Formation - Introduction à Pytorch

Cours : Introduction à PyTorch Lightning

Alexis Lechervy

Sommaire

L'écosystème PyTorch

PyTorch : Librairie bas niveau

- Librairie pour la création de réseau de neurones.
- Très proche de Numpy dans son fonctionnement.
- Portage sur GPU.
- Adapté à des experts pour le développement de fonctionnalité non existante.

PyTorch Lightning Lightning Fabric : Librairies de niveau intermédiaire

- L'objectif de ces librairies est de diminuer le travail d'ingénierie.
- Prise en compte rapide du hardware, de l'apprentissage distribué, du logging, de la visualisation et des mécanismes standards.
- Adapté aux chercheurs et aux ingénieurs en machine learning.
- Lightning est construit au-dessus de PyTorch et peut être facilement enrichie.

Lightning Flash : Librairie haut niveau

- Pour le prototypage rapide, la reproduction de baseline ou la résolution de taches d'apprentissage standard.
- Simple et facile à prendre en main.
- Adapté aux débutants.
- Flash est construit au-dessus de Lightning et peut être facilement enrichie.

L'écosystème PyTorch

Torchvision

Boite à outils pour les applications de vision. La librairie contient

- Les transformations classiques sur les images (noir et blanc, redimensionnement, miroir, translation, rotations...)
- Les modèles classiques de la littérature,
- Les bases de données classiques de la littérature.

TorchMetrics

- À l'origine inclus dans Pytorch Lightning.
- Contient l'implémentation des principales métriques (collection de +80 métriques standards).

Pourquoi l'utiliser ?

Atouts de PyTorch Lightning

Simplicité Utilise un formalisme standard permettant d'avoir un code clair et limitant les possibilités de bugs.

Flexibilité Compatible avec tous les modèles PyTorch et n'importe quel type de données.

Reproductibilité Pipelines d'entraînement facilitant la reproductibilité.

Monitoring Intégration avec TensorBoard pour le suivi en temps réel des métriques.

GPU facilité Gestion automatique de la mémoire GPU.

Scalabilité Possibilité de déployer sur TPU/multi-GPU et cluster de calculs.

Personnalisation Flexibilité pour personnaliser les pipelines d'entraînement.

Sommaire

Les modèles héritant de LightningModule

Principe

Il est nécessaire de créer un objet héritant de la classe LightningModule. Cette classe permet d'organiser le code PyTorch selon les principales étapes :

`__init__` :	définition des éléments de l'architecture
`training_step` :	la boucle d'entraînement
`validation_step` :	la boucle de validation
`test_step` :	la boucle de test
`predict_step` :	la boucle de prédiction
`configure_optimizers` :	les optimiseurs et la gestion du taux d'apprentissage

Exemple de modèle héritant de LightningModule

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

class LitAutoEncoder(pl.LightningModule):
  def __init__(self, encoder, decoder): # definition du modele
    super().__init__()
    self.encoder = encoder
    self.decoder = decoder

  def forward(self,x):
    x = x.view(x.size(0), -1)
    return self.decoder(self.encoder(x)).view(-1,1,28,28)

  def training_step(self, batch, batch_idx): # etape de l'apprentissage
    x, y = batch
    x_hat = self(x)
    loss = nn.functional.mse_loss(x_hat, x)
    return loss

  def configure_optimizers(self): # configuration de l'optimiseur
    optimizer = optim.Adam(self.parameters(), lr=1e-3)
    return optimizer
```

Les paramètres du constructeur __init__

self.save_hyperparameters et self.hparams

Il est possible de sauvegarder tous les hyperparamètres du modèle dans la variable *hparams* avec la commande *self.save_hyperparameters()*. Cela permet notamment de les sauvegarder automatiquement dans les fichiers de sauvegarde et de logs.

```
def __init__(self,
  lr:float=0.1,
  momentum:float=0.9,
  weight_decay:float=1e-4,
  *args,
  **kwargs
  ):
  super().__init__()
  self.save_hyperparameters()
  ...
def configure_optimizers(self):
  return torch.optim.SGD(self.parameters(),
      lr=self.hparams.lr,
      momentum=self.hparams.momentum,
      weight_decay=self.hparams.weight_decay
    )
```

La gestion des Logs

Principe

Il est possible d'enregistrer facilement les évolutions de valeurs au cours de l'apprentissage en utilisant les méthodes *log* ou *log_dict*.

Exemples

```
# Log d'une metrique
def training_step(self, batch, batch_idx):
  ...
  self.log("my_metric", x)

# Log de plusieurs metriques sur la meme figure
def training_step(self, batch, batch_idx):
  ...
  self.log("performance", {"acc": acc, "recall": recall})

# Log de plusieurs metriques sur des figures differentes
def training_step(self, batch, batch_idx):
  ...
  self.log_dict({"acc": acc, "recall": recall})
```

La gestion des Logs

Comportement par défaut de l'enregistrement des logs dans les LightningModule

Les logs peuvent être agrégé par étape ou par époque. Par défaut, on a les paramètres suivant en fonctions des méthodes ou log est appelé :

Hook	on_step	on_epoch
on_train_start, on_train_epoch_start, on_train_epoch_end, training_epoch_end	False	True
on_before_backward, on_after_backward, on_before_optimizer_step, on_before_zero_grad	True	False
on_train_batch_start, on_train_batch_end, training_step, training_step_end	True	False
on_validation_start, on_validation_epoch_start, on_validation_epoch_end, validation_epoch_end	False	True
on_validation_batch_start, on_validation_batch_end, validation_step, validation_step_end	False	True

Le calcul des métriques d'évaluation

Principe

Le calcul de métriques d'évaluation permet de savoir si le réseau est performant sur la tâche cible.

Torchmetrics

La libraire *Torchmetrics* implémente les métriques standard et s'intègre parfaitement à Pytorch Lightning.

Exemple

```
class MyModel(LightningModule):
  def __init__(self, num_classes):
    ...
    self.valid_acc = torchmetrics.Accuracy(task="multiclass",
        num_classes=num_classes)
  def validation_step(self, batch, batch_idx):
    x, y = batch
    preds = self(x)
    self.valid_acc(preds, y)
    self.log('val_acc', self.valid_acc)
```

L'optimiseur

Principe

- On configure l'optimiseur dans la méthode *configure_optimizers*.
- Les valeurs retournées sont soit l'optimiseur, soit une liste d'optimiseur et une liste de *scheduler* pour le *learning rate*, soit un dictionnaire contenant toutes ces informations.

Exemple

```
def configure_optimizers(self):
  self.optim_algo = torch.optim.SGD(self.parameters(),
  lr=self.hparams.lr,
    momentum=self.hparams.momentum,
    weight_decay=self.hparams.weight_decay
  )
  lr_scheduler = torch.optim.lr_scheduler.StepLR(self.optim_algo,
      step_size=100, gamma=0.1)
  return {'optimizer':self.optim_algo,
    'lr_scheduler':{
    'scheduler':lr_scheduler
    }
  }
```

Utilisation de plusieurs Optimiseurs (1/2)

Principe

Il est possible d'utiliser plusieurs optimiseurs par exemple dans le cas des GAN.

```
def training_step(self, batch, batch_idx, optimizer_idx):
  ...
  # train generator
  if optimizer_idx == 0:
    ...
    return g_loss

  # train discriminator
  if optimizer_idx == 1:
    ...
    return d_loss

def configure_optimizers(self):
  gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
  dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
  return gen_opt, dis_opt
```

Utilisation de plusieurs Optimiseurs (2/2)

Principe

Il est possible de les appeler à des fréquences différentes.

```
def training_step(self, batch, batch_idx, optimizer_idx):
  ...
  # train generator
  if optimizer_idx == 0:
    ...
    return g_loss

  # train discriminator
  if optimizer_idx == 1:
    ...
    return d_loss

def configure_optimizers(self):
  gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
  dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
  return (
    {'optimizer': dis_opt, 'frequency': 5},
    {'optimizer': gen_opt, 'frequency': 1}
  )
```

Configuration du Scheduler pour le learning rate

Principe

Il est possible d'ajouter des paramètres pour configurer le scheduler du learning rate.

```
lr_scheduler_config = {
  # Obligatoire
  "scheduler": lr_scheduler,
  # Unite utilise pour les pas du scheduler. Valeur possible: 'step','
      epoch'.
  "interval": "epoch",
  # Frequence utilise pour la mise a jour du learning rate
  "frequency": 1,
  # Metrique utilise pour le monitoring utile pour des scheduler tel
      que `ReduceLROnPlateau`
  "monitor": "val_loss",
  # Si vrai, force le monitor a etre disponible lorsque que le
      scheduler est mis a jours et stop l'apprentissage sinon. Si faux,
       produit uniquement un warning.
  "strict": True,
  # Dans cas de callback `LearningRateMonitor`, permet d'avoir un nom.
  "name": None,
}
```

Sommaire

L'objet Trainer

Principe

L'objet *Trainer* de PyTorch Lightning est une abstraction qui automatise toute la partie ingénierie du code PyTorch. Il permet de contrôler tous les aspects de l'entraînement, comme le nombre d'époques, les accélérateurs, les callbacks, etc. Il s'agit du coeur de fonctionnement du framework PyTorch Lightning.

Exemple

```
# Definition du modele
model = MyLightningModule()
# Definition de la base de donnees
trainset = ...
valset = ...
train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=32,
    shuffle=True)
val_dataloader = torch.utils.data.DataLoader(valset, batch_size=32,
    shuffle=False)
# Apprentissage du modele
trainer = Trainer()
trainer.fit(model, train_dataloader, val_dataloader)
```

Paramétrage du Trainer

Principe

L'objet *Trainer* contient de nombreux paramètres permettant de simplement configurer le code.

Exemple

```
trainer = Trainer(
  accumulate_grad_batches = 4, #accumulation des gradiens sur 4 batches
  deterministic = False, # le code n'est pas deterministique
  check_val_every_n_epoch = 10, # les calculs sur validation se font
      tous les 10 epoques
  max_epochs = 1000, # fixe le nombre max d'epoque a 1000
  max_time = "00:12:00:00", # Stop apres 12h de calcul
  precision = 16, # fait les calculs sur des float 16 au lieu de 32
)
```

Le multi-GPU / multi-nodes (1/2)

Principe

Il est possible de placer un code en CPU, GPU ou en multi-noeuds en changeant uniquement les paramètres du *Trainer*. Pour que cela puisse fonctionner, il ne faut pas utiliser des assignements manuel dans le code.

Exemple

Code PyTorch sans lightning :

```
def forward(self, x):
  z = torch.Tensor(2, 3)
  z = z.cuda(0)
```

Code PyTorch pour lightning :

```
def forward(self, x):
  z = torch.Tensor(2, 3)
  z = z.to(x)
```

Code pour déclarer une variable sans avoir d'autre variable auquel se référer (par exemple dans __init__) :

```
def __init__(self):
  self.register_buffer("sigma", torch.eye(3)) # self.sigma dans la
      suite
```

Le multi-GPU / multi-nodes (2/2)

Exemples

```
# Utilisation d'un GPU
trainer = Trainer(accelerator="gpu", devices=1)
# Utilisation de l'accelerateur sur les puces Apple
trainer = Trainer(accelerator="mps", devices=1)
# Utilisation de 4 GPU
trainer = Trainer(accelerator="gpu", devices=4)
# Utilisation de tous les GPU disponibles
trainer = Trainer(accelerator="gpu", devices=-1)
# Utilisation des GPU 0 et 3
Trainer(accelerator="gpu", devices=[0, 3])

from lightning.pytorch.accelerators import find_usable_cuda_devices
# Utilise deux GPU disponible sur le systeme
trainer=Trainer(accelerator="cuda",devices=find_usable_cuda_devices(2))
# Utilise les GPU 2 de 4 noeud (soit 8 GPU)
trainer = Trainer(accelerator="gpu", devices=2, num_nodes=4)
# Utilise la strategie DDP pour // sur 4 GPU
trainer = Trainer(strategy="ddp", accelerator="gpu", devices=4)
# Utilise la librairie Horovod pour //. gpus=1 dans ce cas meme en
    multi-GPU.
trainer = Trainer(accelerator='horovod', gpus=1)
```

La gestion des Loggueurs

Principe

Il est possible d'utiliser plusieurs méthodes de gestion des logs en parallèle. Par défaut, *Tensorboard* est utilisé (dans le dossier *lightning_logs*).

Exemples

```
from pytorch_lightning import loggers as pl_loggers

# Changement du dossier de sauvegarde
tb_logger = pl_loggers.TensorBoardLogger(save_dir="logs/")
trainer = Trainer(logger=tb_logger)

# Utilisation du site WanDB pour les logs
wandb_logger = pl_loggers.WandbLogger(project="test_My_Net")
trainer = Trainer(logger=tb_logger)

# Utilisation de plusieurs loggueurs
trainer = Trainer(logger=[tb_logger,wandb_logger])
```

La gestion des hyperparamètres en ligne de commande : LightningCLI

Définition d'un fichier main.py

```python
from lightning.pytorch.cli import LightningCLI
def cli_main():
  cli = LightningCLI(DemoModel, BoringDataModule)

if __name__ == "__main__":
  cli_main()
```

Appel depuis le Shell

```
# Lancement d'un apprentissage
python main.py fit
# changement du taux d apprentissage
python main.py fit --model.learning_rate 0.1
# cretion d'un fichier de configuration
python main.py fit --print_config > config.yaml
# lancement d'un apprentissage avec fichier de configuration
python main.py fit --config config.yaml
```

Sauvegarde des paramètres dans un checkpoint

Principe

Par défaut un checkpoint est sauvegardé à chaque époque dans le dossier lightning_logs/version_[num]/checkpoints/....ckpt . On y retrouve :

- La précision des calculs de 16/32/64bits,
- Le numéro d'époque actuelle,
- Le numéro du pas,
- L'état du module LightningModule et notamment les valeurs des poids du réseau (state_dict),
- L'état de tous les optimiseurs,
- L'état de tous les *scheduler* pour les *learning rate*,
- L'état des *callbacks*,
- L'état des modules de données,
- Les hyperparamètres utilisés pour le modèle s'ils sont transmis en tant que hparams (Argparse.Namespace),
- Les hyperparamètres utilisés pour le module de données s'ils sont transmis en tant que hparams (Argparse.Namespace),
- L'état des boucles.

Reprise de l'apprentissage à partir d'un checkpoint

Principe

Il est possible de reprendre l'apprentissage à partir d'un checkpoint en utilisant :

```
model = MyModel()
trainer = Trainer()

# restaure automaticament le model, epoch, step, LR schedulers, ...
trainer.fit(model, ckpt_path="some/path/to/my_checkpoint.ckpt")
```

Utilisation d'un modèle déjà appris

Il est possible de charger un modèle déjà appris pour l'utiliser comme dans PyTorch :

```
model = MyModel.load_from_checkpoint("/path/to/checkpoint.ckpt")
# desactive le hasard, dropout, ...
model.eval()
# predit en utilisant le modele
y_hat = model(x)
```

Sommaire

La gestion des données

Principe

Il est possible d'utiliser des *DataLoader* de PyTorch ou d'implémenter des objets héritant de *LightningDataModule*.

Classe héritant de *LightningDataModule*

Les méthodes à implémenter sont les suivantes :

`__init__` :	Constructeur, permet de récupérer les paramètres.
`prepare_data` :	Permet de télécharger les données et de faire les splits. Appelé que sur le processus principal en cas de parallélisation.
`setup` :	Permet de faire l'initialisation de la base de données. Est appelé pour chaque processus en cas de parallélisation.
`train_dataloader` :	Retourne la base d'apprentissage.
`val_dataloader` :	Retourne la base de validation.
`test_dataloader` :	Retourne la base de test.

Exemple de LightningDataModule (1/2)

```
class Cifar10DataModule(pl.LightningDataModule):
  def __init__(self,
    data_dir:str='./data',
    batch_size:int=32,
    seed:int=42,
    size_val:float=0.1
  ):
  super().__init__()
  self.data_dir = data_dir
  self.batch_size = batch_size
  self.seed = seed
  self.size_val = size_val
  self.name_classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', '
      frog', 'horse', 'ship', 'truck')

  def prepare_data(self):
    datasets.CIFAR10(self.data_dir, train=True, download=True)
    datasets.CIFAR10(self.data_dir, train=False, download=True)

  def train_dataloader(self) -> DataLoader:
    return DataLoader(self.cifar_train, batch_size=self.batch_size,
        shuffle=True)
```

Exemple de LightningDataModule (2/2)

```
def setup(self, stage: Optional[str] = None) -> None:
  if stage == "test" or stage is None:
    transform_test = transforms.Compose([transforms.Resize(224),
        transforms.ToTensor()])
  self.cifar_test = datasets.CIFAR10(
    self.data_dir, train=False, download=False, transform=
        transform_test
  )
  if stage == "fit" or stage is None:
    transform_train = transforms.Compose(
    [transforms.RandomCrop(32, padding=4),
     transforms.Resize(224),
     transforms.ToTensor()])
  cifar_full = datasets.CIFAR10(
    self.data_dir, train=True, download=False, transform=
        transform_train
  )
  self.cifar_train, self.cifar_val = torch.utils.data.random_split(
    cifar_full,
    [len(cifar_full)-int(len(cifar_full)*self.size_val),
    int(len(cifar_full)*self.size_val)],
    generator=torch.Generator().manual_seed(self.seed)
  )
```

Utilisation de LightningDataModule

Principe

On peut passer un objet de type *LightningDataModule* à l'attribut *datamodule* de la méthode *fit* du *Trainer* :

```
cifar = Cifar10DataModule()
model = MyModel()

trainer = Trainer()
trainer.fit(model, datamodule = cifar)
```