PyTorch at Scale

Need for Speed and Efficiency

- Larger Models (self-supervised training)
    - XLS-R up to 2B parameters
    - BigSSL up to 8B parameters
- Larger Datasets (even supervised training)
    - Multilingual LibriSpeech = 50k hours
    - People's Speech = 30k hours
- Carbon footprint

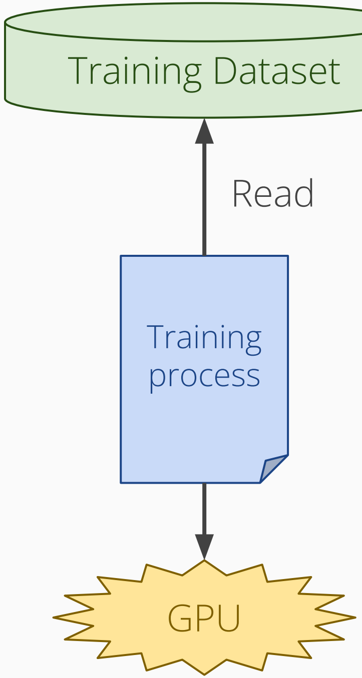
Outline

1. Scaling up
    a. Distributed training
    b. Elastic and fault-tolerant experiments
2. Efficient models
    a. Mixed precision / Quantization
    b. Model compilation
3. Deployment

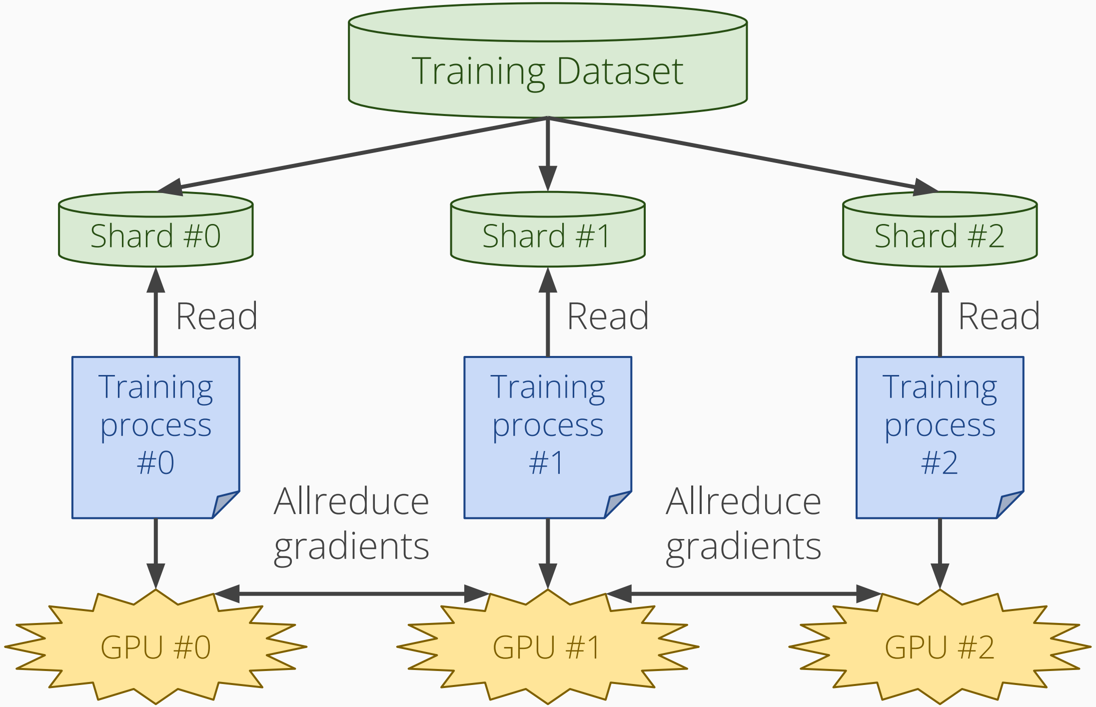
Outline

1. Scaling up
    a. Distributed training
    b. Elastic and fault-tolerant experiments
2. Efficient models
    a. Mixed precision / Quantization
    b. Model compilation
3. Deployment

Distributed Training: Parallel Data



Sequential data training



Parallel data training

Distributed Training: Parallel Data

pytorch_train_script.py

```python
torch.distributed.init_process_group("nccl")

# Use device count to compute local rank from global rank
local_rank = rank % torch.cuda.device_count()
model = Model().to(local_rank)
ddp_model = torch.nn.parallel.DistributedDataParallel(
    model, device_ids=[local_rank]
)

# Create data sampler pinned to global rank
sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=world_size, rank=rank,
)

# Optimize in the usual way…
```

```
torchrun
  --nnodes=2
  --nproc_per_node=8
  --rdzv_id=100
  --rdzv_backend=c10d
  --rdzv_endpoint=
     $MASTER_ADDR:29400
  pytorch_train_script.py
```

Distributed Training: Parallel Data

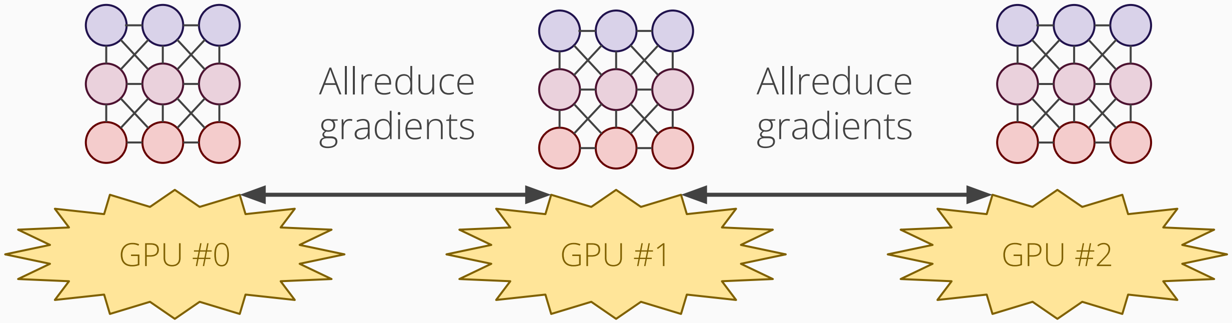
speechbrain_train_script.py

```python
# Initialize distributed communication protocols
speechbrain.utils.distributed.ddp_init_group(run_opts)

# Data preparation, to be run on only one process
speechbrain.utils.distributed.run_on_main(prepare_data_manifest)

# Automatically wraps model and creates distributed sampler
asr_brain.fit(epoch_counter=range(10), train_set=train_dataset)
```

```
torchrun --nproc_per_node=8 speechbrain_train_script.py
  hyperparams.yaml --distributed_launch --distributed_backend=nccl
```

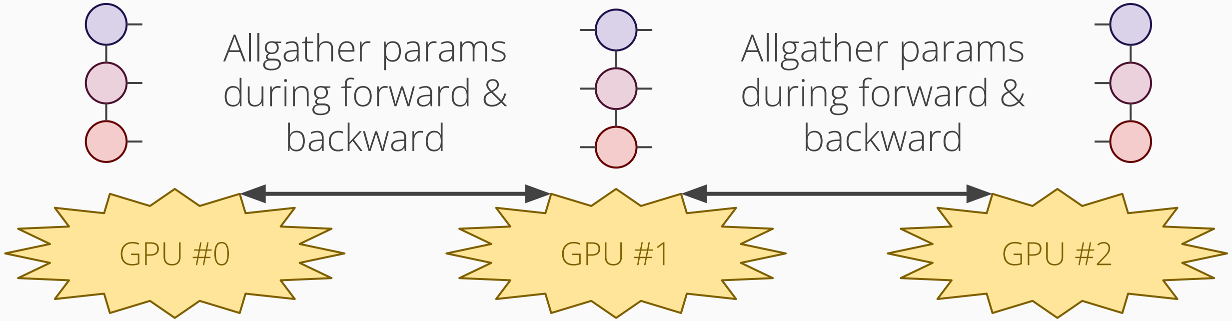
Distributed Training: Parallel Model

Parallel data:
replicate model
across all devices



* Less efficient
in terms of space

Parallel model:
distribute model
shards to devices

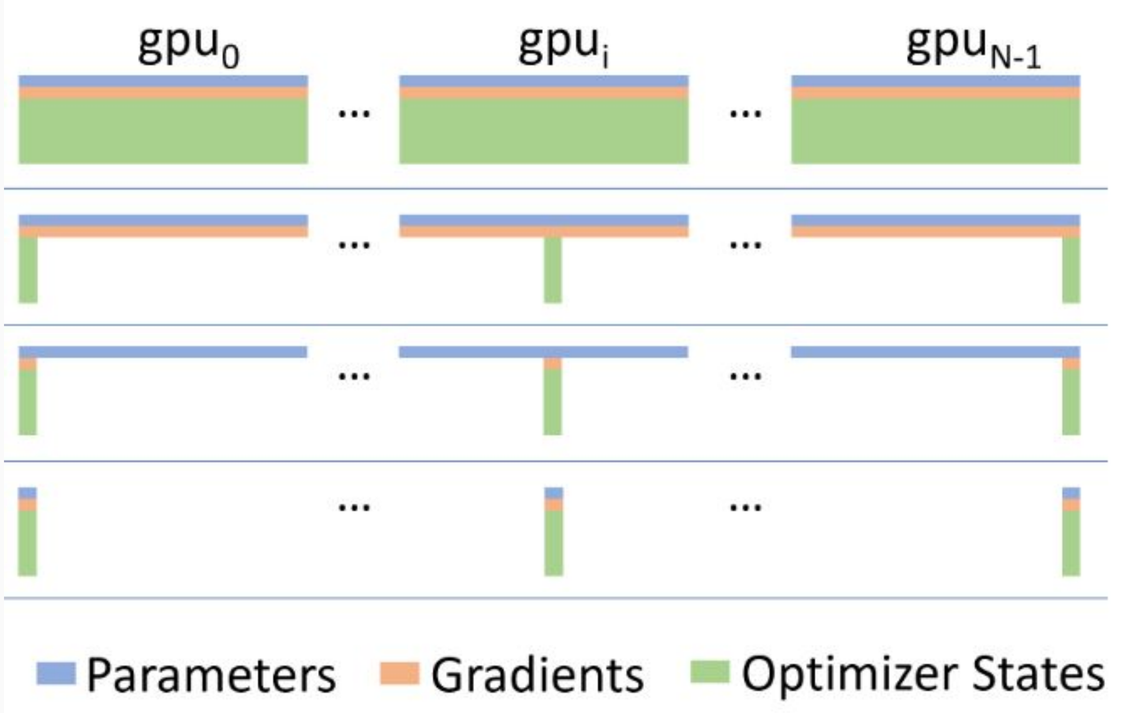


* Communication
overhead, but uses
memory efficiently

Distributed Training: Parallel Model

Origins of PyTorch Fully Sharded Data Parallel (FSDP):

- DeepSpeed
- FairScale



Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "ZeRO: Memory optimizations toward training trillion parameter models." In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-16. IEEE, 2020.

Distributed Training: Parallel Model

```python
# Wrap policy specifies which layers get sharded
custom_auto_wrap_policy = functools.partial(
    torch.distributed.fsdp.wrap.transformer_auto_wrap_policy,
    transformer_layer_cls={ConformerBlock},
)

# CPU offload allows training truly humongous models
sharded_model = torch.distributed.fsdp.FullyShardedDataParallel(
    single_device_model,
    auto_wrap_policy=custom_auto_wrap_policy,
    cpu_offload=torch.distributed.fsdp.CPUOffload(offload_params=True),
)
```

Elastic and Fault-tolerant Experiments

Torch Distributed Elastic, upstreamed version 1.9.0

```
torchrun
    --nnodes=MIN_SIZE:MAX_SIZE
    --nproc_per_node=TRAINERS_PER_NODE
    --max_restarts=NUM_ALLOWED_FAILURES_OR_MEMBERSHIP_CHANGES
    --rdzv_id=JOB_ID
    --rdzv_backend=c10d
    --rdzv_endpoint=HOST_NODE_ADDR
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

https://pytorch.org/docs/stable/elastic/quickstart.html

Outline

1. Scaling up
    a. Distributed training
    b. Elastic and fault-tolerant experiments
2. Efficient models
    a. Mixed precision / Quantization
    b. Model compilation
3. Deployment

Mixed Precision / Quantization

- Reduced memory and bandwidth
- Faster operations on supporting hardware
- Some operations effective at small precision

  e.g. linear, convolution, LSTM

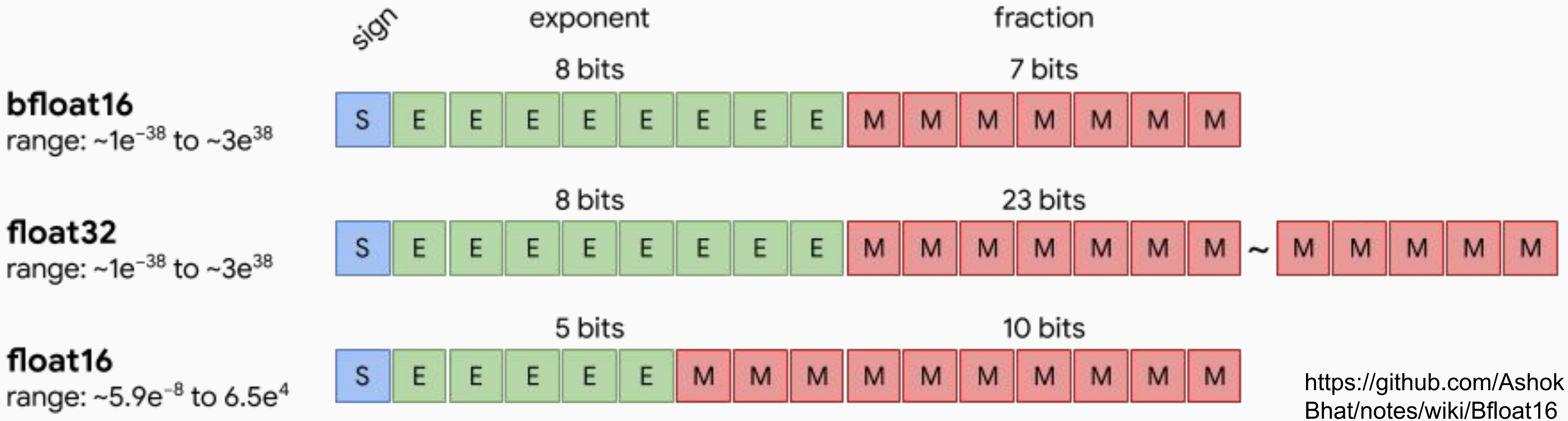
- Some operations require higher precision

  e.g. sigmoid, softmax, cross entropy

Automatic Mixed Precision

```python
scaler = torch.cuda.amp.GradScaler()

# Automatically converts between float16 and float32
with torch.cuda.amp.autocast(dtype=torch.float16):
    output = model(input)
    loss = loss_fn(output, target)

# Perform update operation with scaled gradient
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Mixed Precision: Data Types



https://github.com/AshokBhat/notes/wiki/Bfloat16

`float16` — less stable due to reduced dynamic range, requires loss scaling

`bfloat16` — supported natively by NVIDIA Ampere GPUs and Google TPUs, no scaling needed

Quantization (Aware Training)

- Easiest is post-training, but might lose performance

```
model_int8 = torch.quantization.quantize_dynamic(
    model_fp32,           # the trained, full-precision model
    {torch.nn.Linear},   # a set of layers to dynamically quantize
    dtype=torch.qint8,   # the target dtype for quantized weights
)
```

- Use quantization aware training to maintain scores

```
# QAT preparation inserts observers and fake_quants in the model.
model_fp32_prepared = torch.quantization.prepare_qat(model_fp32)
training_loop(model_fp32_prepared)
```

Model Compilation

- PyTorch is eager execution by default
- Graph mode support via JIT compile to TorchScript
- Can export compiled model to C++ or ONNX
- Op fusion support with NVFuser and NNC
- Removes GIL for multi-threaded inference
- Limitation: not easy to use for model training

Model Compilation

PyTorch has two ways to compile:

1. `torch.jit.trace()` — provides model with example inputs and records the operations.
2. `torch.jit.script()` — analyzes the Python source code and compiles it to TorchScript.

Not sure which to use? Start with `jit.trace()`!

Model Compilation: Limitations

`torch.jit.trace() # Generalization is hard`

- Does not support control flow (if-else statements)
- Sometimes captures variables as constants

`torch.jit.script() # Compilers are hard`

- Limited support for Pythonic syntax
- Requires code changes that can obfuscate code

These two approaches can be combined!

Outline

1. Scaling up
    a. Distributed training
    b. Elastic and fault-tolerant experiments
2. Efficient models
    a. Mixed precision / Quantization
    b. Model compilation
3. Deployment

Deployment

Huge area, deserves its own talk!

- Data sourcing / labeling / versioning / pipelining
- Model versioning / packaging / retraining
- Efficient / scalable model inference
- Evaluating / explaining / monitoring predictions

Deployment: TorchServe

- Widely used (Kubeflow, MLflow, SageMaker)
- Packages all model artifacts to single archive
  - Tool is called `torch-model-archiver`
- Runs on server, responds to inference requests
  - Supports gRPC and HTTP/REST

Thanks for attending! Any questions?