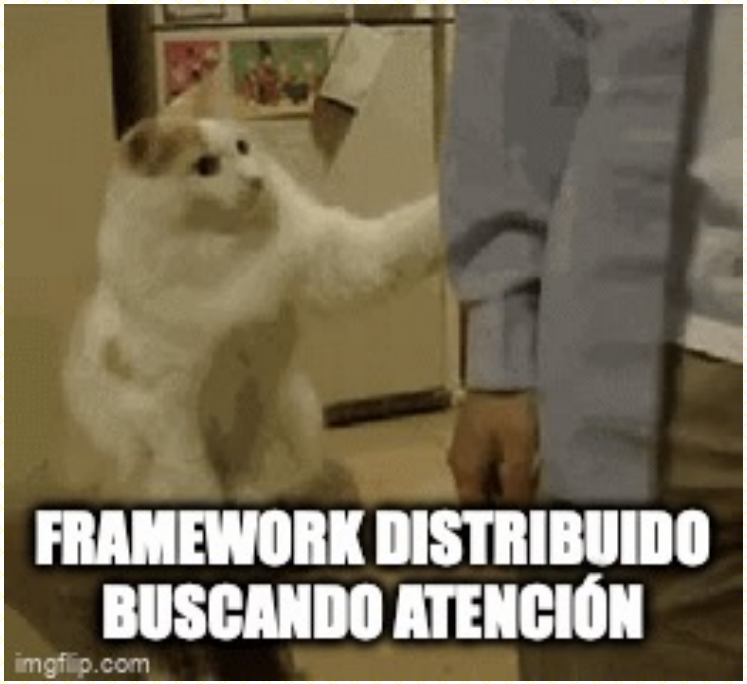
# Pytorch Lightning

Diego Andrade Canosa

# Índice

- Introducción
- Herramientas de entrenamiento distribuido
  - Lightning
  - DeepSpeed
  - Otras herramientas




citic

# Introducción

- Además del soporte nativo de Pytorch para entrenamiento distribuido, existen varias herramientas avanzadas que permiten explotar paralelismo en modelo de ML construidos en base a Pytorch
  - Pytorch Lightning
  - RAY
  - DeepSpeed
  - Accelerate
  - Horovod
  - ...


citic

# Pytorch Lighting

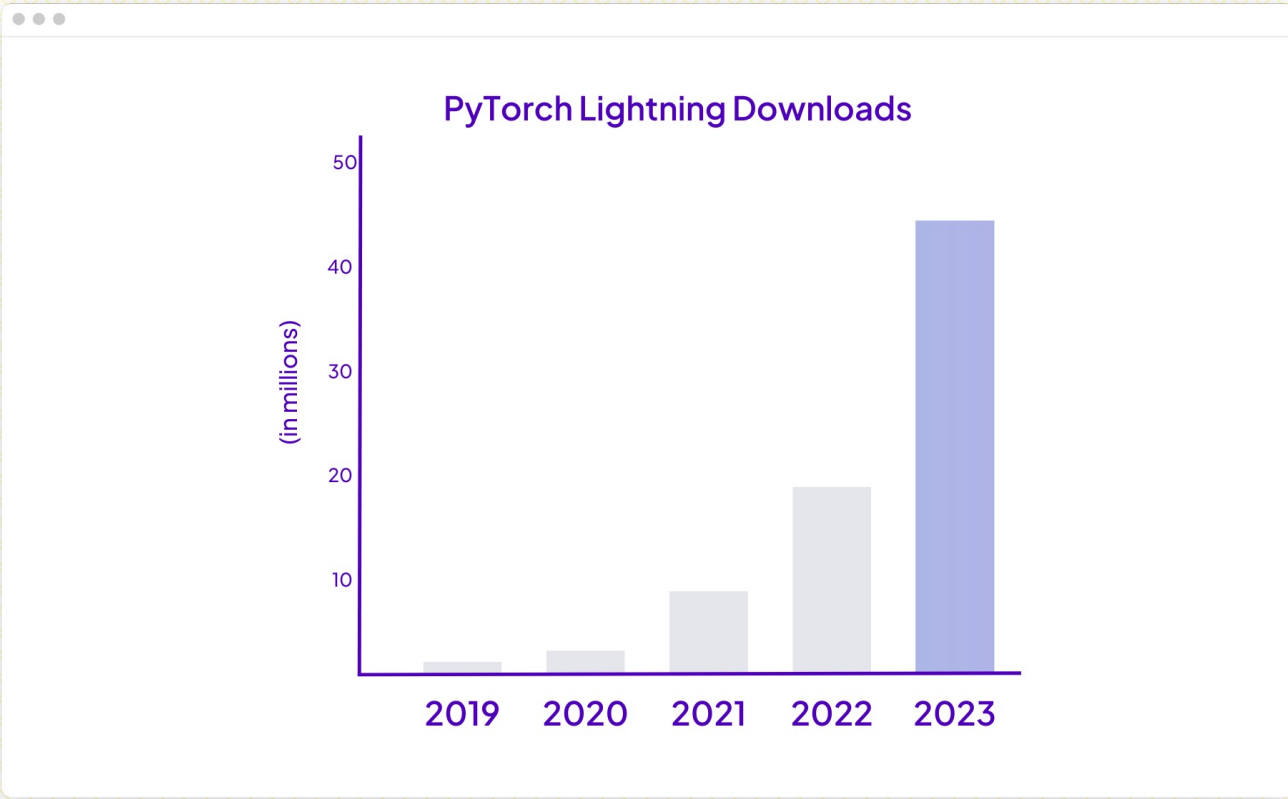
- Framework para escalar modelos a través de la refactorización de código Pytorch
- Su uso proporciona:
  - Ejecución el código en cualquier hardware
    - Entornos distribuidos
  - Profiling de rendimiento y cuellos de botella
    - Y *Logging*
  - Checkpointing
  - Half-precision
- Buen soporte y documentación para Slurm: https://lightning.ai/docs/pytorch/stable/clouds/cluster_advanced.html

Fuente (demo en parte inferior): PyTorch Lightning

Documentación: Basic skills — PyTorch Lightning 1.8.4.post0 documentation (pytorch-lightning.readthedocs.io)

# Pytorch Lightning

- Proyecto en ebullición del ecosistema Pytorch




citic

# Pytorch Lightning

- Versión actual==2.0.4
- Cambios constantes en la interfaz (a veces traumáticos)
  - Publican siempre una guía de actualización para nuestros códigos https://lightning.ai/docs/pytorch/latest/upgrade/migration_guide.html
  - Y también unos scripts que analizan tu código y sugieren los cambios que requiere la adaptación a otra versión

# Otros elementos de Lightning

- Fabric: Alternativa a Lightning que incorpora muchos de sus principios pero que minimiza el código necesario para adaptar código Pytorch
  - Marzo 2023!
  - Mayor flexibilidad que Lightning
  - Veremos algo más adelante en esta presentación
- Bolts: Es una caja de herramientas (*toolbox*) que incorpora varios bloques constructores fáciles de usar:
  - Modelos SOTA preentrenados
  - Componentes de modelos
  - Callbacks
  - Funciones de pérdida
  - Cargadores de datos
  - En revisión!
  - Docs: https://lightning-bolts.readthedocs.io/en/stable/introduction_guide.html

# Pytorch Lightning: Módulos

- En Lightning debemos partir de un modelo definido con el contenedor torch.nn.Module
  - Está integrado con el sistema de autograd (autodiferenciación) de Pytorch
    - Fácil especificar qué parámetros son aprendibles
  - Permiten:
    - Asociar fácilmente a dispositivos computacionales
    - Aplicar operaciones como *quantization* o purgado
  - Podemos:
    - Usar módulos preexistentes
    - Definir módulos propios

# Pytorch Lightning: Módulos

- Definición
  - Hereda de la clase torch.nn.Module
  - Define un estado que se usa en la computación (normalmente con la ayuda de la clase nn.Parameter)
  - Define una función "*Forward*" que realiza el cálculo

Fuente: Modules — PyTorch 1.13 documentation

# Pytorch Lighthing: Módulos

- Definición módulos Pytorch

```
import pytorch_lightning as pl
class Encoder(nn.Module):
    (...)

class Decoder(nn.Module):
    (...)
```

# Pytorch Lighthing: Uso

- Para tales modelos se habilita el uso de Lightning heredando de la clase LightningModule. Cumple la siguiente interfaz
  - __init__: Contiene el constructor del modelo con la definición de las capas que lo componen
  - setup: Aquí se debe incluir todo el código que se corresponde con la inicialización del modelo
    - Complementa al constructor
  - configure_optimizers: contiene la definición del optimizador
  - forward: define la pasada forward

# Pytorch Lighthing: Uso

- En Lightning el modelo se define heredando de la clase LightningModule. Cumple la siguiente interfaz
  - training_step: define un paso de entrenamiento
    - Cómo se procesa en un paso de entrenamiento cada batch del conjunto de entrenamiento
    - Se lanza cuando se llama al método **fit** del Trainer
  - validation_step: define un paso de la tarea de validación
    - Cómo se procesa en un paso del bucle de validación cada batch del conjunto de validación
    - Se lanza cuando se llama al método **eval** del Trainer
  - test_step: define el paso del bucle de test
    - Se lanza cuando se llama al método **test** del Trainer
  - predict_step: define el paso de predicción que por defecto llama al método forward()
    - Se lanza cuando se llama al método **predict** del Trainer

# Pytorch Lighthing: Uso

```python
class LitAutoEncoder(pl.LightningModule):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = F.mse_loss(x_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
        return optimizer
```

# Pytorch lightning: Uso

- Para usar el lightningmodule una vez definido, debemos usar un entrenador (*Trainer*)
  - En su instanciación se le pueden indicar varios parámetros
    - num_nodes
    - gpus
    - precision
    - ...
  - Para desencadenar el proceso de entrenamiento se ejecuta el método *trainer.fit*

# Pytorch Lighthing: Uso

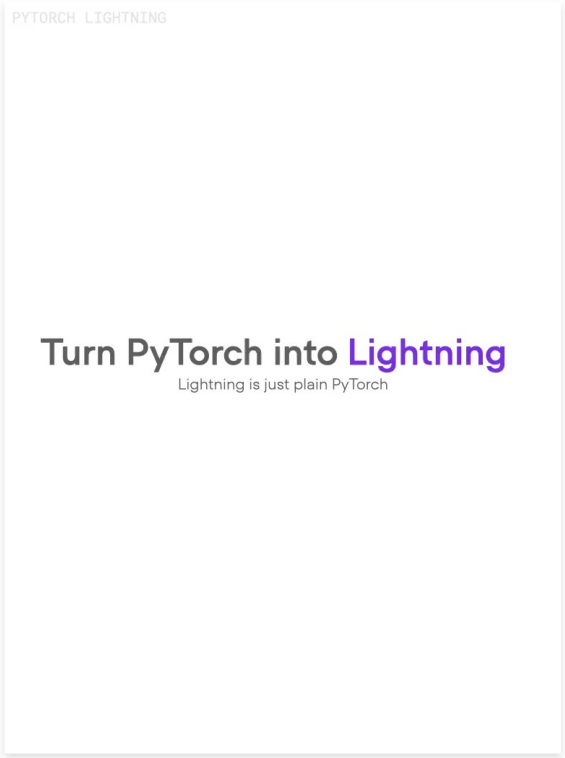
```
dataset = MNIST(os.getcwd(), download=True, transform=transforms.ToTensor())
train_loader = DataLoader(dataset)
# model
autoencoder = LitAutoEncoder(Encoder(), Decoder())
# train model
trainer = pl.Trainer()
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

# Pytorch Lightning: Uso

PYTORCH

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

PYTORCH LIGHTNING




Fuente: https://lightning.ai/docs/pytorch/stable/starter/introduction.html

citic

# ¿Qué nos proporciona el trainer?

- Añade automáticamente 40 características: https://lightning.ai/docs/pytorch/stable/common/trainer.html#trainer-flags
  - Iteración en epochs y en batches
  - Llamadas a los step, backward, zero_grad
  - Llamada a eval al final de cada epoch (habilita y deshabilita autograd)
  - Checkpointing (guardado y carga del modelo)
  - Salidas para TensorBoard
  - Escalado a múltiples nodos y múltiples GPUs por nodo
  - Ajuste de la precisión de los parámetros del modelo (ej. 16-bit precisión)

# ¿Qué nos proporciona el trainer?

- Configuración en una sola línea de distintas estrategias de entrenamiento

```
# train on 4 GPUs
trainer = Trainer(
    devices=4,
    accelerator="gpu",
 )

# train 1TB+ parameter models with Deepspeed/fsdp
trainer = Trainer(
    devices=4,
    accelerator="gpu",
    strategy="deepspeed_stage_2",
    precision=16
 )

# 20+ helpful flags for rapid idea iteration
trainer = Trainer(
    max_epochs=10,
    min_epochs=5,
    overfit_batches=1
 )

# access the latest state of the art techniques
trainer = Trainer(callbacks=[StochasticWeightAveraging(...)])
```

# Operación del Trainer

- Esto es lo que hace el Trainer de forma automática para entrenar al modelo
  - Invocación del método fit() del Trainer
- Otros métodos del Trainer
  - validate
  - test

```
# put model in train mode
model.train()
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)
```

# Flags (parámetros) del Trainer

- accelerator=[“cpu”/”gpu”/”tpu”/”ipu”/”auto”]
- accumulate_grad_batches=*integer*
  - Acumula los gradientes durante *integer* batches antes de ajustar los parámetros
- benchmark=[True/False]
  - Incorpora un autotuner de código que mejora el rendimiento si los tamaños de las entradas no cambian
- deterministic=[True/False]
  - Hace que el comportamiento entre ejecuciones sea determinista. No cambia la semilla de generadores aleatorios

# Flags del Trainer

- callbacks=[*list_of_callbacks*]
  - Configura una lista de callbacks que serán incorporados a diferentes puntos de las llamadas hechas por el Trainer
- default_root_dir
- devices
- [min/max]_epochs
- [min/max]_steps
- num_nodes

```
# train on 8 GPUs (same machine (ie: node))
trainer = Trainer(gpus=8, distributed_backend='ddp')

# train on 32 GPUs (4 nodes)
trainer = Trainer(gpus=8, distributed_backend='ddp', num_nodes=4)
```

Referencia: https://lightning.ai/docs/pytorch/stable/common/trainer.html

# distributed_backend (deprecated) -> strategy

| | |
|---|---|
| `DDPStrategy` | Strategy for multi-process single-device training on one or multiple nodes. |
| `DeepSpeedStrategy` | Provides capabilities to run training using the DeepSpeed library, with training optimizations for large billion parameter models. |
| `FSDPStrategy` | Strategy for Fully Sharded Data Parallel provided by torch.distributed. |
| `HPUParallelStrategy` | Strategy for distributed training on multiple HPU devices. |
| `IPUStrategy` | Plugin for training on IPU devices. |
| `ParallelStrategy` | Plugin for training with multiple processes in parallel. |
| `SingleDeviceStrategy` | Strategy that handles communication on a single device. |
| `SingleHPUStrategy` | Strategy for training on single HPU device. |
| `SingleTPUStrategy` | Strategy for training on a single TPU device. |
| `Strategy` | Base class for all strategies that change the behaviour of the training, validation and test- loop. |
| `XLAStrategy` | Strategy for training multiple TPU devices using the `torch_xla.distributed.xla_multiprocessing.spawn()` method. |

# Lightning

- Instalación de Python Lightning
  - En FT3
  - compute –gpu
  - source $STORE/mypython/bin/activate
  - pip install lightning

- Sigue las instrucciones del README: https://github.com/diegoandradecanosa/CFR24/tree/main/pytorch_dist/lightning/000

# Lightning

- Inspeccionar el log de tensorboard generado por lightning

*tensorboard --logdir . --host `hostname -i`*

# Training Callbacks

- Los *callbacks* son un componente de Pytorch que nos permiten añadir funcionalidades a los entrenamientos que se incorporan en puntos determinados del proceso de entrenamiento, validación y evaluación
- Formalmente, el código de Lightning se debería distribuir de la siguiente forma:
  - Trainer: Es todo lo que tiene que ver con el proceso de ingeniería
  - LightningModule: Lo que tiene que ver con la definición del modelo y su procesamiento
  - Callbacks: Código no esencial
- Fuente: https://lightning.ai/docs/pytorch/stable/extensions/callbacks.html

# Training Callbacks

- Podemos definir nuestros propios Callbacks heredando de la clase *lightning.pytorch.callbacks*
  - Nuestro CallBack puede añadir código en ciertos puntos determinados, y este código se incorpora a través de la definición de Hooks

```python
from lightning.pytorch.callbacks import Callback


class MyPrintingCallback(Callback):
    def on_train_start(self, trainer, pl_module):
        print("Training is starting")

    def on_train_end(self, trainer, pl_module):
        print("Training is ending")


trainer = Trainer(callbacks=[MyPrintingCallback()])
```

*Lista completa de hooks de Callback: https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.callbacks.Callback.html#lightning.pytorch.callbacks.Callback*

# Training Callbacks

- Ejemplo de Hooks que pueden ser definidos:
  - on_[validation/test/train/predict]_epoch_[start/end]
  - on_[validation/test/train/predict]_batch_[start/end]
  - on_before_[backward/optimizer_step/zero_grad]
  - on_after_backward

# Training Callbacks

- Existen una serie de Callbacks ya definidos en Lightning que pueden ser incorporados fácilmente a nuestro trainer. Ejemplos:
  - EarlyStopping: Monitoriza una métrica y detiene el entrenamiento cuando la métrica deja de mejorar
  - Timer: Registra el tiempo que pasamos en entrenamiento, validación y test
  - StochasticWeightAveraging: Implementa un mecanismo de SWA

# Training Callbacks

- El Callback se configura en el momento de instanciar el Trainer

```
early_stop_callback = EarlyStopping(
   monitor='val_loss',patience=3,strict=False,
   verbose=False,mode='min')

timer_callback = Timer()

swa_callback = StochasticWeightAveraging(swa_epoch_start=0.3)

trainer = Trainer(callbacks=[early_stop_callback, timer_callback, swa_callback])
```

# Lightning Callbacks

- Sigue las instrucciones del README: https://github.com/diegoandradecanosa/CFR24/tree/main/pytorch_dist/lightning/001

# Creación de datasets propios con Lightning

- Lightning permite la creación de *datamodule*s (LightningDataModule) que encapsulan todos los pasos necesarios para procesar conjuntos de datos
  - Download/tokenize/process
  - Limpiar y (opcional) guardar en disco
  - Cargar en un *DataSet*
  - Aplicar transformaciones
  - Crear el correspondiente *DataLoader*

Fuente: https://lightning.ai/docs/pytorch/stable/levels/intermediate_level_9.html

# Dataloaders vs LightningDataModule

- A funciones del trainer como fit, le podemos pasar los datos de dos maneras:
  - Como dos Dataloaders (*train_dataloaders,val_dataloaders*)
  - Como un LightningDataModule (*datamodule*)

# API del LightningDataModule

- Algunos métodos incluidos dentro del API de LightningDataModule
  - prepare_data
  - setup
  - train_dataloader
  - val_dataloader
  - test_dataloader
  - predict_dataloader

# DataSets en Lightning: Ejemplo

```
# regular PyTorch
test_data = MNIST(my_path, train=False, download=True)
predict_data = MNIST(my_path, train=False, download=True)
train_data = MNIST(my_path, train=True, download=True)
train_data, val_data = random_split(train_data, [55000, 5000])

train_loader = DataLoader(train_data, batch_size=32)
val_loader = DataLoader(val_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
predict_loader = DataLoader(predict_data, batch_size=32)
```

# DataSets en Lightning: Ejemplo

```python
class MNISTDataModule(pl.LightningDataModule):
    def __init__(self, data_dir: str = "path/to/dir", batch_size: int = 32):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size

    def setup(self, stage: str):
        self.mnist_test = MNIST(self.data_dir, train=False)
        self.mnist_predict = MNIST(self.data_dir, train=False)
        mnist_full = MNIST(self.data_dir, train=True)
        self.mnist_train, self.mnist_val = random_split(mnist_full, [55000, 5000])

    def train_dataloader(self):
        return DataLoader(self.mnist_train, batch_size=self.batch_size)

    def val_dataloader(self):
        return DataLoader(self.mnist_val, batch_size=self.batch_size)

    def test_dataloader(self):
        return DataLoader(self.mnist_test, batch_size=self.batch_size)

    def predict_dataloader(self):
        return DataLoader(self.mnist_predict, batch_size=self.batch_size)

    def teardown(self, stage: str):
        # Used to clean-up when the run is finished
        ...
```

# DataSets en Lightning: Ejemplo

```
mnist = MNISTDataModule(my_path)
model = LitClassifier()

trainer = Trainer()
trainer.fit(model, mnist)
```

# DataModule

Sigue las instrucciones del README:
https://github.com/diegoandradecanosa/CFR24/tree/main/pytorch_dist/lightning/004

Basado en: https://lightning.ai/docs/pytorch/stable/data/datamodule.html

# Parámetros a través de CLI

- Lightning proporciona un mecanismo para configurar los hiperparámetros de nuestro script a través del Command-Line Interface (CLI)
- Puede requerir la instalación de una dependencia adicional

  pip install "pytorch-lightning[extra]"

# Parámetros a través de CLI

- Uso básico

```python
# main.py
from lightning.pytorch.cli import LightningCLI

# simple demo classes for your convenience
from lightning.pytorch.demos.boring_classes import DemoModel, BoringDataModule


def cli_main():
    cli = LightningCLI(DemoModel, BoringDataModule)
    # note: don't call fit!!


if __name__ == "__main__":
    cli_main()
    # note: it is good practice to implement the CLI in a function and call it in the main if block
```

Referencia: https://lightning.ai/docs/pytorch/stable/cli/lightning_cli.html

# Parámetros a través de CLI

- Uso básico

```
usage: main.py [-h] [-c CONFIG] [--print_config [={comments,skip_null,skip_default}+]]
       {fit,validate,test,predict} ...

pytorch-lightning trainer command line tool

optional arguments:
-h, --help            Show this help message and exit.
-c CONFIG, --config CONFIG
                      Path to a configuration file in json or yaml format.
--print_config [={comments,skip_null,skip_default}+]
                      Print configuration and exit.

subcommands:
For more details of each subcommand add it as argument followed by --help.

{fit,validate,test,predict}
    fit               Runs the full optimization routine.
    validate          Perform one evaluation epoch over the validation set.
    test              Perform one evaluation epoch over the test set.
    predict           Run inference on your data.
```

Referencia: https://lightning.ai/docs/pytorch/stable/cli/lightning_cli.html

# Parámetros a través de CLI

- Uso básico

```
$ python main.py fit --help

usage: main.py [options] fit [-h] [-c CONFIG]
                             [--seed_everything SEED_EVERYTHING] [--trainer CONFIG]
                             ...
                             [--ckpt_path CKPT_PATH]
    --trainer.logger LOGGER

optional arguments:
<class '__main__.DemoModel'>:
    --model.out_dim OUT_DIM
                             (type: int, default: 10)
    --model.learning_rate LEARNING_RATE
                             (type: float, default: 0.02)
<class 'lightning.pytorch.demos.boring_classes.BoringDataModule'>:
--data CONFIG         Path to a configuration file.
--data.data_dir DATA_DIR
                        (type: str, default: ./)
```

Referencia: https://lightning.ai/docs/pytorch/stable/cli/lightning_cli.html

# Parámetros a través de CLI

- Uso básico

```
# change the learning_rate
python main.py fit --model.learning_rate 0.1

# change the output dimensions also
python main.py fit --model.out_dim 10 --model.learning_rate 0.1

# change trainer and data arguments too
python main.py fit --model.out_dim 2 --model.learning_rate 0.1 --data.data_dir '~/' --trainer.logger
False
```

Referencia: https://lightning.ai/docs/pytorch/stable/cli/lightning_cli.html

# Profiling en Lightning

- El profiling de nuestro modelo permite conocer sus cuellos de botella
- Hay dos perfiles de profiling disponibles en Pytorch:
  - “simple”
  - “advanced”

```
trainer = Trainer(profiler="simple")
```

Referencia: https://lightning.ai/docs/pytorch/stable/tuning/profiler_basic.html

# Profiling en Lightning: simple

- Hace una medición de la duración de los métodos clave en LightningModule, DataModule y Callback
- Al final de la ejecución de fit() imprime un informe como el siguiente

```
FIT Profiler Report

---------------------------------------------------------------------------------------------
|  Action                                                |  Mean duration (s)     |  Total time (s) |
---------------------------------------------------------------------------------------------
|  [LightningModule]BoringModel.prepare_data             |  10.0001               |  20.00          |
|  run_training_epoch                                    |  6.1558                |  6.1558         |
|  run_training_batch                                    |  0.0022506             |  0.015754       |
|  [LightningModule]BoringModel.optimizer_step           |  0.0017477             |  0.012234       |
|  [LightningModule]BoringModel.val_dataloader           |  0.00024388            |  0.00024388     |
|  on_train_batch_start                                  |  0.00014637            |  0.0010246      |
|  [LightningModule]BoringModel.teardown                 |  2.15e-06              |  2.15e-06       |
|  [LightningModule]BoringModel.on_train_start           |  1.644e-06             |  1.644e-06      |
|  [LightningModule]BoringModel.on_train_end             |  1.516e-06             |  1.516e-06      |
|  [LightningModule]BoringModel.on_fit_end               |  1.426e-06             |  1.426e-06      |
|  [LightningModule]BoringModel.setup                    |  1.403e-06             |  1.403e-06      |
|  [LightningModule]BoringModel.on_fit_start             |  1.226e-06             |  1.226e-06      |
---------------------------------------------------------------------------------------------
```

# Profiling en Lightning: advanced

- Hace un profiling de grano más fino. Llegando a niveles más profundos del grafo de llamadas
- Al final de la ejecución de fit() imprime un informe como el siguiente

```
Profiler Report

Profile stats for: get_train_batch
        4869394 function calls (4863767 primitive calls) in 18.893 seconds
Ordered by: cumulative time
List reduced from 76 to 10 due to restriction <10>
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
3752/1876    0.011    0.000   18.887    0.010 {built-in method builtins.next}
    1876     0.008    0.000   18.877    0.010 dataloader.py:344(__next__)
    1876     0.074    0.000   18.869    0.010 dataloader.py:383(_next_data)
    1875     0.012    0.000   18.721    0.010 fetch.py:42(fetch)
    1875     0.084    0.000   18.290    0.010 fetch.py:44(<listcomp>)
    60000    1.759    0.000   18.206    0.000 mnist.py:80(__getitem__)
    60000    0.267    0.000   13.022    0.000 transforms.py:68(__call__)
    60000    0.182    0.000    7.020    0.000 transforms.py:93(__call__)
    60000    1.651    0.000    6.839    0.000 functional.py:42(to_tensor)
    60000    0.260    0.000    5.734    0.000 transforms.py:167(__call__)
```

# Profiling en Lightning: aceleradores

- Si queremos monitorizar lo que pasa dentro de cada acelerador podemos hacerlo usando el callback *DeviceStatsMonitor*

```
from lightning.pytorch.callbacks import DeviceStatsMonitor
trainer = Trainer(callbacks=[DeviceStatsMonitor()])
```

# Profiling en Lightning

- Podemos usar Lightning para visualizar diferentes métricas que se van generando durante el entrenamiento

  ejs: Accuracy, Loss

```
class LitModel(pl.LightningModule):
    def training_step(self, batch, batch_idx):
        value = ...
        self.log("some_value", value)
```

- También podemos registrar varias métricas al mismo tiempo

```
values = {"loss": loss, "acc": acc, "metric_n": metric_n}
self.log_dict(values)
```

Fuente: https://lightning.ai/docs/pytorch/stable/visualize/logging_basic.html

# Profiling en Lightning

- Visualización en línea durante la ejecución:

```
self.log(..., prog_bar=True)
```

- O en tensorboard

Fuente: https://lightning.ai/docs/pytorch/stable/visualize/logging_basic.html

# PytorchProfiler

- También podemos usar el PytorchProfiler para obtener un profiling más detallado
  - En entornos distribuidos guarda una salida por Rank

```
from lightning.pytorch.profilers import PyTorchProfiler

profiler = PyTorchProfiler(filename="perf-logs")

trainer = Trainer(profiler=profiler)
```

https://lightning.ai/docs/pytorch/stable/tuning/profiler_intermediate.html

```
Profiler Report: rank 0

Profile stats for: training_step
-------------------  --------------  --------------  --------------  -----
---------  --------------
Name                 Self CPU total %  Self CPU total   CPU total %      CPU
total        CPU time avg
-------------------  --------------  --------------  --------------  -----
---------  --------------
t                          62.10%          1.044ms          62.77%
1.055ms          1.055ms
addmm                      32.32%          543.135us        32.69%
549.362us        549.362us
mse_loss                   1.35%           22.657us         3.58%
60.105us         60.105us
mean                       0.22%           3.694us          2.05%
34.523us         34.523us
div_                       0.64%           10.756us         1.90%
32.001us         16.000us
ones_like                  0.21%           3.461us          0.81%
13.669us         13.669us
sum_out                    0.45%           7.638us          0.74%
12.432us         12.432us
transpose                  0.23%           3.786us          0.68%
11.393us         11.393us
as_strided                 0.60%           10.060us         0.60%
10.060us         3.353us
to                         0.18%           3.059us          0.44%
```

# PytorchProfiler

Sigue las instrucciones del README:
https://github.com/diegoandradecanosa/CFR24/tree/main/pytorch_dist/lightning/003

Basado en: https://lightning.ai/docs/pytorch/stable/tuning/profiler_intermediate.html

# Precisión numérica en Lightning

- Los parámetros del modelo son por defecto números en punto  flotante de simple precisión (32 bits/valor)
- Se puede intentar aumentar o reducir la precisión de la representación numérica de estos números (-> 16 bits (half)) o (->64 bits (double))
- La reducción a 16 bits es el caso más habitual
  - Reduce el consumo de memoria del modelo
  - Habilita el uso de tensor cores en aquellas GPUs equipadas con ellos (ej. Ampere, ft3, o Hopper)
    - Realmente se usa precisión *16-mixed*
- Activación

```
Trainer(accelerator="gpu", devices=1, precision=16)
```

*https://lightning.ai/docs/pytorch/stable/common/precision_intermediate.html*

# Precisión numérica en Lightning

## Precision support by accelerator

| Precision | CPU | GPU | TPU | IPU |
|---|---|---|---|---|
| 16 Mixed | No | Yes | No | Yes |
| BFloat16 Mixed | Yes | Yes | Yes | No |
| 32 True | Yes | Yes | Yes | Yes |
| 64 True | Yes | Yes | No | No |

Precision with Accelerators

# Contenidos avanzados recomendados

- TRAIN 1 TRILLION+ PARAMETER MODELS: https://lightning.ai/docs/pytorch/stable/advanced/model_parallel.html
- Integrate your own cluster (definición de un clúster ft3 que facilite la integración con parámetros ad-hoc): https://lightning.ai/docs/pytorch/stable/clouds/cluster_expert.html
- ADD A NEW ACCELERATOR OR STRATEGY: https://lightning.ai/docs/pytorch/stable/levels/expert_level_27.html
- EXTEND THE LIGHTNING CLI: https://lightning.ai/docs/pytorch/stable/levels/expert_level_23.html
- LLM Learning Lab (junio 2023): https://lightning.ai/pages/llm-learning-lab/

# Lightning Fabric

- Fabric es una herramienta del ecosistema Lightning que permite escalar modelos de Pytorch minimizando los cambios necesarios
  - CPU -> GPU,TPU,multi-GPU o distribuido
  - Implementaciones de estrategias populares (DDP, FSDP, DeepSpeed)
  - Pensado y diseñado para LARGE models (varios billones de parámetros)
- Se plantea como un punto intermedio entre Lightning (menos control y más esfuerzo) y Pytorch
- Introducida en el ecosistema en Marzo 2023 (https://lightning.ai/blog/introducing-lightning-2-0/)
- https://lightning.ai/docs/fabric/stable/

# Lightning Fabric

- Fabric frente a Lightning
  - Implementación rápida: Requiere pocos cambios en el código base escrito en Pytorch
  - Mayor flexibilidad en cuanto a escritura del entrenamiento, inferencia, etc...
  - Maximiza el control sobre el código
    - Más fácil de depurar que Lightning

# Lightning Fabric: Ejemplo

```
  import torch
  from lightning.pytorch.demos import WikiText2, Transformer
+ import lightning as L

- device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
+ fabric = L.Fabric(accelerator="cuda", devices=8, strategy="ddp")
+ fabric.launch()

  dataset = WikiText2()
  dataloader = torch.utils.data.DataLoader(dataset)
  model = Transformer(vocab_size=dataset.vocab_size)
  optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

- model = model.to(device)
+ model, optimizer = fabric.setup(model, optimizer)
+ dataloader = fabric.setup_dataloaders(dataloader)

  model.train()
  for epoch in range(20):
      for batch in dataloader:
          input, target = batch
-         input, target = input.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(input, target)
          loss = torch.nn.functional.nll_loss(output, target.view(-1))
-         loss.backward()
+         fabric.backward(loss)
          optimizer.step()
```

# Características

- También podemos hacer uso de los LightningModule con Fabric
- Y definir hooks propios

```python
import lightning as L


class LitModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.model = ...

    def training_step(self, batch, batch_idx):
        # Main forward, loss computation, and metrics goes here
        x, y = batch
        y_hat = self.model(x)
        loss = self.loss_fn(y, y_hat)
        acc = self.accuracy(y, y_hat)
        ...
        return loss

    def configure_optimizers(self):
        # Return one or several optimizers
        return torch.optim.Adam(self.parameters(), ...)

    def train_dataloader(self):
        # Return your dataloader for training
        return DataLoader(...)

    def on_train_start(self):
        # Do something at the beginning of training
        ...

    def any_hook_you_like(self, *args, **kwargs):
        ...
```

# Características

```python
import lightning as L

fabric = L.Fabric(...)

# Instantiate the LightningModule
model = LitModel()

# Get the optimizer(s) from the LightningModule
optimizer = model.configure_optimizers()

# Get the training data loader from the LightningModule
train_dataloader = model.train_dataloader()

# Set up objects
model, optimizer = fabric.setup(model, optimizer)
train_dataloader = fabric.setup_dataloaders(train_dataloader)

# Call the hooks at the right time
model.on_train_start()

model.train()
for epoch in range(num_epochs):
    for i, batch in enumerate(dataloader):
        optimizer.zero_grad()
        loss = model.training_step(batch, i)
        fabric.backward(loss)
        optimizer.step()

        # Control when hooks are called
        if condition:
            model.any_hook_you_like()
```

- Podemos instanciar el módulo y llamar a sus hooks donde creamos conveniente
- Ahora los modules pueden ser intercambiables
  - Si definen los mismos hooks

# Características

- También podemos definir y usar callbacks

```
from lightning.fabric import Fabric

# The code of a callback can live anywhere, away from the training loop
from my_callbacks import MyCallback

# Add one or several callbacks:
fabric = Fabric(callbacks=[MyCallback()])

...

for iteration, batch in enumerate(train_dataloader):
    ...
    fabric.backward(loss)
    optimizer.step()

    # Let a callback add some arbitrary processing at the appropriate place
    # Give the callback access to some varibles
    fabric.call("on_train_batch_end", loss=loss, output=...)
```


Raw PyTorch
Fabric Accelerators
+ Distributed Strategies
+ Managed Logging/ Checkpointing
+ LightningModule
+ Your Own Trainer
PyTorch Lightning
PyTorch
Full control, more boilerplate
Lightning Fabric
PyTorch Lightning
Fully managed, no boilerplate

citic

# Lightning Fabric

Sigue las instrucciones del README:
https://github.com/diegoandradecanosa/CFR24/tree/main/pytorch_dist/fabric

Fuente: https://lightning.ai/docs/fabric/stable/fundamentals/convert.html

# Herramientas entrenamiento distribuido: Ray

- Ray: https://github.com/ray-project/ray
  - Framework unificado para escalar aplicaciones de IA y Python
  - Componentes principales:
    - Ray AIR
      - Datasets: Preprocesado de datos distribuido
      - Train: Entrenamiento distribuido
      - Tune: Ajuste de hiperparámetros distribuido
      - Rlib: *Reinforcement Learning* escalable
      - Serve: *Serving* escalable y programable. Se usa para construir APIs de inferencia online.
    - Ray Core
      - Tasks: Funciones sin estado ejecutadas en un clúster
      - Actors: Procesos trabajadores sin estado creados en un clúster
      - Objects: Valores inmutables accesibles a través de un clúster

# Herramientas entrenamiento distribuido: NebulGym

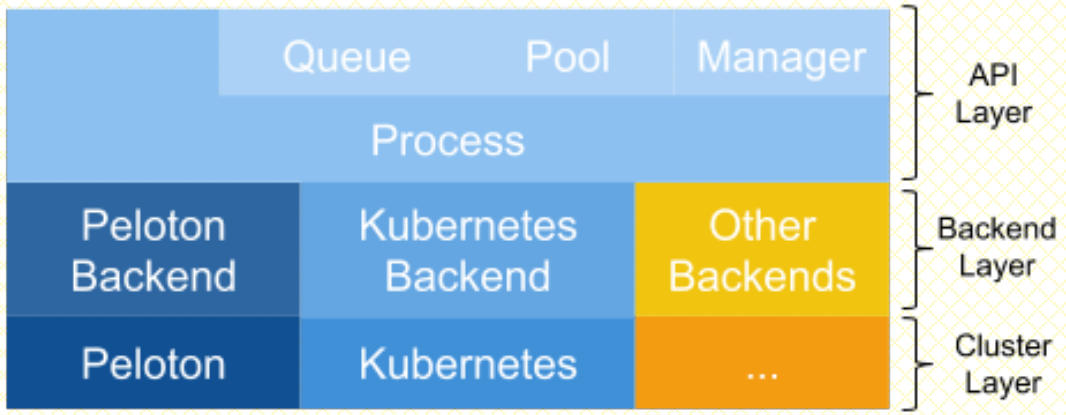
- NebulGym: https://github.com/nebuly-ai/nebulgym
  - Proporciona entrenamiento distribuido añadiendo decoradores al código
  - Pretende ser *framework-agnostic*
    - Por ahora solo soporta Pytorch

# Herramientas entrenamiento distribuido: Mahout

- Apache Mahout: https://mahout.apache.org
- Dos componentes:
  - Librería de álgebra lineal distribuida
  - Lenguaje ScalaDSL idóneo para paralelizar rápidamente algoritmos de ML
- Se combina con Apache Spark para entrenamientos distribuidos
  - Apache Spark es un framework de Big Data

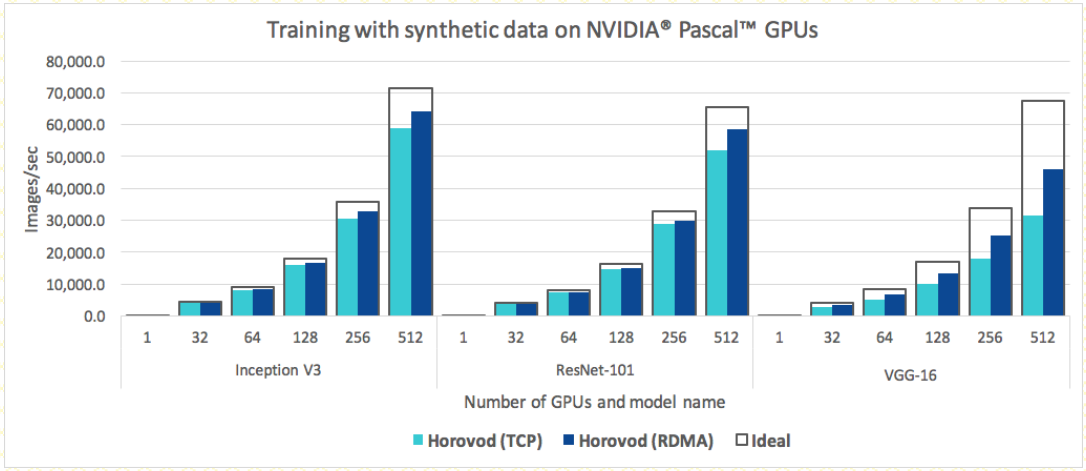
# Herramientas entrenamiento distribuido: Fiber

- Fiber: https://uber.github.io/fiber/
  - Es una herramienta para programación en entornos distribuidos Python
  - La paralelización de entrenamiento es un uso más de esta tecnología



# Herramientas entrenamiento distribuido: Horovod

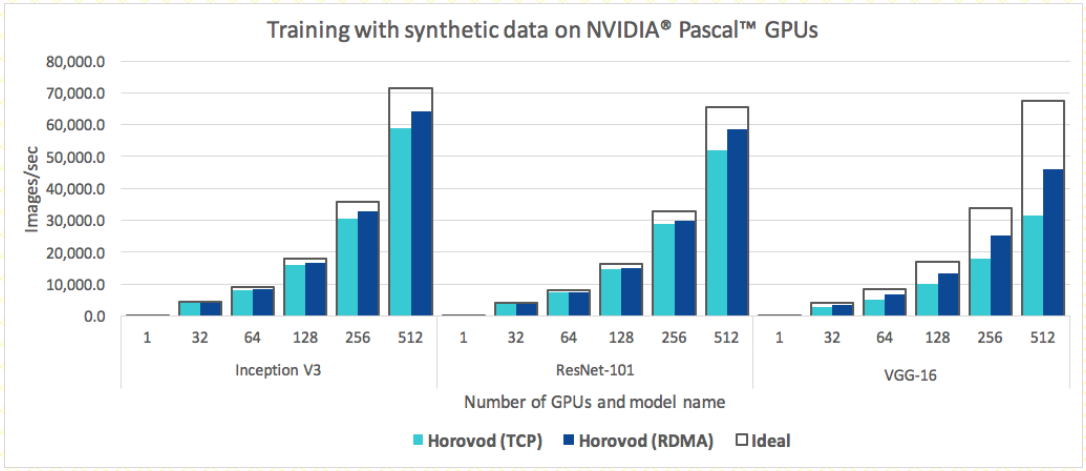
- Horovold: https://github.com/horovod/horovod
  - Herramienta para escalar *fácilmente* el entrenamiento de modelos en entornos distribuidos
  - Inicialmente un desarrollo interno de Uber
  - El objetivo general es escalar un entrenamiento 1-GPU de un modelo y escalarlo a N-GPUs
    - Requiriendo poco esfuerzo por parte del desarrollador
    - Desplegando una implementación eficiente




citic

# Herramientas entrenamiento distribuido: Horovod

- Horovold: https://github.com/horovod/horovod
  - Herramienta para escalar *fácilmente* el entrenamiento de modelos en entornos distribuidos
  - Inicialmente un desarrollo interno de Uber
  - El objetivo general es escalar un entrenamiento 1-GPU de un modelo y escalarlo a N-GPUs
    - Requiriendo poco esfuerzo por parte del desarrollador
    - Desplegando una implementación eficiente




citic

# Horovod

- Tiene soporte para los principales frameworks de ML
  - Tensorflow
    - Keras
  - Pytorch
  - Apache MxNet
- Basado en MPI
  - Usa términos como *size, rank, local rank*
  - El código empieza con un hvd.init() (como MPI_init())

  Fuente: https://horovod.readthedocs.io/en/stable/

# Horovod: Uso básico

Ejemplo TF2 MNIST:

https://github.com/horovod/horovod/blob/master/examples/tensorflow2/tensorflow2_keras_mnist.py

Ejemplos básicos:

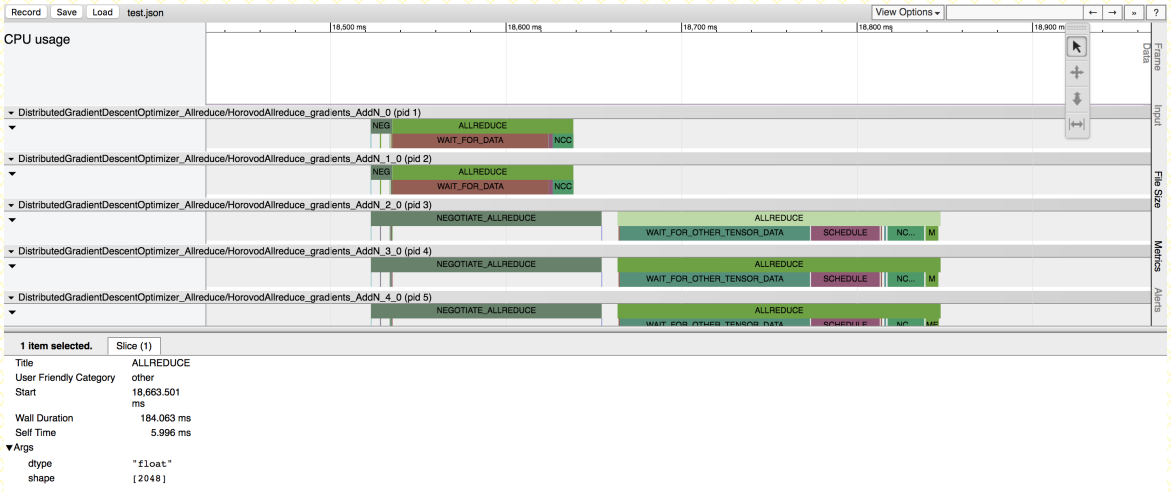
https://github.com/horovod/horovod/tree/master/examples

# Horovod: Ejecución

- Usaremos el script horovodrun
- Ejemplo (4 nodos con 4 GPUs cada uno):

```
$ horovodrun -np 16 -H server1:4,server2:4,server3:4,server4:4 python train.py
```

- Horovod está diseñado para solapar comunicaciones con computación




citic

# Accelerate

- Es una librería para la aceleración de códigos Pytorch
  - Desarrollada por huggingface
  - Utilizada en muchos de sus modelos
  - Requiere pocos cambios en el código
  - Permite escalar el código para usar
    - Varios nodos
    - Varias GPUs
  - Implementado con
    - torch_xla
    - torch.distributed
  - Utiliza otros frameworks aceleradores intermedios como DeeSpeed

Fuentes: https://huggingface.co/docs/accelerate/index
https://github.com/huggingface/accelerate

# Accelerate: Codificación

```
+ from accelerate import Accelerator
+ accelerator = Accelerator()

+ model, optimizer, training_dataloader, scheduler = accelerator.prepare(
+     model, optimizer, training_dataloader, scheduler
+ )

  for batch in training_dataloader:
      optimizer.zero_grad()
      inputs, targets = batch
      inputs = inputs.to(device)
      targets = targets.to(device)
      outputs = model(inputs)
      loss = loss_function(outputs, targets)
+     accelerator.backward(loss)
      optimizer.step()
      scheduler.step()
```

# Accelerate: Lanzamiento

- Involucra a varios comandos disponibles a través del script *accelerate*

  *accelerate config*

  *accelerate launch <<script.py>>*

# Accelerate: Configuración (alternativa)

```yaml
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: MULTI_GPU
fsdp_config: {}
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: fp16
num_machines: 1
num_processes: 2
use_cpu: false
```

```
accelerate launch --config_file {path/to/config/my_config_file.yaml} {script_name.py} {--arg1} {--arg2} ...
```

# Accelerate: con notebooks

- El comando *accelerate config* se tiene que ejecutar en el terminal
- Reiniciar jupyter si ya está iniciado
- Hay que usar un launcher especial llamado *notebook_launcher*

```
from accelerate import notebook_launcher
args = ("fp16", 42, 64)
notebook_launcher(training_loop, args, num_processes=2)
```

Fuente: Launching Multi-Node Training from a Jupyter Environment (huggingface.co)

# Herramientas entrenamiento distribuido: hybrid parallelism

- HybridBackend: https://github.com/alibaba/HybridBackend
  - Se centra en el dominio de los sistemas recomendadores
  - Carga eficiente de datos categóricos
  - Orquestación eficiente de las capas de *embedding*
  - Escalado eficiente de entrenamiento y evaluación
  - Fácil de usar con  flujos de IA existentes
  - Usable como:
    - Paquete pip
    - Contenedor Docker

# Herramientas entrenamiento distribuido: Otros dominios

- Modin: Acelera códigos Pandas cambiando una sola línea de código
- Dask: Centrado en analítica de datos, proporciona escalado del rendimiento usando más recursos
- Rapids: Centrado en ciencia de datos. Desarrollo propio de Nvidia para permitir el uso de varias GPUs de forma eficiente
- H2O: Centrado en Big Data

# Referencias


- https://lightning.ai/docs/pytorch/stable/starter/introduction.html