FAQs

- Quiz #3-5
- Consider a Cassandra storage cluster with 5 storage nodes (A, B, C, D, and E) and an identifier of length m=4. The replication factor of this storage cluster is 1. Suppose that you use a hash function $h(x) = x \mod 16$. For the storage node, the system uses the least-significant byte of the IPv4 address as the input to the hash function. For example, for a node with the IPv4 address, 120.90.3.11, the hash output will be $h(11) = 11 \mod 16 = 11$. The IPv4 addresses of the nodes are the following:
  - A: 120.90.3.11
  - B: 120.90.3.3
  - C: 120.90.3.16
  - D: 120.90.3.39
  - E: 120.90.3.46
Q#3-6

• Create the finger table for the node C.

Q#3-7

• If the node C received a query to retrieve data with the key, k = 31, how many nodes should be visited to retrieve the matching data? Include the node C.
Q#3-8

- Assume that node F is added to this cluster. The IPv4 address of F is 120.9.0.5. How many nodes should update their finger tables? Include the new node F.
This material is built based on

- Jeffrey Dean and Greg S. Corrado and Rajat Monga and Kai Chen and Matthieu Devin and Quoc V. Le and Mark Z. Mao and Marc’Aurelio Ranzato and Andrew Senior and Paul Tucker and Ke Yang and Andrew Y. Ng, Large Scale Distributed Deep Networks, 2012, NIPS

- Martin Abadi and Paul Barham and Jianmin Chen and Zhifeng Chen and Andy Davis and Jeffrey Dean and Matthieu Devin and Sanjay Ghemawat and Geoffrey Irving and Michael Isard and Manjunath Kudlur and Josh Levenberg and Rajat Monga and Sherry Moore and Derek G. Murray and Benoit Steiner and Paul Tucker and Vijay Vasudevan and Pete Warden and Martin Wicke and Yuan Yu and Xiaoqiang Zheng, TensorFlow: A system for large-scale machine learning, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016
Revisit Neural Networks with a handwriting recognition example

- Training your model with a large number of handwritten digits to recognize them

Perceptron

- A perceptron takes several binary inputs, $x_1, x_2, x_3 \ldots$ and produces a single binary output

- Weights $w_1, w_2, w_3 \ldots$, real numbers expressing the importance of the respective inputs

\[
\text{output} = \begin{cases} 
0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\
1 & \text{if } \sum_j w_j x_j > \text{threshold} 
\end{cases}
\]
Perceptron with layers

First layer of perceptron

Second layer of perceptron

Recognizing individual digits

• Suppose we have a multilayer perceptron (MLP)

• “Hidden” layer: neither inputs nor outputs

Simple example: “Recognize number “9”!”

- Encoding the intensities of the image pixels into the input neurons:
  - e.g., if the image is a 64 by 64 greyscale image, then 4,096 = 64 x 64 input neurons with the intensities scaled between 0 and 1

- Output layer:
  - Less than 0.5: input image is not a “9”
  - Greater than 0.5: input image is a “9”
Recognizing individual digits

Training with SGD

- Suppose that our input images are **28 x 28 dimensional vector**
- Output will be 10 dimensional vector
  - For digit “6”, $y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$
- The training algorithm should find weights and biases
  - So that the output from the network approximates $y(x)$ for all training inputs $x$
- Cost function
  \[ C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2. \]
  - Here, $w$ denotes the collection of all weights and $b$ all the biases. $n$ is the total number of training inputs and $a$ is the vector of outputs from the network when $x$ is input
GEAR Session 2. Machine Learning for Big Data
Lecture 3. Distributed Neural Networks
Asynchronized Parallel Optimization
DistBelief

Distributed optimization and Synchronization schemes

• **Large dataset for training**
  - Not only training within a single instance of model
  - Distribute training across multiple model instances

• **SGD**
  - One of the most popular optimization procedure for training deep neural network

• The traditional formulation of SGD is inherently sequential
  - Impractical to apply to very large data sets
  - The time required to move through the data in an entirely serial fashion is prohibitive
Distributed optimization and Synchronization schemes

- **Bounded synchronous parallel (BSP)**
  - Computation phase is followed by a synchronization phase

- **Asynchronous parallel (ASP)**
  - ASP allows local computation to complete as many as possible
  - The central server will not block individual worker that completed their computation

- **Stale synchronous parallel (SSP)**
  - The system determine whether a model instance may perform computation, based on how far it falls behind the fastest model instance

---

DistBelief: Model Parallelism [1/2]

- **DistBelief**
  - Very large deep networks
  - Distributed computation in neural networks and layered graphical models

- **User should define**
  - Computation at each node in each layer of the model
  - Messages passed during the upward and downward phases of computation

- **Framework**
  - Automatically parallelizes computation in each machine using all available cores
  - Manages communication
  - Synchronization
  - Data transfer between machines
DistBelief: Model Parallelism [2/2]

- Performance benefits of distributing a deep network across multiple machines
  - Connectivity structure
  - Computational needs of the model
- Less-than-ideal speedups
  - Variance in processing times across the different machines
  - Waiting for the single slowest machine to finish a given phase of computation

  - A five layer deep neural network with local connectivity partitioned across four machines (blue rectangles)
  - Edges that cross partition boundaries (thick lines) will need to have their state transmitted between machines
  - For the nodes with multiple edges crossing a partition boundary, communication will be bundled
  - Within each partition, computation for individual nodes will the parallelized across all available CPU cores

DistBelief: Distributed Optimization Algorithms

- Second level parallelism
  - Large dataset for training
  - Not only training within a single instance of model
  - Distribute training across multiple model instances
- Two Large-scale Optimization Procedures
  - Downpour SGD: Online method
  - Sandblaster L-BFGS: Batch method
- Common aspects
  - Centralized sharded parameter server
    - Model replicas use to share their parameters
  - Take advantage of the distributed computation within each individual replica
  - Tolerate variance in the processing speed of different model replicas
    - Even the failure of model replicas which may be taken offline or restarted at random
GEAR Session 2. Machine Learning for Big Data
Lecture 3. Distributed Neural Networks
Asynchronized Parallel Optimization

DistBelief: 1. Downpour SGD

Downpour SGD

- Objectives of Downpour SGD
  - Applying SGD to extremely large datasets

- A variant of asynchronous stochastic gradient descent
  - Uses multiple replicas of a single DistBelief model

- This approach is asynchronous in two distinct aspects
  - The model replicas run independently of each other
  - The parameter server shards also run independently of one another
<table>
<thead>
<tr>
<th>Downpour SGD</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Divide the training data into a number of subsets and run a copy of the model on each of these subsets</td>
</tr>
<tr>
<td>• The models communicate updates through a centralized parameter server</td>
</tr>
<tr>
<td>• Before processing each mini-batch</td>
</tr>
<tr>
<td>• Keeps the current state of all parameters for the model</td>
</tr>
<tr>
<td>• Shards parameters across many machines</td>
</tr>
<tr>
<td>• e.g., if we have 10 parameter server shards, each shard is responsible for storing and applying updates to 1/10th of the model parameters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Downpour SGD</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Step 1:</strong> Before processing each mini-batch, a model replica asks the parameter server service for an updated copy of its model parameters</td>
</tr>
<tr>
<td>• Each machine needs to communicate with just the subset of parameter server shards that hold the model parameters relevant to its partition</td>
</tr>
<tr>
<td>• <strong>Step 2:</strong> After receiving an updated copy of its parameters, the DistBelief model replica processes a mini-batch of data to compute a parameter gradient</td>
</tr>
<tr>
<td>• <strong>Step 3:</strong> Sends the gradient to the parameter server</td>
</tr>
<tr>
<td>• Then, applies the gradient to the current value of the model parameters</td>
</tr>
</tbody>
</table>
Reducing the communication overhead of Downpour SGD

- Limit each model replica to request updated parameters only every $n_{fetch}$ steps and send updated gradient values only every $n_{push}$ steps
  - where $n_{fetch}$ might not be equal to $n_{push}$

- Traditional distributed SGD
  - $n_{fetch} = n_{push} = 1$

Fault tolerance with Downpour SGD

- More robust to machines failures

- Synchronous SGD with failures
  - If one machine fails, entire training process is delayed

- Asynchronous SGD with failures
  - If one machine fails, other model replica will continue processing training and updating model parameters
Stochasticity with Downpour SGD

- With DistBelief,
  - no guarantee that:
    - There were the same number of updates in the each shard of parameters
    - There were updates applied in the same order
  - There are subtle inconsistencies in the timestamps of parameters
  - Relaxing consistency requirement is effective to enhance stochasticity

Improving Downpour SGD with Adagrad

- Adagrad
  - Adaptive learning rate procedure
  - Let $\eta_i$ be the learning rate of the $i$-th parameter at iteration $K$ and $\Delta w_{i,x}$ its gradient
  - We set: $\eta_i, K = \frac{\gamma}{\sqrt{\sum_{j=1}^{K} \Delta w_{i,j}^2}}$

- These learning rates are computed only from the summed squared gradients of each parameter
- Adagrad can be implemented locally within each parameter server shard
Improving Downpour SGD with Adagrad

\[ \eta_{i,K} = \frac{\gamma}{\sqrt{\sum_{j=1}^{K} \Delta w_{i,j}^2}} \]

- The value of \( \gamma \)
  - The constant scaling factor for all learning rates
  - Generally larger than the best fixed learning rate used without Adagrad

- The use of Adagrad
  - Extends the maximum number of model replicas that can productively work simultaneously
  - Eliminates stability concerns
    - Combined with a practice of “warmstarting” model training with only a single model replica before unleashing the other replicas

---

GEAR Session 2. Machine Learning for Big Data
Lecture 3. Distributed Neural Networks
Asynchronized Parallel Optimization
DistBelief: 2. Sandblaster L-BFGS
Sandblaster L-BFGS

- Batch methods works well in training small deep networks

- Sandblaster
  - Batch optimization framework
  - Implementation of L-BFGS

- Distributed parameter storage and manipulation

Sandblaster L-BFGS: Coordinator

- Coordinator based system architecture
  - No direct access to the model parameters
  - Issues commands drawn from a small set of operations
    - e.g., dot product, scaling, coefficient-wise addition, multiplication
  - Performed by each parameter server shard independently
  - Results are stored locally on the same shard
Sandblaster L-BFGS: Parameter Server

- Performs **small set of operations**
- Stores results from the local computation
- Stores additional information
  - History cache
  - Provides scalability for the large (e.g. billion parameters) scale learning without incurring the communication overhead
  - To a single central server

Sandblaster L-BFGS: Load balancing

- Typical parallelized implementations of L-BFGS
  - Local machines are responsible for computing the gradient on a subset
  - Results are sent back to the central server
    - Or aggregated via a tree structure

- **Performance is limited by the latency of stragglers!**
Sandblaster L-BFGS: Load balancing

- Coordinator assigns each of the $N$ model replicas a small portion of work
  - Much smaller than $1/N$th of the total size of a batch
  - Assigns replicas new portions whenever they are free

- Faster model replicas do more work than slower replicas

- At the end of a batch
  - Coordinator schedules multiple copies of the outstanding portions
  - Uses result from whichever model replica finishes first

Sandblaster L-BFGS vs. Downpour SGD

- Downpour SGD
  - Requires relatively high frequency, high bandwidth parameter synchronization with the parameter server

- Sandblaster L-BFGS
  - Workers only fetch parameters at the beginning of each batch
    - updated by the coordinator
  - Workers only send the gradients every few completed portions
    - To protect against replica failures and restarts
Performance Evaluation

- Left: **Training accuracy** (on a portion of the training set) for different optimization methods
- Right: **Classification accuracy on the hold out test set** as a function of training time
  - Downpour and Sandblaster experiments initialized using the same ∼10 hour warm-start of simple SGD
TensorFlow: Successor to DistBelief

- DistBelief has been used in Google since 2011
  - Parameter server architecture
  - User defines a neural network as a directed acyclic graph of layers that terminates with a loss function
- fully connected layer
  - Multiplies its input by a weight matrix, adds a bias vector, and applies a non-linear function (such as a sigmoid) to the result
  - The weight matrix and bias vector are parameters
- A loss function is a scalar function
  - Quantifies the difference between the predicted value (for a given input data point) and the ground truth

TensorFlow: Why?

- High-level programming model
  - Customize the code that runs in all parts of the system
  - Adapting/Experiment with new optimization algorithms and model architectures
- Defining new layers
  - Implemented DistBelief layers as C++ classes
- Refining the training algorithms
  - SGD update rules
- Defining new training algorithms
  - DistBelief was suitable for the simple feed-forward style NNs
  - Many of Neural Networks were not applicable
    - e.g. RNN, Adversarial networks, reinforcement learning
- Support GPU acceleration
- Support scale down
  - E.g. Single GPU-powered workstation for development
TensorFlow: Design Principles

- A simple dataflow-based programming abstraction
  - Users can deploy applications on distributed clusters, local workstation, mobile devices

- Dataflow graphs of primitive operators
  - Both TensorFlow and DistBelief use a dataflow representation
  - DistBelief model comprises relatively few complex “layers”
  - TensorFlow model represents individual mathematical operators
    - Such as matrix multiplication, convolution, etc
    - Users can compose their new application easily

TensorFlow: Design Principles

- TensorFlow uses a single dataflow graph to represent all computation and state

- Compared to the batch dataflow systems, e.g. MR, DryLINQ
  - The model supports multiple concurrent executions on overlapping subgraphs of the overall graph
  - Individual vertices may have mutable state that can be shared between different executions of the graph

- Dataflow with mutable state enables TensorFlow to mimic the functionality of a parameter server
  - With additional flexibility

- Phase 1: defining the program (e.g. neural network to be trained and update rules)
- Phase 2: executing an optimized version of the program on the set of available devices
  - Phase 2 (execution phase) is optimized by deferring this phase until the entire program is available
    - e.g. issues a sequence of kernels to the GPU using the graph’s dependency without waiting for intermediate results
  - High GPU utilization
TensorFlow: Design Principles

- Common abstraction for heterogeneous accelerators
  - To incorporate available accelerators
  - Tensor Processing Unit (TPU)
    - Designed specifically for machine learning
    - TPUs yield an order of magnitude improvement in performance-per-watt compared to alternative state-of-the-art technology

- Methods supported by device
  - Issuing a kernel for execution,
  - Allocating memory for inputs and outputs
  - Transferring buffers to and from host memory

- Each operator (e.g., matrix multiplication) can have multiple specialized implementations for different devices

TensorFlow: Dataflow graph elements

- Operations
  - Computation at vertices

- Tensors
  - Values that flow along edges
### Tensors

- Data is modeled as tensors \((n\text{-dimensional arrays})\)
  - Elements having one of a small number of primitive types
  - \text{Int32}, \text{float32}, \text{or} \text{string}
  - E.g. a matrix multiplication with two 2-D tensors, produces a 2-D tensor

- At the lowest level, all TensorFlow tensors are dense
- Representing sparse tensors
  - Variable-length string elements of a dense tensor
  - A tuple of dense tensors
    - \(n\)-D sparse tensor with \(m\) non-zero elements can be represented in coordinate-list format as \(m \times n\) matrix
    - a length-\(m\) vector of values

### Operations

- Takes \(m \geq 0\) tensors as input
- Produces \(n \geq 0\) tensors as output
- An operation has a named “type”
  - May have zero or more compile-time attributes that determine its behavior
    - e.g. \text{Const}, \text{MatMul}, or \text{Assign}
    - e.g. \text{Const}
      - The simplest operation
      - No inputs and a single output
    - e.g. \text{AddN}
      - Sums multiple tensors of the same element type, and it has a type attribute \(T\) and an integer attribute \(N\) that define its type signature
Stateful operations: variables

- Operation with mutable state
- Variable operation
  - Maintains a mutable buffer
  - Store the shared parameters of a model as it is trained
  - No input
  - Produces a reference handle
    - Reading and writing the buffer

Stateful operations: queues

- FIFOQueue
  - an internal queue of tensors
  - concurrent access in first-in-first-out order
  - Produces reference handle
    - Consumed by operations such as Enqueue or Dequeue
Distributed Execution

- Dataflow simplifies distributed execution
  - Communication between subcomputations becomes explicit
  - Abstracts the computation across the heterogeneous cluster
  - Each operation resides on a particular device
    - *CPU or GPU in a particular task*

- TensorFlow runtime
  - places operations on devices, subject to implicit or explicit constraints in the graph
  - computes a feasible set of devices for each operation
  - calculates the sets of operations that must be co-located
  - selects a satisfying device for each colocation group

Distributed Execution: Optimization

- For a graph pruned, placed, and partitioned
  - sub- graphs are cached in their respective devices
  - A client session maintains the mapping from step definitions to cached subgraphs
  - A distributed step on a large graph can be initiated with one small message
Questions?