LESSON NAME

Intro

> Prerequisites
>
> - FIXME
> - …
> - …

20 min	filename

Episode template

> Questions
>
> - What syntax is used to make a lesson?
> - How do you structure a lesson effectively for teaching?
> - `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

> Objectives
>
> - Show a complete lesson page with all of the most common structures.
> - …
>
> This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

> Discussion
>
> g.
>
> Skip to content
>
> ction
>
> - Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

```
print("hello world)
```

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

PyTorch Lightning

Questions

- What is PyTorch lightning and what is it used for?

Skip to content

PyTorch model in Lightning?

ise a Lightning model on several GPUs?

Objectives

- Learn about PyTorch Lightning and why it is useful
- Learn how to wrap a classical torch model in Lightning
- Train Lightning models on multiple GPUs on a SLURM cluster

Introduction

PyTorch Lightning is a lightweight extension of PyTorch that provides structure, automation, and robustness while keeping full flexibility. It does not replace PyTorch, in the sense that you still write PyTorch models, layers, and logic, but it removes much of the repetitive “glue code” around them. Moreover, it simplifies parallelisation over multiple GPUs, requiring virtually no changes to the code if using Lightning types; different parallelisation strategies such as DeepSpeed, DDP and FSDP are readily available and require minimal configuration. In order to achieve this, Lightning requires the developer to wrap their network architecture and logic into Lightning types. This also benefits development since it improves readability of the code, clearly separating ML logic from engineering.

Lightning focuses on three core ideas:

1. **Organize code cleanly**
   Lightning separates the core components of a deep learning project operations:
   - Models
   - Data loading
   - Training loop behavior
   - Logging & checkpointing
   - Distributed execution

   This helps keeping the code modular, readable and more reusable.

2. **Automate engineering, keep architecture flexible**
   The developer can focus on the model and experiments, while device placement or multiprocessing are transparently handled by Lightning. Lightning handles:
   - Training and validation loops
   - Mixed precision
   - Gradient clipping
   - Device and dtype configuration
   - Multi-GPU & multi-node launch logic
   - Checkpointing
   - Logging
3. **Make code portable across hardware and environments**
   With Lightning, the same script can run:
   - on CPU
   - on a single GPU
   - on multiple GPUs using DDP, FSDP or ZeRO

   Switching execution modes becomes a matter of changing a command-line flag rather than rewriting the whole training script.

Lightning building blocks

The core API of Lightning rotates around two objects: `LightningModule` and `Trainer`. `LightningModule` describes the architecture of the network, including forward pass, validation and test loops, optimisers and LR schedulers. Conversely, `Trainer` handles the “engineering” side of things: running training, validation and test dataloaders, calling callbacks at the right time (e.g. checkpointing, logging), transparently handling device placement following the prescribed strategy. In particular, `Callback`s are used to inject custom, non-essential code at appropriate times. This can be very useful for progress tracking, logging and checkpointing.

Demo

In this example, we will build a simple multilayer perceptron to classify flowers belonging to the Iris dataset based on a set of measurements (petal/sepal measurements). We will examine an example script creating this neural network one snippet at a time.

The whole script can be found at `code/iris/iris_example.py` and the submit script at `code/iris/job.slurm`. The code assumes the iris dataset has been downloaded into a `./data` folder. There are scripts to download it included. For people running on Leonardo, all the code and data can be found at `/leonardo/pub/userexternal/ffiusco0/code`.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam
import lightning as L
from lightning.pytorch.callbacks import Callback
```

Our basic imports, plus `Lighthing` and its `Callback`.

```
class IrisClassifier(L.LightningModule):
    def __init__(self):
        super().__init__()
        # Input: 4 features (sepal/petal measurements)
        # Hidden: 16 neurons
        # Output: 3 classes (Setosa, Versicolour, Virginica)
        self.layer_1 = nn.Linear(4, 16)
        self.layer_2 = nn.Linear(16, 3)

    def forward(self, x):
        # Standard Forward Pass
        x = F.relu(self.layer_1(x))
        x = self.layer_2(x)
        return x

    def training_step(self, batch, batch_idx):
        # 1. Unpack batch
        # Tabular data usually comes as (features, labels)
        x, y = batch

        # 2. Forward pass
        logits = self(x)

        # 3. Compute Loss
        # CrossEntropyLoss expects logits for multi-class classification
        loss = F.cross_entropy(logits, y)

        # 4. Log the loss (so our Callback can see it later!)
        self.log("train_loss", loss)

        return loss

    re_optimizers(self):
    dam(self.parameters(), lr=0.01)
```

Here is all that pertains architecture and behaviour of the network:

- In the constructor, we define the network itself (input layer + 16 hidden neurons + output layer)
- The `forward()` method describes the forward pass, which in this case is just a ReLU of the first layer and the output
- The `training_step` method defines the core logic of each training step: get the features and labels from the batch, do the forward pass, compute the loss and log it
- The `configure_optimizers` step schedules an Adam optimiser with a certain learning rate.

```
import numpy as np
from torch.utils.data import TensorDataset, DataLoader

iris = np.load("./data/iris.npz")
X = torch.tensor(iris["X"], dtype=torch.float32) # Features
y = torch.tensor(iris["y"], dtype=torch.long)  # Labels (0, 1, 2)

dataset = TensorDataset(X, y)
train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

model = IrisClassifier()

# Initialize Trainer with the Callback
# We plug a "Spy" Callback into the callbacks list
logger_callback = TrainingLogger()
trainer = L.Trainer(max_epochs=10, callbacks=[logger_callback], accelerator="gpu")

# Train
trainer.fit(model, train_loader)
```

The first part of the snippet loads the Iris dataset from `sklearn` (not crucial, just an easily accessible source). It then converts it into a format digestible by PyTorch. The `IrisClassifier` model we created above is then instantiated. The Lightning `Trainer` object takes care of the *engineering* of the flow: sets a number of epochs, which accelerator to use and possibly number of devices/nodes over which the job can be parallelised with a certain strategy. Note also the inclusion of a `logger_callback`: this exemplifies the use of callbacks to trigger the execution of arbitrary code at certain moments of the training cycle. In this case, our own `TrainingLogger` looks like the following:

```
class TrainingLogger(Callback):
    def on_train_epoch_end(self, trainer, pl_module):
        # This runs automatically at the end of every epoch
        # We access the logged metrics from the module via trainer.callback_metrics
        loss = trainer.callback_metrics.get("train_loss")
        print(f"Spy Report: Epoch {trainer.current_epoch} ended. Loss: {loss:.4f}")
```

Lightning provides some plumbing to create `Callback`s and even some specific types (learning rate schedulers, gradient accumulation, etc.). In the snippet above, the training loss is printed after every epoch. A number of trigger events are available (fit end, fit start, checkpoint loading, test epoch start…).

Lightning data modules

When training neural networks in PyTorch, it is usually needed to find/download the dataset, load it from disk, doing train/validation/test splits and a number of other preprocessing steps. In the previous example, these steps were performed in the launching snippet. In order to improve code organisation and reusability, Lightning introduces a `LightningDataModule`, which takes care of all these steps and returns the correct torch Dataloaders. A generic `LightningDataModule` features a number of hooks:

```
class DataModule(L.LightningDataModule):
  def prepare_data(self):
    ...
  def setup(self, stage=None):
    ...
  def train_dataloader(self):
    ...
  def val_dataloader(self):
    ...
```

- `prepare_data()` runs once and is usually used to download and/or check files. It runs only on one process, even if multiple GPUs are used.
- `setup(stage)` runs on each rank and loads arrays and creates the datasets. The `stage` parameter can be used to prepare data for different steps of the training lifecycle (fit, validate, predict, test)
- `train_dataloader(), val_dataloader(), test_dataloader()` return PyTorch dataloaders for the different splits.

In the next example, we can see how to refactor the Iris training snippet to use a Lightning data module:

Demo

Skip to content

```python
class IrisDataModule(L.LightningDataModule):

    def __init__(
        self,
        data_dir = "./data",
        batch_size = 16,
        num_workers = 0,
        val_size = 0.2,
        random_state = 42,
        shuffle = True,
    ):
        super().__init__()
        self.data_dir = data_dir
        self.data_file = os.path.join(self.data_dir, "iris.npz")
        self.batch_size = batch_size
        self.num_workers = num_workers
        self.val_size = val_size
        self.random_state = random_state
        self.shuffle = shuffle

        self.train_ds = None
        self.val_ds = None

        self.save_hyperparameters(ignore=["num_workers"])

    def prepare_data(self):
        if not os.path.exists(self.data_file):
            raise FileNotFoundError(
                f"Expected dataset at {self.data_file}. "
                f"Run the download script provided to create it."
            )

    def setup(self, stage=None):
        data = np.load(self.data_file)
        X = data["X"].astype(np.float32)   # shape (150, 4)
        y = data["y"].astype(np.int64)     # shape (150,)

        # Train/val split (stratified to keep class balance)
        X_train, X_val, y_train, y_val = train_test_split(
            X, y, test_size=self.val_size, random_state=self.random_state, stratify=y
        )

        # Convert to tensors
        X_train = torch.from_numpy(X_train)
        y_train = torch.from_numpy(y_train)
        X_val = torch.from_numpy(X_val)
        y_val = torch.from_numpy(y_val)

        self.train_ds = TensorDataset(X_train, y_train)
        self.val_ds = TensorDataset(X_val, y_val)

    def train_dataloader(self):
        return DataLoader(
            self.train_ds,
            batch_size=self.batch_size,
            ffle=self.shuffle,
            workers=self.num_workers,


    def val_dataloader(self):
```

```
        return DataLoader(
            self.val_ds,
            batch_size=self.batch_size,
            shuffle=False,
            num_workers=self.num_workers,
        )
```

Now to train the network:

```
# Assume that IrisClassifier and TrainingLogger are available

dm = IrisDataModule(data_dir="./data", batch_size=16, val_size=0.2, random_state=42)
logger_callback = TrainingLogger()
trainer = L.Trainer(max_epochs=10, callbacks=[logger_callback], accelerator="gpu")
model = IrisClassifier()

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

The whole script can be found at `code/iris/iris_with_datamodule.py` and the submit script at `code/iris/job.slurm`.

Lightning works extremely well in SLURM-like environments since the number of nodes/devices and the parallelisation strategy can be passed as arguments to the `Trainer` object and can be fed from SLURM environmental variables (`$SLURM_GPUS_PER_NODE`, `$SLURM_NNODES`). This effectively means that both the Python script itself and the submit script can stay the same. For example, the number of epochs, nodes, GPUs per node and parallelisation strategy can be parsed as arguments:

```
from pytorch_lightning.strategies import FSDPStrategy

parser = argparse.ArgumentParser()
parser.add_argument('--gpus', default=2, type=int, metavar='N',
                        help='number of GPUs per node')
parser.add_argument('--nodes', default=1, type=int, metavar='N',
                        help='number of nodes')
parser.add_argument('--epochs', default=2, type=int, metavar='N',
                        help='maximum number of epochs to run')
parser.add_argument('--accelerator', default='gpu', type=str,
                        help='accelerator to use')
parser.add_argument('--strategy', default='ddp', type=str,
                        help='distributed strategy to use')
args = parser.parse_args()

# Some work needed for FSDP
if args.strategy == "fsdp":
    strategy = FSDPStrategy(
        sharding_strategy="FULL_SHARD",
        cpu_offload=False
    )
if args.strategy == "fsdp1":
    strategy = FSDPStrategy(
        sharding_strategy="SHARD_GRAD_OP",
        cpu_offload=False
    )
if args.strategy == "fsdp2":
    strategy = FSDPStrategy(
        sharding_strategy="NO_SHARD",
        cpu_offload=False
    )
else:
    strategy = args.strategy
```

and then passed to the Trainer object:

```
trainer = L.Trainer(
  devices=args.gpus,
  num_nodes=args.nodes,
  max_epochs=args.epochs,
  accelerator=args.accelerator,
  strategy=strategy,
)
```

Exercises

ResNet50 trained on the Cifar10 dataset

In the following example, a ResNet50 is trained on the Cifar10 image dataset to classify images into 10 categories. Fill in the `#TODO` lines and submit using the script at `code/cifar10/job.slurm`. The submit script also allows you to test different parallelisation strategies.

```python
#!/usr/bin/env python
# train_cifar10_pl_student.py
# PyTorch Lightning training on CIFAR-10 with a ResNet50 backbone.
# Assumes dataset is present in ./data (download disabled) and optional weights at ./model_weights/resne

import os
import argparse
from functools import partial

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader

import pytorch_lightning as L
from pytorch_lightning.callbacks import (
    ModelCheckpoint,
    LearningRateMonitor,
)
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.strategies import FSDPStrategy

from torchvision import datasets, transforms, models

# Optional FSDP imports (no auto_wrap policy here)
try:
    from torch.distributed.fsdp import ShardingStrategy
except Exception:
    ShardingStrategy = None


# -----------------------------
# DataModule for CIFAR-10
# -----------------------------
class CIFAR10DataModule(L.LightningDataModule):
    def __init__(self, data_dir="./data", batch_size=256, num_workers=8):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.num_workers = num_workers

        # Normalize with ImageNet stats (works well with ResNet50 pretraining)
        self.train_transforms = transforms.Compose(
            [
                transforms.Resize(224),
                transforms.RandomHorizontalFlip(),
                transforms.RandomCrop(224, padding=4),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
                ),
            ]
        )
        self.val_transforms = transforms.Compose(
            [
                transforms.Resize(224),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
```

```
                ),
            ]
        )

    def setup(self, stage=None):
        self.train_set = datasets.CIFAR10(
            root=self.data_dir,
            train=True,
            transform=self.train_transforms,
            download=False,
        )
        self.val_set = datasets.CIFAR10(
            root=self.data_dir,
            train=False,
            transform=self.val_transforms,
            download=False,
        )

    def train_dataloader(self):
        # TODO: return DataLoader for self.train_set with shuffle=True
        # Use batch_size=self.batch_size and num_workers=self.num_workers
        # OBS: remember to pass pin_memory=False to the dataloader
        raise NotImplementedError("TODO: Implement train_dataloader()")

    def val_dataloader(self):
        # TODO: return DataLoader for self.val_set with shuffle=False
        # OBS: remember to pass pin_memory=False to the dataloader
        raise NotImplementedError("TODO: Implement val_dataloader()")


# ------------------------------
# LightningModule for ResNet50 on CIFAR-10
# ------------------------------
class LitResNet50(L.LightningModule):
    def __init__(
        self,
        num_classes=10,
        lr=0.1,
        weights_path="./model_weights/resnet50_imagenet.pth",
    ):
        super().__init__()
        self.save_hyperparameters()

        backbone = models.resnet50(weights=None)  # do not trigger internet download
        in_features = backbone.fc.in_features
        backbone.fc = nn.Linear(in_features, num_classes)
        self.model = backbone

        # If weights exist, load them (ignore mismatched classifier with strict=False)
        if os.path.isfile(weights_path):
            state = torch.load(weights_path, map_location="cpu")
            try:
                # Remove fc layer from ImageNet since I have only 10 classes in the end
                state = {k: v for k, v in state.items() if not k.startswith("fc.")}
                missing, unexpected = self.model.load_state_dict(state, strict=False)
                print(
                    f"[Info] Loaded weights from {weights_path}. Missing: {missing}, Unexpected: {unexpe
                )
            except Exception as e:
                print(f"[Warn] Could not load weights from {weights_path}: {e}")
        else:
```

```
            print(
                f"[Info] No external weights found at {weights_path}. Using random initialization."
            )

        self.criterion = nn.CrossEntropyLoss()

        # Fixed (not CLI) optimization hyperparameters
        self._momentum = 0.9
        self._weight_decay = 1e-4

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

    @staticmethod
    def accuracy(logits, targets):
        preds = torch.argmax(logits, dim=1)
        return (preds == targets).float().mean()

    def configure_optimizers(self):
        optimizer = optim.SGD(
            self.parameters(),
            lr=self.hparams.lr,
            momentum=self._momentum,
            weight_decay=self._weight_decay,
            nesterov=True,
        )
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
        return {
            "optimizer": optimizer,
            "lr_scheduler": {"scheduler": scheduler, "interval": "epoch"},
        }

    def training_step(self, batch, batch_idx):
        # TODO: unpack batch, forward pass, compute loss (hint: the criterion() method above returns the
        acc = self.accuracy(logits, y)

        self.log(
            "train_loss",
            loss,
            on_step=True,
            on_epoch=True,
            prog_bar=False,
            sync_dist=True,
        )
        self.log(
            "train_acc",
            acc,
            on_step=False,
            on_epoch=True,
            prog_bar=True,
            sync_dist=True,
        )
        return loss

    def validation_step(self, batch, batch_idx):
        # TODO: implement validation pass with accuracy metric as above
        self.log(
            "val_loss",
            val_loss,
            on_step=False,
            on_epoch=True,
```

```
            prog_bar=False,
            sync_dist=True,
        )
        self.log(
            "val_acc",
            val_acc,
            on_step=False,
            on_epoch=True,
            prog_bar=True,
            sync_dist=True,
        )


# ----------------------------
# Utility to build strategy
# ----------------------------
def build_strategy(name: str):
    """
    Map string to Lightning's strategy or FSDPStrategy.
    Options:
      - 'ddp'
      - 'fsdp_full'
      - 'fsdp_shard_grad'
    """
    if name == "ddp":
        return "ddp"
    if name.startswith("fsdp"):
        if FSDPStrategy is None or ShardingStrategy is None:
            raise RuntimeError(
                "FSDP strategy requested but not available in this environment."
            )
        if name == "fsdp_full":
            return FSDPStrategy(sharding_strategy=ShardingStrategy.FULL_SHARD)
        if name == "fsdp_shard_grad":
            return FSDPStrategy(sharding_strategy=ShardingStrategy.SHARD_GRAD_OP)
    raise ValueError(f"Unknown strategy '{name}'")


# ----------------------------
# Main
# ----------------------------
def parse_args():
    parser = argparse.ArgumentParser(
        description="Train ResNet50 on CIFAR-10 with Lightning."
    )
    parser.add_argument("--data_dir", type=str, default="./data")
    parser.add_argument(
        "--weights_path", type=str, default="./model_weights/resnet50_imagenet.pth"
    )
    parser.add_argument("--batch_size", type=int, default=256)
    parser.add_argument("--num_workers", type=int, default=8)
    parser.add_argument("--lr", type=float, default=0.1)

    # Cluster-related args
    parser.add_argument(
        "--devices", type=int, default=1, help="Number of GPUs to use per node."
    )
    parser.add_argument("--num_nodes", type=int, default=1, help="Number of nodes.")
    parser.add_argument(
        "--strategy",
        type=str,
```

```
        default="ddp",
        choices=["ddp", "fsdp_full", "fsdp_shard_grad"],
    )
    parser.add_argument("--max_epochs", type=int, default=90)
    parser.add_argument("--log_dir", type=str, default="./lightning_logs")

    return parser.parse_args()


def main():
    args = parse_args()

    # Fixed seed (not controlled by CLI)
    L.seed_everything(42, workers=True)

    datamodule = CIFAR10DataModule(
        data_dir=args.data_dir,
        batch_size=args.batch_size,
        num_workers=args.num_workers,
    )

    model = LitResNet50(
        num_classes=10,
        lr=args.lr,
        weights_path=args.weights_path,
    )

    strategy = build_strategy(args.strategy)

    callbacks = [
        ModelCheckpoint(
            dirpath=os.path.join(args.log_dir, args.strategy, "checkpoints"),
            filename="epoch{epoch:03d}-valacc{val_acc:.4f}",
            monitor="val_acc",
            mode="max",
            save_top_k=1,
            save_last=True,
            auto_insert_metric_name=False,
        ),
        LearningRateMonitor(logging_interval="epoch"),
    ]

    logger = TensorBoardLogger(save_dir=args.log_dir, name=args.strategy)

    trainer = L.Trainer(
        accelerator="gpu",
        devices=args.devices,
        num_nodes=args.num_nodes,
        strategy=strategy,
        max_epochs=args.max_epochs,
        logger=logger,
        callbacks=callbacks,
        deterministic=False,
        gradient_clip_val=0.0,
    )

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

```
if __name__ == "__main__":
    main()
```

Skip to content

Solution

Skip to content

```python
#!/usr/bin/env python
# train_cifar10_pl_solution.py

import os
import argparse

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader

import pytorch_lightning as L
from lightning.pytorch.callbacks import (
    ModelCheckpoint,
    LearningRateMonitor,
)
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.strategies import FSDPStrategy

from torchvision import datasets, transforms, models

# Optional FSDP imports (no auto_wrap policy here)
try:
    from torch.distributed.fsdp import ShardingStrategy
except Exception:
    ShardingStrategy = None


# ------------------------------
# DataModule
# ------------------------------
class CIFAR10DataModule(L.LightningDataModule):
    def __init__(self, data_dir="./data", batch_size=256, num_workers=8):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.num_workers = num_workers

        self.train_transforms = transforms.Compose(
            [
                transforms.Resize(224),
                transforms.RandomHorizontalFlip(),
                transforms.RandomCrop(224, padding=4),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
                ),
            ]
        )
        self.val_transforms = transforms.Compose(
            [
                transforms.Resize(224),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
                ),
            ]
        )
```

```
    def setup(self, stage=None):
        self.train_set = datasets.CIFAR10(
            root=self.data_dir,
            train=True,
            transform=self.train_transforms,
            download=False,
        )
        self.val_set = datasets.CIFAR10(
            root=self.data_dir,
            train=False,
            transform=self.val_transforms,
            download=False,
        )

    def train_dataloader(self):
        return DataLoader(
            self.train_set,
            batch_size=self.batch_size,
            shuffle=True,
            num_workers=self.num_workers,
            pin_memory=False,
        )

    def val_dataloader(self):
        return DataLoader(
            self.val_set,
            batch_size=self.batch_size,
            shuffle=False,
            num_workers=self.num_workers,
            pin_memory=False,
        )


# ------------------------------
# LightningModule
# ------------------------------
class LitResNet50(L.LightningModule):
    def __init__(
        self,
        num_classes=10,
        lr=0.1,
        weights_path="./model_weights/resnet50_imagenet.pth",
    ):
        super().__init__()
        self.save_hyperparameters()

        backbone = models.resnet50(weights=None)  # do not trigger internet download
        in_features = backbone.fc.in_features
        backbone.fc = nn.Linear(in_features, num_classes)
        self.model = backbone

        if os.path.isfile(weights_path):
            state = torch.load(weights_path, map_location="cpu")
            try:
                # Remove fc layer from ImageNet since I have only 10 classes in the end
                state = {k: v for k, v in state.items() if not k.startswith("fc.")}
                # allow mismatched classifier layer
                missing, unexpected = self.model.load_state_dict(state, strict=False)
                print(
                    f"[Info] Loaded weights from {weights_path}. Missing: {missing}, Unexpected: {une
                )
```

```
            except Exception as e:
                print(f"[Warn] Could not load weights from {weights_path}: {e}")
        else:
            print(
                f"[Info] No external weights found at {weights_path}. Using random initialization."
            )

        self.criterion = nn.CrossEntropyLoss()

        # Fixed optimization hyperparameters (not exposed via CLI)
        self._momentum = 0.9
        self._weight_decay = 1e-4

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

    @staticmethod
    def accuracy(logits, targets):
        preds = torch.argmax(logits, dim=1)
        return (preds == targets).float().mean()

    def configure_optimizers(self):
        optimizer = optim.SGD(
            self.parameters(),
            lr=self.hparams.lr,
            momentum=self._momentum,
            weight_decay=self._weight_decay,
            nesterov=True,
        )
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
        return {
            "optimizer": optimizer,
            "lr_scheduler": {"scheduler": scheduler, "interval": "epoch"},
        }

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = self.criterion(logits, y)
        acc = self.accuracy(logits, y)

        self.log(
            "train_loss",
            loss,
            on_step=True,
            on_epoch=True,
            prog_bar=False,
            sync_dist=True,
        )
        self.log(
            "train_acc",
            acc,
            on_step=False,
            on_epoch=True,
            prog_bar=True,
            sync_dist=True,
        )
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
```

```
        logits = self(x)
        val_loss = self.criterion(logits, y)
        val_acc = self.accuracy(logits, y)

        self.log(
            "val_loss",
            val_loss,
            on_step=False,
            on_epoch=True,
            prog_bar=False,
            sync_dist=True,
        )
        self.log(
            "val_acc",
            val_acc,
            on_step=False,
            on_epoch=True,
            prog_bar=True,
            sync_dist=True,
        )


# ----------------------------
# Strategy builder
# ----------------------------
def build_strategy(name: str):
    if name == "ddp":
        return "ddp"
    if name.startswith("fsdp"):
        if FSDPStrategy is None or ShardingStrategy is None:
            raise RuntimeError(
                "FSDP strategy requested but not available in this environment."
            )
        if name == "fsdp_full":
            return FSDPStrategy(sharding_strategy=ShardingStrategy.FULL_SHARD)
        if name == "fsdp_shard_grad":
            return FSDPStrategy(sharding_strategy=ShardingStrategy.SHARD_GRAD_OP)
    raise ValueError(f"Unknown strategy '{name}'")


# ----------------------------
# Main
# ----------------------------
def parse_args():
    parser = argparse.ArgumentParser(
        description="Train ResNet50 on CIFAR-10 with Lightning."
    )
    parser.add_argument("--data_dir", type=str, default="./data")
    parser.add_argument(
        "--weights_path", type=str, default="./model_weights/resnet50_imagenet.pth"
    )
    parser.add_argument("--batch_size", type=int, default=256)
    parser.add_argument("--num_workers", type=int, default=8)
    parser.add_argument("--lr", type=float, default=0.1)

    # Cluster-related args
    _argument(
        ices", type=int, default=1, help="Number of GPUs to use per node."
    )
    parser.add_argument("--num_nodes", type=int, default=1, help="Number of nodes.")
    parser.add_argument(
```

```
        "--strategy",
        type=str,
        default="ddp",
        choices=["ddp", "fsdp_full", "fsdp_shard_grad"],
    )
    parser.add_argument("--max_epochs", type=int, default=90)
    parser.add_argument("--log_dir", type=str, default="./lightning_logs")

    return parser.parse_args()


def main():
    args = parse_args()

    # Fixed seed (not exposed via CLI)
    L.seed_everything(42, workers=True)

    datamodule = CIFAR10DataModule(
        data_dir=args.data_dir,
        batch_size=args.batch_size,
        num_workers=args.num_workers,
    )

    model = LitResNet50(
        num_classes=10,
        lr=args.lr,
        weights_path=args.weights_path,
    )

    strategy = build_strategy(args.strategy)

    callbacks = [
        ModelCheckpoint(
            dirpath=os.path.join(args.log_dir, args.strategy, "checkpoints"),
            filename="epoch{epoch:03d}-valacc{val_acc:.4f}",
            monitor="val_acc",
            mode="max",
            save_top_k=1,
            save_last=True,
            auto_insert_metric_name=False,
        ),
        LearningRateMonitor(logging_interval="epoch"),
    ]

    logger = TensorBoardLogger(save_dir=args.log_dir, name=args.strategy)

    trainer = L.Trainer(
        accelerator="gpu",
        devices=args.devices,
        num_nodes=args.num_nodes,
        strategy=strategy,
        max_epochs=args.max_epochs,
        logger=logger,
        callbacks=callbacks,
        deterministic=False,
        gradient_clip_val=0.0,
        _checkpointing=True,


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

```
if __name__ == "__main__":
    main()
```

Try to run this example while changing parallelisation strategy/number of resources from the submit script!

Bonus: add logging of wall time per epoch

You can create a custom callback to plot time per epoch to compare the different approaches. *Hint: you can use the* `on_train_epoch_start()` *and* `on_train_epoch_end()` *hooks*.

Solution

```
class EpochTimingCallback(L.Callback):
    def on_train_epoch_start(self, trainer, pl_module):
        self.epoch_start_time = time.time()

    def on_train_epoch_end(self, trainer, pl_module):
        duration = time.time() - self.epoch_start_time
        trainer.logger.log_metrics({"epoch_time_sec": duration}, step=trainer.current_epoch)
        print(f"[Timing] Epoch {trainer.current_epoch} took {duration:.2f} seconds")
```

Remember to add this to the list of callbacks and to import the `time` module!

Summary

PyTorch Lightning can be used to produce more organised, reusable code to train neural network by clearly separating architecture, data and plumbing by taking advantage of `LightningModule`, `LightningDataModule` and `Trainer` respectively. Parallelising over several GPUs/nodes is made transparent and requires virtually no changes to the code.

See also

- Pytorch Lightning documentation

Quick Reference

Instructor’s guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

Learning outcomes

FIXME

This material is for …

By the end of this module, learners should:

- …
- …

See also

> **Credit**
>
> FIXME
>
> Don’t forget to check out additional course materials from …

License

CC BY-SA for media and pedagogical material

Skip to content

MIT for source code and code snippets

MIT License