Introduction to PyTorch

and Scaling PyTorch Code Using LightningLite

Sebastian Raschka & Adrian Wälchli

Data Umbrella

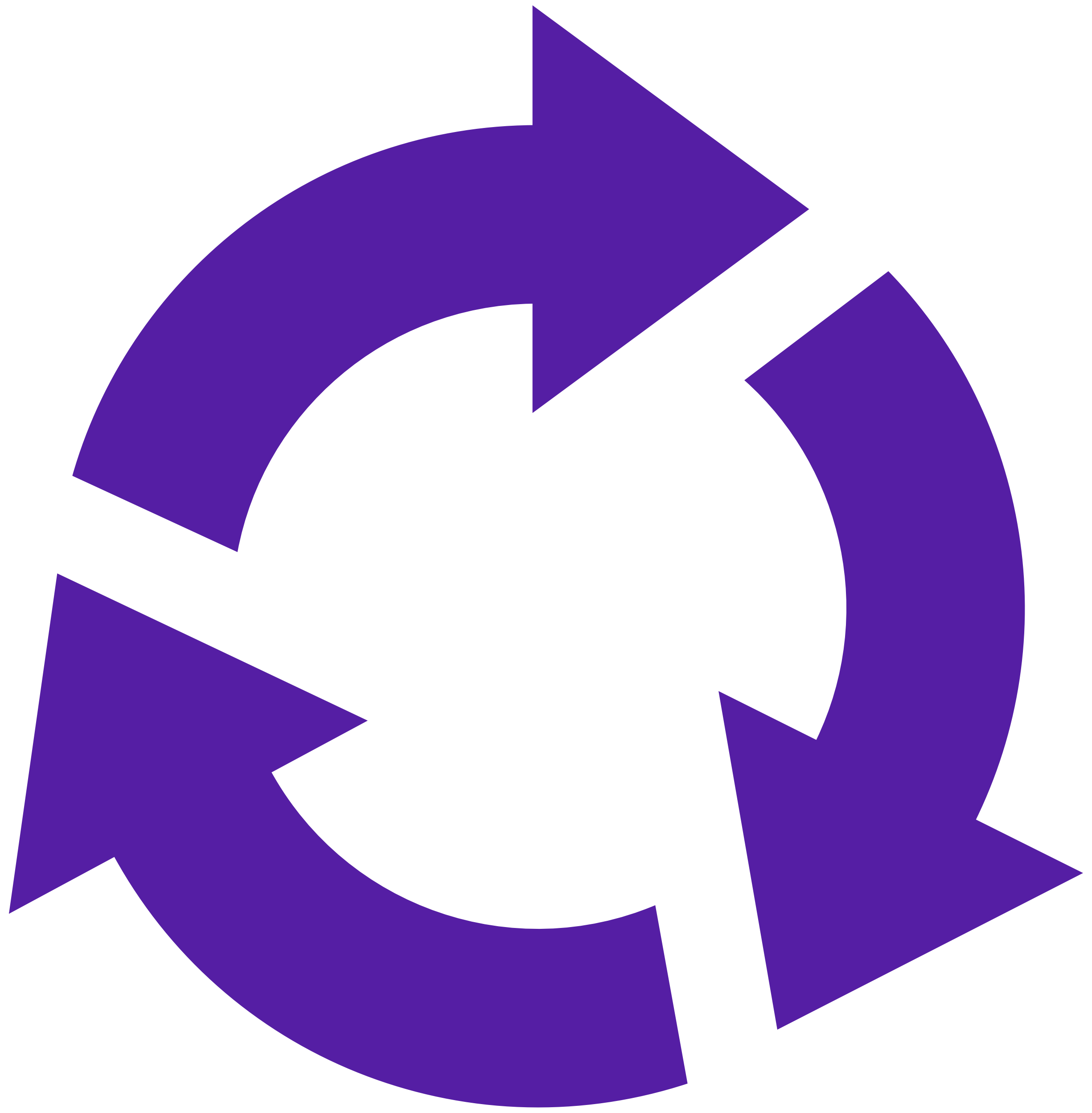
May 10th, 2022

What lies ahead of you

LightningLite

Powered by Lightning Accelerators

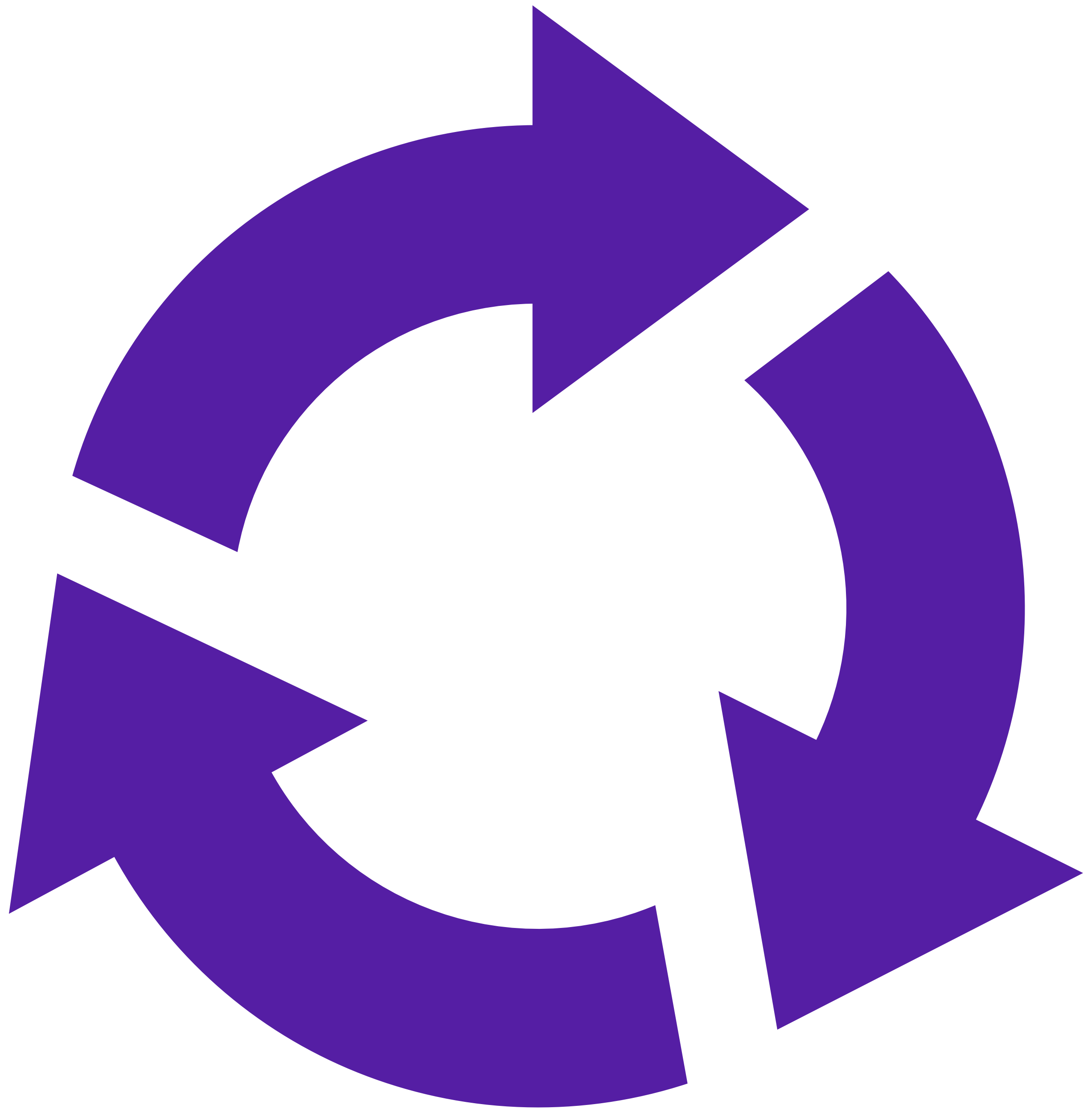
Idea



Code / Debug

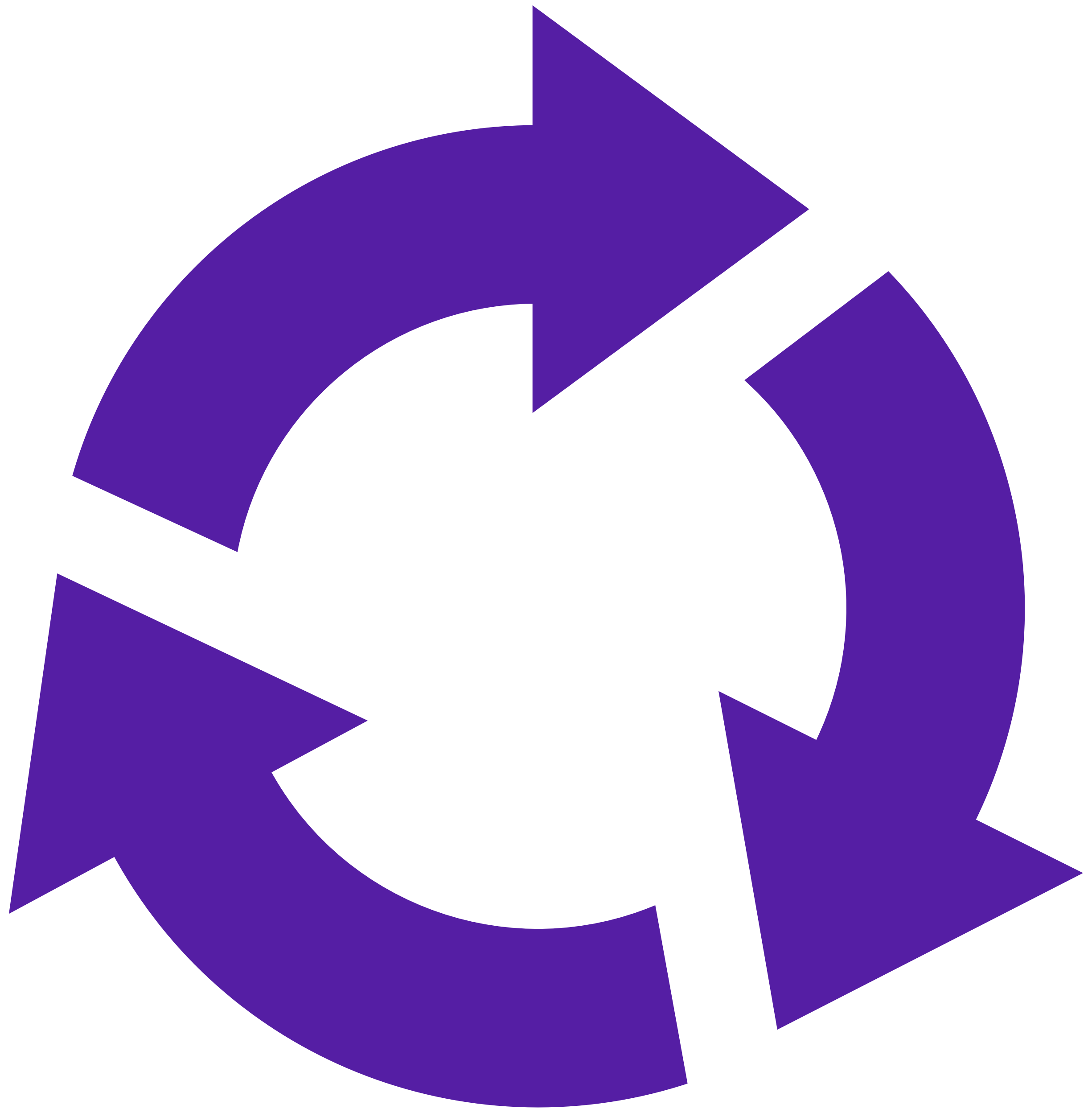
Run Experiments

Test


Idea
Code / Debug
Run Experiments
Test
Many hours!

Idea

Much engineering



Code / Debug

Test

Many hours!

Run Experiments

Image Credit: NVIDIA

What's boilerplate code?

```
if args.distributed:
    if args.dist_url == "env://" and args.rank == -1:
        args.rank = int(os.environ["RANK"])
    if args.multiprocessing_distributed:
        # For multiprocessing distributed training, rank needs to be the
        # global rank among all the processes
        args.rank = args.rank * ngpus_per_node + gpu
    dist.init_process_group(backend=args.dist_backend,
                            init_method=args.dist_url,
                            world_size=args.world_size,
                            rank=args.rank)
```

https://github.com/pytorch/examples/blob/main/imagenet/main.py

What does LightningLite do?

It handles all this boilerplate for you!

You get

CPU, GPU, TPU
Multiple GPUs / TPUs
Multi-node
Mixed precision

For FREE, without the boilerplate

Let's do it!

```
pip install pytorch-lightning
```

PyTorch Lightning

www.pytorchlightning.ai

No changes to the model required!

```python
import torch


class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.num_classes = num_classes
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(
                in_channels=3,
                out_channels=8,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0),
            torch.nn.ReLU(),
            torch.nn.Conv2d(
                in_channels=8,
                out_channels=16,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0),
        )

        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(784, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

No changes to the data required!

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset=train_dset,
    batch_size=batch_size,
    drop_last=True,
    num_workers=4,
    shuffle=True,
)

valid_loader = DataLoader(
    dataset=valid_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)

test_loader = DataLoader(
    dataset=test_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)
```

The LightningLite Skeleton

```python
from pytorch_lightning.lite import LightningLite


class Lite(LightningLite):
    def run(self):

        # Here goes the training code


Lite().run()
```

1. Initializing the model and optimizer

Sets up model and optimizer for distributed training!

```
class Lite(LightningLite):
    def run(self):

        model = PyTorchCNN(num_classes=num_classes)
        model = model.to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

        model, optimizer = self.setup(model, optimizer)

        # ...
```

2. Setting up the data loaders

Automatically moves the data to the right device!

```
class Lite(LightningLite):
    def run(self):

        model = PyTorchCNN(num_classes=num_classes)
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

        model, optimizer = self.setup(model, optimizer)

        train_dataloader = self.setup_dataloaders(train_dataloader)
        val_dataloader = self.setup_dataloaders(val_dataloader)
        teset_dataloader = self.setup_dataloaders(test_dataloader)

        # ...
```

3. Iterating over the training examples

The features and targets are already on the device!

```
class Lite(LightningLite):
    def run(self):

        # ...

        for epoch in range(num_epochs):
            model = model.train()
            for batch_idx, (features, targets) in enumerate(train_loader):

                features, targets = features.to(device), targets.to(device)

                # ...
```

4. Updating the model weights

You only need to replace
loss.backward()
with
self.backward(loss)

```
class Lite(LightningLite):
    def run(self):

        # ...

        for epoch in range(num_epochs):
            model = model.train()
            for batch_idx, (features, targets) in enumerate(train_loader):

                ### Forward pass
                logits = model(features)
                loss = F.cross_entropy(logits, targets)

                ### Backward pass (backpropagation)
                optimizer.zero_grad()
                loss.backward()
                self.backward(loss)

                ### Update model parameters
                optimizer.step()

                # ...
```

We're done. Why did we do this again?

Accelerate your PyTorch code!

```
# Everything on CPU
Lite().run()

# One GPU
Lite(accelerator="gpu", devices=1).run()

# Multiple GPUs
Lite(accelerator="gpu", devices=4).run()

# Specific GPU IDs
Lite(accelerator="gpu", devices=[2, 3]).run()

# TPU
Lite(accelerator="tpu", devices=8).run()

# Select available hardware automatically!
Lite(accelerator="auto", devices="auto").run()
```

Mixed precision saves you memory

```
# Default precision setting is 32-bit
Lite(accelerator="gpu", devices=1, precision=32).run()

# Save memory with mixed 16-bit precision
Lite(accelerator="gpu", devices=1, precision=16).run()

# Double precision is also supported
Lite(accelerator="gpu", devices=1, precision=64).run()
```

Try different strategies for best performance

```
# Best for multi-GPU in Jupyter notebooks
Lite(accelerator="gpu", devices=2, strategy="dp").run()

# Best for single and multi-node training
Lite(accelerator="gpu", devices=4, strategy="ddp").run()

# Sharded training saves memory for very large models!
Lite(accelerator="gpu", devices=8, strategy="ddp_sharded").run()

# For even bigger models
Lite(accelerator="gpu", devices=8, strategy="deepspeed").run()
```

When you're ready, level up in Lightning.

IPU Accelerator

Fault-tolerance

Progress Bar

Mixed Precision

Multi-GPU

Reproducibility

Logging

Checkpointing

Hyperparameter Tuner

Profiling

Cloud Computing

Metrics

Multi-Node

Loops

CLI

Gradient Accumulation

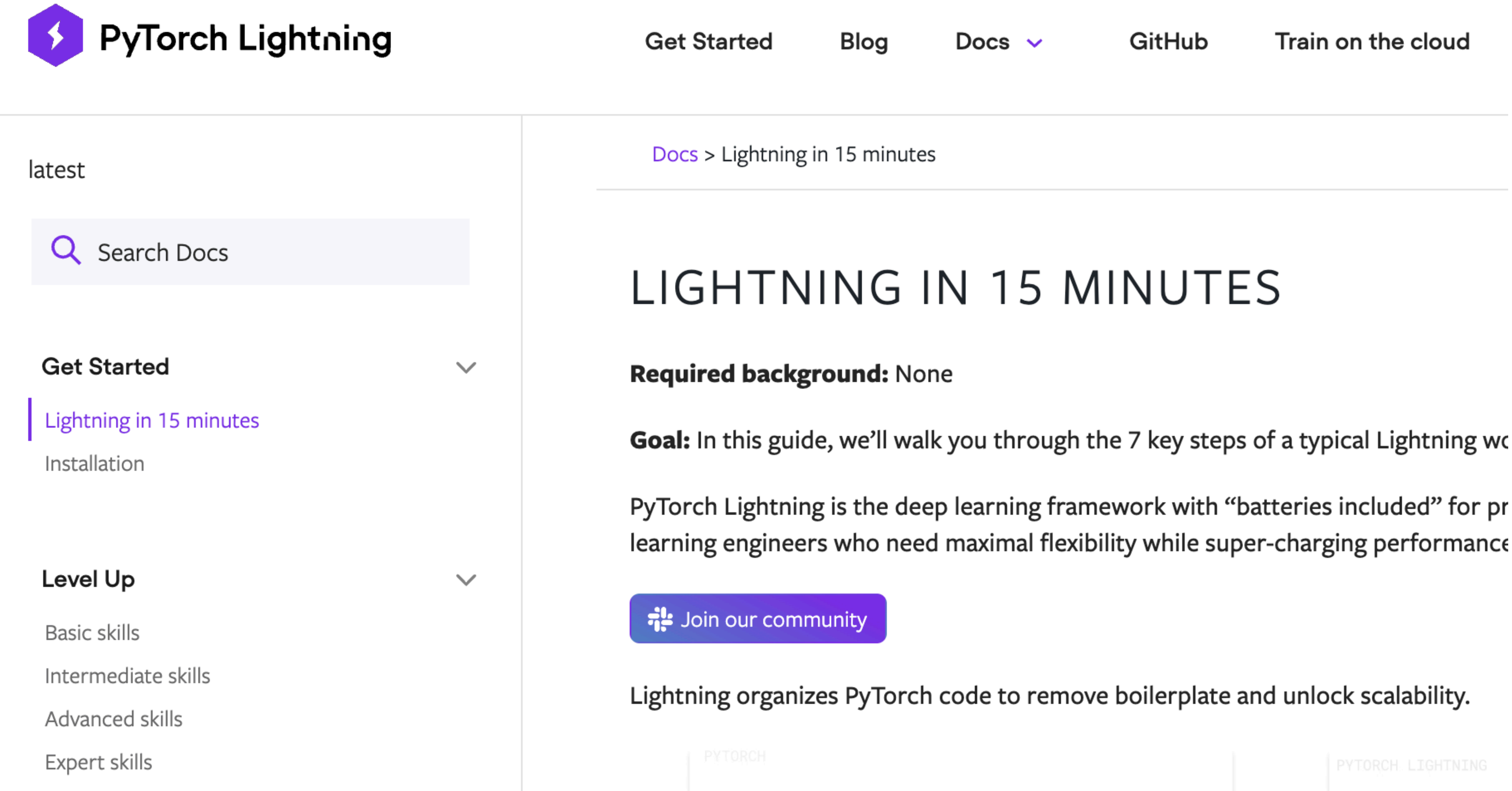
Early Stopping

Quantization

TPU Accelerator

HPU Accelerator

Visit docs.pytorchlightning.ai



The End