Hyper: Distributed Cloud Processing for Large-Scale Deep Learning Tasks

Davit Buniatyan
Snark AI, Menlo Park, CA 94025
Princeton University, Princeton, NJ 08544
e-mail: davit@snark.ai

ABSTRACT

Training and deploying deep learning models in real-world applications require processing large amount of data. This is a challenging task when the amount of data grows to hundred terabyte, or even, petabyte scale. We introduce a hybrid distributed cloud framework with a unified view to multiple clouds and an on-premise infrastructure for processing tasks using both CPU and GPU compute instances at scale. The system implements a distributed file system and failure-tolerant task processing scheduler, independent of the language or Deep Learning framework used. It allows to utilize unstable cheap resources on the cloud to significantly reduce costs. We demonstrate the scalability of the framework on running pre-processing, distributed training, hyperparameter search and large-scale inference tasks utilizing 10,000 CPU cores and 300 GPU instances with overall processing power of 2 petaflops.[1][2]

Keywords

Deep Learning, Cloud Computing, Distributed Systems

1. INTRODUCTION

Deep Learning (DL) based models have outperformed manual feature engineered algorithms in wide range of domains including computer vision, natural language processing, audio processing . The amount of labelled data required for training production ready models achieves a terabyte scale . The unlabelled data to execute those models reaches a petabyte scale . Such computational resources are on-demand available on cloud such as Amazon Web Services (AWS), Google Cloud Platform (GCP) or Azure.

Modern Deep Learning frameworks such as PyTorch, Tensorflow, MXNet, Theano and Caffe are well positioned for training and deploying models on single multicore machines with multiple GPUs or TPUs. Training state-of-the-art models on terascale data often takes weeks or months to converge. To make training faster, DL frameworks such as PyTorch or Tensorflow recently introduced synchronous and asynchronous distributed training methods to achieve almost linear speed up with respect to the number of nodes.

When the number of nodes in a distributed cluster grows, problems such as provisioning, orchestrating, fault-tolerance, distribution data management, task planning and execution arises. For executing Big Data workloads, number of widely accepted synchronous parallel processing systems have been introduced such as MapReduce [2], Apache Spark and Dryad [4]. Additionally, task-parallel frameworks are getting increased usage such as Dask [8]. They provide fine grained task management. These frameworks are very efficient for ETL tasks, but lack native deep learning support. Recent introduction to the family is Ray, which implements dynamic task-parallel framework that is suited for deep learning and reinforcement learning [7].

To manage High Performance Computing (HPC) workloads, container technology has recently become well suite choice for packaging environment libraries [12]. Frameworks such as Kubernetes enable massive distribution of stateless applications packaged in a container on a cluster of nodes in fault-tolerant way [1]. However, it still lacks efficient distributed data management unit, workflow scheduling system and support for stateful applications such as model training. Packages such as KubeFlow and Argo attempt to extend Kubernetes to support implementation of machine learning pipelines [3].

For data intensive workloads, variety of distributed file storage systems have been introduced [9] such as NFS [10] or HDFS [11]. NFS-based file systems significantly decrease multi-read speed and lower bound the compute speed. They often do not scale on multi-write scenarios. There is always trade-off between latency and scalability. For web-based applications, cloud providers offer object storage solution that has high scalability, but suffers in low-latency memory intensive operations.

We introduce a hybrid distributed cloud framework with unified view to multiple clouds and on-premise infrastructure for processing tasks. We make the following contributions

- We design and build distributed framework that unifies preprocessing, training, hyperparameter search and deploying large scale applications on the cloud.
- To achieve scalability we design and deploy distributed file system that has near-zero delay for deep learning jobs in comparison to downloading the data local on the machine with similiar performance.
- Provide fault-tolerance with respect to computational node failures and support utilization of unstable cheap resources on the cloud.

[1]Demo available at https://lab.snark.ai
[2]Documentation available at https://docs.snark.ai

- Abstract away cloud infrastructure setup and provisioning from the user with native cloud integration.

2. COMPUTATIONAL MODEL

In this section, we discuss design decisions made for the system and user interface to specify computation workload.

2.1 Computational Workflows

Workflow is a directed acyclic graph consisting of *Experiment* nodes and their dependency as edges. Single *Experiment* contains multiple *Tasks*. *Tasks* within the same experiment execute the same command with different arguments. Arguments can be templated for efficient parameter space definitions. Each *Experiment* has an associated container that is deployed on all computational workers.

Task is the execution unit, which encapsulates a process. Each *Task* has assigned *Node*, which represents the computation worker. Single *Node* might execute multiple *Tasks*. The number of nodes available corresponds to the number of workers inside the cluster.

2.2 Interface

Workflows are specified using code-as-infrastructure interface defined in YAML recipes as seen in Fig. 1. The recipe is parsed by the server and translated into directed acyclic graph of experiments. The interface lets users specify the environment, hardware settings, number of workers, parameters and parameterized commands. The user can interface the system through CLI or Web UI.

```
experiments:        # Provide list of experiments
  mnist:            # Name the experiment
    framework: pytorch # Or specify Docker name
    parameters:     # Define parameters
     lr:
       range: 0.1-0.3
       sampling: uniform
    samples: 1000   # Number of samples to draw
    workers: 100    # Number of workers
    hardware:       # Set hardware requirements
      gpu: k80
    command:        # Command executed
      - python train.py --lr {{lr}}
```

Figure 1: Example recipe that does hyperparameter search. User specifies DL farmework or docker container url, defines parameters, number of samples to be drawn from parameter space, number of workers inside a cluster, the hardware definition of the computational node and the parameterized command. The recipe is uploaded to REST endpoint and executed on the cloud.

2.3 Parameters

As seen in Fig. 1, user can specify the list of parameters that can be inserted into command arguments during execution. Parameters can be sampled from a discrete class or continuous range. To compute parameters for each *Task*, the algorithm generates Cartesian product of all discrete parameters and samples from the set n times with minimal repetition. n is defined to be the number of samples from a recipe. Then, it samples n times from each continuous parameter range and randomly matches with discrete sampled parameters. This is necessary to support both hyper-parameter search and inference with grid iterators.

3. SYSTEM OVERVIEW

The system receives data, chunks it and stores in an object storage. The recipe is submitted to deploy the deep learning workflow on a cluster of nodes, which mount the distributed file system.

3.1 Distributed Data Management

In order to be framework agnostic and require no further modification of the client program, we chunk the file system itself and store on an object storage provided by the cloud vendor (e.g. AWS S3). The system implementation is similar to closed source ObjectiveFS and tuned for deep learning tasks. The distributed file-system wraps POSIX api and acts as a middle layer with chunking, caching and state synchronization mechanisms across all nodes. When the program queries the file system for a specific file, the integration layer checks which chunk contains the file to download. On the next query the file system can check if the existing chunk contains the next required file before fetching it from the cloud. Within program context, files that are stored on the remote chunked object storage appear to be local files. Any deep learning application without further modification will take the advantage of the highly scalable storage.

Deep Learning frameworks such as PyTorch and TensorFlow natively support asynchronous data fetching from the local storage to the GPU using data loaders. Often deep learning training iteration is bounded by the compute cycles on GPUs. If one combines the distributed remote storage and asynchronous data fetching, the training speed is almost the same as if the data was stored locally on the machine with respect to following constraints.

- Network locality - We assume that computational nodes that access the data are physically located near object storage servers.
- Chunksize - well chosen with respect to latency to maximize the throughput. Should be in the range 25-100MB.

The suggested file system can leverage the scalability of the object storage and provide data access to the cluster of nodes with almost native speed for deep learning jobs.

3.2 Cloud Infrastructure

Distributed frameworks such as Spark, provide functional ecosystem for computational tasks. End-user should only specify the program and apply on large datasets. User is limited to only using supported libraries abstracted in MapReduce framework. State-of-the-art libraries are out of reach.

Provisioning: When constraints of the system does allow arbitrary package support, then the whole environment and necessary packages should be transported to the computational node. We use container technology to bake necessary libraries. Compute nodes need to have docker support to execute arbitrary container and

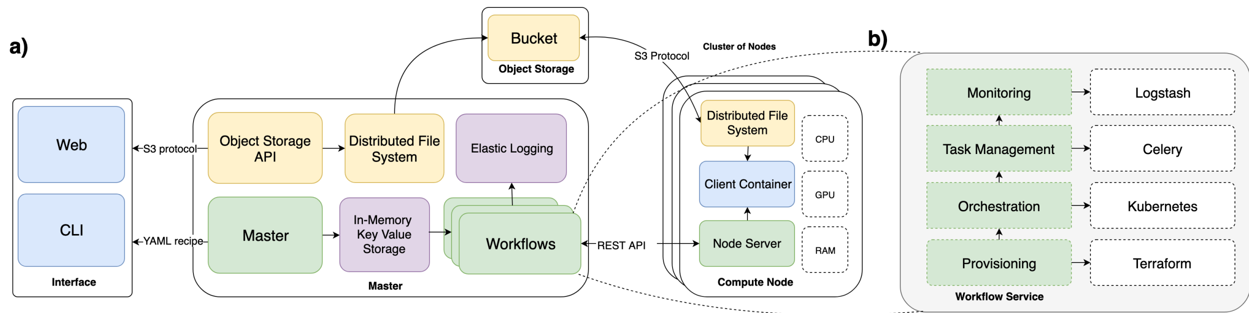

Figure 2: System Architecture a) Interface uploads the training data, source code and YAML recipe to the Master Node. Source files are chunked and uploaded to Object Storage. The Recipe is parsed to create computational graph in in-memory Key Value Storage. For each workflow, cluster is created on the cloud. Each computational node has Node Server that handles management of the node and executing client container. b) Workflow has four main stages. Provisioning the infrastructure, orchestrating nodes, executing tasks and monitoring workers.

Nvidia CUDA [5] libraries for processing deep learning operations on the GPU.

Orchestration: Due to its generic feature of container technology, the Virtual Machine (VM) images necessary to run containers can be based on any Unix based operating systems including CoreOS, Ubuntu and CentOS. VM image is built only once and stored on the cloud. It acts as a proxy to execute custom specified containers. The user can specify the container from public repository. After cloud instances are provisioned, each instance downloads client container. This mechanism allows to support any framework, library or package without constraints. We also cache frequently used containers such as Tensorflow, Pytorch, Jupyter directly inside VM image to reduce loading time. In addition to custom docker management, the system can offload container orchestration to managed Kubernetes [1].

Networking: For cloud infrastructure orchestration, we use Terraform, which provides code-as-infrastructure language for defining cloud resources. For each job execution, the system specifies a Virtual Private Network with Internet Gateway. It makes cluster nodes accessible inside the network for use cases that require state synchronization across nodes such as during distributed training. Alternatively one can use object storage as a parameter server to store the model without networking setup.

3.3 Implementation

As shown in Fig. 2, the architecture of the distributed framework consists of main components: Interface, Master and Node. Master is responsible for receiving the recipe of the pipeline, parsing and creating workflow objects including experiments and tasks. The objects are stored in-memory key-value cache Redis. As a backup alternative, the system stores the state into DynamoDB. Then, master starts a new workflow service as an adjacent container to orchestrate and schedule tasks. During orchestration process, each compute worker runs node server that listens commands executed by the workflow manager. Each node starts to pull client specified container and mount the distributed file system. Once the node is ready, the workflow manager can execute client specified commands.

There are three types of logs that are collected into Elastic Logstash: client application logs, CPU/GPU utilization logs and operating system logs .

In addition to the scheduling system, we deploy our own object storage layer for providing S3-like API to client interface. When the data is uploaded to Minio server , files get chunked and stored on the distributed file system. Celery is used for asynchronous task management. The task management system is similar to Apache Airflow and other workload management systems such as Splunk. It is different from frameworks such as Dask [8] or Ray [7] in terms of execution granularity.

3.4 Fault tolerance

In order to optimize cloud resource allocation, cloud vendors provide spot or preemptible instances. Those instances are usually 2 or 3 times cheaper but can be terminated anytime depending on the demand and the price per hour bid. For stateful long lasting jobs cost optimization is an attractive option, however it requires additional compute logic implementation to recover the state.

Since the system already provides distributed file system backed by remote object storage and a scheduling system, it becomes straightforward to implement fault-tolerant system that can handle instability. When a node fails, the task with exact command arguments gets rescheduled on a different node. For training use case, modern deep learning frameworks provide easy interface to store and retrieve model state. Hence the training can be continued without much additional code modifications.

4. EVALUATION

To evaluate the system we run pre-processing, distributed training, hyper-parameter search and large scale inference using AWS CPU M5 family and GPU P3 family compute instances. We use AWS S3 to store file system chunks.

4.1 Preprocessing

To test the scalability of the system for ETL tasks, we setup a preprocessing experiment. 100 million text files from commoncrawl dataset are uploaded to the distributed storage. The amount of data achieves 10TB. We specify the infrastructure to spin up 110 instances each with 96 cpu cores. The processing script takes 100,000 text files and transforms into tfrecord files. During the transformation, spaCy package is used for filtering, tokenizing and splitting paragraphs. We also enable spot instances for reducing costs and testing the fault tolerance.

4.2 Distributed Training

We defined a recipe for training object recognition model YoloV3. Then, we uploaded COCO dataset [6] to the storage. The training script reads all images and labels from defined path. Furthermore, parameters such as how many epochs, learning rate and what input image size are parameterized in the recipe. We also parametrize model specific parameters such as total number of classes, confidence threshold, nms threshold and iou threshold.

Nvidia K80 GPUs are slow for training the model. By single line configuration change, we deployed the training on Nvidia V100 GPUs with spot enabled instances to reduce costs. The batch size for training was accordingly modified. The cost would be $8.48/h instead of $0.95/h, but the training is 50x faster with 6x efficiency gain. We modified hyper-parameters and started another experiment with zero effort.

4.3 Hyperparameter search

Gradient boosting machine is one of the best off-the-shelf machine learning solvers, however training those models can be computationally very heavy and have a lot of parameters to tune. There are 12 parameters to tune for the tree booster. If you try 2 choices for each one, there will be 4096 different combinations. If each training takes 10 mins to complete, trying out all those 4096 combinations sequentially would take 28.4 days. Using our system, we made the experiments run in 10 minutes by linearly increasing the cluster size without source code modification.

4.4 Large-Scale Inference

For production ready processing, we upload an ImageNet dataset and split it into 300 folders. Each folder contains 1500 images. Using Hyper interface, we easily parallelized the inference execution of Yolo model to 300 GPU instances with overall processing of 2 petaflops.

5. CONCLUSION

Hyper provides a unified view to multiple clouds and on-premise infrastructure without requiring a team of DevOps engineers and saving AI/ML-Ops time. It provides cloud cost savings and transparent compute resource utilization tracking. Hyper Storage solution is significantly better for data-intensive deep learning tasks compared to cloud-native NFS-like offers and much more cost-efficient since it backed by object storage.

Hyper enables data scientists to be highly efficient in machine learning and deep learning. It provides a framework for running experiments, collecting logs and comparing models including one-click Jupyter notebooks or Tensorboard graphs using Web UI or CLI. Data scientists can plug-and-play state-of-the-art deep learning models in Computer Vision, NLP, and other domains to kickstart their project. They can execute large-scale distributed training or batch processing jobs through a very simple interface without knowing about the infrastructure and define continuous workflows for automatic model training, validation, benchmarking and deployment.

Future work includes development of various Bayesian optimization algorithms for hyper-parameter tuning of models, seamless integration with Kubernetes and interactive workflows.

REFERENCES

[1] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] Xinyuan Huang, Amit Kumar Saha, Debojyoti Dutta, and Ce Gao. Kubebench: A benchmarking platform for ml workloads. In *2018 First International Conference on Artificial Intelligence for Industries (AI4I)*, pages 73–76. IEEE, 2018.

[4] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[5] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[6] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[7] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[8] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130-136. Citeseer, 2015.

[9] Mahadev Satyanarayanan. A survey of distributed file systems. *Annual Review of Computer Science*, 4(1):73–104, 1990.

[10] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and Dave Noveck. Network file system (nfs) version 4 protocol. Technical report, 2003.

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[12] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.