Robust Scheduling and Elastic Scaling of Deep Learning Workloads

Jayaram K. R.

IBM Research AI, New York, USA

DIDL Workshop, Middleware 2018

Joint work with

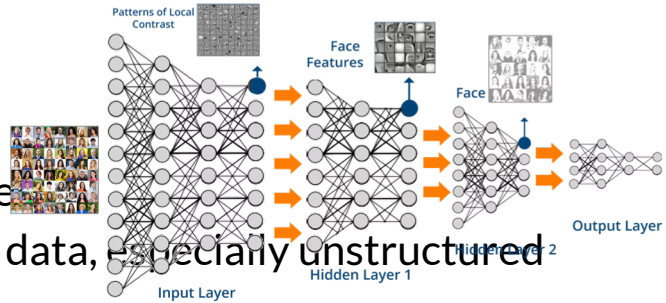
Scott Boag, Vinod Muthusamy, Benjamin Herta, Waldemmar Hummer, Vatche Ishakian, Michael Garod, Archit Verma, Parijat Dube, Atin Sood and Rania Khalaf

Diana Arroyo, Asser Tantawi, Stefania Costache and Alaa Youssef

Yogish Sabharwal, Ashish Verma, Saurav Basu and Vaibhav Saxena

Motivation[1]

- Popularity of deep learning in several fields
  - Ability to learn features in an unsupervised manne
  - Availability and ability to collect large amounts of data, especially unstructured data
  - Recent improvements to GPU technologies
  - Advances in interconnection technologies (NVLink
  - Easy-to-use open source deep-learning framework
    - Caffe, Caffe2, Torch, Tensorflow, etc.



E, etc

Caffe2

PyTorch

TensorFlow

Need for scalable and robust deep learning platforms

- A large organization with a private data center
  - Multiple teams, frameworks and application domains
  - Goals:
    - Make effective use of expensive hardware
    - Run deep learning workloads in a robust and secure manner
    - Avoid repetitive work, and situations where each team has to set up and maintain its own deep-learning “software stack”
    - Reduce barrier to entry : data scientists should focus on their algorithms, data, hyperparameter optimization, etc. and not on installations, maintenance, failure handling etc.
- A cloud provider
  - Enable small, medium and large businesses to address said goals

This Talk

Robust Scheduling and Elastic Scaling of Deep Learning Workloads

- Context: A deep learning platform developed and used at IBM Research
  - DLaaS : Deep Learning as a Service
    - Released March 20, 2018
    - Part of IBM Watson Studio, available on IBM Cloud
    - https://www.ibm.com/cloud/deep-learning (trials are free)
  - FfDL : Fabric for Deep Learning
    - Open source release of major portions of DLaaS
    - https://github.com/IBM/FfDL
- Built by composing several open-source technologies
  - Simple concepts (compared to academic papers)
  - Simplicity → Maintainability

DLaaS : Key Challenges

- Training jobs typically run continuously for 1-7 days
  - Make several passes over a large data set (several TB)
  - Consequence of failure is significant (loss of several days of work)
  - Need (user configurable/directed) reliable checkpointing
- GPU-heavy
  - Designed to maximize GPU utilization
  - Hardware failures (reboots, bad GPUs) in DL clusters are more common than other clusters
- Impose a heavier load on the datacenter network
- Job deployment is not instantaneous
- Users need reliable status updates (e.g., QUEUED, DOWNLOADING, FAILED)
- Reliable streaming of logs during training
- Isolation (multi-tenancy)
- Resilience to node and job crashes (with reliable notifications)

DLaaS : Goals

- Horizontal scalability
- Flexibility -- supports popular DL frameworks; like programming languages data scientists have an affinity towards frameworks
- Dependability -- highly available, robust (timely, handle hardware and software faults), secure and maintainable
- Efficiency -- overheads introduced to achieve (above) goals and response time to external requests should be minimal
- Elasticity -- user driven and system-driven
- Priorities and pre-emption

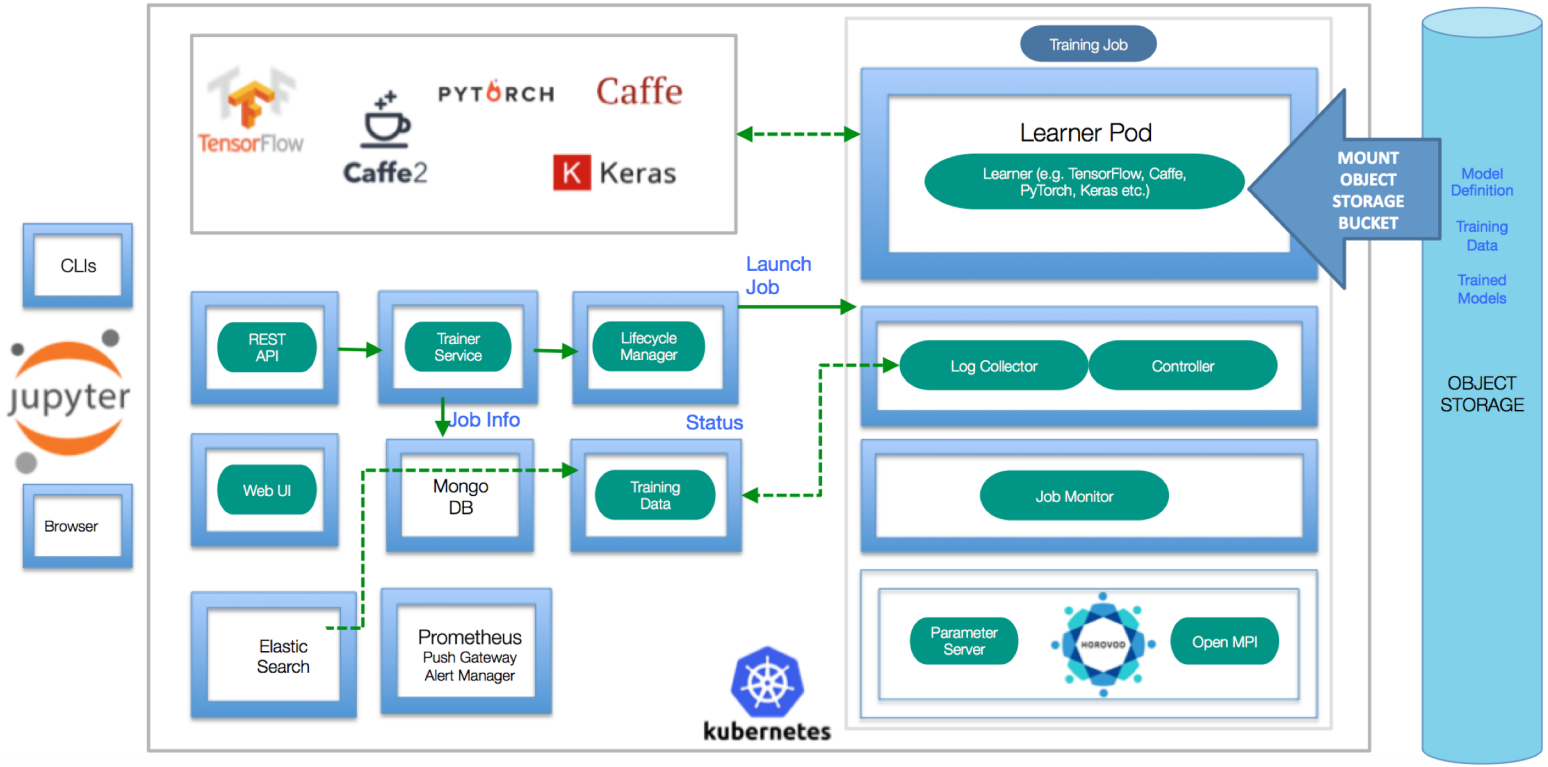
Outline

1. Motivation and Goals of DLaaS/FfDL
2. Architecture
3. Scheduling
4. Elastic Scaling
5. Lessons learned and future research

A Training Job

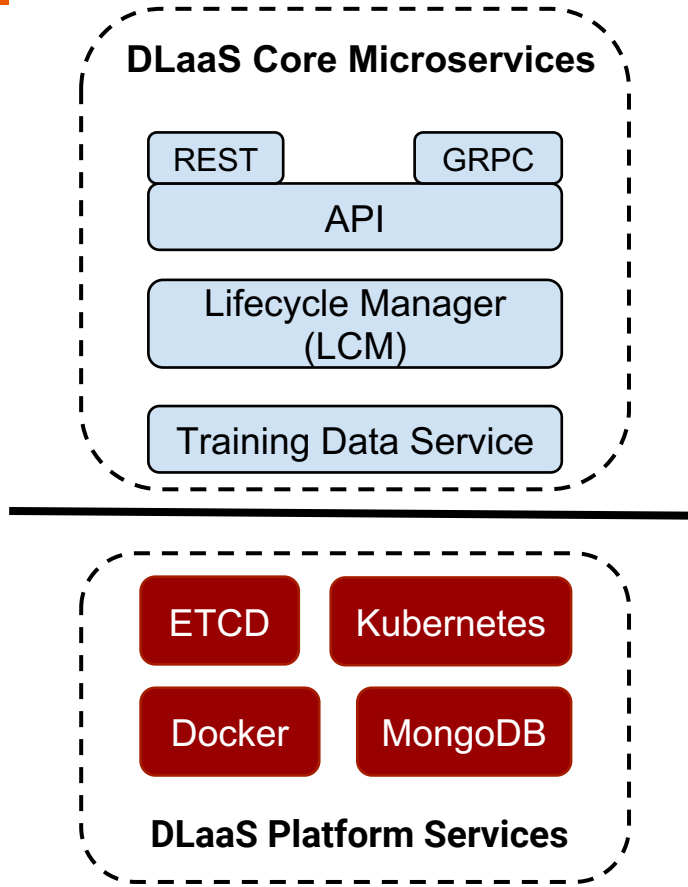
- Consists of several “training”/”learning” processes, each using GPUs and synchronizing over MPI or by using parameter servers
- DLaaS view : a set of Docker containers instantiated using a manifest file
  - Docker images corresponding to popular DL frameworks
  - DLaaS instantiates docker images with user code to create the training job
  - Each learning process → a DLaaS learner
  - Manifest file : framework to use, #CPUs, #GPUs, RAM, location of training data/checkpoints/results, credentials to access said locations, etc.
- Isolation and Confidentiality
  - Using Docker containers
  - Policies on network traffic to/from training jobs
  - End-to-end encryption of data transferred to the training job and model parameters during synchronization

DLaaS/FfDL Architecture

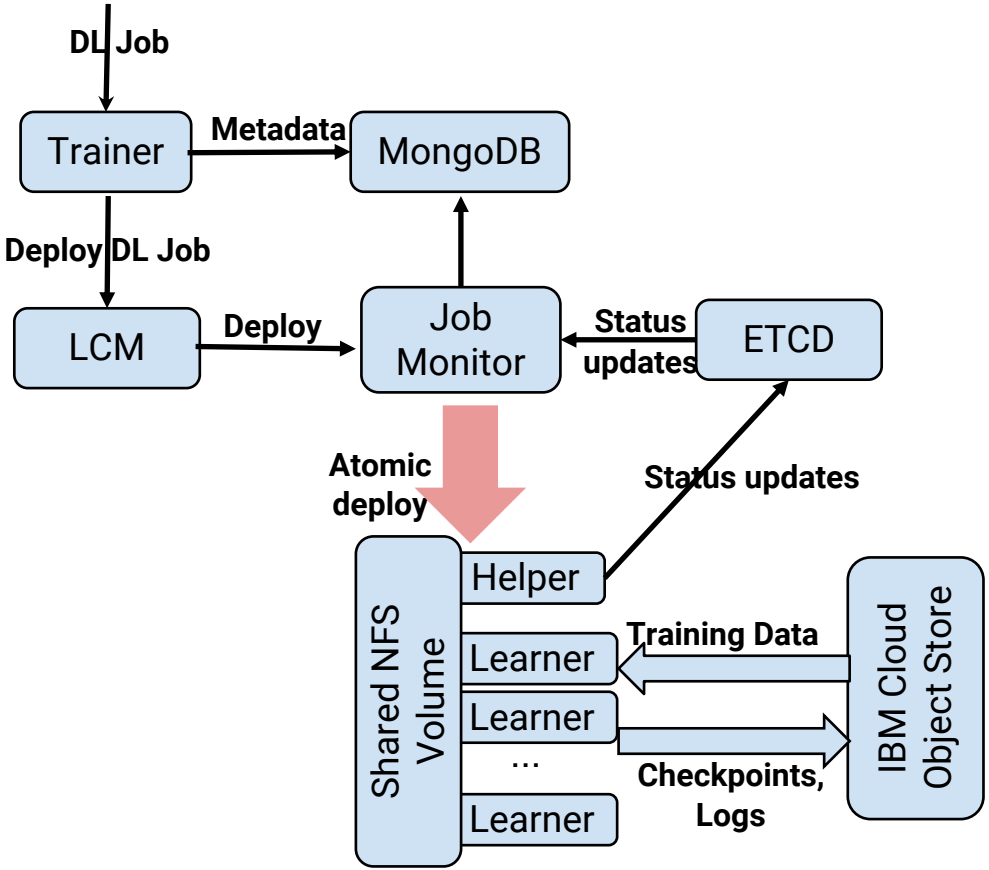


DLaaS Architecture

- Middleware
  - Above cluster manager (Kubernetes)
- Loosely coupled microservices
- GRPC
- Kubernetes for cluster management
  - Docker containers for training jobs encapsulated using stateful sets
  - Ordered start, guaranteed restarts
- ETCD for coordination



DLaaS Job Deployment and Management



Lifecycle Manager (LCM)

- Responsible for Creating, Deleting and Halting a Job
- Creating a training Job by interacting with Kubernetes
  - Job Monitor to babysit the job
  - Learners
  - Helper pod -- controller, log collector, download-data, store-results
  - Persistent volume claims
- Controller
  - Direct monitoring of learners through shared NFS
  - Updates status to ETCD (DOWNLOADING, PROCESSING, STORING, COMPLETED, FAILED)
  - Monitors exit state of learning process (Caffe, etc.)

Job Monitor

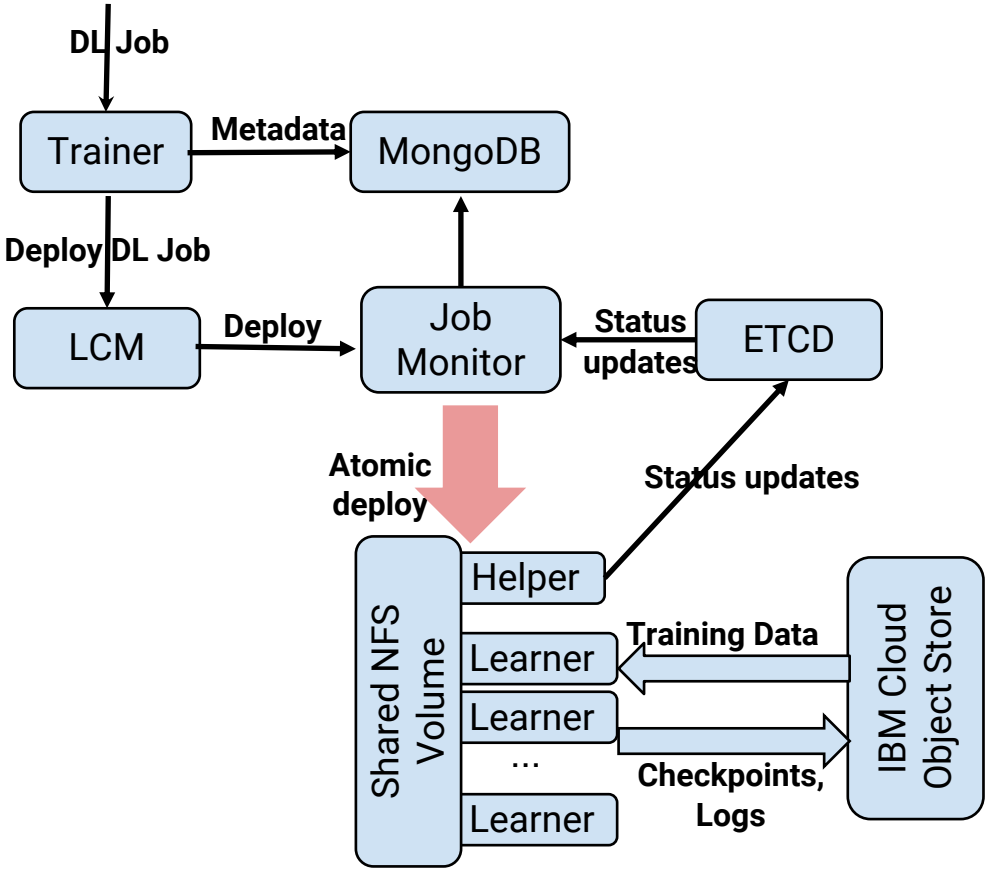
- 1 Job Monitor per Job
- Monitors the status of the learner pods by talking to Kubernetes
  - Image pull errors
  - Volume mount errors
  - Insufficient resources
  - Pods stuck in Container Creating/Terminating
- Monitors the status updates from Controller in ETCD
  - Reads status from each learner, aggregates status and updates status in Mongo through a Trainer API call
- Responsible for initiating garbage collection

Job Deployment in DLaaS

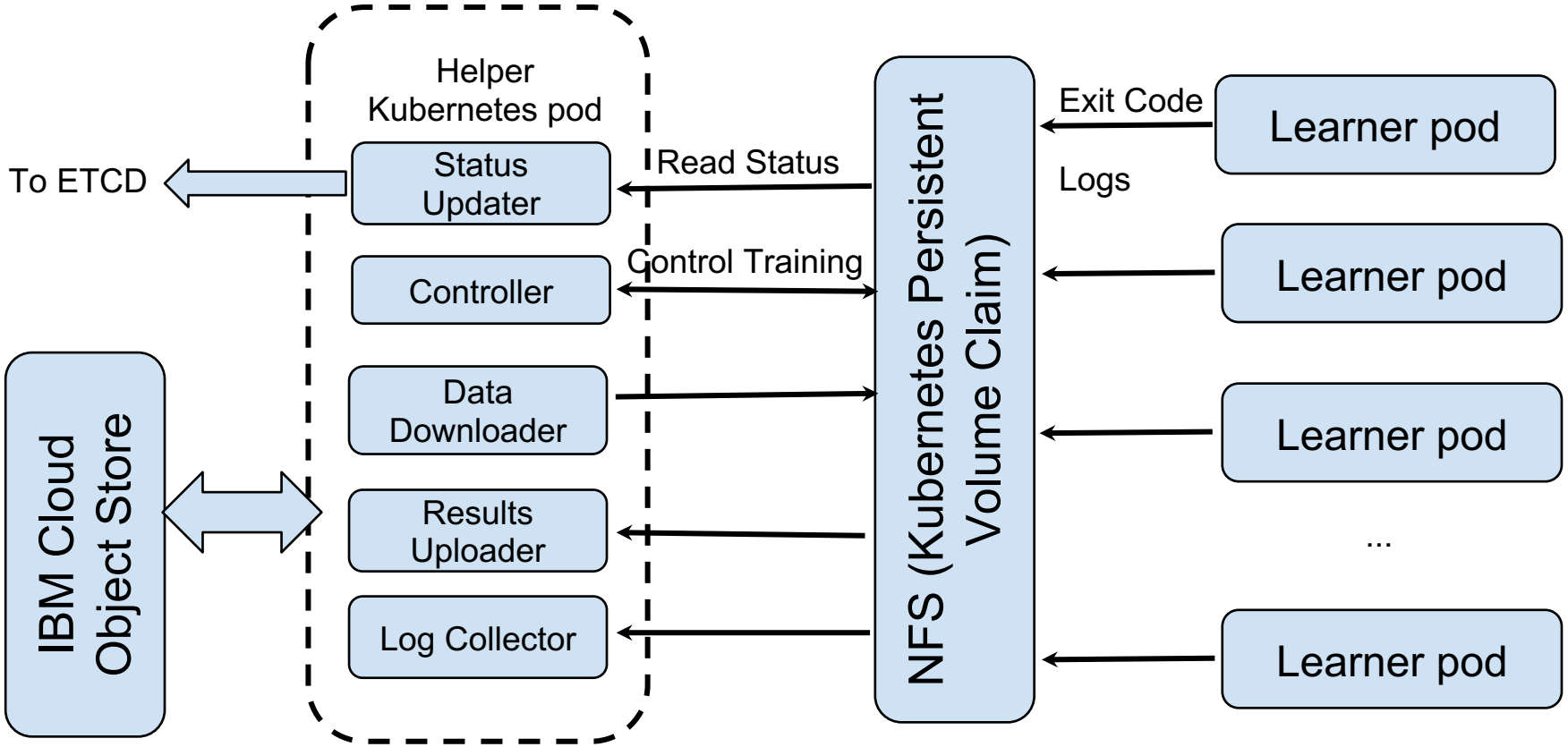
- Not instantaneous
  1. Create Network policies
  2. Create Secrets for data access
  3. Create learners as stateful sets
     - Ordered deployment of learners
     - N learners : learner-0...learner-n-1
     - Guaranteed restart upon crash failure
- LCM is 3 way replicated

- Q: What happens if a replica crashes in the middle of the steps outlined above?

DLaaS Job Deployment and Management



Helper


Helper
Kubernetes pod
To ETCD
Status
Updater
Read Status
Controller
Control Training
Data
Downloader
Results
Uploader
Log Collector
IBM Cloud
Object Store
NFS (Kubernetes Persistent
Volume Claim)
Exit Code
Logs
Learner pod
Learner pod
Learner pod
...
Learner pod

Overhead (vs. Bare Metal)

Benchmark	Framework	#PCIe GPUs	GPU type	Decrease in Performance
VGG-16	Caffe	1	K80	**3.29%**
VGG-16	Caffe	2	K80	**0.34%**
VGG-16	Caffe	3	K80	**5.88%**
VGG-16	Caffe	4	K80	**5.2%**
InceptionV3	Tensorflow	1	K80	**0.32%**
InceptionV3	Tensorflow	2	K80	**4.86%**
InceptionV3	Tensorflow	3	K80	**5.15%**
InceptionV3	Tensorflow	4	K80	**1.54%**

Overhead (vs. NVIDIA DGX-1)

Benchmark	Framework	#PCIe GPUs	GPU type	Decrease in Performance
InceptionV3	Tensorflow	1	P100	**3.30%**
Resnet-50	Tensorflow	1	P100	**7.07%**
Resnet-152	Tensorflow	1	P100	**8.13%**
VGG-16	Tensorflow	1	P100	**7.84%**
InceptionV3	Tensorflow	2	P100	**10.06%**
Resnet-50	Tensorflow	2	P100	**10.53%**
Resnet-152	Tensorflow	2	P100	**12.29%**
VGG-16	Tensorflow	2	P100	**13.69%**

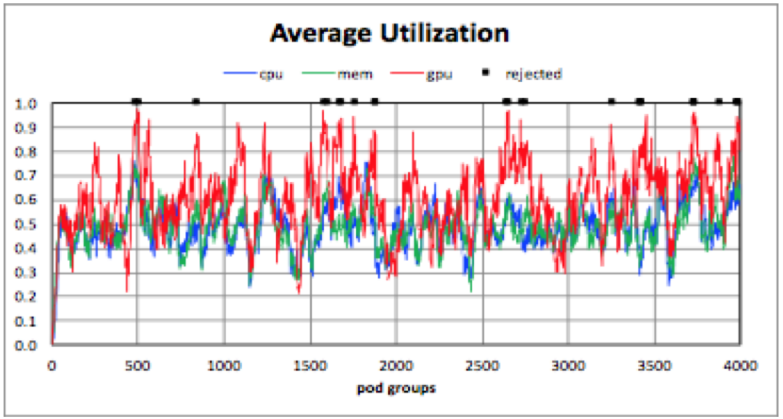
Recovery Times

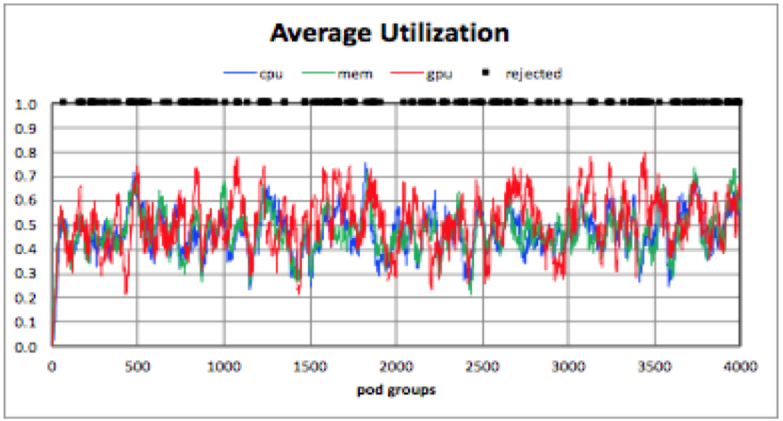
Component	Recovery Time
API	3-5s
LCM	4-6s
Guardian	1-2s
Helper	3-4s
Learner	10-20s

Scheduling Deep Learning Jobs

- Kubernetes default scheduler
  - Scheduling at the pod level
  - FCFS
- PACK
  - Pack components of deep learning jobs (i.e. kubernetes pods) into as few physical servers as possible
  - 4 machines (4 GPUs each), a job with 2 learners (2 GPUs/learner) → 1 machine is used
- SPREAD
  - Opposite of PACK, ideal for replicated services
  - 4 machines (4 GPUs each), a job with 2 learners (2 GPUs/learner) → 2 machines are used
- Locality awareness
  - Try not to place across racks
  - Try not to place across network “areas”
  - Kubernetes labels

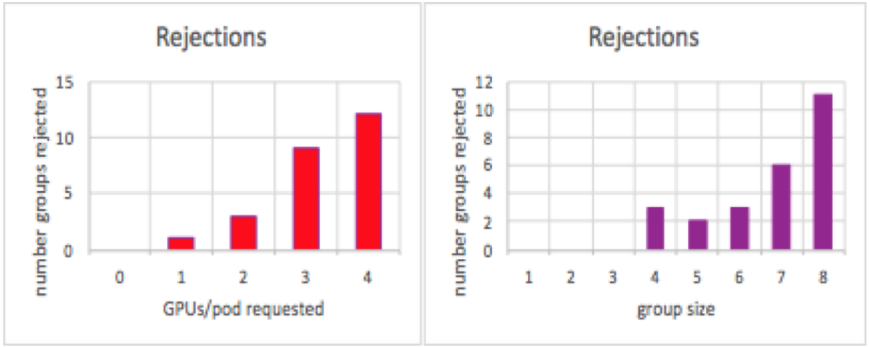
PACK vs. SPREAD

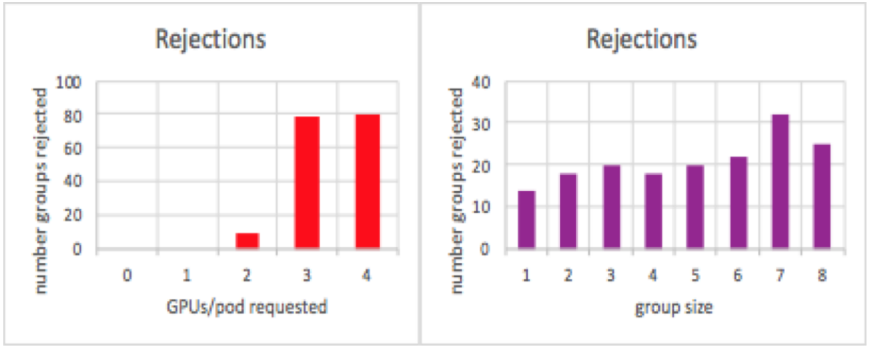




- 100 machines, 32 CPU cores, 4 K80 GPUs, 128GB RAM per machine
- 4000 Jobs
  - 1-4 GPUs/learner, avg 2.5
  - 4-16 CPUs/learner, avg 5
  - 1-8 learners/job, avg 4.5

PACK vs. SPREAD





- 100 machines, 32 CPU cores, 4 K80 GPUs, 128GB RAM per machine
- 4000 Jobs
  - 1-4 GPUs/learner, avg 2.5
  - 4-16 CPUs/learner, avg 5
  - 1-8 learners/job, avg 4.5

Gang Scheduling

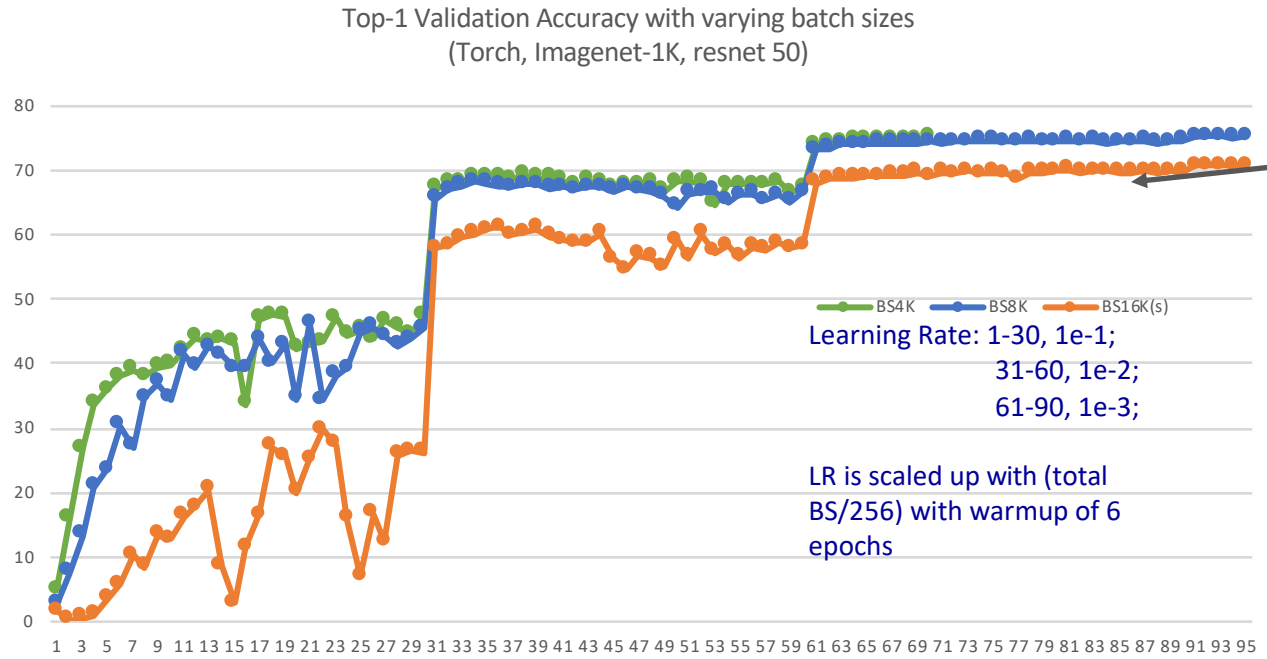
- Enhancement to Kubernetes default scheduler
- Scenario : Cluster has 8 GPUs, 4 jobs arrive, with 4 learners and 1 GPU/learner
  - Desired outcome : 2 jobs running, 2 jobs “pending”
  - Kubernetes default scheduler
    - 4 jobs with 2 learners/job running, 2 learners/job queued. Deadlock!
    - 4 jobs, 1st job 1 learner running, 2nd job 2 learners running, 3rd job 3 learners running, 4th job 2 learners running. Deadlock!
- Gang scheduling needed
  - A distributed deep learning job is a “gang” of learners
  - Either the whole gang should be scheduled or none at all
- Gang scheduling is different from atomic job deployment
- Atomic job deployment – all artifacts of a job, implemented above cluster manager
- Gang scheduling – learners only, implemented inside the cluster manager

Elasticity in Deep Learning

- Ability to dynamically scale-up or scale-down training resources (GPUs, CPUs)
- Need for elasticity:
  - User: Ability to complete the job faster by increasing the batch-size (using more resources) in the middle of training
  - System: maintain desired utilization levels, support job priorities, spot pricing

User-driven Elasticity : Motivation

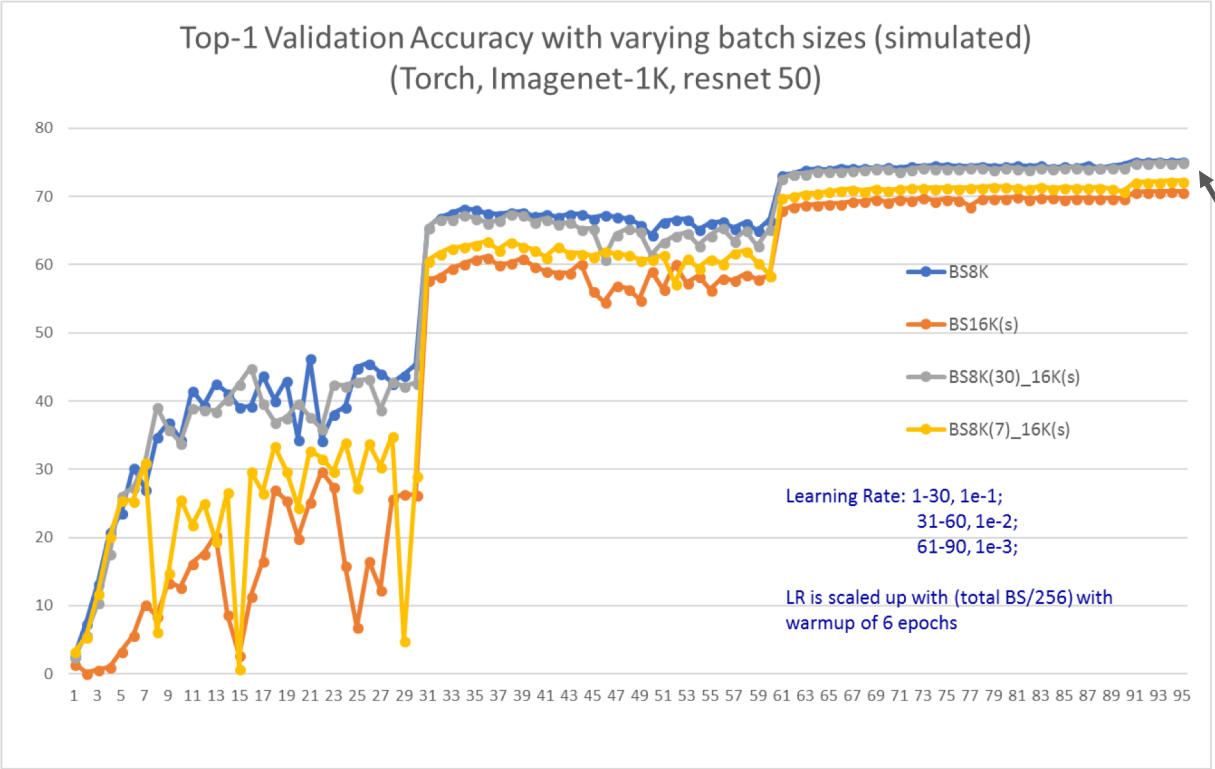
Most deep learning models yield poor accuracy when using a very large batch size



- Similar accuracy till 8K batch size
- Accuracy drops at 16K BS

Max Accuracy: BS4K (70 epochs) - **75.09%**, BS8K - **75.26%**, BS16K - **70.8%**

Larger batch sizes can be used after initial phase of training



- Run with 8K till 30 epochs, change/simulate to 16K after that.
- LR doubles as per 16K BS after 30 epochs

Observation:

- Doubling the BS at 31st epoch provides same accuracy.
- Accuracy drops when doubling after 7 epochs.

	BS8192	BS16K(s)	BS8K(30)_16K(s)	BS8K(7)_16K(s)
Max Accuracy	75.26	70.8	75.06	72.32

Reference: *Smith et al. paper, Don't Decay the Learning Rate, Increase the Batch Size, ICLR 2016* (https://openreview.net/forum?id=B1Yy1BxCZ)

User-driven Elasticity

- Static/Pre-defined
  - Larger batch size can be used (hence more resources can be employed) after certain # of epochs
  - Scenario: epochs 1-30 with <x GPUs, config1>, 30 to 60 with <y GPUs, config2>

```
Resume from last checkpoint
Setup hyper-parameters
For every epoch
    train batches for this epoch
    If (epoch == 30)
        module.call_as_restart(4 /*GPUs*/)
```

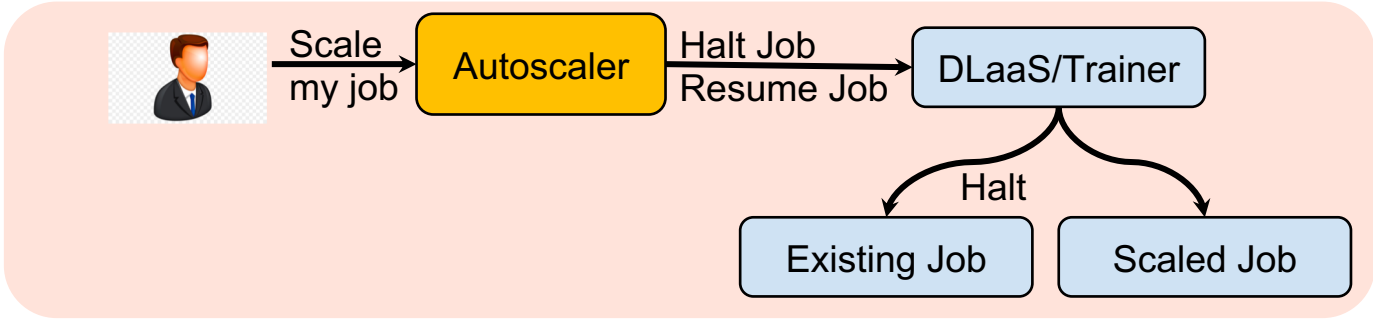
- Dynamic
  - User decides when to scale elastically through a UI/CLI command
  - Scenario: User analyzes logs and decides to change hyper-parameters on the fly

System-driven Elasticity -- Scenarios

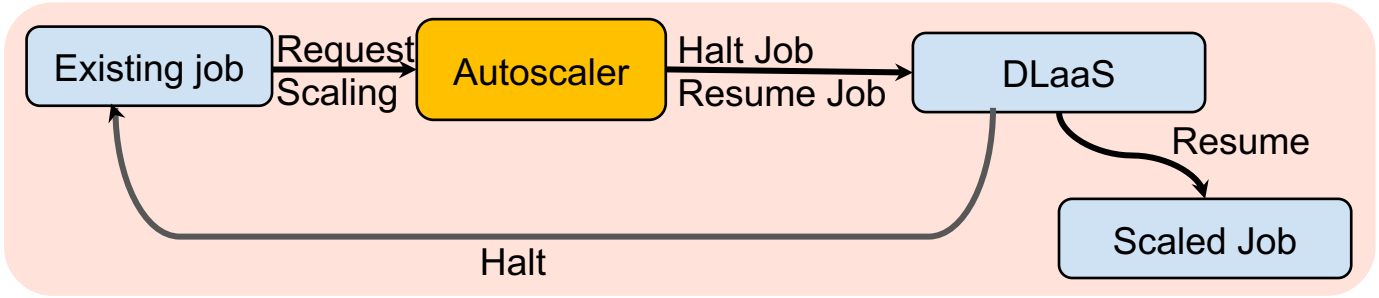
- Optimal cluster utilization
  - Cluster does not have spare capacity but maintenance needs to be performed → Elastically scale down jobs instead of terminating them
  - Automatically scale-up jobs when cluster is under-utilized
- Supporting priorities
  - Scale-down a lower priority job if a higher priority job arrives and resources aren't available
  - Currently, higher priority users may not be entertained if lower priority jobs are occupying the system
- Support *some* spot pricing models
  - When cluster is under-utilized, offer GPUs at a cheaper price if the user is willing to scale-down the job later on
- Increase flexibility while starting jobs
  - Start a job with smaller number of available GPUs and later scale as more GPUs become available

User-driven Elastic Scaling

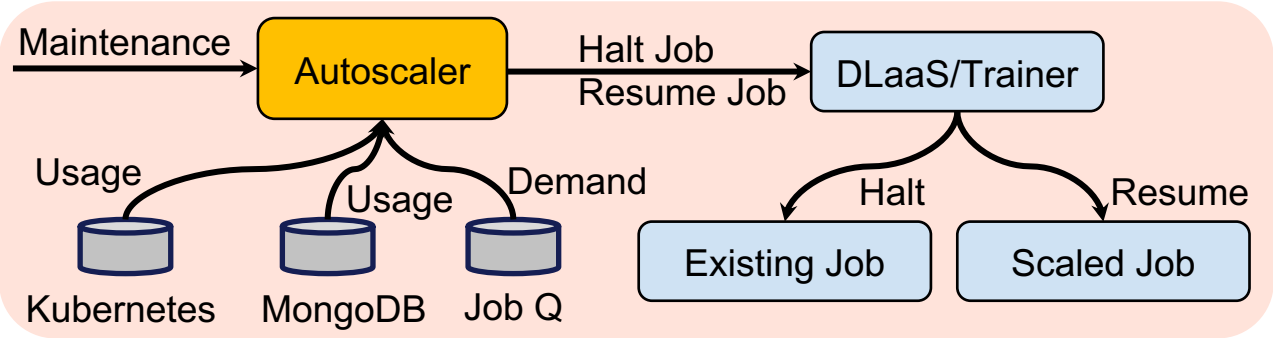
Dynamic



Static/Predefined



System-driven Elastic Scaling



Optimize cluster utilization

- Scale jobs up when cluster under-utilized (and vice-versa)
- min,max GPUs specified by user

Better handle planned maintenance and outages

- Avoid terminating jobs on nodes going down
- Scale jobs down instead

Handle priorities better [Ongoing work]

- Scale down jobs to admit higher priority jobs
- Choose jobs to scale-up based on priorities
- Interact with BSA scheduler and kube arbitrator

Impact on Users

- Requires (minimal) code changes from the user
  - Regular Checkpointing (which the users mostly use and are already familiar with)
  - For static/predefined scenario, user invokes simple functions by importing modules that hide details
- User specifies range of acceptable resources
  - Currently the user specifies number of GPUs; now the user will specify range of GPUs (min and max) per learner
  - System can schedule job with any number of GPUs between min and max
  - User code (Learner), upon startup queries number of resources allocated and sets up hyper-parameters accordingly
- Non-intrusive
  - Works without modifying any framework – can work with all DLaaS frameworks

Lessons Learned

- Cluster managers alone are insufficient to run DL workloads effectively
- DL workloads are similar in many ways but are also different in important ways to regular datacenter workloads (gang, no overcommitment, etc.)
- Users hate jobs being queued with no estimate of how long the job will remain in the queue
- Non-distributed DL workloads are not rare; distributed DL workloads are not large scale (< 10 learners)
- Futile to try and predict job arrival trends
- Simplicity is key to scalability, fault tolerance and maintainability of DL platforms
- Popular open source technologies (ETCD, Kubernetes, Docker, GoLang) can be helpful, but need to be augmented where necessary

Avenues for further research

- Priority and pre-emption while scheduling deep learning jobs
- Dynamic priority (based on number of jobs submitted)
- Priority + elasticity
- Runtime estimation and estimating the amount of time jobs remain in the queue
- “Smarter” scheduling and load prediction for user-driven static elasticity scenario