

Automate the boring stuff with `pytorch-lightning`

Jacob Nilsson

13 May, 2020

Luleå University of Technology

What's the boring stuff?



LIVE CODING EXAMPLE



An example of boilerplate code

With boring stuff:

```
class MyClass():  
    def __init__(self, a, b):  
        self._a = a  
        self._b = b
```

Without boring stuff:

```
@dataclass  
class MyDataClass():  
    a: int  
    b: float
```

Flexibility vs. Readability

Higher flexibility

Messier code

More error prone:

Lower flexibility

Cleaner code

Less error prone:

```
class MyClass():  
    def __init__(self, a, b):  
        self._a = a  
        self._b = b
```

```
@dataclass  
class MyDataClass():  
    a: int  
    b: float
```

Opinions on PyTorch vs TensorFlow

PyTorch:

Flexible models (“normal” code)*

Easy debugging (`print()` works)*

Deployment is manual and requires a lot of boilerplate code

TensorFlow:

Models are difficult to change*

Difficult debugging*

Deployment is streamlined

*Common thoughts about PyTorch vs. TensorFlow online

Opinions on PyTorch vs TensorFlow

PyTorch:

Flexible models (“normal” code)*

Easy debugging (`print()` works)*

Deployment is manual and requires a lot of boilerplate code

TensorFlow:

Models are difficult to change*

Difficult debugging*

Deployment is streamlined

*Common thoughts about PyTorch vs. TensorFlow online

Introduction to `pytorch-lightning`

What is `pytorch-lightning`

Pytorch-lightning is a tool to organize pytorch code.

Pytorch-lightning organizes code into 3 categories:

1. Research code
2. Engineering code (Boilerplate)
3. Non-essential research code (logging)



LIVE CODING EXAMPLE



Research code

PyTorch

```
# model
class Net(nn.Module):
    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

# train loader
mnist_train = MNIST(os.getcwd(), train=True, download=True,
                    transform=transforms.ToTensor())
mnist_train = DataLoader(mnist_train, batch_size=64)

net = Net()

# optimizer + scheduler
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
scheduler = StepLR(optimizer, step_size=1)

# train
for epoch in range(1, 100):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)

        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{}] ({:.0f}%)\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

PyTorch Lightning

```
# model
class Net(LightningModule):
    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

    def train_dataloader(self):
        mnist_train = MNIST(os.getcwd(), train=True, download=True,
                            transform=transforms.ToTensor())
        return DataLoader(mnist_train, batch_size=64)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
        scheduler = StepLR(optimizer, step_size=1)
        return optimizer, scheduler

    def training_step(self, batch, batch_idx):
        data, target = batch
        output = self.forward(data)
        loss = F.nll_loss(output, target)
        return {'loss': loss}
```

PyTorch

```
# model
class Net(nn.Module):
    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

# train loader
mnist_train = MNIST(os.getcwd(), train=True, download=True,
                    transform=transforms.ToTensor())
mnist_train = DataLoader(mnist_train, batch_size=64)

net = Net()

# optimizer + scheduler
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
scheduler = StepLR(optimizer, step_size=1)

# train
for epoch in range(1, 100):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)

        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{}] ({:.0f}%) \tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

PyTorch Lightning

```
# model
class Net(LightningModule):
    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

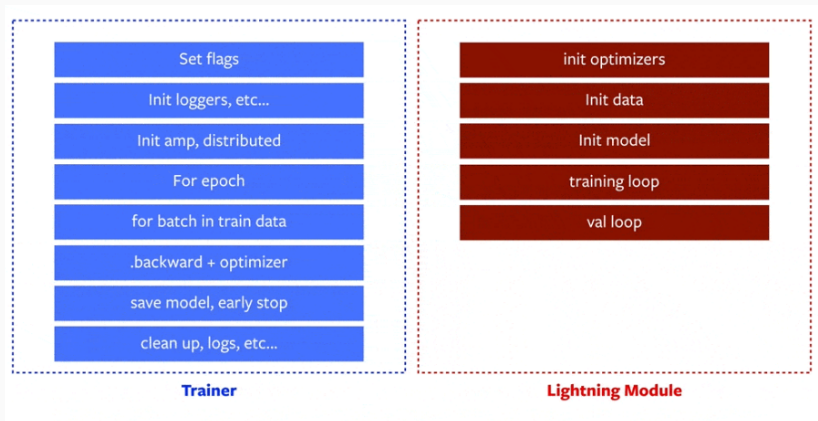
    def train_dataloader(self):
        mnist_train = MNIST(os.getcwd(), train=True, download=True,
                            transform=transforms.ToTensor())
        return DataLoader(mnist_train, batch_size=64)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
        scheduler = StepLR(optimizer, step_size=1)
        return optimizer, scheduler

    def training_step(self, batch, batch_idx):
        data, target = batch
        output = self.forward(data)
        loss = F.nll_loss(output, target)
        return loss

if __name__ == '__main__':
    net = Net()
    trainer = Trainer()
    trainer.fit(net)
```

Summary thus far



Question break

Validation, Testing, Progress Bar, and Logging

Validation

To do validation, you add two methods and a dataloader:

```
def validation_step(self, batch, batch_idx):  
    """ Calculate validation for batch """  
  
def validation_epoch_end(self, outputs):  
    """ Average results over all batches """  
  
def val_dataloader(self):  
    """ Returns a validation dataloader """
```

For testing, replace `validation` with `test`

Progress Bar and Logging

All `_step` and `_epoch_end` methods *must* return a dictionary where you can choose what is logged and shown in the progress bar.

```
def training_step(self, batch, batch_idx):  
    """  
  
    """  
    return {'loss': loss} # Mandatory for training_step  
  
def validation_epoch_end(self, batch, batch_idx):  
    """  
  
    """  
    return {'progress_bar': pb_dict, # dict of progress bar  
            'log': {'val_loss', val_loss} # dict of variables to log
```

Customizable loggers

Logging is done automatically done with Tensorboard by default, but can be swapped to any other logger compatible with Pytorch Lightning, see <https://pytorch-lightning.readthedocs.io/en/latest/loggers.html>. For example TestTube:

```
def main():  
    from pytorch_lightning.loggers import TestTubeLogger  
    ttlogger = TestTubeLogger('tb_logs', name='my_model')  
    trainer = pl.Trainer(logger=ttlogger)
```



LIVE CODING EXAMPLE



Summary of section

To add *validation* or *testing* functionality, you add `step` and `epoch_end` (collation) methods, as well as a dataloader for each functionality.

Logging is easily added, and logger behaviour can be changed.

Debugging

Validation sanity checks

Pytorch Lightning runs a few validation steps (5 by default) before training to find errors before costly training begins. Can be customized:

```
trainer = Trainer(num_sanity_val_steps=5)
```

To quickly check that the model runs through all steps correctly, you can set the `fast_dev_run` flag:

```
trainer = Trainer(fast_dev_run=True)
```

Overfitting checks

You can tell the trainer to overfit on a small subset of your data to check that the loss goes to 0.

```
trainer = pl.Trainer(  
    overfit_pct=0.01, # Overfits on 1% of the data  
    training_percent_check=0.01, # Overfits on 1% of the training data
```


Customizing the training loop

Callbacks

Non-essential functionality is put in callbacks, some important ones are already defined such as early stopping and model checkpointing

```
from pytorch_lightning.callbacks import EarlyStopping, \
                                     ModelCheckpoint

early_stopping = EarlyStopping('val_loss')
checkpoint_callback = ModelCheckpoint(filepath='my/path/')

trainer = Trainer(early_stop_callback=early_stopping,
                  checkpoint_callback=checkpoint_callback)
```

Each callback can be further customized to suit specific tasks, and new callbacks can be written if needed, see <https://pytorch-lightning.readthedocs.io/en/latest/callbacks.html>.

The *model hooks* are the methods that comprise the model interface.

training_step is a model hook.

Each hook represent a behaviour of the model, and all hooks can be changed to change the behaviour.

Hooks example: on_epoch_start

Used if you have something you need to do at the start of each epoch:

```
class Net(pl.LightningModule):  
    def on_epoch_start():  
        print('new epoch started')  
        self.a *= 1.1
```

Consult the documentation for more hooks, some are very specific and they are open to suggestions for new hooks.

```
https://pytorch-lightning.readthedocs.io/en/latest/api/  
pytorch\_lightning.core.hooks.html
```

Summary

Pytorch Lightning is a framework for Pytorch that strives to reduce the amount of engineering code in your projects, letting you focus on research.

Logging is automatic.

Debugging functionality is built-in.

Little flexibility is lost due to flexible callbacks and model hooks.

“Standardizes” training code in Pytorch (to those who use it).

Questions and End