TensorFlow: A System for large-scale machine learning

Presented by,

K

arthik Mohan

What is TF?

- Machine Learning system that operates at Large scale heterogeneous environments.
- Uses Dataflow graph to represent computation, shared state and operations.
- Maps nodes of Dataflow graph across many machine in a cluster and within machine across multiple computation devices like CPUs, GPUs and TPUs.
- Offers flexibility to developers to experiment with novel optimizations and training algorithms.

- Unified dataflow graph to represent both the computation in an algorithm and the state in which the algorithm operates.
- Inspiration from high level data flow programming model and low level efficiency of parameter servers.
- Traditional Data flow: Graph vertices represent functional computation on immutable data.
- Tensorflow: Computations that own or update mutable state.
- Edges carry tensors.
- By unifying computation + State, allows for experimentation with different parallelization schemes.

DistBelief

- Distributed system for training Neural networks.
- Parameter server architecture.
- Job: Two disjoint set of processes. 1. Stateless worker processes
- 2. Stateful Parameter Server
- Neural Network -> DAG of Layers -> Loss function.
- |-----> Composition of Mathematical operators.
- For example, FC Layer: Wx + b

Limitations of DistBelief

- Defining new layers
  - For efficiency -> Layers as C++ classes.
  - Less familiar programming language.
- Refining the training Algorithms
  - Many NN uses SGD
  - Refinements to SGD requires modifying parameter server implementation.
  - More efficient to offload computations on parameter server.

:

- DistBelief uses fixed execution pattern
  - Read Input data and current parameter values
  - Compute Loss function
  - Compute Gradients
  - Write Gradients to Parameter Server.
- Fails for RNN, RL and Adversarial Networks.
- Other algorithms like Expected Maximization, Decision Tree, Random Forest could benefit from a common, well optimized distributed runtime.

Design Principles

Dataflow graphs from Primitive Operators:

- DistBelief uses few complex “layers”
- TF uses individual mathematical operators in Dataflow graph.
- Reason: Makes it easier for programmers to compare novel layers using high level scripting interface.
- Makes it easier for automatic differentiation.
- Enables experimentation with different update rule.

Deferred Execution:

- TF application has two phases
  - Program -> Symbolic dataflow graph with placeholders for input data and variables that represent state.
  - Execute an optimized version of program on a set of available devices.
- By delaying, optimal execution of dataflow graph by using global information about the computation.
- Execution is more efficient.

Common Abstraction of heterogeneous accelerators

- Special purpose accelerators in addition to GPUs and CPUs.
- TPU achieves O(magnitude) compared to state of the art.
- To support these accelerators, define common abstraction of devices.
- At minimum device must implement methods for,
  - Issuing kernel for execution.
  - Allocate memory for inputs and outputs.
  - Transfer buffers to and from host memory.
- Each operator can have multiple implementations for different devices.
- Same TF program for different devices.
  - GPU, TPU, CPU for Training, Serving and Inference

Key difference

- In TF, there is no parameter server.
- Only PS taks and worker tasks.
- PS tasks can arbitrarily run TF graphs, hence it is more flexible than conventional parameter server.
- It can be programmed with same scripting interface that is used to define models.

Related Work

- Single Machine frameworks
  - Caffe: Similar to DistBelief, same limitations.
  - Theano: Closest to TF, provides same flexibility in a single machine.
  - Torch: Imperative programming model, fine grained control over execution order and memory utilization.
- Batch Dataflow Systems:
  - Starting with MR, batch flow systems have been applied to large number of ML Algorithms.
  - DryardLINQ: Adds a high level query language supports more sophisticated algo than MR
  - Spark: Extends DryardLINQ with ability to cache computation in memory and therefore suited for iterative ML algorithms when input data is in memory.
  - Dandelion: Extends DryardLINQ with code generation for GPUs and FPGAs.

Limitations:

- Input data must be immutable, and all subcomputations must be deterministic, so systems can re-execute subcomputations when machines in clusters fail.
- Makes updating machine learning models an expensive operation.
- Sparknet: 20 Seconds to broadcast weights collect and update from 5 workers.
- Thus each model update step must process large batches, slowing convergence.
- TF takes 2 seconds.

Parameter Servers

Set of servers that manages shared state that is updated by a set of parallel workers.

Ex: DistBelief, MXNet

Parameter server specialized for use with GPUs can achieve speedups on small clusters.

MXNet uses Dataflow graph and parameter server.

It supports a key value store interface, which enables aggregating updates sent from multiple devices in each worker using an arbitrary function to combine incoming updates with current value.

Disadvantage: Does not allow sparse gradient updates within single values, crucial for distributed training of larger model.

TensorFlow Execution Model

- TF uses single dataflow graph to represent all computations and state in ML Algorithms.
  - This includes Math. operations, parameters + update rules, Input preprocessing.
- Dataflow graph, explicitly expresses the communication between subcomputations, thus making it easy to execute independent computations in parallel and to partition computation across multiple devices.
- **Difference from Batch Dataflow systems:**
- Model supports multiple concurrent execution on overlapping subgraphs of overall graphs.
- Individual vertices may have mutable state that can be shared between different execution of the graph.

- **Key Observation:** In parameter server architecture, mutable state makes it possible to make in place updates to very large parameters, and propagate those updates to parallel training steps quickly.
- Data flow with mutable graph mimic the functionality of parameter server, but with the added ability to execute arbitrary dataflow subgraphs on machines that host the shared model parameters.
- Allows for experimentation with different optimization algorithms, parallelization strategies.

DataFlow graph elements

- Computation at vertices: Operations
- Values that flow along edges: Tensors
- Tensors: models all data as tenors with elements being int32, float32 or string.
    - Naturally represents Input to and Output from common mathematical operations in many ML Algorithms.
    - Matmul operation -> Input: two 2D tensors Output: 2D tensor.
    - At lowest level, all tensors are dense.

- **Operations**
- An operation takes m>=0 tensor as input and produces n>=0 tensors as output.
- An operation has a named type and can have 0 or more compile time attributes.
- Operations can be polymorphic and variadic at compile time: Attributes determine both expected types and arity of inputs and outputs.
- **Stateful Operations: Variables**
  - Owns a mutable buffer for storing shared parameters of model as it is trained.
  - No input and outputs a reference handle r.
  - Read (r) -> State[r] as dense tensor.
  - AssignAdd(r, x) -> State[r] + x -> State'[r]

Stateful operation: Queues

- **FIFO Queues:**
  - Internal queue of tensors.
  - Concurrent access in FIFO order.
  - Produces a reference handle r.
  - Useful when enque(r) -> block when queue is full and deque(r) -> block when queue is empty.
  - Provides back pressure and supports synchronisation, which can be used for streaming computation between subgraphs.

Partial and Concurrent Execution

- API allows clients to specify which part of the subgraph to execute.
- Client selects 0 or more edges as inputs tensors into the dataflow graph.
- Client selects 1 or more edges as output tensors into the dataflow graph.
- Runtime prunes the graph to necessary set of operations.
- Each invocation of API -> **Step**
- Supports multiple concurrent steps on same graph.
- Stateful operations: Steps share data and synchronize when necessary.

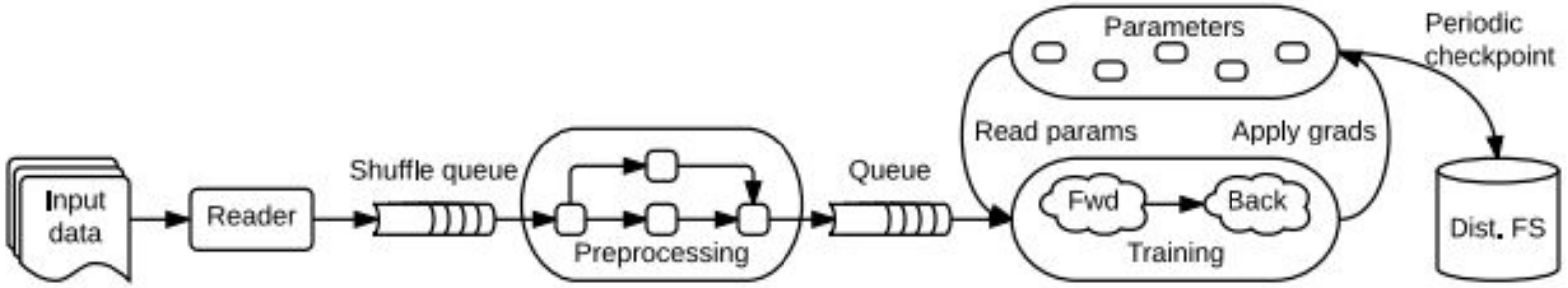


Figure 2: A schematic TensorFlow dataflow graph for a training pipeline, containing subgraphs for reading input data, preprocessing, training, and checkpointing state.

- Mutable state + Coordination via queues: Possible to specify model architecture in user level code.
- Concurrent executions of subgraph run asynchronously with one other.
- This asynchronicity makes it straightforward to implement ML Algorithms with weak consistency.

Distributed Execution

- Dataflow makes communication between subgraphs explicit.
- Allows for same TF program to be used for training (GPU), serving (TPU) and inference (Mobile CPUs).
- Each operation resides on a particular device.
- Device is responsible for executing a kernel for each operation assigned to it.
    - TF allows multiple kernels to be registered for a single operation based on data or device type.

- Runtime places operations subject to constraints in the graph.
- **Placement algorithm:**
  - Computes feasible set of devices for each operation.
  - Calculates set of operations that must be located.
  - Select a device for each Colocation group.
- Implicit colocation constraint: Stateful operation and state on same device.
- Explicit constraints: “GPU in a task”, “* device in a particular task”
- Client side programming constraints: Parameters are distributed among a set of “PS” tasks.
- Offers flexibility in mapping operations in DF graph to devices. Novice users can get away with simple heuristics yielding an adequate performance.

- Separates **placement directives** from **model definitions.**
- Since **placement can change** after training the model.
- After placing the operation, compute partial subgraph for a step.
- Then TF **partitions** operation into per device subgraphs.
- Per device subgraphs for d contains all the operations assigned for d with Send and Recv that replace edges in device boundaries.
- Send (Input | Key) -> specified device.
- Recv -> Block until the key is available locally.
- For a step after pruning, placing and partitioning its' subgraphs are cached in their respective devices.
- Client session manager manages maintain step definitions ---> cached subgraphs.
- Distributed step on large graph can be initiated with one small message to each participating task.

Dynamic Control Flow

- TF supports advanced ML Algorithms that contain conditional and iterative control flow.
- Ex: RNN, LSTM models.
- Core of RNN is a recurrence relation.
- Dynamic control flow allows for iteration over sequence with variable lengths without unrolling the graph to length of longest sequence.
- Deferred execution is used to offload larger chunk of work to accelerators.
- Hence, the conditional and iterative constructs are added in the dataflow graph itself using primitives like map(), fold(), scan()

- Borrows Switch and Merge primitives from dynamic dataflow architecture.
- Switch -> Demultiplexer
  - Input: (D, C) Output: (O1 = v, O2 = dead | c=0) ; (O1 = dead, O2 = v | c=1)
- Merge -> Multiplexer
  - Input: (I1, I2) Output: (frwd at most one | I1!= dead & I2!= dead); (dead | I1== dead or I2== dead)
- Conditional operator uses Switch to use one of the branches and merge to combine the outputs of the branches.
- Similarly while loop using Enter, Exit and NextIterator.
- Execution of graphs overlap, TF also partitions conditional branches and loops across multiple devices.
- Partitioning step adds logic to coordinate the start and termination of each iteration on each device, and also determines the termination of loop.

Extensibility Case Studies

Differentiation and Optimization:

Many ML Algorithms uses some variant of SGD algorithm.

TF includes a user library that differentiates symbolic expression of loss function and produces new symbolic expression representing gradients. (Backprop code is automatic.)

The differentiation algorithm performs BFS to identify all paths from loss functions to set of parameters, and sums all the partial gradients that each path contributes.

For conditional and iterative subcomputations, differentiation is done by adding nodes to the graphs that record the decision in forwards pass and replays in backward pass.

Differentiating iterative computations over long sequences leads to large amount of intermediate state accumulated in memory, and techniques have been developed.

- TF makes it easy to experiment with optimization algorithms.
- SGD is easy to implement in a parameter server.
- Update Rule: W’ -> W - learning_rate * (dL/dW)
- Write operation: -= Write: learning_rate * (dL/dW) to each W after a training step.
- But Momentum based algorithms are difficult to implement in DistBelief as it will require changes to the representation of parameter data and complex write operation.
- Ex: Momentum, Ada-Grad, AdaDelta, RMSProp, Adam.
- These algorithms can be implemented in TF using the variable operations and primitive math operations.

Training very large models

- To train model on high dimensional data, it is common to use a distributed representation.
- Embedding Layers: It embeds a training example as a pattern of activity of several neurons.
- In recommender Systems, we learn an embedding matrix of size: n x d where n is the number of words and d is the dimension.
- During inference, this embedding matrix is multiplied by a sparse vector b.
- The Multiplication gives, gives a smaller b x d dense matrix representation.
- In TF, models that process sparse data, n x d can amount to gigabytes of parameters. Ir., a large model.
- These are too large to copy to a worker on every use, or even store in RAM of single host.

- It is implemented as a composition of primitive operations.

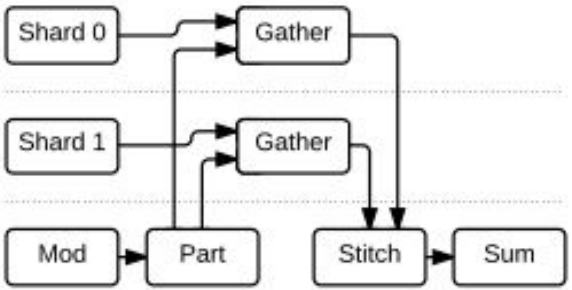


Figure 4: Schematic dataflow for an embedding layer (§4.2) with a two-way sharded embedding matrix.

- Gather -> Extract sparse set of rows from a tensor, and TF collocates this operation with variable on which it operate
- Part -> Divide incoming into variable sized tensors that contain the Indices destined for each shard.
- Stitch -> Combine partial results from each shard into a single tensor.

Fault Tolerance

- Training large models can take days.
- Resources are typically non dedicated for that long.
- Therefore likely to experience failure or preemption, hence we need fault tolerance.
- No need for strong consistency.
- Solution: User-level checkpointing.
- Save: Write one or more tensor to checkpoint file.
- Restore: Reads one or more tensors from a checkpoint file.
- Assign: Stores the restored value in respective variable.
- Typically, clients periodically calls save and during start up, attempts to Restore from latest checkpoint.
- TF includes a library for constructing appropriate graph structure and for invoking save and restore as necessary.
- It does not produce consistent checkpoints.
- If training and checkpointing concurrently, all or some or none of the updates from the training step.
- Compatible with relaxed guarantees of asynchronous SGD.
- For consistent checkpoint, additional synchronization needed to ensure update do not interfere with checkpointing.

Synchronous Replica Coordination

- SGD is robust to asynchrony, and many systems train deep neural networks using asynchronous parameter updates.
- Increased throughput comes at cost of using stale parameter values in training steps.
- Since, GPUs work with 100s rather than 1000s of machines, synchronous training may be faster.
- TF graph enables users to change how parameters are read and written when training a model.
- Three models:
    - **Asynchronous replication:** At the beginning read current values, and applies gradient to (different) values at the end.
    - **Synchronous replication:** Using queues to coordinate execution. Blocking queue acts as a barrier - > all workers read same parameter values. Per-variable queue -> Accumulates gradient updates from all workers to apply atomically. Simple version accumulates gradients before applying, but slow workers limit throughput.
    - **Synchronous with backup worker:** To mitigate stragglers. Here the backup worker runs proactively, and aggregation takes first m of n updates produced. Exploiting the fact it is not a problem if a batch is ignored since SGD selects examples randomly and each worker works on different random batch.

Implementation

TF runtime is a cross-platform library.

C API separates user level code in different languages from core runtime.

Core is implemented in C++: runs on Linux, Mac OS X, Windows, Android, iOS, x86 and various arm based CPU; NVIDIA's Kepler, Maxwell and Pascal GPU microarchitecture.

Distributed master: User request -> Execution across set of tasks.

Given graph and step definition, it prunes and partitions the graph to obtain subgraphs for each participating device, and caches these subgraphs to be reused in subsequent steps.

It also applies optimizations since it has a global view.

Dataflow executor: Each task handles requests from the master, and schedules execution of the kernels that comprise a local subgraph.

It is optimized and can execute 10,000 subgraphs per second, which enables large number of replicas to make rapid, fine-grained training steps.

It dispatches kernels to local devices and runs these kernels in parallel when possible across multiple CPU cores or GPU streams.

Runtime contains over 200 Standard operations and liberally uses libraries like cuDNN.

Also implemented quantization, for faster inference in mobile devices.

Send and Recv operations between a pair of source and destination device types. Transfers between CPUs and GPUs use cudaMemoryAsync() API to overlap computation and data transfer.

Between local GPUs: DMA to relieve pressure on the host. gRPC over TCP, and RDMA over covered ethernet.

- Everything is implemented above the C API, in user level code.
- Users compose standard operations to build high level abstraction such as Neural Network layers, optimization algorithms and shared embedded computations.
- TF supports multiple client languages but C++ and Python are prioritized.
- It is inefficient to represent sub computations as a composition of operations, hence users can register additional kernels that provides an efficient implementation written in C++.
- It is also profitable to hand-implement fused kernels for some performance critical operations, such as ReLU and sigmoid activation functions and their corresponding gradients.

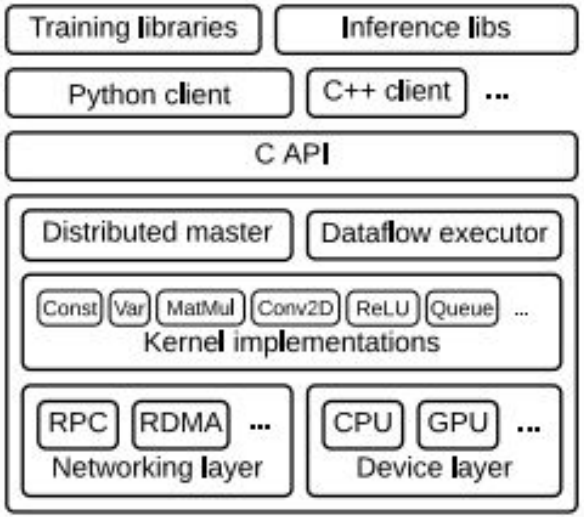


Figure 6: The layered TensorFlow architecture.

Evaluation

Single Machine Benchmarks:

Scalability does not mask poor performance at small scale.

HW: Six-core Intel Core i7-5930K CPU at 3.5 GHz + NVIDIA Titan X GPU

It achieves shorter runtime than Caffe.

Performance within 6% percent of the latest version of Torch.

Torch and TF use same version of cuDNN, which implements convolution and pooling operations on critical path for training.

Caffe uses open source implementations which are less efficient.

Neon outperforms ny using hand optimized convolutional kernels implemented in assembly language.

	Training step time (ms)			
Library	AlexNet	Overfeat	OxfordNet	GoogleNet
Caffe [38]	324	823	1068	1935
Neon [58]	87	**211**	**320**	**270**
Torch [17]	**81**	268	529	470
TensorFlow	**81**	279	540	445

Table 1: Step times for training four convolutional models with different libraries, using one GPU. All results are for training with 32-bit floats. The fastest time for each model is shown in bold.

Synchronous Replica Microbenchmark:

The performance of coordination implementation is the main limiting factor for scaling with additional machines.

Figure shows number of null training steps that TF performs per second for varying model size and increasing number of synchronous workers.

Null training step -> fetch shared model from 16 PS, perform computation, and send update to parameters.

Scalar curve shows best performance. Because only 4k byte value is fetched from each PS task.

Median step time: 1.8 ms with 100 workers. It measures the overhead of synchronization mechanism and noise expected when running on shared cluster.

Dense curve shows performance of null step when worker fetches entire model.

Experiment is repeated with models of size 100MB to 1GB with equally shared parameter over 16 PS tasks.

Median Step time at 100MB: 147 ms with 1 worker to 613 ms with 100 workers.

Median Step time at 1GB: 1.01s with 1 worker to 7.16s with 100 workers.

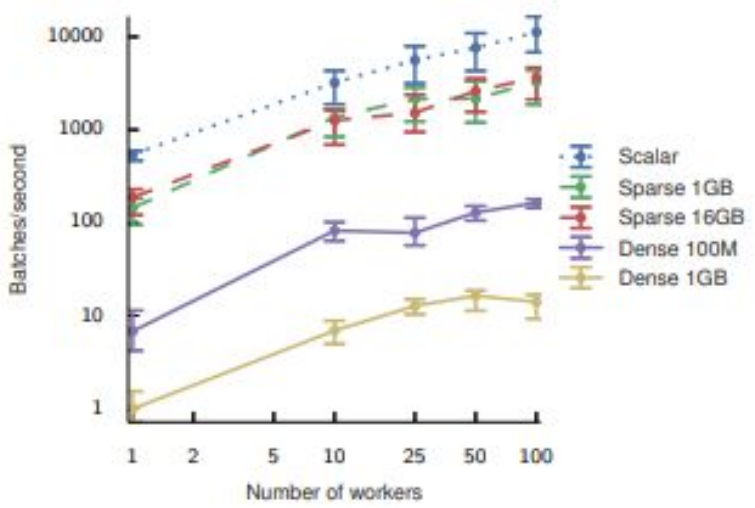


Figure 7: Baseline throughput for synchronous replication with a null model. Sparse accesses enable TensorFlow to handle larger models, such as embedding matrices (§4.2).

For large models, training steps access only a subset of parameters.

The sparse curve shows the throughput of the embedding lookup operation.

Each worker read 32 randomly selected entries from the Embedding matrix containing 1GB or 16 GB of data.

Step times do not vary with the size of embedding with step size ranging from 5 ms to 20 ms,

Image Classification:

Training CV models to high accuracy requires large amount of computation across a cluster of GPU enable servers.

Model used in experiments: Google Inception v3 ; Accuracy : 78.8% on ILSVRC 2012 Image classification challenge

Experiments: Scalability of training model using multiple replicas, vary the number of worker tasks with 7 PS tasks on two different clusters.

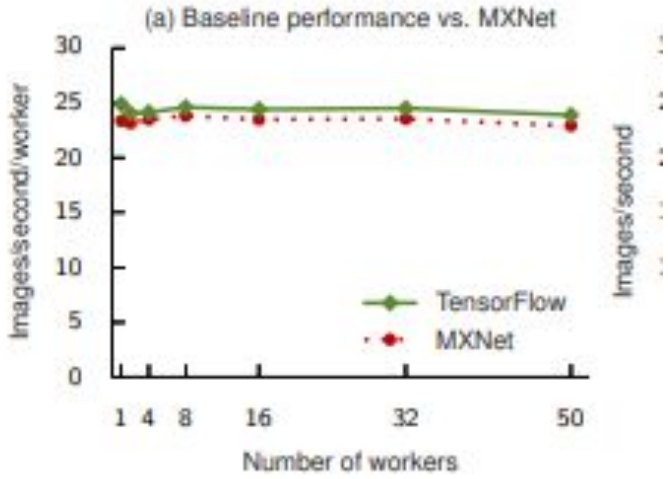
First Experiment: Performance of training Inception using Asynchronous SGD on TensorFlow and MXNet.

HW: GCE VM's running Xeon E5 servers with NVIDIA K80 GPUs with 8 vCPUS, 16 Gbps of NW bandwidth with one GPU perVM.

Both: 7 PS tasks on separate VM's with no GPU.

Tensorflow achieves performance that is marginally better than MXNet.

Results largely determined by single GPU performance, and both systems use cuDNN version 5.1



TensorFlow achieves slightly better throughput than MXNet for asynchronous training.

Second Experiment: Effect of coordination on training performance

HW: Large internal cluster with NVIDIA K40 GPUs and a shared datacenter network.

Ideally, With efficient synchronous, model will train in fewer steps and converge to higher accuracy than asynchronous training.

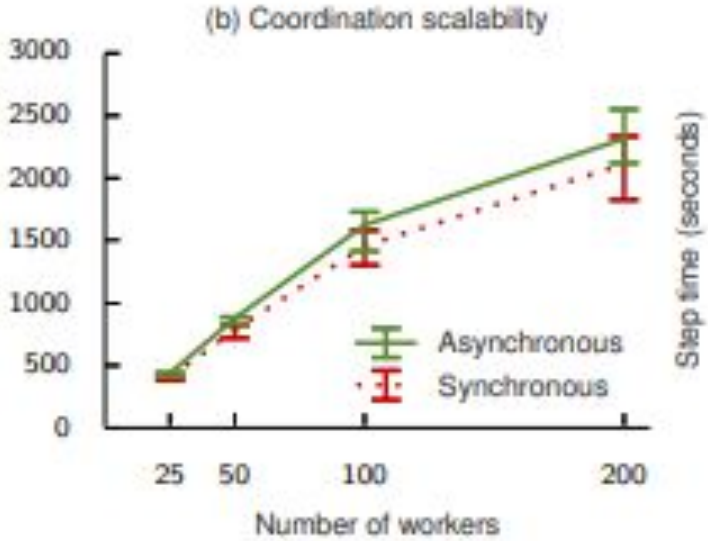
Training throughput improves to 2,3000 images/second as the number of workers is increased to 200, but with diminishing returns.

Adding more workers leads to more contention on PS tasks both at NW interface and aggregation of update weights.

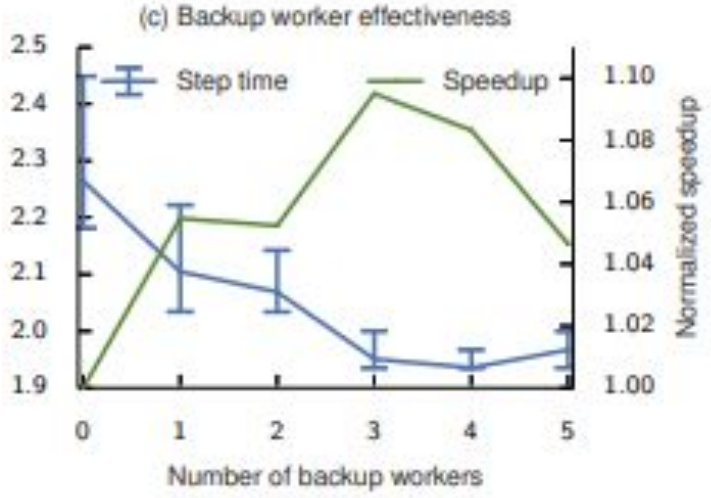
Synchronous steps are longer than Asynchronous steps in all configurations.

Median of Synchronous step is appx. 10 % longer than asynchronous step with same workers.

Above 90th percentile, the performance degrades sharply, because stragglers impact tail latency.



- To mitigate tail latency, add backup workers so a step completing m of n tasks produces gradients.
- Shows the effect of adding backup worker to a 50 Worker training job.
- Each addition of backup worker upto 4 reduces the mean step time, because probability of straggler affecting step decreases.
- Adding 5th backup worker degrades performance, because the first 51st worker (first result being discarded) is more likely to be non straggler that generate incoming traffic for PS tasks.
- Figures also show normalized speedup, calculated as t(b) / t(0) * 50 / (50 + b). It discounts speedup by fraction of additional resources consumed.
- t(b) -> Median step time and b is backup workers.
- Adding 4 backup worker achieves shortest overall step time (1.93s)
- Adding 3 backup worker achieves highest normalized speedup (9.5%), and hence uses less aggregate GPU runtime.



Adding backup workers to a 50-worker training job can reduce the overall step time, and improve performance even when normalized for resource consumption.

Language Modeling:

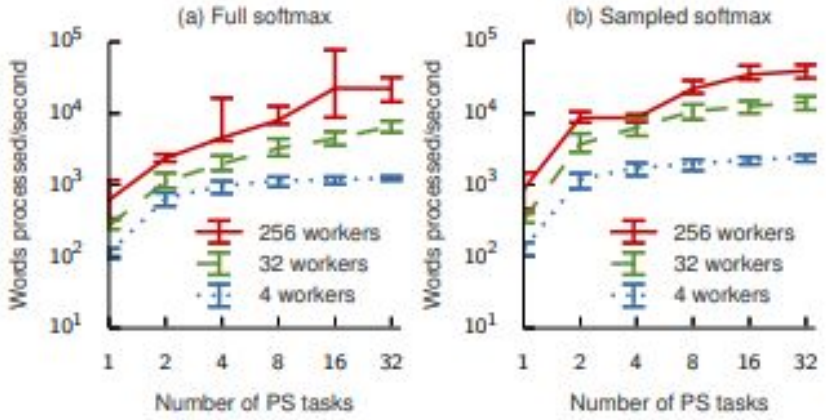


Figure 9: Increasing the number of PS tasks leads to increased throughput for language model training, by parallelizing the softmax computation. Sampled softmax increases throughput by performing less computation.

Discussion

Question 1:

Eager execution vs Lazy Execution. Which is better and when? What are the advantages and disadvantages?

Answer 1:

Eager execution:
i) No need for session.

ii) More Pythonic way, ease of use.

iii) Performance is slower.

iv) Pytorch uses similar dynamic graph approach -> Imperative programming.

TensorFlow2.0 Eager Execution is implemented by default therefore no longer need to create a session to run the computational graph, you can see the result of your code directly without the need of creating Session.

Question 2: Why integrate Spark with TF when there is a distributed TF framework?

Answer 2

The ideal Spark paradigm is to have every executor independently executing the same DAG on a subset of your dataset. MLLib also utilizes the RDD structure, providing methods for training large ML models like matrix factorizations over big RDD matrices.

To spin up multiple, independent TensorFlow jobs for hyperparameter tuning. (one way to think is training different versions of the model.)

Question 3:

When to use Spark and when to use Tensorflow?

Answer 3:

Spark is more suitable for data processing.

TF is more suitable for Machine Learning.

Question 4:

Tensorflow vs Pytorch: Ease of use. Which is easier to use as an application developer?

Answer 4:

Pytorch is more pythonic, and python is widely used for programming deep neural networks.

Tensorflow may feel like a new language for the developer. (as per the papers version 1.0)