self.log()`

3.2.3 The LightningModule – Validation Loop.

Validation Loop (validation_step())

```python
def validation_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self(x)
    loss = F.cross_entropy(y_hat, y)

    # Compute accuracy
    preds = torch.argmax(y_hat, dim=1)
    acc = (preds == y).float().mean()

    # Log validation metrics
    self.log("val_loss", loss, on_epoch=True)
    self.log("val_acc", acc, on_epoch=True)

    # Optional: Return values for custom epoch reduction
    return {"val_loss": loss, "val_acc": acc}

# OPTIONAL: Custom epoch-level aggregation
def on_validation_epoch_end(self):
    # Access all validation_step outputs if needed
    avg_loss = torch.stack([x["val_loss"] ...
    for x in self.validation_step_outputs]).mean()
```

- **What it does:**
  - ✓ Runs **during training** (after each epoch) and **during validation**
  - ✓ No gradient computation (automatic `torch.no_grad()`)
  - ✓ Metrics logged with `on_epoch=True` for proper averaging

3.2.4 The LightningModule – Test Loop.

Test Loop (test_step())

```python
def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)

        # Compute test metrics
        preds = torch.argmax(y_hat, dim=1)
        acc = (preds == y).float().mean()

        # Log test metrics
        self.log("test_acc", acc)

        # Optional: Save predictions for analysis
        return {"preds": preds, "targets": y}

    def on_test_epoch_end(self):
        # Analyze all test results
        all_preds = torch.cat([x["preds"] for x in self.test_step_outputs])
        all_targets = torch.cat([x["targets"] for x in self.test_step_outputs])
        # Compute confusion matrix, F1 score, etc.
```

- **What it does:**
  - ✓ Runs **once** after training is complete
  - ✓ For final evaluation on unseen test data
  - ✓ Identical structure to `validation_step()` but semantically different

3.2.5 The LightningModule – Prediction Loop.

Prediction Loop (predict_step())

```python
def predict_step(self, batch, batch_idx, dataloader_idx=0):
        # Batch could be raw data, no labels
        x = batch  # Just features, no labels

        # Forward pass
        y_hat = self(x)

        # Post-processing (softmax, thresholding, etc.)
        probs = F.softmax(y_hat, dim=1)
        preds = torch.argmax(probs, dim=1)

        # Return predictions in any format
        return {"predictions": preds, "probabilities": probs}
```

- **What it does:**
  - ✓ **Inference-only mode** (`no gradients, no logging`)
  - ✓ For **production inference** or **batch predictions**
  - ✓ Handles data **without labels**
  - ✓ Returns raw **predictions for external use**

3.2.6 The LightningModule – Optimizers.

Optimizers (configure_optimizers())

```python
def configure_optimizers(self):
    # Basic: Single optimizer
    optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)

    # With learning rate scheduler
    optimizer = torch.optim.SGD(self.parameters(), lr=1e-2, momentum=0.9)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30)

    # Return format depends on what you need
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": scheduler,
            "interval": "epoch",  # or "step"
            "frequency": 1,
            "monitor": "val_loss",  # For ReduceLROnPlateau
        }
    }

    # Multiple optimizers (GANs, meta-learning)
    # return [opt1, opt2], [sched1, sched2]
```

- **What it does:**
  - ✓ **Centralizes optimizer configuration**
  - ✓ Supports single/multiple optimizers
  - ✓ Automatic learning rate scheduling
  - ✓ Automatic **`gradient clipping`**, **`frequency`**, etc.

Putting it altogether …

Complete Example

```python
class LitMNIST(pl.LightningModule):
    # 1. INITIALIZATION
    def __init__(self, hidden_dim=128):
        super().__init__()
        self.save_hyperparameters()
        self.layers = nn.Sequential(
            nn.Linear(28*28, hidden_dim),
            nn.Linear(hidden_dim, 10)
        )

    # 2. TRAIN LOOP
    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        self.log("train_loss", loss)
        return loss

    # 3. VALIDATION LOOP
    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        self.log("val_loss", loss)
        return loss

    # 4. TEST LOOP
    def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.log("test_acc", acc)

    # 5. PREDICTION LOOP
    def predict_step(self, batch, batch_idx):
        x = batch
        return self(x).argmax(dim=1)

    # 6. OPTIMIZERS
    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=1e-3)
```

Final Thought: "This isn't a *new way to write PyTorch* → it's just *organizing* what you already write into *logical, reusable components*."

3.3 The Trainer

- Once you've organized your PyTorch code into a LightningModule,
  - the Trainer automates everything else.

YOUR CONTROL	AUTOMATED BEST PRACTICES
You write PyTorch code in LightningModule	Trainer executes it optimally using community-proven patterns
You define WHAT to compute (forward pass, loss, metrics)	Trainer handles HOW to compute it (distribution, optimization, logging)
You can disable ANY automation if you need custom behavior	But defaults come from top AI labs

The Core Philosophy

3.3.1 Basic Use

- This is the basic use of the trainer:

Basic Use of Trainer

```
model = MyLightningModule()

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

3.3.1.1 Under the hood ...

1. **SETUP PHASE**
   - Moves model to correct device(s)
   - Sets up distributed strategy (DDP/FSDP)
   - Initializes precision (fp16/bf16)
   - Configures all callbacks
   - Sets up logging

2. TRAINING LOOP - AUTOMATED

```
for epoch in range(max_epochs):
        # Training
        for batch in train_dataloader:
            loss = model.training_step(batch)
            loss.backward()          # ← AUTOMATIC
            optimizer.step()         # ← AUTOMATIC
            optimizer.zero_grad()    # ← AUTOMATIC

        # Validation
        if epoch % val_check_interval == 0:
            model.eval()
            with torch.no_grad():
                for batch in val_dataloader:
                    model.validation_step(batch)
            model.train()

        # Callbacks execution
        callbacks.on_validation_end()
```

3. **AUTOMATIC CHECKPOINTING**
   - Saves best model(s) automatically
   - Saves latest model every N epochs
   - Manages checkpoint directory

4. **CLEANUP & RESTORABILITY**
   - Handles keyboard interrupts (Ctrl+C)
   - Can resume from any checkpoint
   - Proper distributed teardown

"You write the `training_step()`, Lightning handles **everything else**."

3.3.2 Community Wisdom, Built In

Embedded Best Practices from top AI Labs

```
trainer = pl.Trainer()
# Defaults include:
# From FAIR's DDP experience:
# • Proper process group initialization
# • Correct barrier synchronization
# • Gradient bucketing optimizations

# From Google's TPU research:
# • XLA compilation strategies
# • BF16 mixed precision handling
# • Distributed checkpointing

# From Stanford's ML research:
# • Smart validation scheduling
# • Gradient accumulation patterns
# • Learning rate warmup strategies

# From MIT's optimization research:
# • Gradient clipping defaults
# • Adaptive batch sizing
# • Numerical stability checks
```

3.3.3 Opt – out of Any

Opt out of Any

```python
trainer = pl.Trainer(
    # Disable training loop automation
    automatic_optimization=False,  # ← Manual optimization control

    # Manual validation control
    val_check_interval=None,       # ← Run validation manually

    # Custom checkpointing
    enable_checkpointing=False,    # ← Implement your own

    # Disable automatic logging
    logger=False,                  # ← Log manually
)
# Or override at the LightningModule level:
class CustomModel(pl.LightningModule):
    def training_step(self, batch, batch_idx):
        # Manual optimization example (GANs, Meta-learning)
        opt1, opt2 = self.optimizers()

        # Compute generator loss
        self.toggle_optimizer(opt1)
        loss1 = self.generator_loss(batch)
        self.manual_backward(loss1)
        opt1.step()
        opt1.zero_grad()

        # Compute discriminator loss
        self.toggle_optimizer(opt2)
        loss2 = self.discriminator_loss(batch)
        self.manual_backward(loss2)
        opt2.step()
        opt2.zero_grad()
```

3.3.4 Real – World Trainer Configurations

One Trainer Infinite Configurations

```python
# 1. RESEARCHER ON LAPTOP (Quick Experiments)
researcher = pl.Trainer(
    max_epochs=10,
    fast_dev_run=True,           # Quick sanity check
    overfit_batches=0.01,        # Debug overfitting
    limit_train_batches=0.1,     # Faster iteration
)
```

One Trainer Infinite Configurations

```python
# 2. GRAD STUDENT WITH 2 GPUs
grad_student = pl.Trainer(
    accelerator="gpu",
    devices=2,                   # Uses both GPUs automatically
    strategy="ddp",              # Distributed Data Parallel
    precision="16-mixed",        # 2x memory, 1.5x speed
    max_epochs=100,
)
```

"Same code, same LightningModule, completely different scale."

3.3.4 Real – World Trainer Configurations

One Trainer Infinite Configurations

```python
# 3. INDUSTRY TRAINING CLUSTER
production = pl.Trainer(
    accelerator="gpu",
    devices=8,                    # 8 GPUs across nodes
    strategy="fsdp",              # Fully Sharded Data Parallel
    precision="bf16-mixed",       # Modern precision for large models
    max_epochs=1000,
    gradient_clip_val=1.0,
    accumulate_grad_batches=8,    # Effective batch size = 64
    callbacks=[
        ModelCheckpoint(monitor="val_loss", mode="min"),
        EarlyStopping(monitor="val_loss", patience=20),
        LearningRateFinder(),     # Automatic LR tuning
    ],
    logger=[WandbLogger(), MLFlowLogger()],
    profiler="advanced",          # Performance optimization
)
```

One Trainer Infinite Configurations

```python
# 4. GOOGLE TPU POD
tpu_trainer = pl.Trainer(
    accelerator="tpu",
    devices=8,                    # TPU v3-8 pod
    strategy="xla",               # TPU-specific strategy
    precision="bf16-true",        # Full bfloat16 on TPU
)
```

"Same code, same LightningModule, completely different scale."

Thank You!!!