Pytorch Lightning

Quentin Febvre, PhD student

“Simple” training routine walkthrough

```
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

"Simple" learning routine contains many considerations intertwined:

- Training logic:
    - What do I optimize for ?
    - How do I compute it ?
    - How do I optimize it ?

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

"Simple" learning routine contains many considerations intertwined:

- Training logic:
    - What do I optimize for ?
    - How do I compute it ?
    - How do I optimize it ?

- Training flow:
    - Loop for each epoch
    - Iterate over each minibatch

- Autograd/Pytorch flow
    - train/eval mode
    - Backprop + parameter update
    - No_grad (disable grad computation)

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

“Simple” learning routine contains many considerations intertwined:

- Training logic:
    - What do I optimize for ?
    - How do I compute it ?
    - How do I optimize it ?
- Training flow:
    - Loop for each epoch
    - Iterate over each minibatch
- Autograd/Pytorch flow
    - train/eval mode
    - Backprop + parameter update
    - No_grad (disable grad computation)
- Engineering flow / Plugins
    - Training hardware (.to(device))
    - Logging training metrics
    - Saving best model

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

“Simple” learning routine contains many considerations intertwined:

- Training logic:
  - What do I optimize for ?
  - How do I compute it ?
  - How do I optimize it ?

Extract

- Training flow:
  - Loop for each epoch
  - Iterate over each minibatch
- Autograd/Pytorch flow
  - train/eval mode
  - Backprop + parameter update
  - No_grad (disable grad computation)

Abstract

- Engineering flow / Plugins
  - Training hardware (.to(device))
  - Logging training/validation metrics
  - Saving best model

Configure

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```python
class LitMLP(pl.LightningModule):
    def __init__(self): ...

    def forward(self,x): ...

    def training_step(self, batch, batch_idx):
        data, label = batch
        output = self(data)
        return F.cross_entropy(output, label)

    def validation_step(self, batch, batch_idx):
        data, label = batch
        output = self(data)
        return F.cross_entropy(output, label)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(),lr = 0.01)
```

Training logic extraction:

- Change the nn.Module to pytorch_lightning.LightningModule

- Compute and return the optimization objective in the training and validation step method

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```python
class LitMLP(pl.LightningModule):
    def __init__(self):⋯

    def forward(self,x): ⋯

    def training_step(self, batch, batch_idx):
        data, label = batch
        output = self(data)
        return F.cross_entropy(output, label)

    def validation_step(self, batch, batch_idx):
        data, label = batch
        output = self(data)
        return F.cross_entropy(output, label)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(),lr = 0.01)

```

```python
model = LitMLP()
trainer = pl.Trainer(max_epochs=1)
trainer.fit(
    model, train_dataloaders=train_loader, val_dataloaders=valid_loader)
```

Training flow abstraction:

- Instantiate the pytorch_lightning.Trainer object
- Pass the Lightning module and the training data to the fit method to perform the optimization procedure

```python
n_epochs = 30
model_1 = MLP()
model_1.to(device=device)
optimizer = torch.optim.SGD(model_1.parameters(),lr = 0.01)
criterion = nn.CrossEntropyLoss()

train_losses, valid_losses = [], []
valid_loss_min = np.Inf

for epoch in range(n_epochs):
    train_loss, valid_loss = 0, 0

    model.train()
    for data, label in train_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, label in valid_loader:
        data = data.to(device=device, dtype=torch.float32)
        label = label.to(device=device, dtype=torch.long)
        with torch.no_grad():
            output = model(data)
        loss = criterion(output,label)
        valid_loss += loss.item() * data.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('epoch: {} \ttraining Loss: {:.6f} \tvalidation Loss: {:.6f}'.format(
        epoch+1, train_loss, valid_loss))

    if valid_loss <= valid_loss_min:
        print('validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```python
class LitMLP(pl.LightningModule):
    def __init__(self):⋯

    def forward(self,x): ⋯

    def training_step(self, batch, batch_idx):
        data, label = batch
        output = self(data)
        loss = F.cross_entropy(output, label)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch, batch_idx):
        data, label = batch
        output = self(data)
        loss = F.cross_entropy(output, label)
        self.log('val_loss', loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(),lr = 0.01)
```

```python
model = LitMLP()
model_checkpoint = pl.callbacks.ModelCheckpoint(monitor='val_loss')
logger = pl.loggers.CSVLogger('logs', name='mlp_mnist')
trainer = pl.Trainer(
    max_epochs=10,
    gpus=1,
    callbacks=[model_checkpoint],
    logger=logger
    )

trainer.fit(model, train_dataloaders=train_loader, val_dataloaders=valid_loader)
```

Engineering/Plugins configuration:

- Instantiate the logger and model checkpoint callback
- Pass as parameter to the trainer, as well as other flags (gpus…)

Vanilla Pytorch: VS Pytorch Lightning:

Vanilla Pytorch:

- \+ All the logic is exposed sequentially: easier to write and reason about
- \+ Fewer classes/concept to handle
- \- Code harder to read
- \- Reinvent the wheel each time

Pytorch Lightning:

- \+ Code is organized : easier to read/share
- \+ Less code
- \+ Benefit from SOTA implementations
- \- Additional concepts to understand: Trainer, Logger, ModelCheckpointCallback
- \- Some modifications are less straight-forward (find the method, the parameter…)

Give it a try :
notebooks/notebook_PytorchLightning_MNIST_CNN_students.ipynb