HUGGING FACE ACCELERATE: MAKING DEVICE-AGNOSTIC ML TRAINING AND INFERENCE EASY AT SCALE

WHO AM I?

- Zachary Mueller
- Technical Lead for the 🤗 Accelerate project
- Maintain the `transformers` Trainer
- API design geek

WHAT IS 🤗 ACCELERATE?

- A training framework
- An inference framework
- A command-line interface

A TRAINING FRAMEWORK

- Powered by PyTorch
- Change a few lines of code, gain device *and* hardware-agnostic capabilities
- Low-code, with minimal magic aimed at easy hackability and use without high-level abstractions
- We handle the intracies so you don’t have to

A TRAINING FRAMEWORK

- Support for any hardware-accelerator on the market:
  - CPU, GPU, TPU, XPU, NPU, MLU
- Automatic mixed-precision training *safely* in whatever fashion you may choose:
  - FP16, BF16, FP8 (through either `TransformerEngine` or `MS-AMP`)
- Automatic and efficient gradient accumulation
- Support for quantization through `bitsandbytes`
- Support your favorite experiment trackers (`aim`, `clearml`, `comet_ml`, `dvc-lite`, `ml-flow`, `tensorboard`, `wandb`)
- Easy to configure plugin or YAML-level API for setting up advanced frameworks like `FSDP`, `DeepSpeed`, and `Megatron-LM`

LOW-CODE

- Biggest friction with “wrapper” libraries is control of your code
- By being minimally intrusive, your code just “works” while still giving you complete control

```diff
  import torch
  import torch.nn.functional as F
  from datasets import load_dataset
+ from accelerate import Accelerator

+ accelerator = Accelerator()
- device = 'cpu'
+ device = accelerator.device

  model = torch.nn.Transformer().to(device)
  optimizer = torch.optim.Adam(model.parameters())
  dataset = load_dataset('my_dataset')
  data = torch.utils.data.DataLoader(dataset, shuffle=True)

+ model, optimizer, dataloader = accelerator.prepare(model, optimizer, dataloader

  model.train()
  for epoch in range(10):
      for source, targets in dataloader:
          source, targets = source.to(device), targets.to(device)
          optimizer.zero_grad()
          output = model(source)
          loss = F.cross_entropy(output, targets)
-         loss.backward()
+         accelerator.backward(loss)
          optimizer.step()
```

EASY TO INTEGRATE

- Due to the low-code nature, it's trivial to integrate into existing PyTorch frameworks:
  1. Create an `Accelerator`

```
  import torch
  import torch.nn.functional as F
  from datasets import load_dataset
+ from accelerate import Accelerator

+ accelerator = Accelerator()
  device = 'cpu'

  model = torch.nn.Transformer().to(device)
  optimizer = torch.optim.Adam(model.parameters())
  dataset = load_dataset('my_dataset')
  data = torch.utils.data.DataLoader(dataset, shuffle=True)

  model.train()
  for epoch in range(10):
      for source, targets in dataloader:
          source, targets = source.to(device), targets.to(device)
          optimizer.zero_grad()
          output = model(source)
          loss = F.cross_entropy(output, targets)
          loss.backward()
          optimizer.step()
```

EASY TO INTEGRATE

- Due to the low-code nature, it's trivial to integrate into existing PyTorch frameworks:

  2. Wrap your PyTorch objects with `accelerator.prepare` and remove device-placements

```
  import torch
  import torch.nn.functional as F
  from datasets import load_dataset
  from accelerate import Accelerator

  accelerator = Accelerator()
- device = 'cpu'

  model = torch.nn.Transformer().to(device)
  optimizer = torch.optim.Adam(model.parameters())
  dataset = load_dataset('my_dataset')
  data = torch.utils.data.DataLoader(dataset, shuffle=True)

+ model, optimizer, dataloader = accelerator.prepare(model, optimizer, dataloader

  model.train()
  for epoch in range(10):
      for source, targets in dataloader:
          source, targets = source.to(device), targets.to(device)
          optimizer.zero_grad()
          output = model(source)
          loss = F.cross_entropy(output, targets)
          loss.backward()
          optimizer.step()
```

EASY TO INTEGRATE

- Due to the low-code nature, it's trivial to integrate into existing PyTorch frameworks:

  3. Use `accelerator.backward` for the backward pass

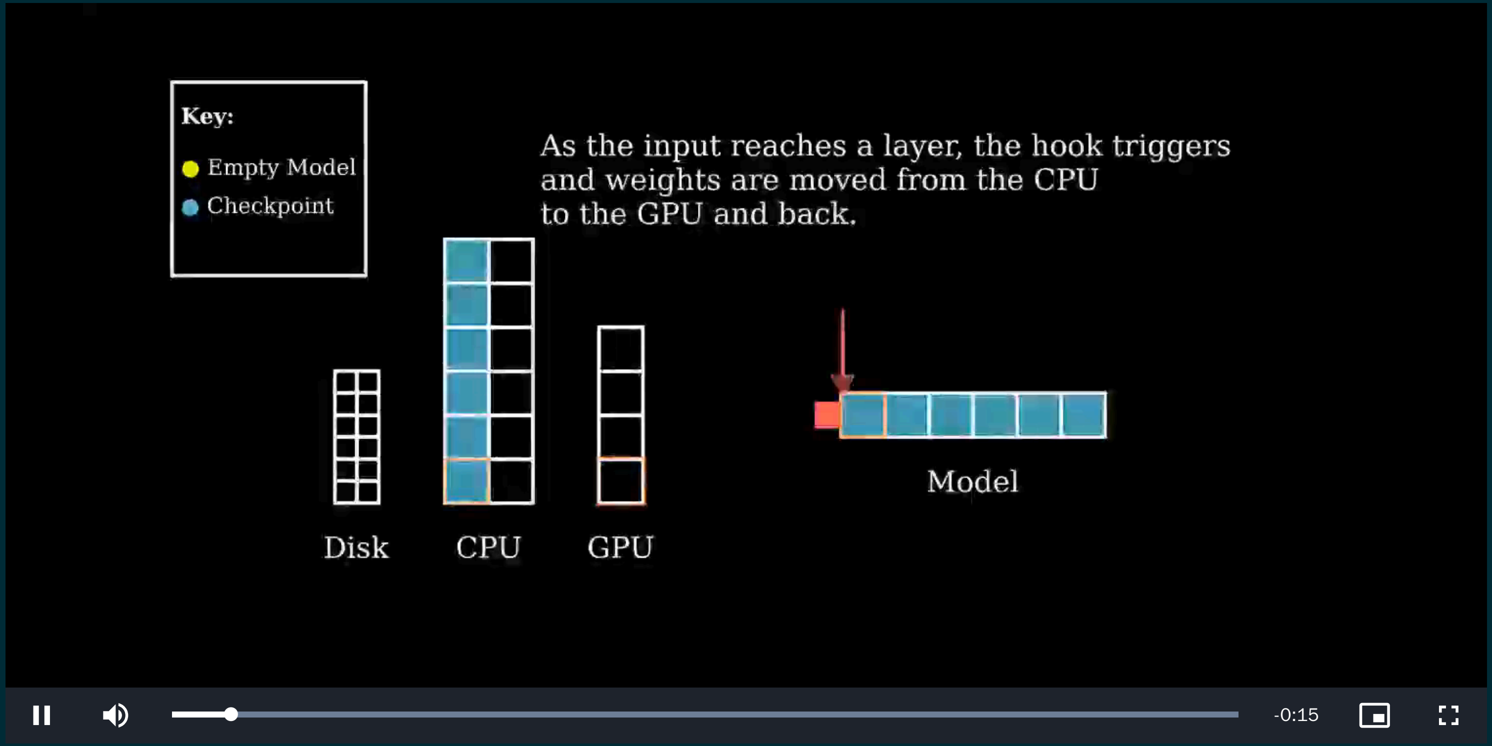
```
  import torch
  import torch.nn.functional as F
  from datasets import load_dataset
  from accelerate import Accelerator

  accelerator = Accelerator()

  model = torch.nn.Transformer().to(device)
  optimizer = torch.optim.Adam(model.parameters())
  dataset = load_dataset('my_dataset')
  data = torch.utils.data.DataLoader(dataset, shuffle=True)

  model, optimizer, dataloader = accelerator.prepare(model, optimizer, dataloader

  model.train()
  for epoch in range(10):
      for source, targets in dataloader:
          source, targets = source.to(device), targets.to(device)
          optimizer.zero_grad()
          output = model(source)
          loss = F.cross_entropy(output, targets)
-         loss.backward()
+         accelerator.backward(loss)
          optimizer.step()
```

BUT WHAT ABOUT INFERENCE?

- 🤗 Accelerate is not just for training, and has helped make the GPU-Poor take control of the narrative
- Using tools like Big Model Inference, users with *tiny* compute can run large models locally
- Started with the boom of stable diffusion, and now has scaled to having the ability to run huge LLMs locally with a single graphics card

HOW DOES IT WORK?

- PyTorch introduced `device="meta"`
- 🤗 Accelerate introduced `device_map="auto"`



A CLI INTERFACE

- `accelerate config`
  - Configure the environment
- `accelerate launch`
  - How to run your script

LAUNCHING DISTRIBUTED TRAINING IS HARD

```
python script.py
```

vs.

```
torchrun --nnodes=1 --nproc_per_node=2 script.py
```

vs.

```
deepspeed --num_gpus=2 script.py
```

How can we make this better?

accelerate launch

```
accelerate launch script.py
```

```
accelerate launch --multi_gpu --num_processes 2 script.py
```

```
accelerate launch \
  --multi_gpu \
  --use_deepspeed \
  --num_processes 2 \
  script.py
```

`accelerate config`

- Rely on `config.yaml` files
- Choose to either running `accelerate config` or write your own:

ddp_config.yaml

```yaml
compute_environment: LOCAL_MACHINE
distributed_type: MULTI_GPU
main_training_function: main
mixed_precision: bf16
num_machines: 1
num_processes: 8
```

fsdp_config.yaml

```yaml
compute_environment: LOCAL_MACHINE
distributed_type: FSDP
fsdp_config:
  fsdp_auto_wrap_policy: TRANSFORMER_BASED_WRA
  fsdp_backward_prefetch: BACKWARD_PRE
  fsdp_cpu_ram_efficient_loading: true
  fsdp_forward_prefetch: false
  fsdp_offload_params: false
  fsdp_sharding_strategy: FULL_SHARD
  fsdp_state_dict_type: SHARDED_STATE_DICT
  fsdp_sync_module_states: true
  fsdp_use_orig_params: false
main_training_function: main
mixed_precision: bf16
num_machines: 1
num_processes: 8
```

NOW THAT YOU'RE UP TO SPEED, WHAT'S NEW?

WE'VE HAD A BUSY LAST YEAR, AND SO HAS THE ML COMMUNITY!

NEW TRAINING TECHNIQUES

- Quantization has taken the field by storm
- New ideas such as FSDP + QLoRA to train huge models on tiny compute!
- New precision backends as we train natively on smaller precision
- Optimizing futher how much we can push on a single machine through efficient RAM and timing techniques

LARGER COMPUTE LANDSCAPE

- As we search for alternatives to NVIDIA, new compilers rise:
  - XPU (Intel)
  - NPU (Intel)
  - MLU (Cambricon)

All of which are supported by 🤗 Accelerate

LOWER ABSTRACTIONS

- While the `Accelerator` was great, needed better abstractions focused on controlling behaviors
- Introduced the `PartialState`

```python
from accelerate import PartialState

if PartialState().is_main_process:
  # Run on only 1 device

with PartialState().main_process_first:
  # Useful for dataset processing

# Device-agnostic without the bulk of the `Accelerator`
device = PartialState().device
```

FASTER AND BETTER INFERENCE ALTERNATIVES

- `PiPPy` gives us efficient pipeline-parallelism in distributed environments to increase throughput while keeping a simple torch-bound API
- Rather than having to wait for each GPU, every GPU can be busy in parallel

```python
import torch
from transformers import AutoModelForSequenceClassification

from accelerate import PartialState, prepare_pippy

model = AutoModelForSequenceClassification.from_pretrained("gpt2")
model.eval()

input = torch.randint(
    low=0,
    high=model.config.vocab_size,
    size=(2, 1024),  # bs x seq_len
    device="cpu",
)

model = prepare_pippy(model, split_points="auto", example_args=(input,

with torch.no_grad():
    output = model(input)
```

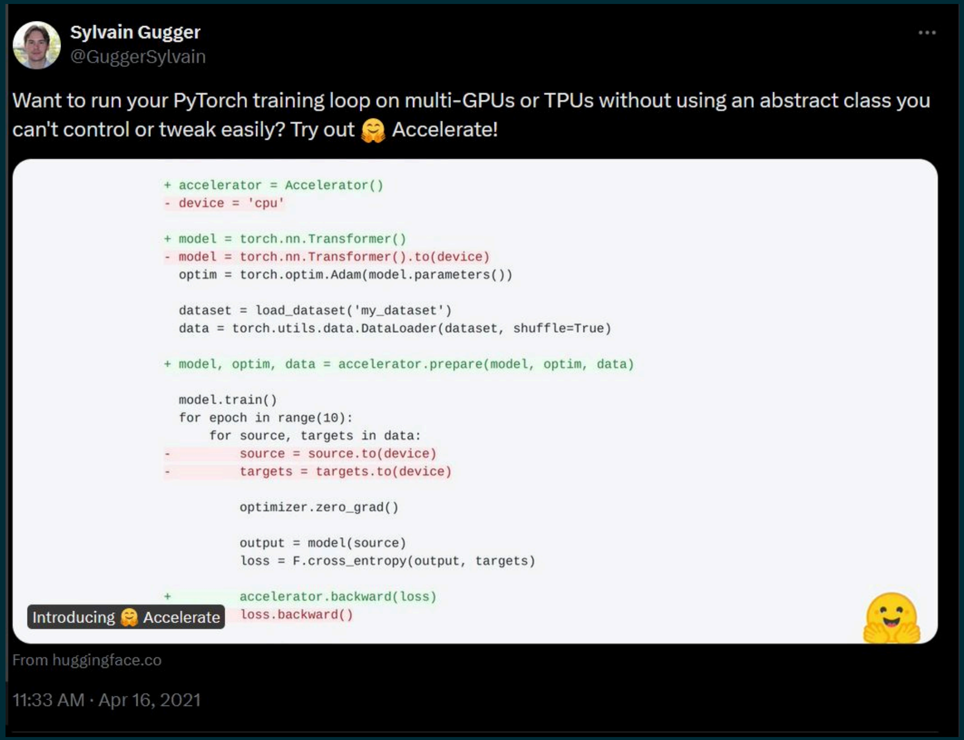
ADOPTION: ACCELERATE IN THE ECOSYSTEM

ACCELERATE IN THE ECOSYSTEM

- Many of the frameworks you use daily already rely on 🤗 Accelerate!
  - Nearly all of 🤗
  - `axolotl`
  - `fastai`
  - `FastChat`
  - `lucidrains`
  - `kornia`

ACCELERATE IN THE ECOSYSTEM

- Started as a way to isolate out distributed code on TPU and `DistributedDataParallelism`



ACCELERATE IN THE ECOSYSTEM

- Now is the backbone of some of the largest PyTorch training frameworks in the ecosystem

WHAT'S NEXT?

ELEVATING THE COMMUNITY

- Now that more advanced training techniques are reachable (FSDP, DeepSpeed, etc), we need to focus on educating the community on how to use it best
- Goes beyond how to use the `Trainer` or `Accelerator`, but how to use *what* where
- Keep Accelerate as a tool for the community to utilize when new techniques come out and play with, to push new ideas to scale quickly

1.0.0: SOON!

- Tried and battle-tested by over 7M users/month
- As we’ve been stable for over a year now, we’re near ready to release 1.0.0

THANKS FOR JOINING!

- 🤗 Accelerate documentation
- Launching distributed code
- Distributed code and Jupyter Notebooks
- Migrating to 🤗 Accelerate easily
- Big Model Inference tutorial
- DeepSpeed and 🤗 Accelerate
- Fully Sharded Data Parallelism and 🤗 Accelerate
- FSDP vs DeepSpeed In-Depth