Aprendizado Profundo

PyTorch Lightning e Monitoramento de Experimentos

Professor: Lucas Silveira Kupssinskü

PyTorch Lightning

- PyTorch puro oferece grande flexibilidade
- Por consequência, é comum encontrarmos códigos confusos e sem padronização
- Uma alternativa é usar o PyTorch Lightning
  - Perdemos em:
    - flexibilidade
  - Ganhamos em:
    - Menos código *boilerplate*
    - Código mais padronizado

PyTorch Lightning

- Temos dois objetos importantíssimos no Lightning
  - `LightningModule`
  - `Trainer`
- A lógica do Lightning é implementar as funções pré-determinadas e adicionar comportamento customizado via *callbacks*

PyTorch Lightning

- `LightningModule`
  - É uma extensão do `torch.nn.Module`
  - Organiza o código PyTorch em diversos métodos
    - `__init__` usado para definir a inicialização
    - `forward()` usado para definir o forward pass do modelo
    - `training_step()` usado para definir um passo de treinamento, deve retornar a loss
    - `validation_step` usado para definir um passo de validação
    - `test_step()` usado para definir o teste
    - `configure_optimizers` usado para definir o otimizador do modelo

PyTorch Lightning

- `Trainer`
  - É usado para gerenciar o loop de treino e validação
  - Organiza as chamadas do seu `LightningModule`

Você escreve isso

```
model = MyLightningModule()

trainer = Trainer()
trainer.fit(model,
            train_dataloader,
            val_dataloader)
```

O `Trainer` executa isso

```
# enable grads
torch.set_grad_enabled(True)
losses = []
for batch in train_dataloader:
   # calls hooks like this one
   on_train_batch_start()

   # train step
   loss = training_step(batch)

   # clear gradients
   optimizer.zero_grad()

   # backward
   loss.backward()

   # update parameters
   optimizer.step()
   losses.append(loss)
```

PyTorch Lightning

```python
def create_dataloaders(
    data_dir: str,
    batch_size: int,
    num_workers: int
):
    training_data = datasets.MNIST(
        root=data_dir,
        train=True,
        download=True,
        transform=ToTensor(),
    )
    test_data = datasets.MNIST(
        root=data_dir,
        train=False,
        download=True,
        transform=ToTensor(),
    )

    train_dataloader = DataLoader(
        training_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
        persistent_workers=True)
    test_dataloader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False)
    return train_dataloader, test_dataloader
```

```python
class MNISTDataModule(L.LightningDataModule):
    def __init__(self, data_dir: str, batch_size: int, num_workers: int):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.num_workers = num_workers

    def prepare_data(self):
        datasets.MNIST(root=self.data_dir, train=True, download=True)
        datasets.MNIST(root=self.data_dir, train=False, download=True)

    def setup(self, stage=None):
        if stage == "fit" or stage is None:
            self.train_dataset = datasets.MNIST(
                root=self.data_dir, train=True, transform=ToTensor()
            )
            self.val_dataset = datasets.MNIST(
                root=self.data_dir, train=False, transform=ToTensor()
            )

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

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

PyTorch Lightning

```python
def create_dataloaders(
    data_dir: str,
    batch_size: int,
    num_workers: int
):
    training_data = datasets.MNIST(
        root=data_dir,
        train=True,
        download=True,
        transform=ToTensor(),
    )
    test_data = datasets.MNIST(
        root=data_dir,
        train=False,
        download=True,
        transform=ToTensor(),
    )

    train_dataloader = DataLoader(
        training_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
        persistent_workers=True)
    test_dataloader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False)
    return train_dataloader, test_dataloader
```

```python
class MNISTDataModule(L.LightningDataModule):
    def __init__(self, data_dir: str, batch_size: int, num_workers: int):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.num_workers = num_workers

    def prepare_data(self):
        datasets.MNIST(root=self.data_dir, train=True, download=True)
        datasets.MNIST(root=self.data_dir, train=False, download=True)

    def setup(self, stage=None):
        if stage == "fit" or stage is None:
            self.train_dataset = datasets.MNIST(
                root=self.data_dir, train=True, transform=ToTensor()
            )
            self.val_dataset = datasets.MNIST(
                root=self.data_dir, train=False, transform=ToTensor()
            )

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

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

PyTorch Lightning

```
class LeNet5(torch.nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.feature_extractor = torch.nn.Sequential(
            torch.nn.Conv2d(
                in_channels=1, out_channels=6, kernel_size=5,
            padding=2),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(
                in_channels=6, out_channels=16,
           kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(1),
            torch.nn.Linear(16*5*5, 120),
            torch.nn.ReLU(),
            torch.nn.Linear(120, 84),
            torch.nn.ReLU(),
            torch.nn.Linear(84, 10)
        )

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

```
class LeNet5(L.LightningModule):
    def __init__(self):
        ...

    def forward(self, x):
        ...

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = torch.nn.functional.cross_entropy(y_hat, y)
        self.log("train_loss", loss)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = torch.nn.functional.cross_entropy(y_hat, y)
        self.log("val_loss", loss, prog_bar=True)
        self.log("val_acc", (y_hat.argmax(1) ==
                 y).float().mean(), prog_bar=True)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.001)
```

PyTorch Lightning

```python
def main(batchsize: int, epochs: int, workers: int, force_cpu: bool):
    torch.manual_seed(42)

    torch.backends.cudnn.benchmark = True
    device = 'cuda' if torch.cuda.is_available() and not force_cpu else 'cpu'

    train_dataloader, test_dataloader = create_dataloaders(
        '../data', batchsize, workers)

    lenet5 = LeNet5()
    lenet5.to(device)

    loss_fn = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(params=lenet5.parameters(), lr=0.1)

    train_time_start_on_cpu = timer()

    for epoch in range(epochs):
        with tqdm(train_dataloader, desc=f'{epoch=}', unit='batch') as tepoch:

            train_loss = train_step(lenet5, tepoch, loss_fn, optimizer, device)

            test_loss, test_acc = test_step(
                lenet5, loss_fn, test_dataloader, device)

            # Print out what's happening
            print(
                f"Train loss: {train_loss:.5f} | Test loss: {test_loss:.5f}, Test
acc: {test_acc*100:.4f}%")
```

```python
def main(batchsize: int, epochs: int, workers: int, force_cpu: bool):
    model = LeNet5()
    data_module = MNISTDataModule(
        data_dir='../../../../data', batch_size=batchsize,
num_workers=workers)
    trainer = L.Trainer(max_epochs=epochs)
    trainer.fit(model, datamodule=data_module)
```

PyTorch Lightning

- Outras vantagens do Lightning
  - Já implementa TQDM (podemos customizar)
  - Controla log de métricas e loss
  - Controla checkpoints

PyTorch Lightning

```
from lightning.pytorch.callbacks import TQDMProgressBar, ModelCheckpoint

checkpoint_callback = ModelCheckpoint(monitor='val_loss', save_top_k=3, mode='min')
trainer = L.Trainer(max_epochs=120,
                    callbacks=[TQDMProgressBar(refresh_rate=5), checkpoint_callback])
```

PyTorch Lightning

- Outras vantagens do Lightning
  - Já implementa TQDM (podemos customizar)
  - Controla log de métricas e loss
  - Controla checkpoints
  - Integra com diferentes plataformas de MLOps
  - Facilita treinamento distribuído

PyTorch Lightning

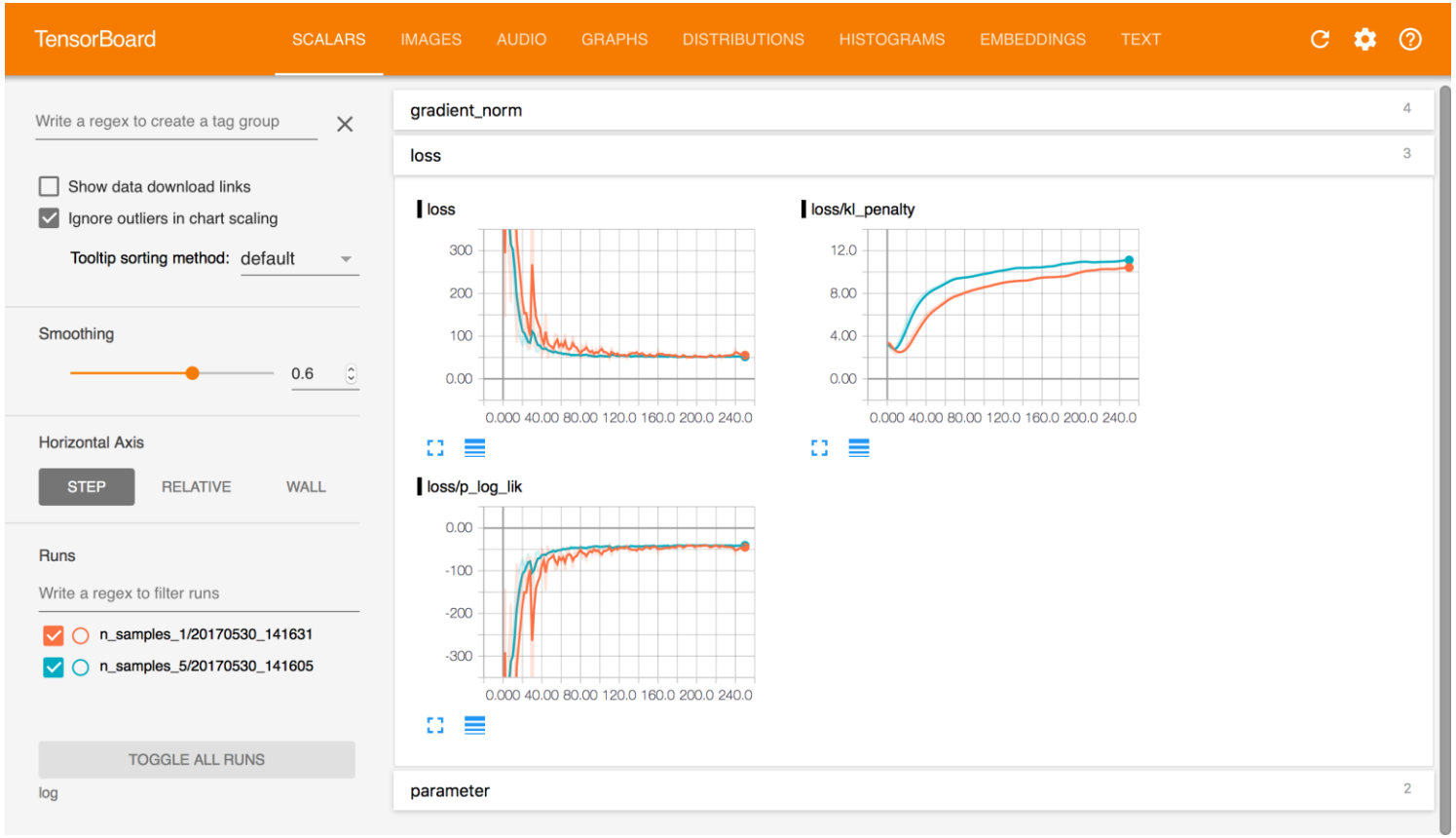
```
from lightning.pytorch.callbacks import TQDMProgressBar, ModelCheckpoint

logger = TensorBoardLogger('lightning_logs', name='model_name')
checkpoint_callback = ModelCheckpoint(monitor='val_loss', save_top_k=3, mode='min')
trainer = L.Trainer(max_epochs=120, logger=logger,
                    callbacks=[TQDMProgressBar(refresh_rate=5), checkpoint_callback])
```

Log de Experimentos

- Para acompanhar experimentos e tomar decisões de projeto é interessante realizar log e acompanhar as métricas do modelo
- Existem diversas ferramentas que permitem tal abordagem, vamos falar de duas delas
  - Tensorboard
  - wandb

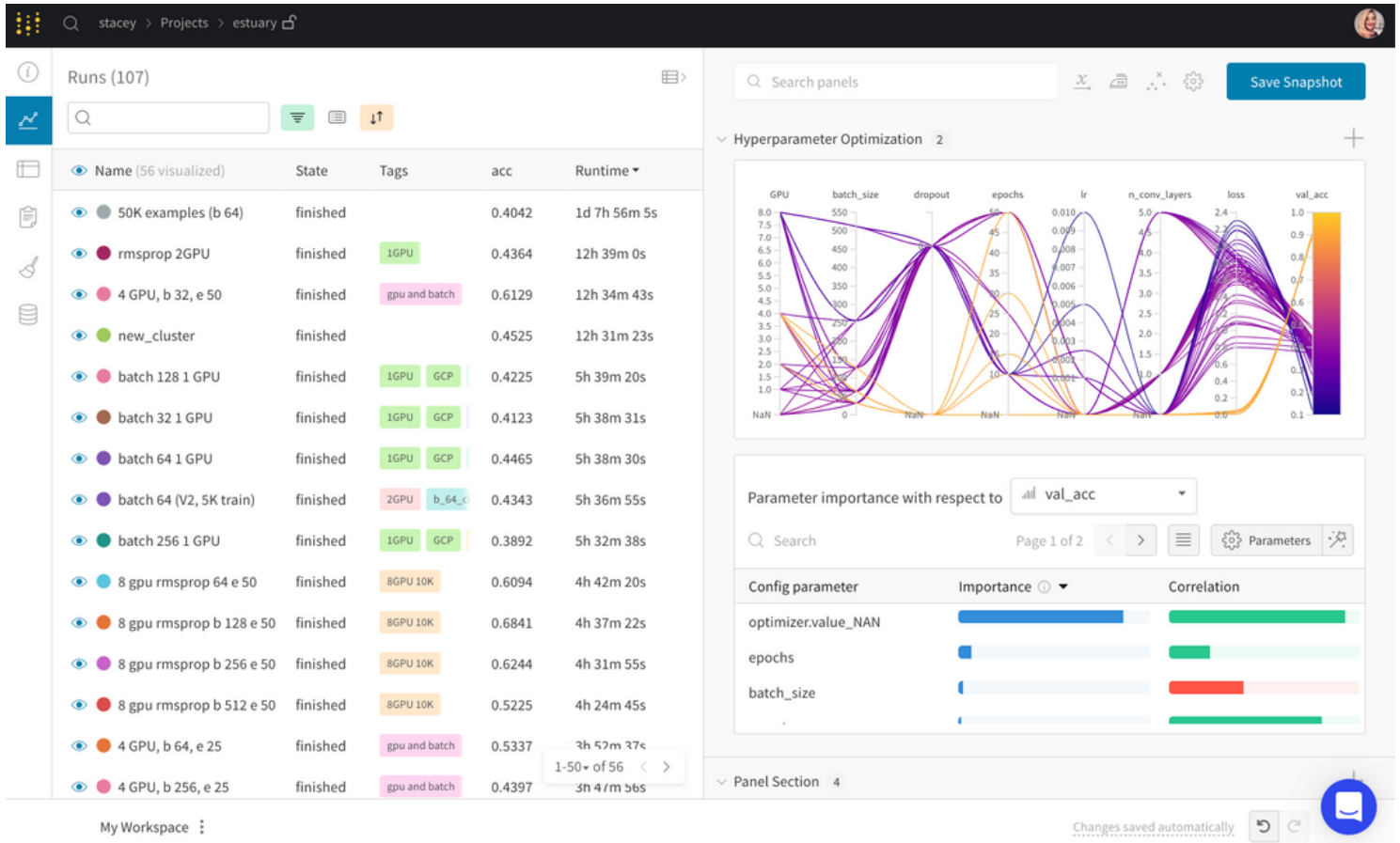
Tensorboard

- É a ferramenta padrão de monitoramento de treinamento de modelos
- Possibilita logar imagens, áudio, métricas, histogramas para diversos experimentos



wandb

- É uma plataforma online com diversas funcionalidades para log e acompanhamento de experimentos
- Permite geração de relatórios online para compartilhamento de informações
- Possibilita otimização de hiperparâmetros usando *"sweeps"*
  - Inclusive em máquinas diferentes



https://wandb.ai/

Referências

- https://lightning.ai/docs/pytorch/stable/
- https://wandb.ai