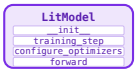
PyTorch Lightning

The key pieces of PyTorch Lightning, which strips away training boilerplate and organizes your model, loops, and logging around a Trainer.

blog.thecoatlessprofessor.com
updated 06/10/2026

The LightningModule

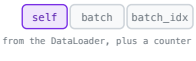
One class that bundles model, data flow, and training logic.

PURPOSE	COMMAND	RESULT
Subclass the base module.	`class LitModel(L.LightningModule):`	
Build layers in the constructor.	`def __init__(self): super().__init__()` `self.net = nn.Sequential(...)`	
Save constructor args as hparams.	`self.save_hyperparameters()`	
Define the forward pass.	`def forward(self, x): return self.net(x)`	
Instantiate the model.	`model = LitModel(lr=1e-3)`	

a LightningModule is a torch.nn.Module with fixed slots that say what to do, not how to loop

Step Hooks

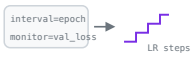
Lightning calls these per-batch; you just return the loss.

PURPOSE	COMMAND	RESULT
Train on one batch.	`def training_step(self, batch, batch_idx):` `loss = F.cross_entropy(self(x), y);return loss`	
Validate on one batch.	`def validation_step(self, batch, batch_idx):` `self.log("val_loss", loss)`	
Test on one batch.	`def test_step(self, batch, batch_idx):`	
Predict on one batch.	`def predict_step(...): return self(batch)`	
Hook into the epoch end.	`def on_train_epoch_end(self):`	
The signature contract.	`(self, batch, batch_idx)`	

training_step returns the loss (Lightning calls backward + step); other steps log metrics or return predictions

Optimizers & Schedulers

Return optimizers from one method; Lightning steps them.

PURPOSE	COMMAND	RESULT
Configure one optimizer.	`def configure_optimizers(self):` `  return torch.optim.Adam(self.parameters())`	
Add an LR scheduler.	`return {"optimizer": opt,` `  "lr_scheduler": sched}`	
Scheduler step interval.	`{"scheduler": s, "interval": "epoch",` `  "monitor": ..}`	
Lightning runs the loop.	~~`loss.backward(); opt.step()`~~ `# automatic`	
Multiple optimizers (manual).	`self.automatic_optimization =False`	
Clip gradients.	`L.Trainer(gradient_clip_val=1.0)`	

by default Lightning runs zero_grad, backward, step for you; clipping and accumulation are set on the Trainer

The Trainer

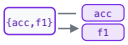
The engine that owns the loop and the hardware.

PURPOSE	COMMAND	RESULT
Build the trainer.	`trainer =L.Trainer(max_epochs=10)`	
Pick hardware automatically.	`L.Trainer(accelerator="auto",` `  devices="auto")`	
Run the full fit loop.	`trainer.fit(model, train_dl, val_dl)`	
Evaluate on a held-out set.	`trainer.test(model, test_dl)`	
Generate predictions.	`preds =trainer.predict(model, dl)`	
Smoke-test the whole pipeline.	`L.Trainer(fast_dev_run=True)`	

device placement, precision, and distribution live in the Trainer, so the same module runs anywhere

Logging Metrics

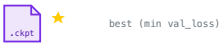
Call self.log; Lightning aggregates and routes it.

PURPOSE	COMMAND	RESULT
Log a scalar.	`self.log("train_loss", loss)`	
Show it in the progress bar.	`self.log("val_acc", acc, prog_bar=True)`	
Control step vs epoch reduce.	`self.log(.., on_step=True, on_epoch=True)`	
Log several at once.	`self.log_dict({"acc": acc, "f1": f1})`	
Pick a backend logger.	`L.Trainer(logger=CSVLogger("logs"))`	
Track a real metric object.	`self.acc =torchmetrics.Accuracy(...)`	

self.log aggregates across batches and devices; a torchmetrics object accumulates the correct value per epoch

Callbacks

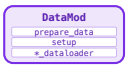
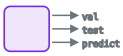
Plug behavior into the loop without touching the model.

PURPOSE	COMMAND	RESULT
Save the best checkpoint.	`ModelCheckpoint(monitor=.., mode="min", save_top_k=1)`	
Stop when it plateaus.	`EarlyStopping(monitor="val_loss", patience=3)`	
Log the learning rate.	`LearningRateMonitor( logging_interval="step")`	
Attach them to the trainer.	`L.Trainer(callbacks=[ckpt, es, lr])`	
Resume from a checkpoint.	`trainer.fit(model, dl, ckpt_path="last.ckpt")`	
Write your own.	`class MyCb(L.Callback): on_train_epoch_end(...)`	

attach callbacks as a list to Trainer(callbacks=[...]); ckpt_path resumes optimizer state too

DataModule

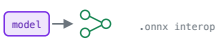
Package download, split, and dataloaders in one class.

PURPOSE	COMMAND	RESULT
Subclass the data base.	`class DataMod(L.LightningDataModule):`	
Download once, on rank 0.	`def prepare_data(self):`	
Build splits per process.	`def setup(self, stage):` `  self.train, self.val = random_split(...)`	
Expose the train loader.	`def train_dataloader(self): DataLoader(bs=32)`	
Add val / test / predict loaders.	`def val_dataloader(self):` `  ... (test_, predict_)`	
Hand it to the trainer.	`trainer.fit(model, datamodule=dm)`	

prepare_data downloads once; setup(stage) splits per process; pass datamodule=dm instead of loose loaders

Save, Load & Export

Checkpoints capture everything; export for serving.

PURPOSE	COMMAND	RESULT
Save a checkpoint by hand.	`trainer.save_checkpoint("model.ckpt")`	
Reload weights + hparams.	`LitModel.load_from_checkpoint(` `  "model.ckpt")`	
Override hparams on load.	`load_from_checkpoint("model.ckpt", lr=1e-4)`	
Get just the weights dict.	`torch.load("model.ckpt",` `  weights_only=False)["state_dict"]`	
Export to ONNX.	`model.to_onnx("model.onnx", sample)`	
Export the graph (TorchScript-free).	`torch.export.export(model, (sample,))`	

a checkpoint stores weights + hparams + optimizer + epoch; to_torchscript is deprecated, prefer torch.export