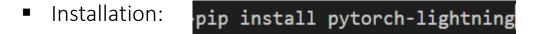
Introduction

- Python package
- framework built on top of PyTorch
- claims "making coding complex networks simple"
- nice features
 - logging of training / validation metrics
 - creating checkpoints
 - early stopping
 - training on multiple GPUs, TPUs, and CPUs





Introduction

- requires a class with four functions:
 - init_()
 - forward()
 - configure optimizers()
 - training_step()
- optional functions
 - prepare_data()
 - validation_step()
 - test step()
 - predict_step()



Comparison PyTorch vs. PyTorch Lightning

your package imports

your package imports

import pytorch_lightning as pl

PyTorch

PyTorch Lightning

Comparison PyTorch vs. PyTorch Lightning

```
class LinearRegressionTorch(nn.Module):
   def init (self, input size, output size):
        super(LinearRegressionTorch, self). init ()
        self.linear = nn.Linear(input_size, output_size)
   def forward(self, x):
       return self.linear(x)
model = LinearRegressionTorch(input size=1, output size=1)
model.train()
# %% Mean Squared Error
loss fun = nn.MSELoss()
#%% Optimizer
learning rate = 0.02
optimizer = torch.optim.SGD(model.parameters(), lr=learning rate)
#%% perform training
losses = []
slope, bias = [], []
number epochs = 1000
for epoch in range(number epochs):
   for j, (X, y) in enumerate(train_loader):
       optimizer.zero grad()
       y_pred = model(X)
       loss = loss fun(y pred, y)
       losses.append(loss.item())
       loss.backward()
       # update weights
       optimizer.step()
   losses.append(float(loss.data))
```

Class Inheritance

```
class LitLinearRegression(pl.LightningModule):
    def __init__(self, input_size, output_size):
        super(LitLinearRegression, self). init ()
        self.linear = nn.Linear(input size, output size)
        self.loss fun = nn.MSELoss()
    def forward(self, x):
        return self.linear(x)
    def configure optimizers(self):
        learning rate = 0.02
        optimizer = torch.optim.SGD(self.parameters(), lr=learning_rate)
        return optimizer
    def training step(self, train batch, batch idx):
        X, y = train batch
        # forward pass
        v pred = model(X)
        loss = self.loss fun(y pred, y)
        self.log('train loss', loss, prog bar=True)
        return loss
#%% model instance and training
# model instance
model = LitLinearRegression(input size=1, output size=1)
# training
trainer = pl.Trainer(gpus=1, precision=16, max_epochs=100)
trainer.fit(model, train loader)
```

PyTorch

Comparison PyTorch vs. PyTorch Lightning

```
class LinearRegressionTorch(nn.Module):
   def init (self, input size, output size):
       super(LinearRegressionTorch, self). init ()
       self.linear = nn.Linear(input_size, output_size)
   def forward(self, x):
       return self.linear(x)
model = LinearRegressionTorch(input size=1, output size=1)
model.train()
# %% Mean Squared Error
loss fun = nn.MSELoss()
#%% Optimizer
#%% perform training
losses = []
slope, bias = [], []
number epochs = 1000
for epoch in range(number epochs):
   for j, (X, y) in enumerate(train_loader):
       # forward pass
       y_pred = model(X)
       loss = loss fun(y pred, y)
       losses.append(loss.item())
       loss.backward()
       # update weights
   # store loss
   losses.append(float(loss.data))
```

Optimizer

```
class LitLinearRegression(pl.LightningModule):
    def __init__(self, input_size, output_size):
        super(LitLinearRegression, self). init ()
        self.linear = nn.Linear(input_size, output_size)
        self.loss fun = nn.MSELoss()
    def forward(self, x):
        return self.linear(x)
    def training_step(self, train_batch, batch_idx):
        X, y = train batch
        # forward pass
        v pred = model(X)
        loss = self.loss fun(y pred, y)
        self.log('train loss', loss, prog bar=True)
        return loss
#%% model instance and training
# model instance
model = LitLinearRegression(input size=1, output size=1)
trainer = pl.Trainer(gpus=1, precision=16, max_epochs=100)
trainer.fit(model, train loader)
```

PyTorch

Comparison PyTorch vs. PyTorch Lightning

```
class LinearRegressionTorch(nn.Module):
   def init (self, input size, output size):
       super(LinearRegressionTorch, self). init ()
        self.linear = nn.Linear(input_size, output_size)
   def forward(self, x):
       return self.linear(x)
model = LinearRegressionTorch(input size=1, output size=1)
model.train()
# %% Mean Squared Error
loss fun = nn.MSELoss()
#%% Optimizer
learning rate = 0.02
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Training

```
class LitLinearRegression(pl.LightningModule):
   def __init__(self, input_size, output_size):
        super(LitLinearRegression, self). init ()
       self.linear = nn.Linear(input_size, output_size)
        self.loss fun = nn.MSELoss()
    def forward(self, x):
        return self.linear(x)
   def configure optimizers(self):
        learning rate = 0.02
       optimizer = torch.optim.SGD(self.parameters(), lr=learning_rate)
        return optimizer
       self.log('train loss', loss, prog bar=True)
#%% model instance and training
# model instance
model = LitLinearRegression(input_size=1, output_size=1)
```

PyTorch

Early Stopping

Early Stopping

Additional Features: Early Stopping

from pytorch_lightning.callbacks.early_stopping import EarlyStopping

```
early_stop_callback = EarlyStopping(monitor="train_loss",
min_delta=0.00, patience=2, verbose=True, mode="min")
```

```
trainer = pl.Trainer(accelerator='gpu', devices=1,
max_epochs=500, log_every_n_steps=2, callbacks=
[early_stop_callback])
trainer.fit(model=model, train_dataloaders=train_loader)
```

