COMP 642 — Machine Learning April 06, 2021

Lecture 18

Lecturer: Anshumali Shrivastava
Scribe By: Junyuan Sheng,Oluwatonamise (Tona) Akerele, Zhenghui Guo

Disclaimer: *These lecture notes are intended to develop the thought process and intuition in machine learning. The materials are not thoroughly reviewed and can contain errors.*

Review

- Most important aspect of Machine Learning: Being able to validate hypothesis
  - We generate hypothesis much faster than we can validate them
- Parallelism is used in matrix multiplication in Neural Networks
  - Gradient is another way of doing this

Gradient Descent Function:

$$\nabla L = \frac{1}{k} \nabla L(F(X_i), y_i)$$

Forms of Parallelism

- Gradient Computation
- Model is a Neural Network then it's F

Training Large Models

The main bottleneck for training very large neural network models is the intense demand for a large amount of GPU memory, way above what can be hosted on an individual GPU machine. Besides the model weights (e.g. tens of billions of floating point numbers), it is usually even more expensive to store intermediate computation outputs such as gradients and optimizer states (e.g. momentums variations in Adam). Additionally training a large model often pairs with a large training corpus and thus a single process may just take forever. As a result, parallelism is necessary. Parallelism can happen at different dimensions, including data, model architecture, and tensor operation.

- Multi-core Parallelism: Multiple cores from a single machine can be used for fitting the data and model, where these cores share the memory (PRAM model). The use of multiple cores can be performed in the following ways:
  - To process multiple images at once using the multiple cores in each layer. This is a core parallel process.

  - GPU:A computationally intensive subroutine like matrix multiplication can be performed using GPU
  - Specialized processor with specialized memory access to do a lot of things.SGD of multiple mini-batches can be performed in parallel by using multiple cores
  - Single device set-up

- Multi-Node Parallelism : It's basically a cluster
  - Hundreds of nodes capable of doing a lot of computation and want to leverage all of them to do the training
  - average computing power of multiple processors to make things faster
  - 2 Forms of Parallelism

- Training Data
  - Data Parallel - popular: The most naive way for Data parallelism (DP) is to copy the same model weights into multiple workers and assign a fraction of data to each worker to be processed at the same time.
    * Data resides in multiple nodes, gets generated fast, and there is a lot of data.
    * Why cluster computing is really important: much faster.
    * Naive DP cannot work well if the model size is larger than a single GPU node's memory. Methods like GeePS (Cui et al. 2016) offload temporarily unused parameters back to CPU to work with limited GPU memory when the model is too big to fit into one machine. The data swapping transfer should happen at the backend and not interfere with training computation.At the end of each minibatch, workers need to synchronize gradients or weights to avoid staleness. There are two main synchronization approaches and both have clear pros cons.

      · Bulk synchronous parallels (BSP): Workers sync data at the end of every minibatch. It prevents model weights staleness and good learning efficiency but each machine has to halt and wait for others to send gradients.

      · Asynchronous parallel (ASP): Every GPU worker processes the data asynchronously, no waiting or stalling. However, it can easily lead to stale weights being used and thus lower the statistical learning efficiency. Even though it increases the computation time, it may not speed up training time to convergence.
  - Model Parallel - very rare and big company use this. Aims to solve the case when the model weights cannot fit into a single node. The computation and model parameters are partitioned across multiple machines. Different from data parallelism where each worker hosts a full copy of the entire model, MP only allocates a fraction of model parameters on one worker and thus both the memory usage and the computation are reduced.Since deep neural networks usually contain a stack of vertical layers, it feels straightforward to split a large model by layer, where a small consecutive set of layers are grouped into one partition on one worker. However, a naive implementation for running every data batch through multiple such workers with sequential dependency leads to big bubbles of waiting time and severe under-utilization of computation resources.

Algorithm 1: DistributedDataParallel

Input: Process rank r, bucket size cap c, local model *net*

Function `constructor(`*net*`)`**:**
  if *r=0* **then**
    broadcast *net* states to other processes
  init buckets, allocate parameters to buckets in the reverse order of net.parameters()
  for *p* ***in*** *net.parameters()* **do**
    acc ← *p*.grad_accumulator
    acc → add_post_hook(autograd_hook)

Function `forward(`*inp*`)`**:**
  out = *net*(inp)
  traverse autograd graph from out and mark unused parameters as ready
  return out

Function `autograd_hook(`*param_index*`)`**:**
  get bucket b_i and bucket *offset* using param_index
  get parameter *var* using *param_index*
  view ← b_i.*narrow*(*offset*, *var.size*())
  view.copy_(var.grad)
  if *all grads in* b_i *are ready* **then**
    mark b_i as ready
  launch AllReduce on ready buckets in order
  if *all buckets are ready* **then**
    block waiting for all AllReduce ops

Figure 1: pytorch-ddpn

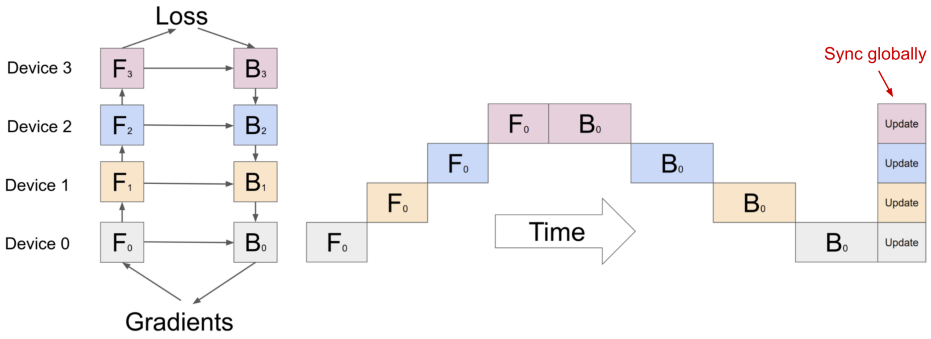

Figure 2: naive-data-parallelism)

A naive model parallelism setup where the model is vertically split into 4 partitions. Data is processed by one worker at a time due to sequential dependency, leading to large "bubbles" of idle time. (Image source: Huang et al. 2019)

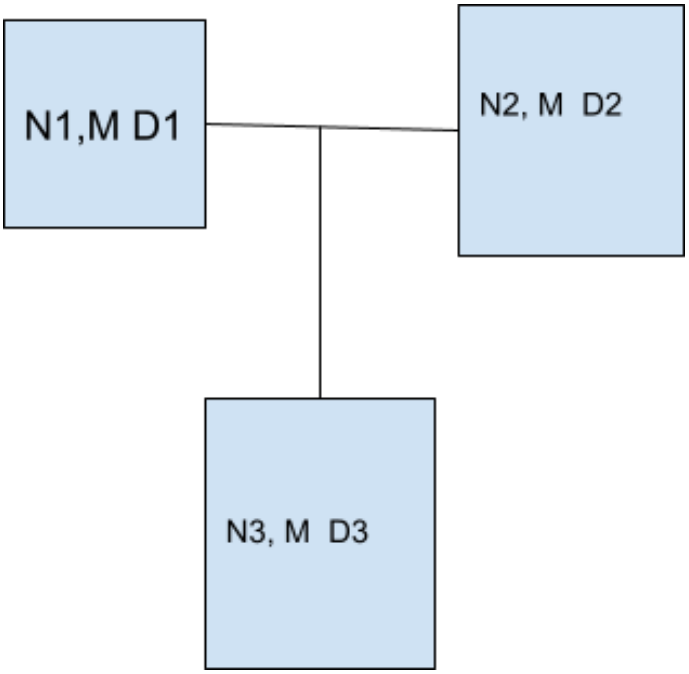

Figure 3: image2

Example:Computing disjoint nodes

$$D = D1UD2UD3$$

$$|D1 \cap D2| = 0$$

Federated Learning

It works like this: your device downloads the current model, improves it by learning from data on your phone, and then summarizes the changes as a small focused update. Only this update to the model is sent to the cloud, using encrypted communication, where it is immediately averaged with other user updates to improve the shared model. All the training data remains on your device, and no individual updates are stored in the cloud.
see comics `https://federated.withgoogle.com`

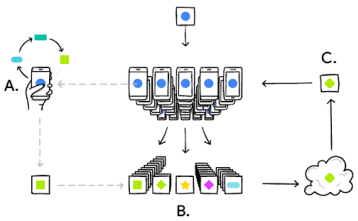

Figure 4: FederatedLearning$_F inal Files_F low Chart1$

Your phone personalizes the model locally, based on your usage (A). Many users' updates are aggregated (B) to form a consensus change (C) to the shared model, after which the procedure is repeated. Federated Learning allows for smarter models, lower latency, and less power consumption, all while ensuring privacy. And this approach has another immediate benefit: in addition to providing an update to the shared model, the improved model on your phone can also be used immediately, powering experiences personalized by the way you use your phone.

Assuming there are 3 parties ..

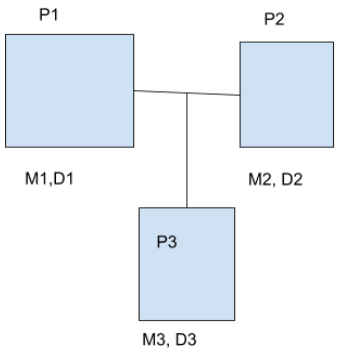

Figure 5: image2

- All have data but don't want to communicate certain private data
  - Communication channels does not allow to communicate anything that isn't private
  - Want to train this model
  - Transporting gradient of the data
- Data will be significantly biased: Heterogeneity
  - Cannot share gradients between the data
  - Everything is the same besides this

$$\text{Model} : (m1 + m2 + m3\ ) / 3$$

- Do one step of gradient descent and average, as good as doing gradient descent for modeling

References

[1] How to Train Really Large Models on Many GPUs? `https://lilianweng.github.io/posts/2021-09-25-train-large/`

[2] Federated Learning: Collaborative Machine Learning without Centralized Training Data `https://ai.googleblog.com/2017/04/federated-learning-collaborative.html`

[3] Federated Learning `https://federated.withgoogle.com/`