IDRIS

Pytorch Lightning

"The ultimate PyTorch research framework. Scale your models, without the boilerplate."

Léo Hunout, Idris/Inria

Postulat de départ (à choix multiple) :

- ✓ Je sais à quoi ressemble une boucle d'apprentissage en deep learning
- ✓ J'ai des bases (solide ou non) en Pytorch
- ✓ Je fais des codes qui sont difficiles à maintenir dans le temps
- ✓ Je n'aime pas perdre du temps à faire de l'ingénierie parce que la recherche n'attend pas !
- ✓ J'adore tester les dernières nouveautés, je suis SOTA

Je coche toutes les cases ? => heureusement que j'assiste à cette présentation

Retour aux bases

Un modèle

```
class Net(nn.Module) :

   def __init__(self) :
      super().__init__()
      self.conv1 = nn.Conv2d(1, 32, 3, 1)
      self.conv2 = nn.Conv2d(32, 64, 3, 1)
      self.dropout1 = nn.Dropout(0.25)
      self.dropout2 = nn.Dropout(0.5)
      self.fc1 = nn.Linear(9216, 128)
      self.fc2 = nn.Linear(128, 10)

   def forward(self, x) :
      x = self.conv1(x)
      x = F.relu(x)
      x = self.conv2(x)
      x = F.relu(x)
      x = F.max_pool2d(x, 2)
      x = self.dropout1(x)
      x = torch.flatten(x, 1)
      x = self.fc1(x)
      x = F.relu(x)
      x = self.dropout2(x)
      x = self.fc2(x)
      output = F.log_softmax(x, dim=1)

      return output
```

Une boucle d’apprentissage

```
for epoch in range(EPOCHS) :

   for i, data in enumerate(trainloader) :
      inputs, labels = data

      optimizer.zero_grad()
      # forward + backward + optimize
      outputs = net(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()

      # print statistics
      ...
```

Retour aux bases

Une boucle d'apprentissage

```
for epoch in range(EPOCHS) :

  for i, data in enumerate(trainloader) :
    inputs, labels = data

    optimizer.zero_grad()
    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # print statistics
    ...
```

En marche vers l'ajout

On peut atteindre de meilleure performance avec un lr scheduler ?

```
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
for epoch in range(EPOCHS) :

    for i, data in enumerate(trainloader) :
        inputs, labels = data

        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()

        # print statistics
        ...
```

En marche vers l’ajout x2

Mais faut que je suive l’évolution de mon entraînement ... Go mettre un tensorboard

```
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
writer = SummaryWriter()
for epoch in range(EPOCHS) :

    for i, data in enumerate(trainloader) :
        inputs, labels = data

        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()

        # print statistics
        writer.add_scalar(‘Loss/train’, loss)
```

En marche vers l'ajout x3

Mon code sur CPU tourne pas très vite. Il faut que j'utilise le GPU de Jean Zay

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)

scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
writer = SummaryWriter()
for epoch in range(EPOCHS) :

    for i, data in enumerate(trainloader) :
        inputs, labels = data

       inputs = inputs.to(device)
       labels = labels.to(device)

        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()

        # print statistics
        writer.add_scalar('Loss/train', loss)
```

En marche vers l’ajout x4

Mon entraînement prend trop de temps. Cela irai plus vite en utilisant la précision mixe !

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)

scaler = GradScaler()
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
writer = SummaryWriter()
for epoch in range(EPOCHS) :

    for i, data in enumerate(trainloader) :
        inputs, labels = data

       inputs = inputs.to(device)
       labels = labels.to(device)

        optimizer.zero_grad()
        # forward + backward + optimize
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        scheduler.step()
```

En marche vers l’ajout x5

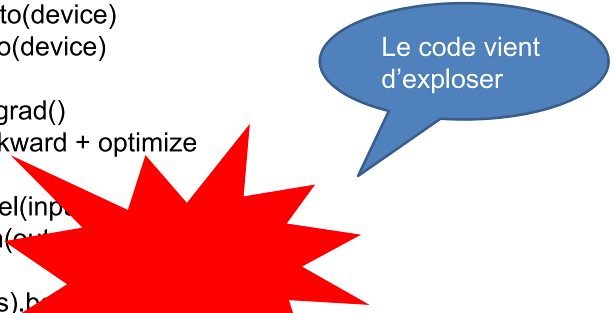
Euh, Jean Zay dispose de plusieurs GPU. Je pourrais faire de la distribution

```
dist.init_process_group(backend='nccl',init_method='env ://',
                        world_size=idr_torch.size,rank=idr_torch.rank)
torch.cuda.set_device(idr_torch.local_rank)
gpu = torch.device('cuda')
model = model.to(device)
ddp_model = DDP(model, device_ids=[idr_torch.local_rank])
scaler = GradScaler()
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
writer = SummaryWriter()
for epoch in range(EPOCHS) :

    for i, data in enumerate(trainloader) :
        inputs, labels = data

         inputs = inputs.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        # forward + backward + optimize
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)

        scaler.scale(loss).backward()
```

En marche vers l'ajout x666

Mais avec autant de GPU mes heures de calcul vont se réduire rapidement.
Go mettre de l'Early stopping ! (pas encore natif sous Pytorch)

```
dist.init_process_group(backend='nccl',init_method='env ://',
                        world_size=idr_torch.size,rank=idr_torch.rank)
torch.cuda.set_device(idr_torch.local_rank)
gpu = torch.device('cuda')
model = model.to(device)
ddp_model = DDP(model, device_ids=[idr_torch.local_rank])
scaler = GradScaler()
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
writer = SummaryWriter()
for epoch in range(EPOCHS) :

    for i, data in enumerate(trainloader) :
        inputs, labels = data

         inputs = inputs.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        # forward + backward + optimize
        with autocast():
            output = model(inp
            loss = loss_fn(

        scaler.scale(loss).b
```



Trop de complexité

Notre petite boucle initiale est devenu complexe, étape par étape.
Le code est maintenant moins lisible et par conséquence sa maintenabilité est plus difficile.
De plus, pendant tout ce temps, on n’a pas avancé sur le modèle et la méthodologie

On a ça :

Alors que l’on voudrait plutôt ça :

La solution :

PyTorch Lightning

You do the research. Lightning will do everything else.
The ultimate PyTorch research framework. Scale your models, without the boilerplate.

PyTorch Lightning

Lightning est un wrapper pure Pytorch. Pas besoin d'apprendre un nouveau langage. (ou pire faire du TF..)

Il sépare le code de recherche avec le code d'ingénieries.
Cela permet une meilleure reproductibilité des expériences ainsi qu'une lecture plus simple.
Il s'occupe des tâches d'ingénieries les plus répétitives.

Il permet d'accéder facilement à des fonctions avancés
(ex : apprentissage distribué, AMP, parallélisme de modèle)

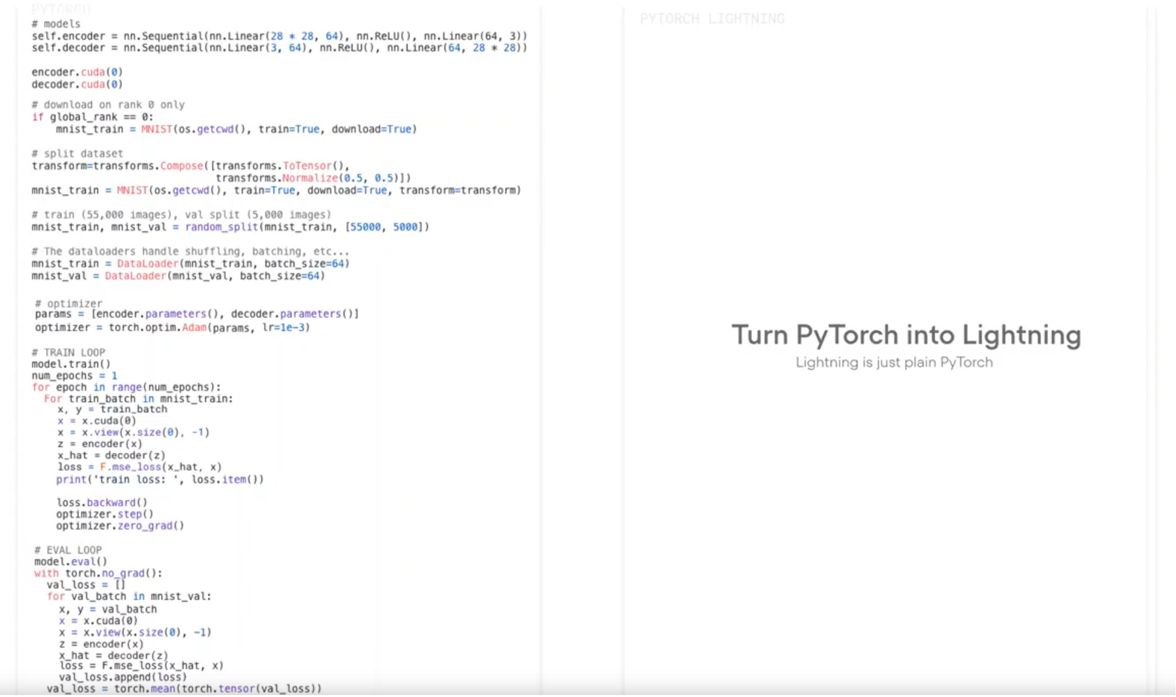
WHAT IS PYTORCH LIGHTNING?

Lightning makes coding complex networks simple.

Spend more time on research, less on engineering. It is fully flexible to fit any use case and built on pure PyTorch so there is no need to learn a new language. A quick refactor will allow you to:

- Run your code on any hardware
- Performance & bottleneck profiler
- Model checkpointing
- 16-bit precision
- Run distributed training
- Logging
- Metrics
- Visualization
- Early stopping
- ... and many more!

Démonstration refactorisation d’un code Pytorch vers Pytorch-Lightning




```
# models
self.encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
self.decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))

encoder.cuda(0)
decoder.cuda(0)

# download on rank 0 only
if global_rank == 0:
    mnist_train = MNIST(os.getcwd(), train=True, download=True)

# split dataset
transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize(0.5, 0.5)])
mnist_train = MNIST(os.getcwd(), train=True, download=True, transform=transform)

# train (55,000 images), val split (5,000 images)
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])

# The dataloaders handle shuffling, batching, etc...
mnist_train = DataLoader(mnist_train, batch_size=64)
mnist_val = DataLoader(mnist_val, batch_size=64)

# optimizer
params = [encoder.parameters(), decoder.parameters()]
optimizer = torch.optim.Adam(params, lr=1e-3)

# TRAIN LOOP
model.train()
num_epochs = 1
for epoch in range(num_epochs):
  for train_batch in mnist_train:
    x, y = train_batch
    x = x.cuda(0)
    x = x.view(x.size(0), -1)
    z = encoder(x)
    x_hat = decoder(z)
    loss = F.mse_loss(x_hat, x)
    print('train loss: ', loss.item())

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

# EVAL LOOP
model.eval()
with torch.no_grad():
  val_loss = []
  for val_batch in mnist_val:
    x, y = val_batch
    x = x.cuda(0)
    x = x.view(x.size(0), -1)
    z = encoder(x)
    x_hat = decoder(z)
    loss = F.mse_loss(x_hat, x)
    val_loss.append(loss)
  val_loss = torch.mean(torch.tensor(val_loss))
```

[Source](#)

Les hooks

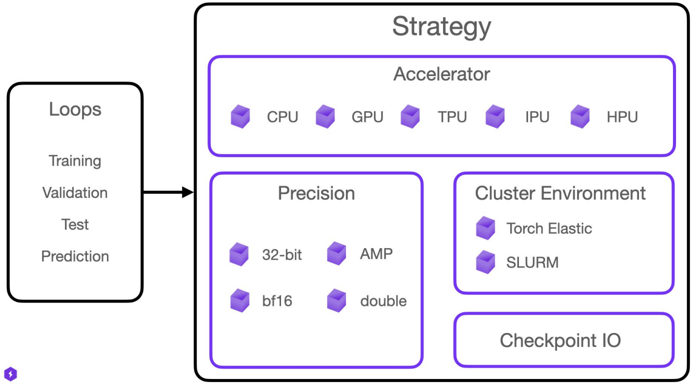
- Hooks
  - backward
  - on_before_backward
  - on_after_backward
  - on_before_zero_grad
  - on_fit_start
  - on_fit_end
  - on_load_checkpoint
  - on_save_checkpoint
  - load_from_checkpoint
  - on_hpc_save
  - on_hpc_load
  - on_train_start
  - on_train_end
  - on_validation_start
  - on_validation_end
  - on_test_batch_start
  - on_test_batch_end
  - on_test_epoch_start
  - on_test_epoch_end
  - on_test_start
  - on_test_epoch_end
  - on_test_start
  - on_test_end
  - on_predict_batch_start
  - on_predict_batch_end
  - on_predict_epoch_start
  - on_predict_epoch_end
  - on_predict_start
  - on_predict_end
  - on_train_batch_start
  - on_train_batch_end
  - on_train_epoch_start
  - on_train_epoch_end
  - on_validation_batch_start
  - on_validation_batch_end
  - on_validation_epoch_start
  - on_validation_epoch_end
  - on_post_move_to_device
  - configure_sharded_model
  - on_validation_model_eval
  - on_validation_model_train
  - on_test_model_eval
  - on_test_model_train
  - on_before_optimizer_step
  - configure_gradient_clipping
  - optimizer_step

Lightning dispose de +20 hooks permettant d'ajouter du code personnalisé dans la boucle d'apprentissage

Deep Learning tricks

ACCELERATORS

Accelerator	The Accelerator Base Class.
CPUAccelerator	Accelerator for CPU devices.
GPUAccelerator	Accelerator for GPU devices.
HPUAccelerator	Accelerator for HPU devices.
IPUAccelerator	Accelerator for IPUs.
TPUAccelerator	Accelerator for TPU devices.




```
# CPU accelerator
trainer = Trainer(accelerator="cpu")

# Training with GPU Accelerator using 2 GPUs
trainer = Trainer(devices=2, accelerator="gpu")

# Training with TPU Accelerator using 8 tpu cores
trainer = Trainer(devices=8, accelerator="tpu")

# Training with GPU Accelerator using the DistributedDataParallel strategy
trainer = Trainer(devices=4, accelerator="gpu", strategy="ddp")
```

dp, ddp, FairScale, Bagua, Collaborative, FairScale, DeepSpeed, Horovod ...

Deep Learning tricks

```
# accumulate every 4 batches (effective batch size is batch*4)
trainer = Trainer(accumulate_grad_batches=4)
```

```
# run learning rate finder, results override hparams.learning_rate
trainer = Trainer(auto_lr_find=True)
# call tune to find the lr
trainer.tune(model)
```

```
trainer = Trainer(sync_batchnorm=True)
```

```
# run batch size scaling, result overrides hparams.batch_size
trainer = Trainer(auto_scale_batch_size=  binsearch)
# call tune to find the batch size
trainer.tune(model)
```

+ Trainer flags
  - accelerator
  - accumulate_grad_batches
  - amp_backend
  - amp_level
  - auto_scale_batch_size
  - auto_select_gpus
  - auto_lr_find
  - benchmark
  - deterministic
  - callbacks
  - check_val_every_n_epoch
  - default_root_dir
  - devices
  - enable_checkpointing
  - fast_dev_run
  - gpus
  - gradient_clip_val
  - limit_train_batches
  - limit_test_batches
  - limit_val_batches
  - log_every_n_steps
  - logger
  - max_epochs
  - min_epochs
  - max_steps
  - min_steps
  - max_time
  - num_nodes
  - num_processes
  - num_sanity_val_steps
  - overfit_batches
  - plugins
  - precision
  - profiler
  - enable_progress_bar
  - reload_dataloaders_every_n_epochs
  - replace_sampler_ddp
  - resume_from_checkpoint
  - strategy
  - sync_batchnorm
  - track_grad_norm
  - tpu_cores
  - val_check_interval
  - weights_save_path
  - enable_model_summary

la liste complète est là :
https://pytorch-lightning.readthedocs.io/en/latest/common/trainer.html#trainer-flags

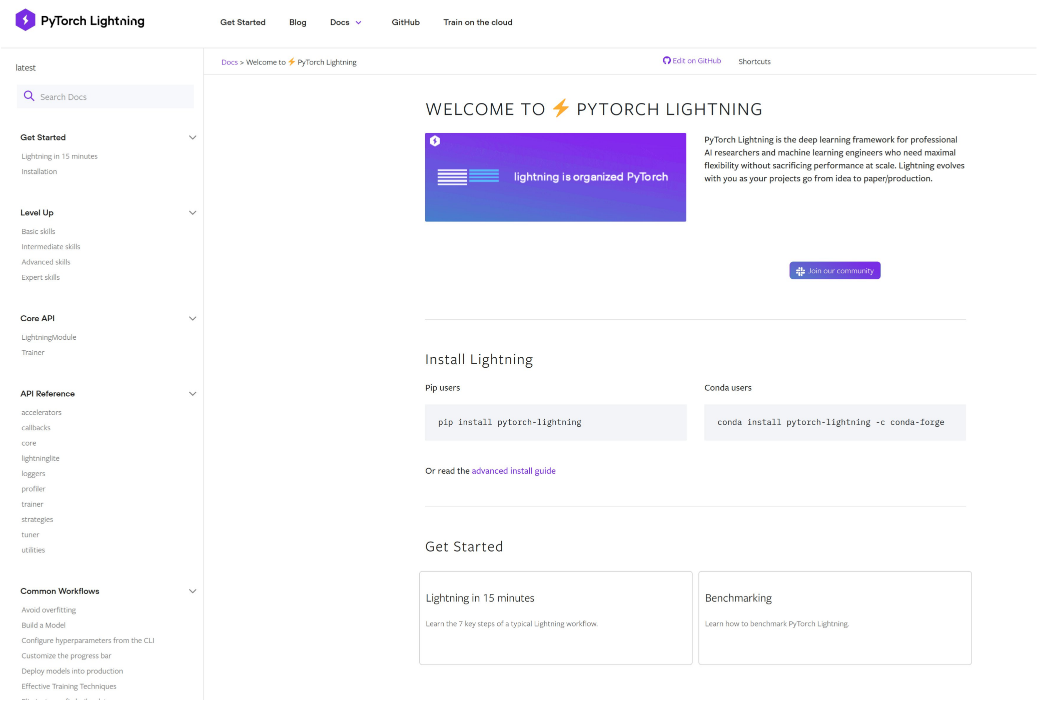
Deep Learning tricks

Survole d’un LitResNet

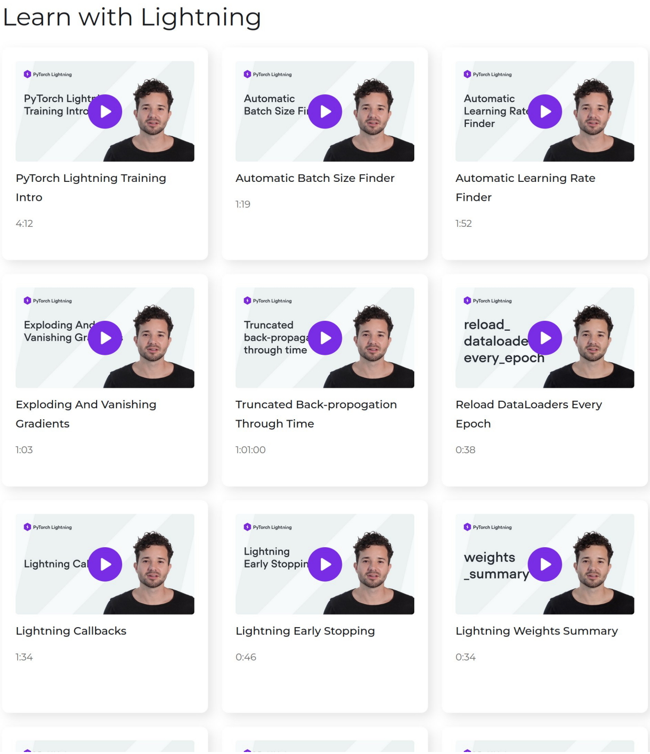
```
class LitResNetClassifier(pl.LightningModule):
    def __init__(self, num_classes,
                 resnet_version,
                 optimizer='adam',
                 lr=1e-3,
                 batch_size=16):

        super().__init__()

        self.__dict__.update(locals())
        resnets = {
            18: models.resnet18, 34: models.resnet34,
            50: models.resnet50, 101: models.resnet101,
            152: models.resnet152
        }

        optimizers = {'adam': Adam, 'sgd': SGD}
        self.optimizer = optimizers[optimizer]

        # instantiate loss criterion
        self.criterion = nn.BCEWithLogitsLoss() if num_classes == 2 else nn.CrossEntropyLoss()

        # instantiate model
        self.resnet_model = resnets[resnet_version]()

        # Replace old FC layer with Identity so we can train our own
        linear_size = list(self.resnet_model.children())[-1].in_features
        # replace final layer for fine tuning
        self.resnet_model.fc = nn.Linear(linear_size, num_classes)


    def forward(self, X):
        return self.resnet_model(X)

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        if self.num_classes == 2:
            y = F.one_hot(y, num_classes=2).float()

        loss = self.criterion(preds, y)
        acc = (y == torch.argmax(preds,1)).type(torch.FloatTensor).mean()
        # perform logging
        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True )
        self.log("train_acc", acc, on_step=True, on_epoch=True, prog_bar=True, logger=True )


        return loss
```

http://www.idris.fr/static/notebook-ia/Example_DataParallelism_PyTorch_Lightning.ipynb

La documentation

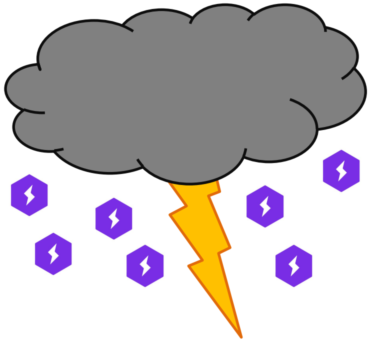


https://pytorch-lightning.readthedocs.io/en/latest/index.html



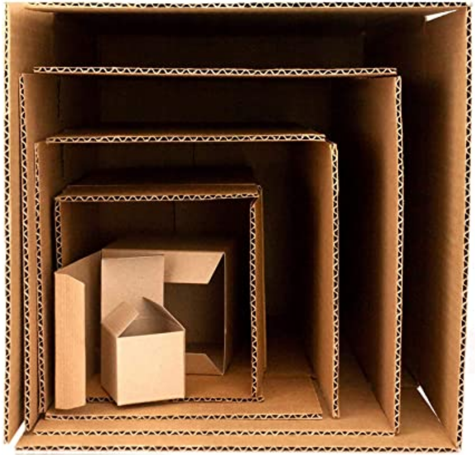
https://www.youtube.com/c/pytorchlightning

C’est trop beau pour être vrai ?!



Effectivement, tout n’est pas parfait !
Lightning a quelques points négatifs

Lightning est une surcouche !



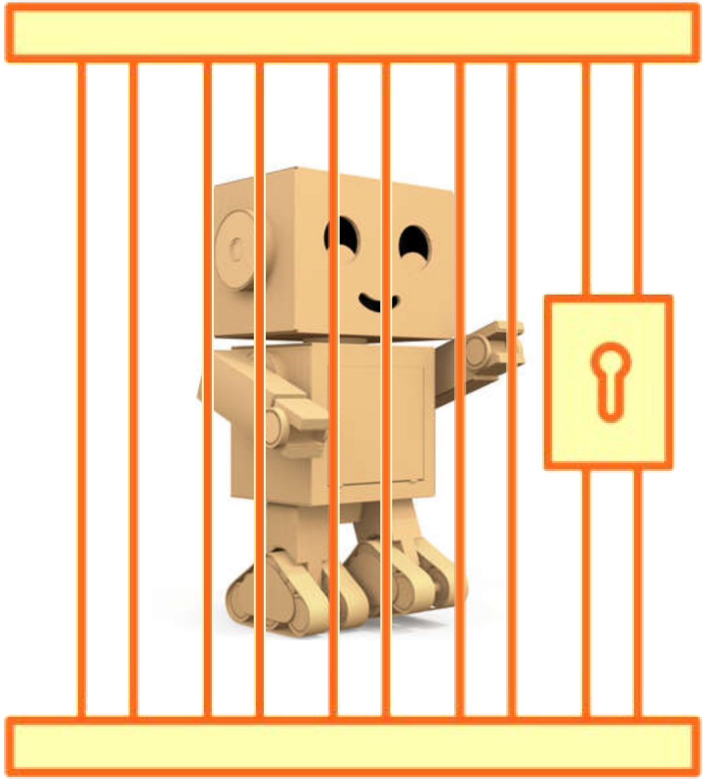
Lightning fait perdre en performance

Comparé à Pytorch et à entraînement équivalent :

Pytorch Lightning est entre 3% et 8% moins rapide

Lightning est (un peu) une boite noire

Lightning est une barrière à l'apprentissage



Lightning AI propose aussi :

Lightning Bolts

Lightning Transformers

TorchMetrics

Lightning Flash

GRID.AI

Intérêt pour les utilisateurs :

- Permet aux utilisateurs novices d’accéder facilement à de l’entraînement distribué
- Un code plus formaté et strict permet une meilleur reproductibilité des résultats

La conclusion