How to train DNNs efficiently?

Data parallelism:
- Replicate model on every worker
- Train on disjoint samples

But:
- Communication (weight resync) is very expensive
- SOTA models are huge and don’t fit on a single worker

Instead use model parallelism:
- Partition the model
- Transfer intermediate activations between workers

To get high worker utilization, use pipelining:
- Once the first sample goes to Machine 2, Machine 1 can start processing the second sample, etc.
- For training (forward + backward pass), schedules were proposed by PipeDream and GPipe

How to split the DNN graph?

- Assign every node to a machine/device
- Problem called device placement
- Want to balance computation out, but also minimize communication
- Usually done by human experts; growing need for automated methods

Prior work

Approach 1:
- Treat objective function as black box (e.g. measure time of 10 training steps)
- Optimize it with generic heuristics such as Reinforcement Learning [Mihroseti et al., Spotlight] or MCMC [FlexFlow]
- Learn a placement policy and generalize to unseen graphs [Placeto, GDP, REGAL]
- Pros: realistic-by-definition performance model
- Cons: very expensive to evaluate cost of each partition tried; heuristics w/o guarantees

Approach 2 (ours):
- Build cost model that closely reflects real performance
- Solve resulting “offline” optimization problem with principled algorithmic techniques
- Previously done in PipeDream, but only for linear computation graphs (i.e. path-graphs)

Challenges:
- Formulate a correct (close-to-reality) cost model
- Resulting problem is highly non-trivial

Dynamic Programming Approach

Objective: maximize throughput
That is, minimize time-per-sample, which is the max load of any machine (load = computation + communication)

For each downward-closed set \( I \) of nodes (\textit{ideal}), compute:

\[
dp[I, k] = \min_{I' \subseteq I \setminus k} \max \left( \frac{\text{load}(I' \setminus I)}{k} \right)
\]

DP recursion:

\[
dp[I, k] =  \min_{I' \subset I} \left( \frac{\text{load}(I' \setminus I)}{k} \right) + \min_{I' \subset I} \max(dp[I', k - 1], \text{load}(I' \setminus I))
\]

Notes:
- \( S = I \setminus I' \) is a contiguous subgraph, and one gets any contiguous subgraph this way
- The function \( \text{load}(S) \) can be arbitrary, should take computation and communication into account
- Can set \( \text{load}(S) = \infty \) if \( S \) doesn’t fit on one device (OOM)
- Runtime is \( O((\text{number of machines}) \cdot (\text{number of ideals})) \) - exponential in theory, good in practice

Yields a general framework; can also handle:
- Multiple device types
- Hybrid mode with data parallelism
- Hierarchical communication costs

Our contributions:

We isolate the structured combinatorial optimization problem at the core of device placement, for both training and inference

And we give efficient algorithms to find optimal splits:

Integer Programming Approach

that can find non-contiguous splits

- The green/checkered subgraph is non-contiguous
- Such splits are predicted to yield up to 27% higher throughput for some DNN workloads

Evaluation

- Several modern DNN workloads (BERT, GNMT, Inception, Resnet)
- We find provably optimal splits on operator-level graphs, within seconds to minutes
- Higher throughput than human experts, PipeDream, some other baselines


Some figures courtesy of PipeDream