Large Scale Training

argonne-lcf / ai-science-training-series

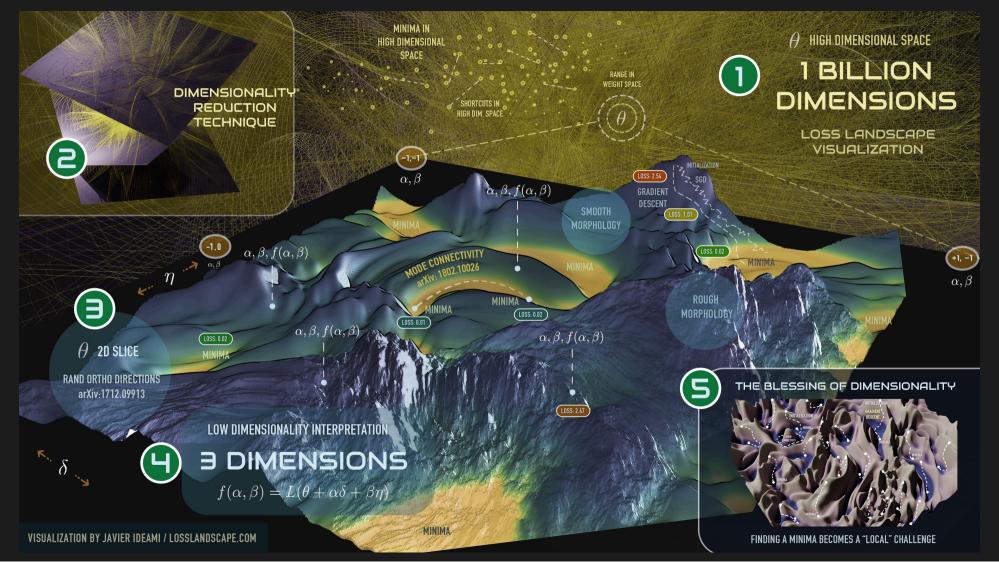
Sam Foreman 2022-11-01



Why Distributed Training?

- Large batches may not fit in GPU memory
- Splitting data across workers → larger batch size
- Smooth loss landscape
- Improved gradient estimators
- Less iterations needed for same number of epochs
 - May need to train for more epochs if another change is not made
 - e.g. scaling learning rate
- See Large Batch Training of Convolutional Networks





Recent Progress

Year	Author	Batch Size	Processor	DL Library	Time	Accuracy
2016	He et al. [1]	256	Tesla P100 x8	Caffe	29 Hrs	75.3%
	Goyal et al. [2]	8192	Tesla P100	Caffe 2	1 hour	76.3%
	Smith et al. [3]	8192 → 16,384	full TPU pod	TensorFlow	30 mins	76.1%
	Akiba et al. [4]	32,768	Tesla P100 x1024	Chainer	15 mins	74.9%
	Jia et al. [5]	65,536	Tesla P40 x2048	TensorFLow	6.6 mins	75.8%
	Ying et al. [6]	65,536	TPU v3 x1024	TensorFlow	1.8 mins	75.2%
	Mikami et al. [7]	55,296	Tesla V100 x3456	NNL	2.0 mins	75.29%
2019	Yamazaki et al	81,920	Tesla V100 x 2048	MXNet	1.2 mins	75.08%

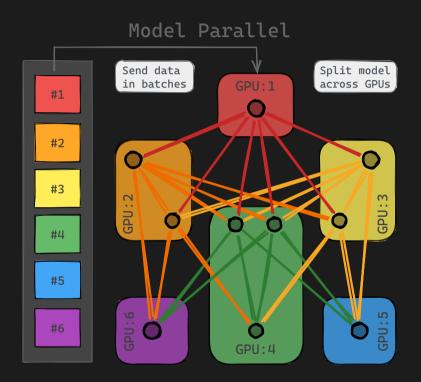


Model Parallel Training



Model Parallel Training

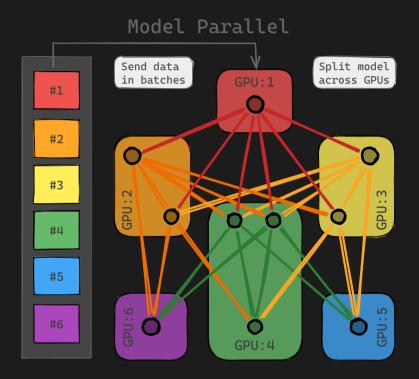
- Split up network over multiple workers
 - Each receives disjoint subset
 - All communication associated with subsets are distributed
- Communication whenever dataflow between two subsets
- Typically more complicated to implement than data parallel training
- Suitable when the model is too large to fit onto a single device (CPU / GPU)





Model Parallel Training

- Suitable when the model is too large to fit onto a single device
 - Partitioning the model into different subsets is not an easy task
 - Might introduce load imbalancing issues limiting scale efficiency
- huggingface/transformers useful reference
 - Excellent series of posts in their documentation on Model Parallelism

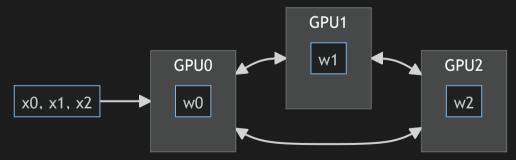




Model Parallel Training: Example

$$y = w_0 * x_0 + w_1 * x_1 + w_2 * x_2$$

- 1. Compute $y_0 = w_0 * x_0$ and send to \longrightarrow GPU1
- 2. Compute $\overline{y}_1=y_0+w_1*x_1$ and send to \longrightarrow GPU2
- 3. Compute $y=y_1*w_2*x_2$

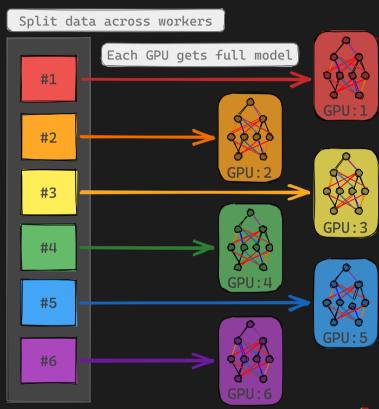






- Typically easier to implement
- Existing frameworks (Horovod, DeepSpeed, DDP, etc)
- Relatively simple to get up and running (minor modifications to code)¹
- Recent presentation on data-parallel training available on YouTube

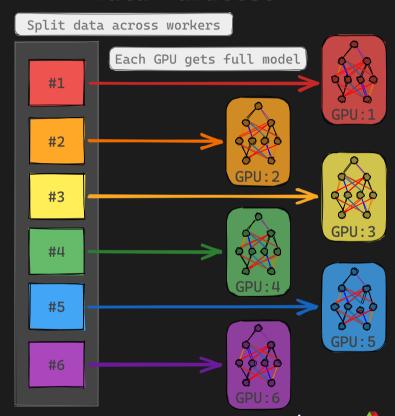
Data Parallel

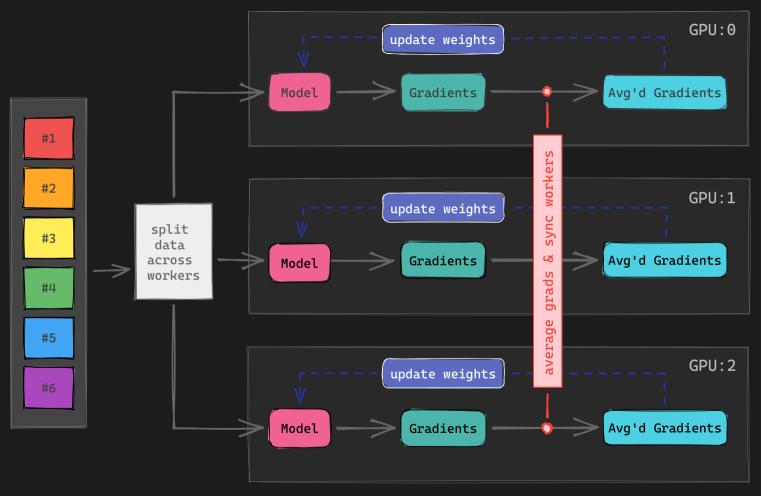




- Each worker has copy of complete model
- Global batch of data split into multiple minibatches
 - Each worker computes the corresponding loss and gradients from local data
- Before updating parameters, loss and gradients averaged across workers

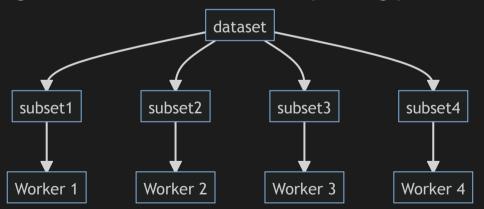
Data Parallel





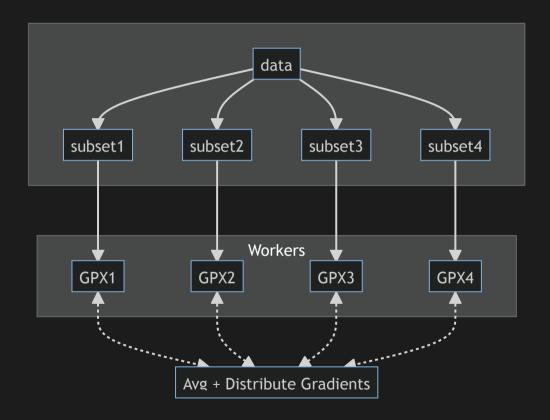


- Each worker has identical copy of model
- Global batch of data split across workers
- Loss + Grads averaged across workers before updating parameters

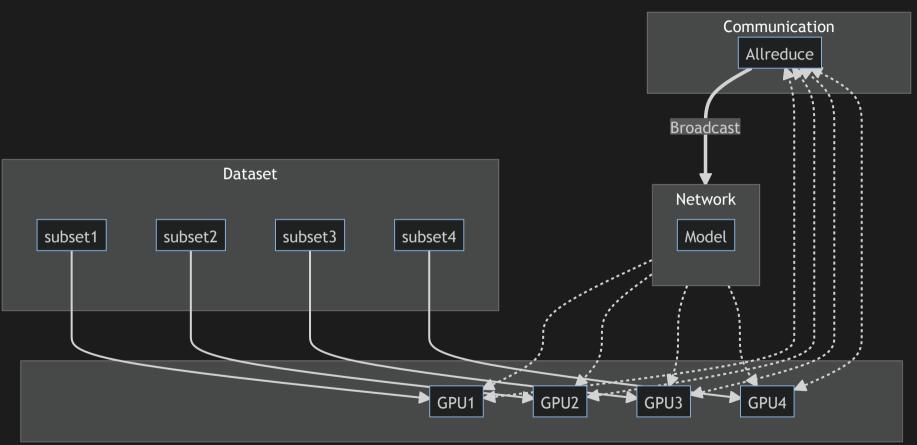




- Each worker receives:
 - identical copy of model
 - unique subset of data



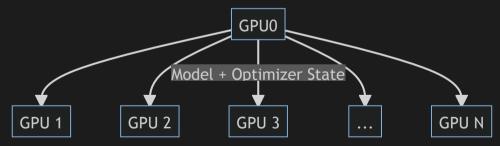






Broadcast Initial State

- At the start of training (or when loading from a checkpoint), we want all of our workers to be initialized consistently
 - Broadcast the model and optimizer states from hvd.rank() == 0 worker





Best Practices

- Use parallel IO whenever possible
 - Feed each rank from different files
 - Use MPI IO to have each rank read its own batch from a file
 - Use several ranks to read data, MPI to scatter to remaining ranks
 - Most practical in big at-scale training

▲ Computation stalls during communication!

Keeping the communication to computation ratio small is important for effective scaling

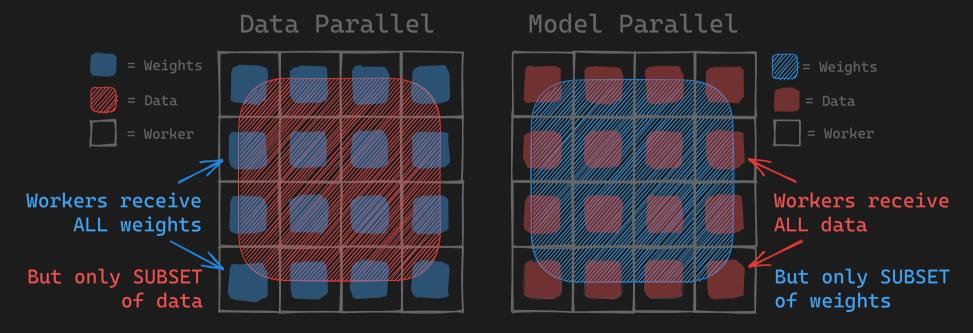


Best Practices

- Take advantage of data storage
 - Use striping on lustre
 - Use the right optimizations for Aurora, Polaris, etc.
- Preload data when possible
 - Offloading to a GPU frees CPU cycles for loading the next batch of data
 - minimize IO latency this way



Comparison





Horovod: Overview

- 1. Initialize Horovod¹
- 2. Assign GPUs to each rank
- 3. Scale the initial learning rate by num. workers
- 4. Distribute gradients + broadcast state
 - Distribute gradients by wrapping tf.GradientTape with hvd.DistributedGradientTape
 - Ensure consistent initialization by broadcasting model weights and optimizer state from rank == 0 to other workers
- 5. Ensure workers are always receiving unique data
- 6. Take global averages when calculating loss, acc, etc. using hvd.allreduce (\ldots)
- 7. Save checkpoints only from rank == 0 to prevent race conditions



TensorFlow + Horovod

Initialize Horovod:

```
import horovod.tensorflow as hvd
hvd.init()
```

Set one GPU per process ID (hvd.local_rank())

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    local_rank = hvd.local_rank()
    tf.config.experimental.set_visible_devices(gpus[local_rank], 'GPU')
```



Scale the Learning Rate

1. Scale by the number of workers to account for increased batch size

```
import horovod.tensorflow as hvd
optimizer = tf.optimizers.Adam(lr_init * hvd.size())
```



TensorFlow + Horovod

Training step then looks like:

```
@tf.function
def train_step(data, model, loss_fn, optimizer, first_batch):
    batch, target = data
    with tf.GradientTape() as tape:
        output = model(batch, training=True)
        loss = loss_fn(target, output)
    tape = hvd.DistributedGradientTape(tape)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    if first_batch:
        hvd.broadcast_variables(model.variables, root_rank=0)
        hvd.broadcast_variables(optimizer.variables, root_rank=0)
    return loss, output
```



Deal with Data

- At each training step, we want to ensure that each worker receives unique data
- This can be done in one of two ways:
 - 1. Manually partition data (ahead of time) and assign different sections to different workers
 - 1. Each worker can only see their local portion of the data
 - 2. From each worker, randomly select a mini-batch
 - 1. Each worker can see the full dataset

▲ Don't forget your seed!

When randomly selecting, it is important that each worker uses different seeds to ensure they receive unique data



Deal with Data

```
(images, labels), (xtest, ytest) = (
   tf.keras.datasets.mnist.load data(path='mnist.npz')
dataset = tf.data.Dataset.from tensor slices(
    (tf.cast(images[..., None] / 255.0, tf.float32),
    tf.cast(labels, tf.int64))
test dataset = tf.data.Dataset.from tensor slices(
    (tf.cast(xtest[..., None] / 255.0, tf.float32),
    tf.cast(ytest, tf.int64)))
dataset = dataset.repeat().shuffle(1000).batch(args.batch size)
test dataset =
   test dataset.shard(
       num shards=hvd.size(),
       index=hvd.rank()
    ).repeat().batch(args.batch size)
```



Average Across Workers

 Typically, we will want to take the global average of the loss across all our workers, for example:

```
global_loss = hvd.allreduce(loss, average=True)
global_acc = hvd.allreduce(acc, average=True)
```



ai4sci



Hands-On

- 1. Navigate to ai-science-training-series
- 2. git pull
- 3. Navigate into 07_largeScaleTraining/src/ai4sci
- 4. To run (with a batch_size=512):

```
./main.sh batch_size=512 > main-bs-512.log 2>&1 &
```

5. View output:

```
tail -f "main-bs-512.log" $(tail -1 logs/latest)
```



Thank you!

- Organizers
- ALCF Data Science & Operations
- Feel free to reach out!



Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.



