Hewlett Packard Enterprise

HPE Reference Architecture for AI on HPE Elastic Platform for Analytics (EPA) with TensorFlow and Spark

Using HPE Asymmetric Cluster Architecture (Apollo 2000/4200)

Contents

Executive summary 3
Introduction 4
HPE Elastic Platform for Analytics (EPA) 5
Apache Spark 5
Hadoop YARN 6
TensorFlow 6
Distributed TensorFlow 7
TensorFlowOnSpark 7
HPE Deep Learning Cookbook 8
Solution overview 8
Single rack elastic architecture solution 9
Software 11
Configuration guidance for the solution 12
General hardware guidance 12
Base Hadoop configuration guidance for HPE EPA asymmetric reference architecture 12
Spark configuration guidance 16
TensorFlow configuration guidance 16
Capacity and sizing 17
HPE Sizer for the Elastic Platform for Big Data Analytics 17
Implementing a proof-of-concept 17
MNIST digit classification with Fully Connected model 18
CIFAR-10 Image classification with Convolutional Neural Network model 21
ImageNet with Inception v3 model 25
Criteo – Ad Click-Through Rate (CTR) prediction use case 28
Summary 34
Resources and additional links 35

Executive summary

Artificial Intelligence (AI) is increasingly being used to accelerate innovation and drive competitive advantage across a number of industries and applications. Organizations are now utilizing machine learning (ML) and deep learning (DL) techniques to explore a new breed of analytics, predictive tools and speeding time to insight. Deep learning, a branch of machine learning (ML), is about enabling computers to learn new concepts from raw data - much like the human mind - using Deep Neural Networks (DNN). The key enablers for the AI revolution are recent ground breaking advances in deep learning research making use of vast computing power of multi-core Graphical Processing Units (GPU), and the availability of big data to train the Deep Neural Networks. The age of big data has made machine learning and deep learning much easier as the availability of huge amounts of data eliminates the need to rely on statistical estimations used in the past. The accuracy of the ML models has been plateauing prior to the use of DL techniques and the recent DNN innovations in last decade have improved the accuracy by a huge margin as evidenced by the results of the ImageNet challenge.

TensorFlow™, originally developed by researchers and engineers from the Google Brain team within Google's AI organization, has become a popular choice for AI practitioners who are implementing ML and DL algorithms. Its flexible architecture allows easy deployment of computation across a variety of platforms and its support for distributed training allows multiple machines to train a model faster and to improve the productivity of AI teams.

However the challenge aspect of AI isn't necessarily the development AI code. The AI code represents a very small fraction of a real-world AI system. The rest is the ecosystem surrounding it which enables the AI data pipeline. The majority of the AI effort is spent in configuring the ecosystem, collecting and cleansing the data, verifying the data, extracting the features from the raw data to feed to the AI algorithms, managing the model serving infrastructure, and monitoring it.

Spark provides a well understood and effective solution to the ecosystem needs surrounding AI, as it can connect to all the AI ecosystem components and has become the de-facto processing framework for big data, due to its ability to speed up batch, interactive, and streaming analytics and its ability to scale out to big data sets. It also offers the flexibility to run analytics on data stored not just in Hadoop Distributed File System (HDFS), but also across object stores and traditional databases, making Spark the ideal platform for accelerating cross-platform analytics on-premises and in the cloud.

While distributed AI training with TensorFlow improves the productivity through faster training turnaround times, it is also difficult to configure and does not come with a resource manager to manage the cluster resources. This problem can be solved effectively by running Spark on Hadoop YARN and integrating Spark and TensorFlow together to make ML/DL drastically easier to use and scale.

While enterprises utilizing AI already have systems in place for big data ecosystems with ML/DL, they are frequently siloed, resulting in operational complexities tied to copying, duplicating, and maintaining the freshness of data. Integrating AI and big data ecosystem components into a consolidated platform can help organizations avoid these costly missteps.

The HPE Elastic Platform for Big Data Analytics (EPA) is designed as a modular infrastructure foundation to address the need for a scalable, multi-tenant platform, enabling independent scaling of compute and storage through infrastructure building blocks that are optimized for different workloads. HPE EPA can accommodate AI workloads by using AI compute blocks. These blocks are based on platforms that support GPUs (e.g. HPE Apollo 6500) and can be added independently to the cluster from the storage tier. For an in-depth analysis of the Hewlett Packard Enterprise architecture for scalable and shared enterprise analytics platforms, and the benefits of separating compute and storage, review the HPE Elastic Platform for Big Data Analytics technical white paper at, http://h20195.www2.hpe.com/V2/GetDocument.aspx?docname=4AA6-8931ENW.

Document purpose: This document describes a reference architecture for deploying AI workloads on HPE EPA asymmetric architecture using TensorFlow and Spark. In addition to outlining the key solution components, this white paper also provides guidelines for configuring and deploying this combined solution, including recommendations for Spark, YARN and TensorFlow.

This white paper highlights recognizable benefits of integrating AI workloads into existing HPE EPA infrastructure simply by adding AI compute blocks and combining Hadoop YARN and Spark with the TensorFlowOnSpark framework. It provides guidance on selecting the appropriate configuration for an AI-focused Hadoop/Spark cluster. The configurations are based on the HPE EPA architecture leveraging modular compute building blocks based on the HPE Apollo 2000 with HPE ProLiant XL190r as the AI compute block, the HPE Apollo 2000 with HPE ProLiant XL170r as a general purpose analytics compute block and the HPE Apollo 4200 as a capacity-optimized storage building block.

This Reference Architecture demonstrates the flexibility of HPE EPA elastic architecture to include AI workloads seamlessly. The paper demonstrates the capabilities of Spark scalable preprocessing using Apollo 2000 based compute blocks combined with the ML/DL training using Apollo 2000 with HPE ProLiant XL190r AI compute blocks. Since Spark is run with the YARN resource manager on HPE EPA, preprocessing and

ML/DL training jobs can be managed independently by making use of YARN labels, thus providing isolation and efficient resource sharing between the different nodes in a single cluster. With the HPE EPA architecture, analytics compute blocks and AI compute blocks can share the same data on the HPE Apollo 4200 storage blocks, avoiding expensive data duplication and providing independent compute and storage scaling flexibility.

Target audience: The intended audience of this document includes, but is not limited to, Chief Data/Information Officers, VP of Analytics, Data Scientists, Data Engineers, IT managers, pre-sales engineers, services consultants, partner engineers, and customers that are interested in deploying TensorFlow and Spark in their existing or new big data deployments to include AI capabilities in their analytic workloads. This solution will be frequently employed by enterprises that already have data in an HDFS cluster and who are considering or already have invested in the Spark ecosystem.

This white paper describes testing performed in September 2018. Although the HPE EPA architecture cluster used for testing utilized HPE Apollo 2000 Gen9 servers with HPE ProLiant XL190r servers as AI accelerator nodes, HPE ProLiant XL170r servers for compute nodes and HPE Apollo 4200 Gen9 servers for storage nodes, similar results could be obtained with Gen10 servers.

Introduction

Enterprises embarking on their AI journey may already have existing big data infrastructure for their analytical requirements and need to integrate AI with the existing infrastructure. A typical AI pipeline includes three main stages: data ingest and preparation, training and serving the AI models for inference. Usually the collected raw data is not amenable for training models and needs to undergo some preparation before it can be used for training. The training data for deep learning is typically unstructured e.g. images and speech while for traditional machine learning the data typically comes from various structured sources like Hive, MySQL etc. The data preparation steps for DL could include packing the data into an efficient format, while for ML they could include exploring data for feature selection, doing joins, computing aggregates, generating embeddings for some of the features, etc. Traditionally, tools like Hadoop MapReduce and Spark have been used for this ETL stage. Spark has almost replaced MapReduce to become the de-facto processing framework for big data, due to its ability to speed up batch, interactive, and streaming analytics and its ability to make use of memory efficiently to process big data sets.

Training the AI models takes multiple forms. First, data scientists try out experiments on a single node - laptop or a server – using a small sample of data. After coming up with a reasonable model, they try to fine tune the model and improve the accuracy by using bigger data sets which will run into longer training times and iterations. The training times can be sped up using distributed training using multiple GPUs, e.g. Google has shown 58x speed up in training throughput using 64 GPUs and Facebook has shown reducing training time from two weeks to one hour, using 256 GPUs spread over 32 servers.

TensorFlow has become the most popular choice with ML/DL practitioners for its flexible architecture, multiple language support and its support for distributed training. Distributed training with TensorFlow accelerates the training, but is difficult and doesn't come bundled with infrastructure/ecosystem support. There are two essential components that are required by distributed TensorFlow:

- A resource manager to manage the cluster resources
- A shared file system for input data and saving model checkpoints in case of failures

Distributed TensorFlow can work with multiple resource managers like Hadoop YARN, Kubernetes, Mesos and SLURM, but for the enterprises with existing big data infrastructure, YARN is an ideal choice. While parallel file systems like WekaIO are suitable for HPC customers, HDFS offers a good choice of the shared file system requirement for enterprises with existing big data infrastructure since they already have the data in HDFS and TensorFlow has HDFS support since version 0.11. Running distributed TensorFlow on YARN itself can be still challenging, that's where frameworks like TensorFlowOnSpark come to the rescue. Since many enterprises already have data pipelines implemented using Spark and YARN is the most popular resource manager for Spark, using TensorFlowOnSpark offers a smooth transition for enterprises to get started on AI projects due to its ability to migrate existing TensorFlow programs with just a few lines of code changes, and yet still supporting various TensorFlow functionalities: synchronous/asynchronous training, model/data parallelism, inferencing and TensorBoard support.

According to recent technology industry market research surveys, over three-quarters of enterprises are investing in AI, but the majority expect significant barriers to AI business benefit realization, with a lack of IT infrastructure and lack of access to talent among the top challenges. By extending the existing investments in big data infrastructure like HPE's Elastic Platform for Analytics with AI compute blocks and data processing frameworks like Hadoop, YARN, and Spark with ML/DL frameworks like TensorFlow, enterprises can get started with AI and leverage the full potential of this revolutionary technology.

HPE Elastic Platform for Analytics (EPA)

The HPE Elastic Platform for Analytics is a premier modular infrastructure foundation to accelerate business insights, enabling organizations to rapidly deploy, efficiently scale, and securely manage the explosive growth in volume, speed, and variety of big data workloads. By leveraging a building block approach, customers can simplify the underlying infrastructure needed to address a myriad of different business initiatives around Data Warehouse modernization, analytics, and BI, while building large-scale data lakes which incorporate a diverse set of data. As workload, compute and data storage requirements change (each often growing uncorrelated to the other), the elastic nature of the HPE EPA architecture allows customers to easily scale by adding compute and storage blocks independently.

One of the main benefits of HPE EPA platform is that newer workloads like Deep Learning can be easily integrated into existing infrastructure simply by adding additional relevant EPA blocks. For the entry-level DL workloads, enterprises can get started with the addition of HPE Apollo 2000 AI compute blocks and can slowly graduate to HPE Apollo 6500 AI computes block for more demanding DL workloads. An Apollo 2000 AI building block includes two HPE ProLiant XL190r Gen10 servers and two GPU accelerators per server. The HPE Apollo 6500 AI compute block supports up to eight GPUs with a fast GPU interconnect and high bandwidth fabric.

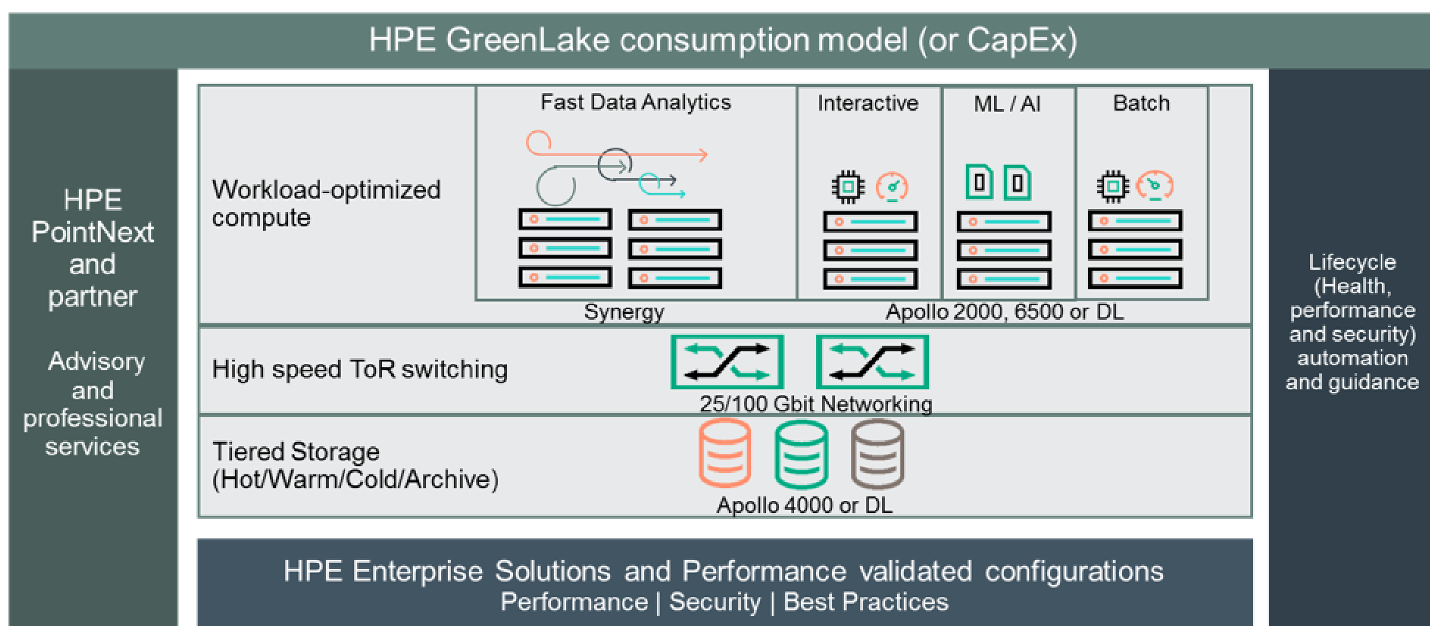


Figure 1. HPE Elastic Platform for Big Data Analytics

The HPE GreenLake Flex Capacity service allows customers to consume their analytics infrastructure using a flexible, on-demand consumption model, paying only for what they use in terms of servers, storage, networks, software, and services. HPE Pointnext provides powerful, scalable IT solutions for organizations across every industry and offers experts in key service areas — Advisory and Professional, Consumption-based IT, and Operational.

Apache Spark

Apache Spark is a fast general-purpose engine for large-scale data processing. Spark was developed in response to limitations in Hadoop's two-stage disk-based MapReduce processing framework. Spark maintains MapReduce's linear scalability and fault tolerance, and extends the processing capabilities for in-memory analytics, iterative analytics, and near real-time analytics with Spark streaming.

Spark consists of core and a set of libraries. The core is the distributed execution engine, and the Java, Scala, Python, and R APIs offer a platform for scalable distributed application development. Additional libraries, built atop the core, allow diverse workloads for streaming, SQL, machine learning, and graph processing. Spark's key advantage is speed, with an advanced DAG (Directed Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing; hence, it runs programs much faster than Hadoop/MapReduce. It offers ease of use to write applications quickly in Java, Scala, Python, and R in addition to providing interactive modes from Scala, Python, and R shells. Spark libraries can be combined seamlessly in the same application. Spark jobs executed in YARN client/cluster mode are ideally suited for generic Spark workloads;

although, Spark jobs can be run in standalone cluster mode with the ability to access data in HDFS, HBase, Hive, S3, and any Hadoop data source. Spark complements Hadoop as it provides a unified solution to manage various big data use cases and requirements.

Figure 2 shows the various components of the Apache Spark ecosystem.

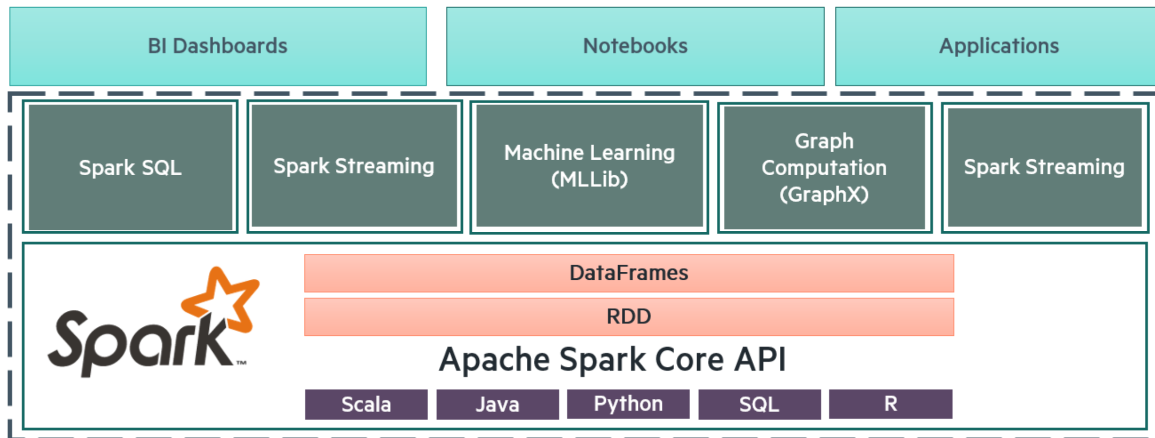


Figure 2. Apache Spark – Components

Spark can be deployed with different cluster managers – Spark's standalone cluster manager, Apache Mesos, or Hadoop YARN. Most organizations that already have deployed Hadoop will look to integrate Spark with Hadoop's YARN framework to manage all data processing under one resource manager.

Hadoop YARN

Part of the core Hadoop project, YARN allows multiple data processing engines such as interactive SQL, real-time streaming, data science and batch processing. One of the most significant benefits of Hadoop YARN is to separate processing from resource management. This enables a variety of new tools like Spark to identify data of value interactively and in real time, without being hampered by the often I/O intensive, high latency MapReduce framework.

Spark on YARN is an optimal way to schedule and run Spark jobs on a Hadoop cluster alongside a variety of other data-processing frameworks, leveraging existing clusters using queue placement policies, and enabling security by running on Kerberos-enabled clusters.

YARN has the concept of node labels for groupings of compute nodes. Jobs submitted through YARN can be flagged to perform their work on a particular set of nodes when the appropriate label name is included with the job. Thus, an organization can create groups of compute resources that are designed, built, or optimized for particular types of work, allowing for jobs to be passed out to subsets of the hardware that are optimized to the type of computation, making it an ideal choice for HPE EPA with multiple, diverse building blocks. In addition, the use of labels allows for isolating compute resources that may be required to perform a specialized job involving accelerators like GPUs.

GPUs have become the key enablers of AI workloads with the help of deep learning frameworks that exploit the 1000s of GPU cores efficiently to reduce the training time from weeks to days and from days to hours. As the servers with GPUs tend to be a precious commodity in the data center, they tend to be managed more closely and YARN node labels provide an excellent mechanism to group them separately and make efficient use of them.

TensorFlow

TensorFlow is an open source software library for high performance numerical computation. Among all of the deep learning frameworks, TensorFlow is the most popular choice[1]. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning.

[1] Deep Learning Framework Power Scores 2018

TensorFlow includes a core distributed execution engine that runs on top of various devices and offers an application programming interfaces (API) in Python, Java, Go, as well as a lower level set of functions implemented using C++. For easier machine learning model development, it provides higher level APIs e.g. Layers, Datasets, Estimators etc. It also includes XLA (Accelerated Linear Algebra) domain-specific that optimizes TensorFlow computations. The results are improvements in speed, memory usage, and portability on server and mobile platforms.

Figure 3 shows the various features of TensorFlow architecture.

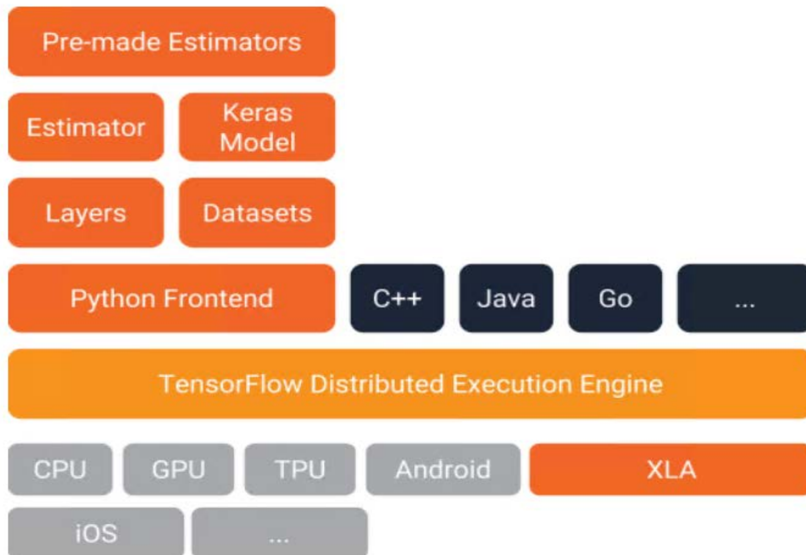


Figure 3. TensorFlow architecture – components (Source: Google)

Distributed TensorFlow

Training on a large amount of data improves the performance of DL models. However, training a DNN with millions of parameters with millions or billions of training samples may take days, or weeks. Distributed training allows multiple machines to train a model faster by distributing the DL training across the machines. We can scale out distributed training to 100s of GPUs resulting in massive reduction of experimentation time. Google has shown 58x speed up in training throughput using 64 GPUs (8 GPUs spread over 8 servers). Facebook released a paper showing that they reduced training time for a big convolutional neural network (CNN) model, RESNET-50, on ImageNet dataset from two weeks to one hour, using 256 GPUs spread over 32 servers.

There are multiple DL frameworks supporting distributed training e.g. Distributed TensorFlow, Horovod, Apache MXNet. Microsoft® CNTK etc., and each has its own merits. For existing HPE EPA customers, Distributed TensorFlow offers a good choice as TensorFlow has HDFS support since early versions.

Distributed TensorFlow training is quite different from the training on a single machine where all parameters and gradients computation are on the same node. The computation is coordinated across multiple machines by employing a centralized repository for parameters that maintains a unified, single copy of model parameters. Each individual machine sends gradient updates to the centralized parameter repository which coordinates these updates and sends back updated parameters to the individual machines running the model training. Each machine that runs a copy of the training is called a "worker" or "replica". Each machine that maintains model parameters is called a "PS", short for parameter server. There may be more than one machine acting as a PS as the model parameters may be shared across multiple machines. Variables may be updated with synchronous or asynchronous gradient updates.

TensorFlowOnSpark

Distributed TensorFlow makes the training go much faster, but having separate clusters for deep learning and data processing results in transferring large datasets between them, introducing unwanted system complexity and end-to-end training latency. Several open source projects like TensorFrames from DataBricks and SparkNet from UC Berkeley AMPLab tried to integrate TensorFlow with Spark to address this limitation. However, they require significant effort to migrate existing TensorFlow programs.

TensorFlowOnSpark enables distributed TensorFlow training and inference on Spark and Hadoop clusters. As illustrated in Figure 4, TensorFlowOnSpark is designed to work along with SparkSQL, MLlib, and other Spark libraries in a single pipeline. Additionally, it makes use of Spark's built-in fault-tolerance to recover from failures during training.

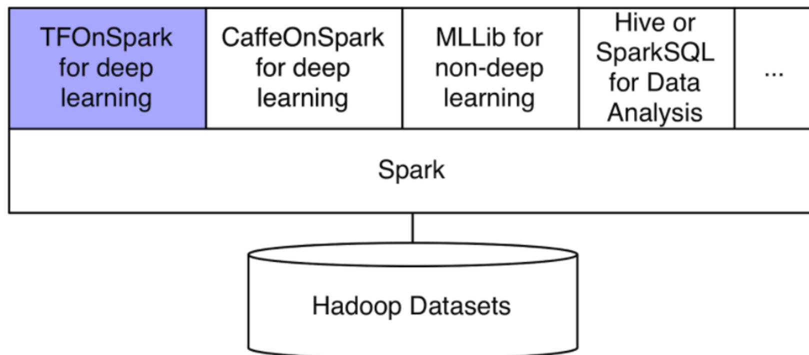


Figure 4. TensorFlowOnSpark for deep learning on Hadoop/Spark cluster

TensorFlowOnSpark supports both asynchronous and synchronous training and inferencing. It supports model parallelism and data parallelism, as well as TensorFlow tools such as TensorBoard on Spark clusters. Any TensorFlow program can be easily migrated to TensorFlowOnSpark.

TensorFlowOnSpark programs are launched by the standard Apache Spark command, `spark-submit`. Users can specify the number of Spark executors, the number of GPUs per executor, and the number of parameter servers as part of the command line options.

HPE Deep Learning Cookbook

Choosing optimal hardware/software stack for any given workload is not obvious and needs careful evaluation with respect to the choice of GPU box, how many GPUs to put in a system, how many systems to put in a cluster and which interconnect to use etc. HPE Deep Learning Cookbook is a set of tools to characterize deep learning workloads and to make the optimal recommendations. It consists of several key assets:

- HPE Deep Learning Benchmarking Suite: automated benchmarking tool to collect performance measurements on various HW/SW configurations in a unified way.
- HPE Deep Learning Performance Guide: a web-based tool which provides access to a knowledge base of benchmarking results. It enables querying and analysis of measured results as well as performance prediction based on analytical performance models.
- Reference Designs: hardware/software recipes for selected workloads.

Solution overview

At the core of this Reference Architecture for AI on the HPE EPA platform are the underlying infrastructure building blocks. These blocks form the foundation of the EPA solution designs and can be pieced together in different ways to solve unique customer needs. This section will detail storage, compute and AI building blocks based on the HPE Apollo 2000 and HPE Apollo 4200 servers required to build an example configuration. This exact configuration was used for benchmarking a few representative workloads as described in "Implementing a proof-of-concept" section. The blocks defined in this section may be modified (e.g., processor model, memory, etc.) to address new or changing workloads and environments. Specific workload services and behavior will drive the final configuration requirements including compute and storage definition and quantity.

Refer to the HPE Reference Configuration for Elastic Platform for Big Data Analytics white paper, which contains detailed information on the various building blocks used for traditional and asymmetric architectures.

Note

In the Reference Configuration document at the link above, traditional cluster design is referred to as a Balanced and Density Optimized (BDO) solution/Symmetric Architecture, and Asymmetric Architecture is referred to as Elastic/ Workload and Density Optimized (WDO) solution. Refer to the WDO section with reference to this Reference Configuration document.

Single rack elastic architecture solution

The solutions highlighted in this document are geared towards most organizations that already have deployed Hadoop and Spark and are looking to integrate AI into their existing ecosystem. One of the benefits of the HPE EPA platform is that newer workloads like Deep Learning can be easily integrated into the existing infrastructure by adding additional EPA building blocks optimized for those workloads. The solution for this RA uses and validates this approach by adding an AI compute block to a previously released RA solution for Spark, available at HPE Reference Architecture for Apache Spark 2.1 on HPE Elastic Platform for Big Data Analytics. HPE has performed extensive testing with multiple workloads including Spark (e.g., MapReduce, Hive, and HBase) to determine the optimal building block configurations to balance the compute power, storage capacity, and network bandwidth. Based on this testing for the Spark RA, HPE recommends a “balanced” single rack elastic cluster design with three memory accelerated compute blocks of HPE Apollo 2000 chassis with XL170r servers and four standard storage blocks of HPE Apollo 4200 Gen9 servers. To extend this solution to include AI workloads, an AI compute block of HPE Apollo 2000 chassis with XL190r servers is added to the solution for entry-level DL workloads. Please note that this a starter configuration example. HPE has developed the HPE Sizer for the Elastic Platform for Analytics to assist customers with proper sizing of EPA environments. Based on design requirements, the sizer will provide a suggested bill of materials (BOM) and metrics data for an HPE EPA cluster which can be modified further to meet customer requirements

HPE Apollo 2000 compute block delivers a scalable, high-density layer for compute tasks and includes four HPE ProLiant XL170r Gen10 nodes in a single 2U chassis. Each XL170r node harnesses the performance of dual Intel® Xeon® Gold 6130 processors each with 16 cores and 768GB of memory, expandable to 1.5TB for memory-rich Spark workloads. The local SSDs on the ProLiant XL170r servers provide ideal speed and flexibility for Shuffle operations during a Spark job. Local storage can scale up to 6 SFF disks on each node.

HPE Apollo 2000 AI compute block includes two HPE ProLiant XL190r Gen10 nodes in a single 2U chassis. The ProLiant XL190r Gen10 Server is a 2U half-width, 2P server with the same configuration options as the XL170r for CPU and memory, but has additional PCIe slots in multiple configurations providing support for additional expansion cards and support for two GPU accelerators per server. For the entry-level DL workloads, NVDIA Tesla P100 (PCIe based) with 16GB of memory is the recommended option. Tesla P100 includes 3584 CUDA cores and offers 4.7 teraFLOPS of peak double-precision performance and 9.3 teraFLOPS of peak single-precision performance.

HPE Apollo 4200 Gen9 servers make up the HDFS storage layer, providing a single repository for big data. The HPE Apollo 4200 saves valuable data center space through its unique density optimized 2U form factor which holds up to 28 LFF disks with a capacity of 112TB per server using 4TB disks.

The single rack RA example provides a core-to-disk ratio of approximately 2:1 that can easily accommodate interactive Spark workloads and more than the canonical 1:1 ratio recommended for batch workloads using MapReduce, etc. In addition, as the HPE EPA architecture provides great flexibility in deploying disparate workloads and managing data growth, by decoupling storage growth from compute through high-speed networking, it enables a wide range of compute-to-storage ratios by adding more compute or storage as needed without having to add both in lock-step. Thus, a “hot” single rack elastic reference architecture for compute-intensive machine learning Spark workloads could have a higher compute-to-storage ratio with more compute blocks; and, a “cold” single rack elastic reference architecture for Spark ETL workloads on storage-demanding use cases could have a lower compute-to-storage ratio with more storage blocks. This logic also applies as the solution scales beyond a single rack. Similarly, the Apollo 2000 AI compute block could be replaced with a higher performing Apollo 6500 AI compute block for demanding DL workloads. From a GPU perspective, the NVDIA Tesla P100 could be replaced by NVIDIA® Tesla V100 which has 640 Tensor cores and 5120 CUDA cores and offers 7.8 teraFLOPS of peak double-precision performance and 15.7 teraFLOPS of peak single-precision performance. This level of flexibility provides a wide range of configuration options to address varying analytics requirements.

Note

HPE recommends to use HPE Deep Learning Cookbook to guide the choice of the best hardware/software environment for a given deep learning workload.

The following summarizes the baseline single rack elastic configuration building blocks which can be adjusted to suit the customer workload requirements:

- Three memory accelerated compute blocks of HPE Apollo 2000 chassis with four HPE ProLiant XL170r Gen10 servers in each compute block. Since Spark is an in-memory analytics engine, memory accelerated compute blocks, with 768GB on each HPE ProLiant XL170r Gen10 server, are used.
- One AI compute block of HPE Apollo 2000 chassis with two HPE ProLiant XL190r Gen10 servers in each block with 768GB memory on each server. Each XL190r Gen10 server is configured with two NVIDIA Tesla P100 GPU accelerator cards.
- Four standard storage blocks of HPE Apollo 4200 Gen9 servers.
- Control block of three HPE ProLiant DL360 Gen10 servers, with an optional fourth server acting as an edge or gateway node depending on the customer enterprise network requirements.
- Network block
- Rack block

Figure 5 shows the conceptual diagram of the baseline singe rack EPA architecture cluster.

Network Block
3x Network switches
Control Block (Edge Module)
1x Edge node
Compute, Storage and AI Accelerator Blocks
1x - AI Accelerator Node (NodeManager, Spark, TensorFlow)
3x – Compute Worker Nodes (NodeManager, Spark)
4x - Storage Worker Nodes (DataNode (HDFS))
HPE 5900AF-48G-4XG-2QSFP+
HPE FlexFabric 5950 48SFP28
HPE FlexFabric 5950 48SFP28
HPE DL360 Gen10 1u
HPE DL360 Gen10 1u
HPE DL360 Gen10 1u
HPE DL360 Gen10 1u
HPE Apollo 2000 Gen10 2u with ProLiant XL190r Gen10
HPE Apollo 2000 Gen10 2u with ProLiant XL170r Gen10
HPE Apollo 4200 Gen9 2u
42u rack enclosure
Control Block Module
1x Management node (CMU, ZK, Ambari, MySQL server, Hive, WS, Zeppelin Notebook)
2x Head nodes, HA enabled (NN, ZK, SNN, RM, Spark History Server, Spark Thrift Server, MR2, Yarn ATS)

Legend: RM – ResourceManager, NN - NameNode, SNN - Secondary NameNode, WS - WebHCat server, ZK - ZooKeeper, MR2 - MapReduce2, CMU – HPE Insight Cluster Management Utility

Figure 5. Basic conceptual diagram of single rack HPE EPA architecture cluster with Apache Spark and TensorFlow

Figure 6 shows a rack-level view of the single rack HPE EPA cluster Reference Architecture.

1 Edge Node
HPE DL360 Gen10 8SFF
Dual Intel Xeon-Silver 4116 (2.1GHz/12-core)
192GB RAM (12x16GB 2Rx8 PC4-2666V-R)
7.2 TB Disks (8x 900 GB SAS 15K HDD)
1x HPE Smart Array P408i-a Gen10 Controller
1x Ethernet 10/25Gb 2-port 640FLR-SFP28 Adapter

1 Ethernet Switch
HPE 5900AF-48G-4XG-2QSFP+ Switch

2 Ethernet Switches
HPE FlexFabric 5950 48SFP28 Switch

1 Management Block
HPE DL360 Gen10 8SFF
Dual Intel Xeon-Silver 4116 (2.1GHz/12-core)
192GB RAM (12x 16GB 2Rx8 PC4-2666V-R)
7.2TB Disks (8x 900GB 6G SAS 15K HDD)
1x HPE Smart Array P408i-a Gen10 Controller
1x Ethernet 10/25Gb 2-port 640FLR-SFP28 Adapter

1 AI Compute Block
HPE Apollo 2000 Gen10 ProLiant XL190r Gen10
Dual Intel Xeon-Gold 6140 (2.3GHz/18-core)
768GB RAM (12x 64GB 4Rx4 PC4-2666V-L)
960GB Disks (4x 480GB 6G SATA MU-2 SSD)
1x HPE Smart Array P408i-p SR Gen10 Controller
1x Ethernet 10/25Gb 2-port 640FLR-SFP28 Adapter
2x NVIDIA Tesla P100 PCIE 16GB

Three-phase PDU (4 PDUs per rack):
HPE 4.9k VA/L6-30/NA/J PDU

3 Compute Blocks
HPE Apollo 2000 Gen10 with ProLiant XL170r Gen10
Dual Intel Xeon-Gold 6130 (2.1GHz/16-core)
768GB RAM (12x 64GB 4Rx4 PC4-2666V-L)
960GB Disks (4x 480GB 6G SATA MU-2 SSD)
1x HPE Smart Array P408i-p SR Gen10 Controller
1x Ethernet 10/25Gb 2-port 640FLR-SFP28 Adapter

4 Storage Blocks
HPE Apollo 4200 Gen 9
Dual 14-Core Intel E5-2680 v4 2.4GHz
128GB RAM (8x 16GB 2Rx4 PC4-2400T-R)
112TB Disks (28x 4TB 6G SATA 7.2K HDD)
Dual 120GB Value Endurance Solid State M.2
1x HPE Smart Array P840ar/2G Controller
1x Ethernet 10/25Gb 2-port 640FLR-SFP28 Adapter

Software
Operating System: 64-bit Red Hat Enterprise Linux 7.3
Hortonworks HDP 2.6 and Ambari 2.6
Spark 2.1, TensorFlow 1.8, TensorFlowOnSpark 1.3
HPE Insight Cluster Management Utility v8.2

Figure 6. Single rack HPE EPA asymmetric architecture cluster with Apache Spark and TensorFlow – Rack-level view

Software

Hortonworks Data Platform (HDP 2.6.4) with Spark 2.2 was used for the testing documented in this paper. TensorFlow 1.8.0 and TensorFlowOnSpark version 1.3 were used as AI frameworks. In addition, NVIDIA's CUDNN v7.3.0 (GPU-accelerated library of primitives for deep neural networks) with CUDA 9.0 toolkit were used on GPU accelerated nodes.

The details of the service distribution among the HPE EPA cluster nodes are as follows:

- Management node – hosts the applications that manage the Hadoop cluster and is also used to submit jobs to the Hadoop cluster. Recommended services for this node: Ambari Server, HPE Insight Cluster Management Utility (CMU) 8.0, MySQL server, Hive Server, WebHCat server, Zeppelin Notebook, and ZooKeeper.
- Head Node 1 – hosts the HDFS NameNode service. Other recommended services for this node: Spark 1 History Server, Spark 1 Thrift Server, and ZooKeeper.
- Head Node 2 – hosts the YARN Resource Manager service. Other recommended services for this node: Spark 2 History Server, Spark 2 Thrift Server, App Timeline Server, and ZooKeeper.
- Compute Worker Nodes – host the YARN NodeManager service. Will also host Spark executors. Optionally, can host TensorFlow and TensorFlowOnSpark.
- AI Compute Nodes - host the YARN NodeManager service. Also host Spark executors, TensorFlow and TensorFlowOnSpark frameworks, NVIDIA's CUDA and CUDNN toolkits.
- Storage Worker Nodes – host the HDFS DataNode service.

Configuration guidance for the solution

General hardware guidance

Following are some general hardware guidelines for Spark and TensorFlow.

CPUs

Spark scales well to tens of CPU cores per machine because it performs minimal sharing between threads. Provisioning at least 8-16 cores per machine is a good start; and RA recommended CPU is Intel Xeon Gold 6130 processor (16 core/2.1GHz).

For DL workloads, recommended CPU is Intel Xeon Gold 6140 processor (18-core/2.3GHz) or Intel Xeon Gold 6148 processor (20 core/2.4 GHz).

GPUs

The RA recommended GPU for entry level cluster is NVDIA Tesla P100 with 16GB of memory and to accelerate training, NVDIA Tesla V100 with 32GB of memory may be used.

Memory

In general, Spark can make use of as much memory as is available. This RA recommends 768GB RAM which should be sufficient for the majority of Spark workloads. HPE Gen10 systems support a variety of flexible memory configurations, but for optimal performance, it is recommended to balance the total memory capacity across all installed processors making use of all six memory channels per CPU with up to two DIMM slots per channel.

For DL workloads, this amount of memory is more than sufficient for storing variables of most of the TensorFlow graphs. The training batch size can be affected by the amount of memory available and may needs to be adjusted accordingly.

Storage

Most Spark jobs will likely have to read input data from a storage system (e.g., the Hadoop File System, or NoSQL data stores), so having fast access to data from Spark's compute nodes is important. SSDs are the recommended choice for the intermediate data on compute nodes.

DL workloads can also benefit from faster storage systems, but for enterprises with existing Hadoop/Spark clusters, the HDFS storage using 6G SATA 7.2K HDDs can be sufficient for entry level clusters. For distributed training, HDFS provides ideal shared storage for entry level requirements as demonstrated through the testing results, but for higher performance, faster parallel systems like WekaIO can be used as the shared storage.

Network

Spark "Distributed Reduce" applications such as group-bys, reduce-bys, and SQL joins result in a lot of network traffic between the compute nodes. The two 25GbE network adapter cards used in this reference architecture provide the best option to avoid network bottlenecks between compute nodes during shuffle operations.

Network also plays an important role in distributed training with TensorFlow. 25GbE cards have been found to be sufficient for testing done for this RA and are recommended option for entry level requirements with fewer GPUs and Inception v3 like DL architectures. For more demanding DL workloads using Apollo 6500s with lot of powerful GPUs like NVDIA Volta V100 and large DL architectures like RESNET, 100GbE network adapter cards are recommended.

Base Hadoop configuration guidance for HPE EPA asymmetric reference architecture

The configuration changes mentioned in this section are intended to assist in optimizing the setup of the various services of this reference architecture. They are recommended to be used as a starting point for actual deployment scenarios. Customizations may be required in real life deployments. The configuration examples provided here are for HDP on which the testing was performed.

HDFS configuration guidance for TensorFlow

With local (single node) TensorFlow, it is normal to use local storage for datasets, but with distributed TensorFlow it is imperative that you use a shared file system (NFS/Distributed File System/Parallel File System) as multiple workers can share the datasets without copying and more importantly, the workers can save the checkpoints to the shared file system so that the training can restart from the check pointed state in case of failure avoiding retraining from scratch. For enterprises using Hadoop/Spark clusters, HDFS is a natural choice for the shared file system.

Normally the HDFS native client library (`libhdfs.so`) is only installed on edge nodes and might not be available on the compute/AI compute nodes. But when accessing datasets from HDFS, TensorFlow requires the native libraries to be present on the local nodes and

`LD_LIBRARY_PATH` environment variable point to the location of the library. The library needs to be copied to all the compute/ AI compute nodes from where TensorFlow code is executed.

HDFS configuration optimizations for storage nodes

Make the following changes to the HDFS configuration for optimal HDFS performance for Spark jobs using HDFS:

- Increase the dfs.blocksize value to allow more data to be processed by each map task, thus reducing the total number of mappers and NameNode memory consumption:

  `dfs.blocksize 512`

- Increase the dfs.namenode.handler.count value to better manage multiple HDFS operations from multiple clients:

  `dfs.namenode.handler.count 180`

- Increase the Java heap size of the NameNode to provide more memory for NameNode metadata:

  `Java Heap Size of NameNode in Bytes 4096MiB`

  `Java Heap Size of Secondary NameNode in Bytes 4096MiB`

- Increase the following timeout value to eliminate timeout exceptions when dealing with large datasets:

  `dfs.client.block.write.locateFollowingBlock.retries 30`

YARN configuration guidance for compute nodes

YARN Node labels feature, available with Capacity Scheduler, allows to group nodes with similar characteristics and applications can specify where to run. A cluster is partitioned into several disjoint sub-clusters by node partitions. As GPU resources are more expensive, YARN node labels provide an excellent mechanism to group the nodes with GPUs separately and make efficient use of them. Using the YARN capacity scheduler queues, administrators can allocate resources in controlled manner and can monitor the jobs submitted using those queues closely. While this document uses YARN node labels feature of capacity scheduler for resource grouping, similar functionality can be achieved by using CDH custom hardware profiles feature.

One can setup an elaborate queue control mechanism to share the resources, but following is a simple scheme (used for the testing of this RA) to have two queues: one for GPU nodes and another for CPU nodes.

Steps to set up two node partitions for CPUs and GPUs using node labels:

1. Set up the following yarn properties in Ambari.

```
yarn.node-labels.fs-store.root-dir    hdfs://namenode:port/path/to/store/node-labels/
yarn.node-labels.enabled              true
```

2. Make sure `yarn.node-labels.fs-store.root-dir` is created and ResourceManager has permission to access it. (Typically from “yarn” user).

```
su - hdfs -c "hadoop fs -mkdir -p /yarn/node-labels"
su - hdfs -c "hadoop fs -chown -R yarn:yarn /yarn"
su - hdfs -c "hadoop fs -chmod -R 700 /yarn"
```

3. Add the cluster node labels list.

```
su - yarn rmadmin -addToClusterNodeLabels "GPU(exclusive=true),CPU(exclusive=true)"
su - yarn yarn rmadmin -replaceLabelsOnNode "m16wn01.localdomain=GPU m16wn02.localdomain=GPU"
su - yarn yarn rmadmin -replaceLabelsOnNode "m16wn03.localdomain=CPU m16wn04.localdomain=CPU
                                             m16wn05.localdomain=CPU m16wn06.localdomain=CPU
                                             m16wn07.localdomain=CPU m16wn08.localdomain=CPU
                                             m16wn09.localdomain=CPU m16wn10.localdomain=CPU
                                             m16wn11.localdomain=CPU m16wn12.localdomain=CPU"
```

4. Check the cluster node labels list.

```
yarn cluster --list-node-labels
```

Figure 7 shows the YARN Resource Manager GUI showing the nodes and associated node labels after following the above steps.

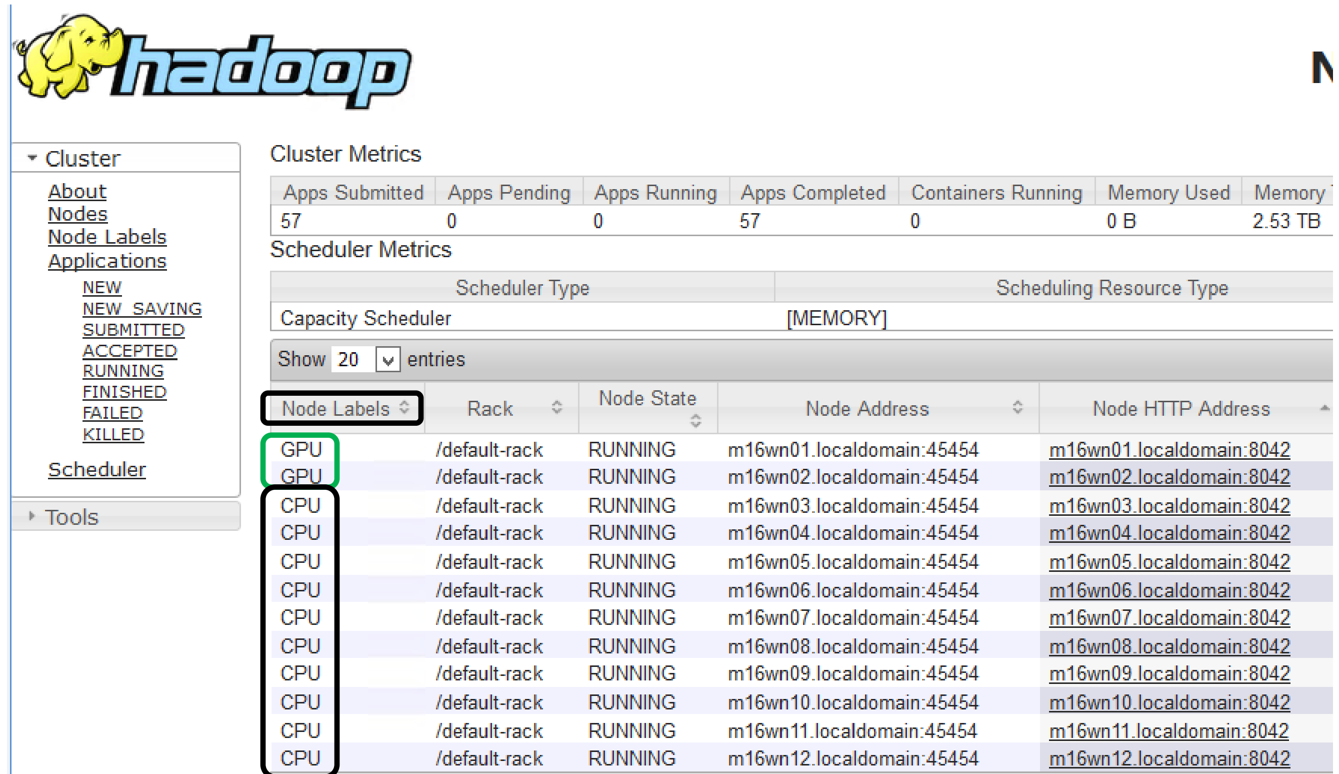


Figure 7. YARN Resource Manager GUI with Node Labels

5. Next, set up the YARN queues for CPU and GPU nodes in capacity scheduler and associate the node labels to their respective queues by adding the following lines in Ambari advanced capacity scheduler configuration:

```
yarn.scheduler.capacity.default.minimum-user-limit-percent=100
yarn.scheduler.capacity.maximum-am-resource-percent=0.2
yarn.scheduler.capacity.maximum-applications=10000
yarn.scheduler.capacity.node-locality-delay=40
yarn.scheduler.capacity.root.GPU_Q.accessible-node-labels=GPU
yarn.scheduler.capacity.root.GPU_Q.accessible-node-labels.GPU.capacity=100
yarn.scheduler.capacity.root.GPU_Q.accessible-node-labels.x.maximum-capacity=100
yarn.scheduler.capacity.root.GPU_Q.acl_administer_queue=*
yarn.scheduler.capacity.root.GPU_Q.acl_submit_applications=*
yarn.scheduler.capacity.root.GPU_Q.capacity=50
yarn.scheduler.capacity.root.GPU_Q.maximum-capacity=100
yarn.scheduler.capacity.root.GPU_Q.default-node-label-expression=GPU
yarn.scheduler.capacity.root.GPU_Q.state=RUNNING
yarn.scheduler.capacity.root.GPU_Q.user-limit-factor=1
yarn.scheduler.capacity.root.accessible-node-labels=*
yarn.scheduler.capacity.root.accessible-node-labels.GPU.capacity=100
yarn.scheduler.capacity.root.accessible-node-labels.GPU.maximum-capacity=100
yarn.scheduler.capacity.root.accessible-node-labels.CPU.capacity=100
yarn.scheduler.capacity.root.accessible-node-labels.CPU.maximum-capacity=100
yarn.scheduler.capacity.root.acl_administer_queue=*
yarn.scheduler.capacity.root.CPU_Q.accessible-node-labels=CPU
yarn.scheduler.capacity.root.CPU_Q.accessible-node-labels.CPU.capacity=100
yarn.scheduler.capacity.root.CPU_Q.accessible-node-labels.CPU.maximum-capacity=100
yarn.scheduler.capacity.root.CPU_Q.acl_administer_jobs=*
yarn.scheduler.capacity.root.CPU_Q.acl_submit_applications=*
```

```
yarn.scheduler.capacity.root.CPU_Q.capacity=50
yarn.scheduler.capacity.root.CPU_Q.maximum-capacity=100

yarn.scheduler.capacity.root.CPU_Q.default-node-label-expression=CPU
yarn.scheduler.capacity.root.CPU_Q.state=RUNNING
yarn.scheduler.capacity.root.CPU_Q.user-limit-factor=1
yarn.scheduler.capacity.root.capacity=100
yarn.scheduler.capacity.root.queues=CPU_Q,GPU_Q
```

6. After finishing configuration of CapacityScheduler, execute the "`yarn rmadmin -refreshQueues`" command to apply changes.
7. Go to the scheduler page of Resource Manager Web UI to check if you have successfully set the configuration.

Figure 8 shows the YARN Resource Manager GUI with the configured queues after following the above steps.

NEW,NEW_SAVING,SUBMITTED,A

Cluster: About, Nodes, Node Labels, Applications (NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED), Scheduler

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved
57	0	0	57	0	0 B	2.53 TB	0 B

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	
Capacity Scheduler	[MEMORY]	<memory:204

Dump scheduler logs 1 min

Application Queues

Legend: Capacity Used Used (over capacity) Max Capacity

Partition: <DEFAULT_PARTITION> <memory:0, vCores:0>
Queue: root
Partition: CPU <memory:2211840, vCores:450>
Queue: root
Queue: CPU_Q
Partition: GPU <memory:442368, vCores:90>
Queue: root
Queue: GPU_Q

Figure 8. YARN Resource Manager GUI with GPU and CPU queues

YARN configuration optimizations for compute nodes

As mentioned above, Spark on Hadoop uses YARN as the cluster manager, so it is important that YARN and Spark configurations are tuned in tandem. Settings of Spark executor memory and executor cores result in allocation requests to YARN with the same values and YARN should be configured to accommodate the desired Spark settings.

It is recommended that 75% of memory be allocated for Spark, leaving the rest for OS and buffer cache. For a recommended configuration of 768GB memory, this means NodeManager should be configured to use 576GB memory.

To configure NodeManager to use 576GB memory and 64 vcores use the following configuration settings:

- Amount of physical memory that can be allocated for containers:

```
yarn.nodemanager.resource.memory-mb  576 GiB
```

- Amount of vcores available on a compute node that can be allocated for containers:

```
yarn.nodemanager.resource.cpu-vcores 64
```

Configuring the number of YARN containers depends on the nature of the workload. The general guideline is to configure containers in a way that maximizes the utilization of the vcores and memory on each node in the cluster.

- The node-locality-delay specifies how many scheduling intervals to let pass attempting to find a node local slot to run on prior to searching for a rack local slot. This setting is very important for small jobs that do not have a large number of maps or reduces as it will better utilize the compute nodes. We highly recommend this value be set to 1.

```
yarn.scheduler.capacity.node-locality-delay 1
```

- Specify the location of YARN local log files on the compute nodes to point to SSDs:

```
yarn.nodemanager.log-dirs  /data1/hadoop/yarn/log, /data2/hadoop/yarn/log

yarn.nodemanager.local-dirs  /data1/hadoop/yarn/local, /data2/hadoop/yarn/local
```

Spark configuration guidance

Of the various ways to run Spark applications on Hadoop cluster, Spark on YARN mode is best suited to run them, as it leverages YARN services for resource allocation, runs Spark executors in YARN containers, and supports workload management and security features. Spark on HPE EPA is configured to run as Spark on YARN in two ways: client and cluster modes. In YARN cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away once the application is initiated. In YARN client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN. Typically, most development is done in YARN client mode and uses Spark interactively. YARN-cluster mode is ideal for production workloads.

Normally Spark executors can make use of multiple cores efficiently, but for TensorFlowOnSpark, it is recommended to have just one core per executor. This will help in mapping one TensorFlow worker node to one executor and all the executor logs will contain the logs for just one TensorFlow worker node which helps in debugging.

TensorFlow configuration guidance

TensorFlow needs a Python distribution (2.7+ or 3.5+) to run. While it can be zipped and can be shipped to the Spark executors at runtime, it might be better to install it on all compute/AI compute nodes thereby avoiding copying it each time.

Once Python is installed on compute/AI compute nodes, you can "`pip install tensorflow tensorflowonspark`" on compute nodes and "`pip install tensorflow-gpu tensorflowonspark`" on AI compute nodes (along with any other dependencies you might need for your application). For an Intel optimized version of TensorFlow, you can follow the instructions provided at Intel Optimization for TensorFlow Installation Guide.

As mentioned above, TensorFlow access to HDFS requires HDFS native client library (`libhdfs.so`), it has to be installed on all compute/AI compute nodes and `LD_LIBRARY_PATH` environment variable point to the location of the library. Also, TensorFlow requires the path to libjvm.so library to be set in LD_LIBRARY_PATH and path to `libcuda*.so` libraries to be set on AI compute nodes to make use of GPUs. The environment variable can be set in user profile or as part of spark submit job using `--spark.executorEnv.LD_LIBRARY_PATH` option.

The following are example environment settings:

```
export LIB_HDFS=/usr/hdp/2.6.4.0-91/usr/lib
export LIB_JVM=/usr/java/jdk1.8.0_151/jre/lib/amd64/server
export LIB_CUDA=/usr/local/cuda-9.0/bin         # on GPU nodes
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$LIB_JVM:$LIB_HDFS:$LIB_CUDA
```

TFRecord input format

TensorFlow project strongly suggests using TFRecord input format as TensorFlow code is optimized for TFRecord input format. It also combines multiple files into records of a single large file making it suitable for storage systems like HDFS. So we recommend that all input should be converted into TFRecord format where feasible. To use the TFRecord format on Hadoop/Spark cluster, you need to build a jar file using the following instructions:

```
git clone https://github.com/tensorflow/ecosystem.git
cd ecosystem/hadoop
mvn versions:set -DnewVersion=1.8.0
mvn clean package install
hdfs dfs -put target/hadoop-1.8.0.jar tensorflow-hadoop-1.8.0.jar
```

Once the `tensorflow-hadoop-1.8.0.jar` is built and copied to HDFS, it can be passed to the Spark job using `spark-submit --jars` option.

Capacity and sizing

Storage

HDFS storage sizing requires careful planning and identifying the current and future storage and compute needs. Use the following as general guidelines for data inventory:

- Sources of data
- Frequency of data
- Raw storage
- Processed FS storage
- Replication factor
- Default compression turned on
- Space for intermediate files

YARN/Spark

As mentioned before, configuring the number of YARN containers depends on the nature of the workload. For Spark workloads, the containers host Spark executors and the number of executor containers depends on the executor vcore and memory requirements. It is recommended to have one core per executor for easier debugging purposes.

TensorFlow/DL

The choice of GPU and the number of GPUs depends on the DL workloads and HPE recommends to make use of HPE Deep Learning Cookbook which provides a massive collection of performance results for various deep learning workloads on different HW/SW stacks, and analytical performance models. The combination of real measurements and analytical performance models enables estimation of the performance of any workload.

HPE Sizer for the Elastic Platform for Big Data Analytics

HPE has developed the HPE Sizer for the Elastic Platform for Big Data Analytics to assist customers with proper sizing of these environments. Based on design requirements, the sizer will provide a suggested bill of materials (BOM) and metrics data for an HPE EPA asymmetric architecture cluster which can be modified further to meet customer requirements.

To download the HPE Sizer for the Elastic Platform for Big Data Analytics, visit hpe.com/info/sizers.

Implementing a proof-of-concept

To evaluate TensorFlowOnSpark on HPE EPA for AI, a couple of use cases were chosen: Computer Vision/Image classification and Ad Click-Through Rate prediction. For image classification use case, three different workloads were used for evaluation: MNIST hand-written digit classification using a simple Fully Connected (FC) model, CIFAR10 image classification using a convolutional neural network (CNN) model and ImageNet classification using more complex Inception v3 model from Google.

Note

This evaluation was done mainly to demonstrate that HPE EPA can be extended with AI compute nodes to perform distributed DL training using Spark and TensorFlow on representative workloads. There has been no attempt to get the best possible metrics from this exercise, particularly there was no hyper parameter tuning done to get the best model parameters and achieve best possible metrics.

MNIST digit classification with Fully Connected model

The MNIST database of handwritten digits (0 to 9) is a good dataset for people who want to try learning techniques and pattern recognition methods on simple image data. It has a training set of 60,000 examples, and a test set of 10,000 examples, where each image size is 28 x 28 x 1 (grayscale). When getting started with Deep Learning, it is a good idea to start with the MNIST dataset and get familiarized with Deep Learning frameworks and environment by training a model on MNIST dataset.

Experiments were conducted to get familiar with distributed TensorFlow training and with TensorFlowOnSpark framework, to evaluate the training metrics using CPUs and GPUs and to evaluate the scalability of the training with the addition of nodes. The model used is a very simple linear (fully connected) model with one input layer, one hidden layer and a softmax layer. It is to be emphasized that distributed training using multiple nodes on such a simple workload will not improve training times as the overhead of the framework is more than the performance gains. MNIST workload is used here only to get familiar with TensorFlowOnSpark framework and will help in resolving any configuration issues while getting started with distributed training. Experiments were conducted using CPUs and GPUs but GPUs were not found to speed up the training as the model is very small. Figure 9 shows the model architecture used for this evaluation.

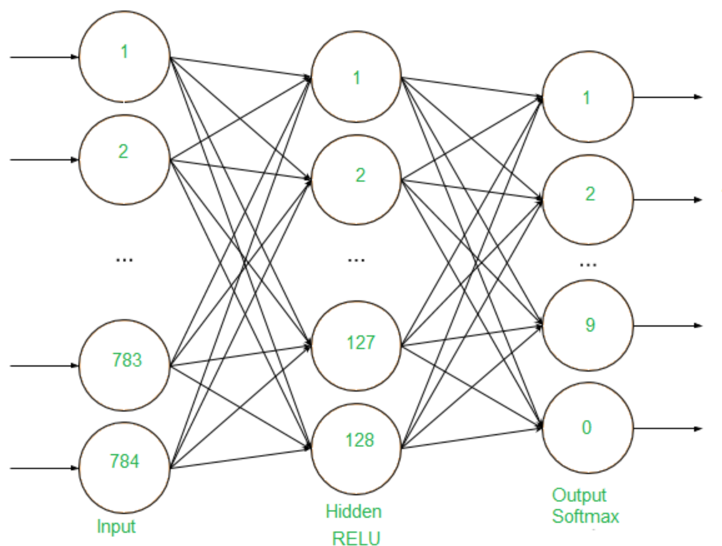


Figure 9. Graph of MNIST training model

Preprocessing

The downloaded dataset (mnist.zip) was preprocessed to convert the images into more efficient TFRecord format and to store the dataset on HDFS. TFRecord format combines the separate image and label information into a single record and also allows to combine multiple images into a single large file suitable for HDFS storage. TensorFlow code is optimized for TFRecord input format so it is strongly recommended to use it.

The preprocessing was done using the following command:

```
spark-submit    --master yarn -queue CPU_Q --deploy-mode cluster --num-executors 10 \
                --archives mnist/mnist.zip#mnist \
                --jars hdfs:///user/root/tensorflow-hadoop-1.8.0.jar examples/mnist/mnist_data_setup.py \
                --output mnist/tfr \
                --format tfr
```

Distributed MNIST training

Experiments were conducted to perform distributed training on MNIST dataset using CPUs and GPUs.

For training using CPUs the following command was used:

```
spark-submit     --master yarn --deploy-mode cluster --queue CPU_Q --num-executors 3
                 --py-files tfspark.zip,examples/mnist/tf/mnist_dist.py  \
                    examples/mnist/tf/mnist_spark.py \
                 --images hdfs://default/user/root/mnist/tfr/train
                 --format tfr --mode train --model mnist_model --steps 100000
```

For training using GPUs the following command was used:

```
spark-submit     --master yarn --deploy-mode cluster --queue GPU_Q --num-executors 3
                 --py-files tfspark.zip,examples/mnist/tf/mnist_dist.py  \
                    examples/mnist/tf/mnist_spark.py \
                 --images hdfs://default/user/root/mnist/tfr/train
                 --format tfr --mode train --model mnist_model --steps 100000
```

Figure 10 shows the accuracy during the training process using 2 worker nodes with CPUs and GPUs for 100000 steps. We can see that GPUs do not improve the training process for this simple model.

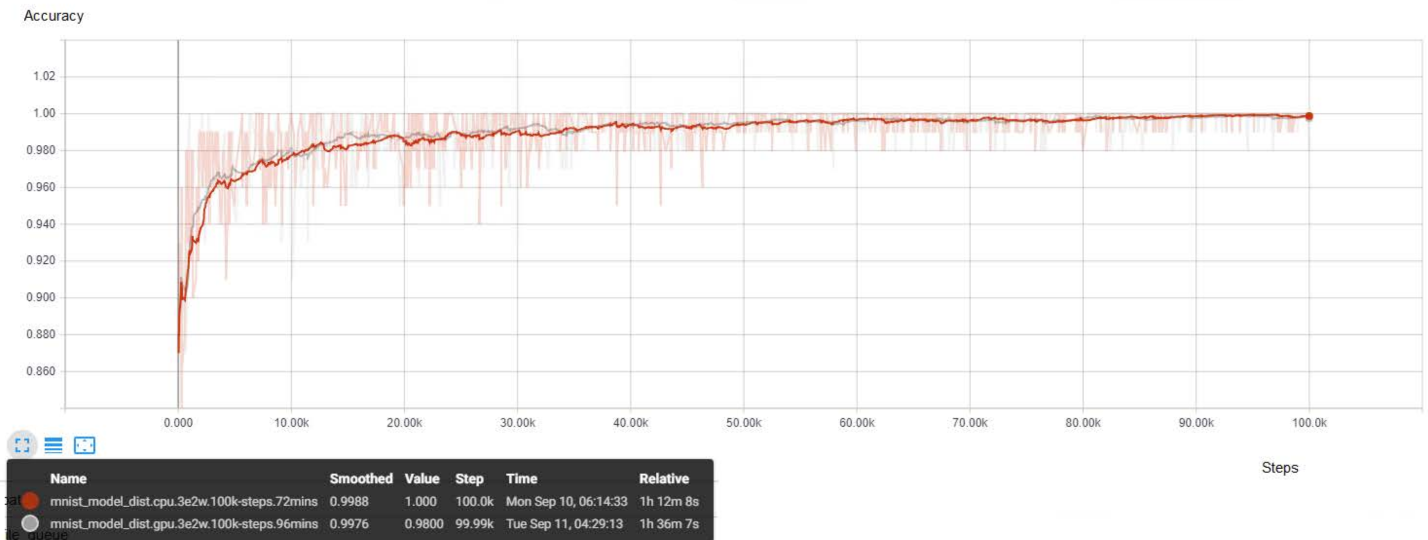


Figure 10. Training accuracy using 2 worker nodes with CPUs and GPUs

Figure 11 shows the accuracy during the training process using 2, 4 and 10 worker nodes with CPUs for 100000 steps. We can see that adding more nodes improve the training process by achieving faster convergence for a given level of accuracy and result in shorter training times.

Figure 11. Training accuracy using 2, 4 and 10 worker nodes with CPUs

Figure 12 shows the loss (metric that measures the inaccuracy of the prediction) during the training process using 2, 4 and 10 worker nodes with CPUs for 100000 steps. We can see that adding more nodes accelerates the training process and result in shorter training times.

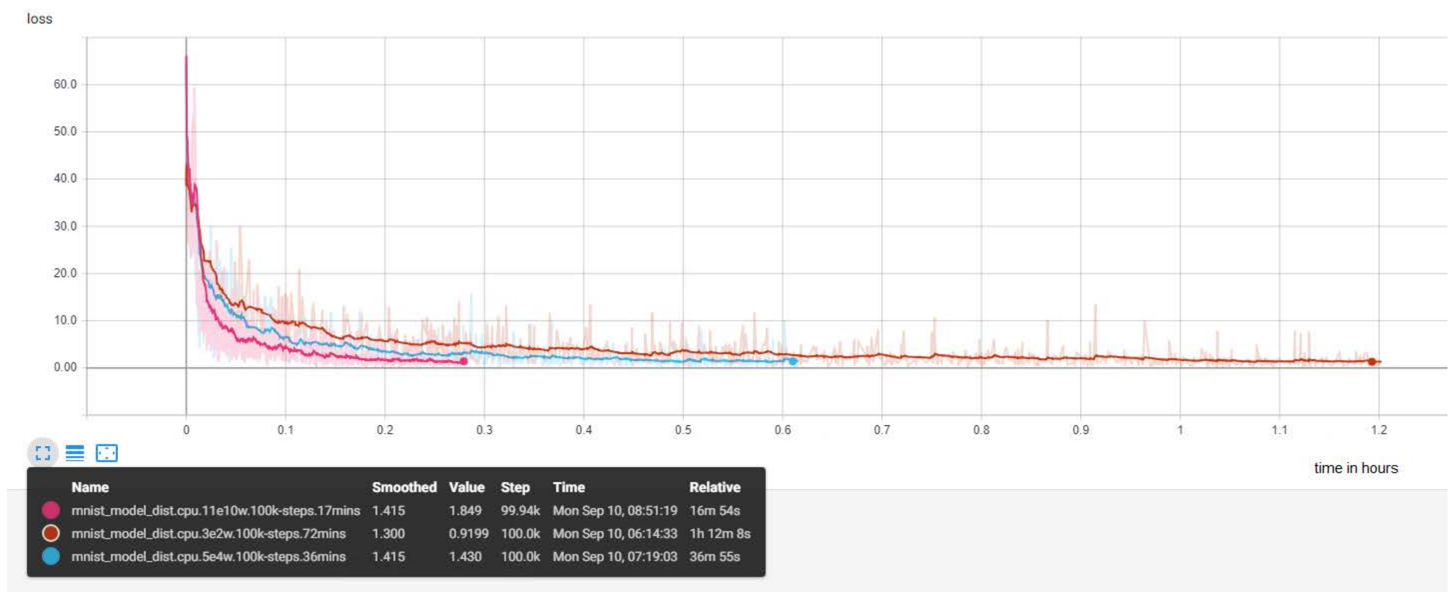


Figure 12. Training loss using 2, 4 and 10 worker nodes with CPUs

CIFAR-10 Image classification with Convolutional Neural Network model

While MNIST dataset is good for learning purposes, it does not represent a real world use case. CIFAR-10 dataset provides a better real world image workload for image classification use case that is not simple as MNIST dataset and not as complex and resource intensive as ImageNet dataset (described below). CIFAR-10 has 10 categories of images: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are completely mutually exclusive e.g. there is no overlap between automobiles and trucks.

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

A convolutional neural network (CNN) model was trained for recognizing images. The model is a multi-layer architecture consisting of alternating convolutions and nonlinearities. These layers are followed by fully connected layers leading into a softmax classifier. The model follows the architecture similar to Alexnet. The model consists of 1,068,298 learnable parameters and requires about 19.5M multiply-add operations to compute inference on a single image. This model achieves a peak performance of about 87% accuracy within a few hours of training time on a GPU.

Figure 13 shows the model graph as generated by TensorBoard.

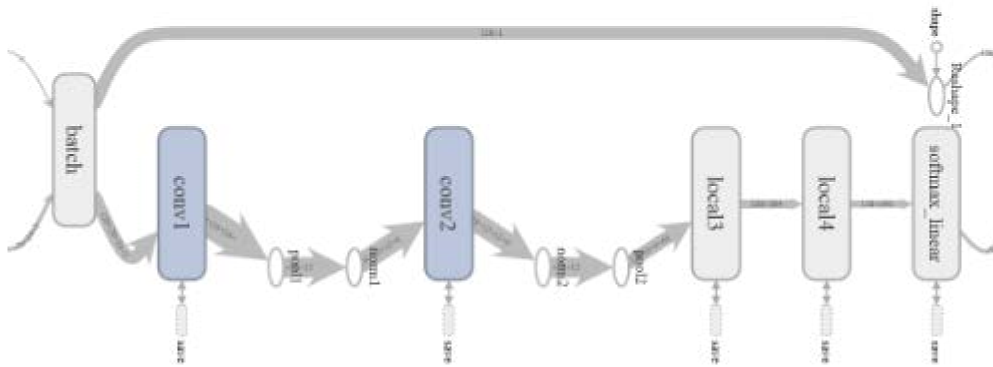


Figure 13. Graph of CIFAR-10 training model

Training

Experiments were done with 1 and 2 GPUs to see how the training improves when the number of GPUs is increased.

First, set some environment variables for convenience:

```
export TFoS_HOME= /home/viplav/TensorFlowOnSpark                          #<path to TensorFlowOnSpark>
pushd ${TFoS_HOME}/examples/cifar10; zip -r ~/cifar10.zip .; popd
export CIFAR10_DATA= hdfs://default/user/${USER}/cifar10_data       #<HDFS path to downloaded CIFAR-10 dataset>
export CIFAR10_TRAIN= hdfs://default/user/${USER}/cifar10_train     #<HDFS path to trained model directory>
export CIFAR10_EVAL= hdfs://default/user/${USER}/cifar10_eval       #<HDFS path to evaluation directory>
export NUM_GPU=2
```

Following command was used for start the training:

```
spark-submit \
--master yarn \
--deploy-mode cluster \
--queue GPU_Q \
--num-executors 1 \
--py-files ${TFoS_HOME}/tfspark.zip,cifar10.zip \
${TFoS_HOME}/examples/cifar10/cifar10_multi_gpu_train.py \
--data_dir ${CIFAR10_DATA} \
--train_dir ${CIFAR10_TRAIN} \
--max_steps 1000000 \
--num_gpus ${NUM_GPU}
```

Training loss

Figure 14 shows the graph of training loss with 1 GPU after training for 200,000 and 300,000 steps. As can be seen, training loss value is 0.3054 after 200K steps and improves to 0.1935 after 300K steps.

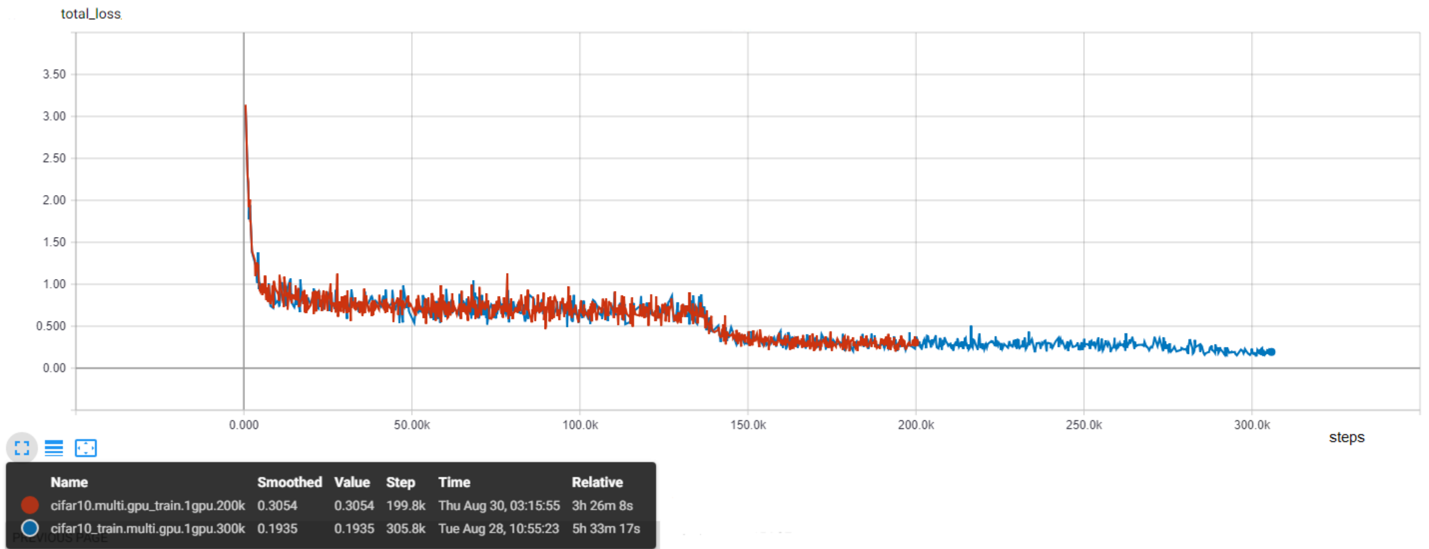


Figure 14. CIFAR-10 training loss progress with 1 GPU

Figure 15 shows the graph of training loss with 2 GPUs after training for 200,000 steps. As can be seen, training loss value improves to 0.1358 after only 200K steps.

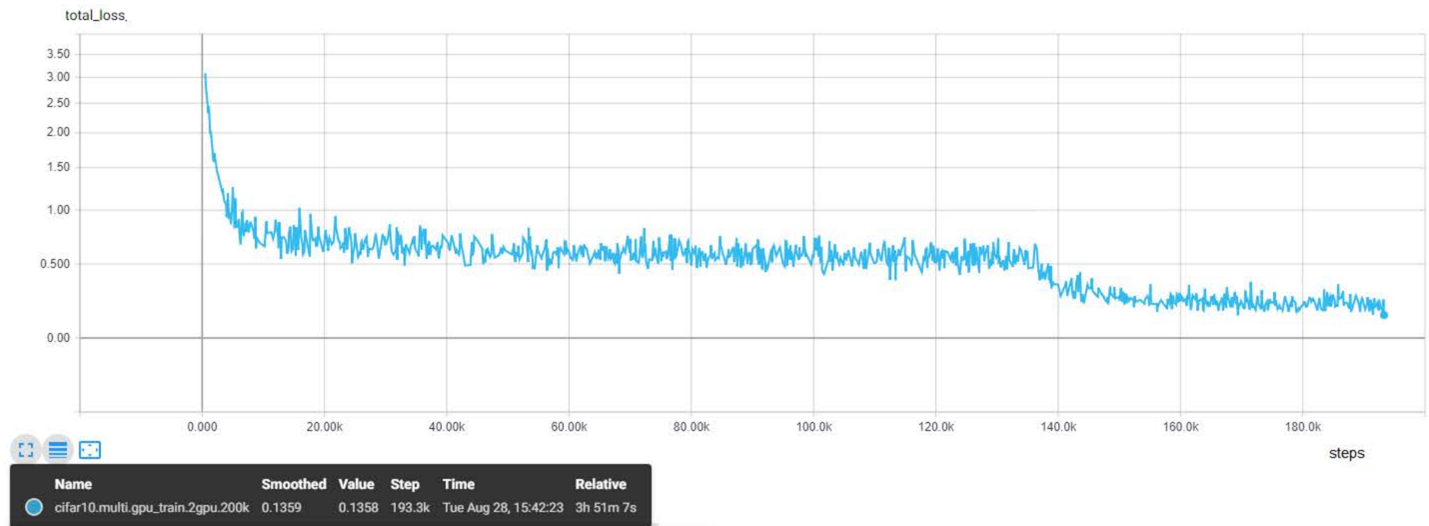


Figure 15. CIFAR-10 training loss progress with 2 GPUs

Training with CPUs using one node and 45 cores, after 1 million steps and more than a day, we get a training loss of 0.1226 as shown below. Figure 16 shows the graph of the training loss progress for this training.

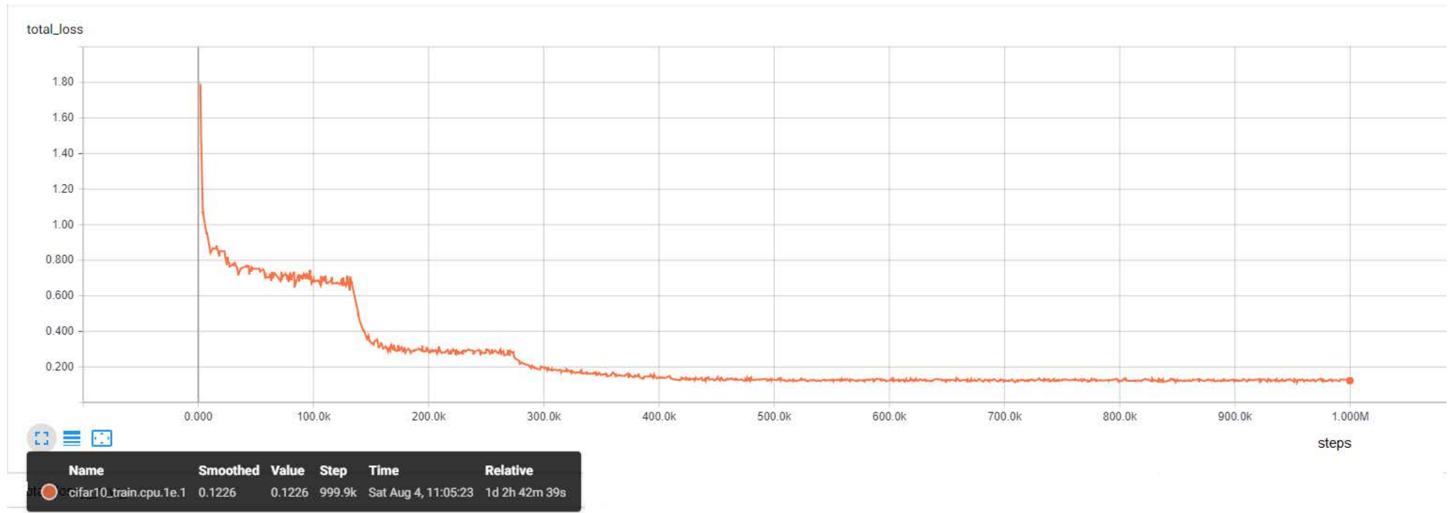


Figure 16. CIFAR-10 training loss progress with CPU node and 1 million steps

Training precision/accuracy

While training with 1 GPU, the precision metric value reaches the peak of 0.8702 (accuracy of 87.02% in classifying images) after 250,000 steps and 4-1/2 hours. Figure 17 shows the graph of the training loss progress for this training.

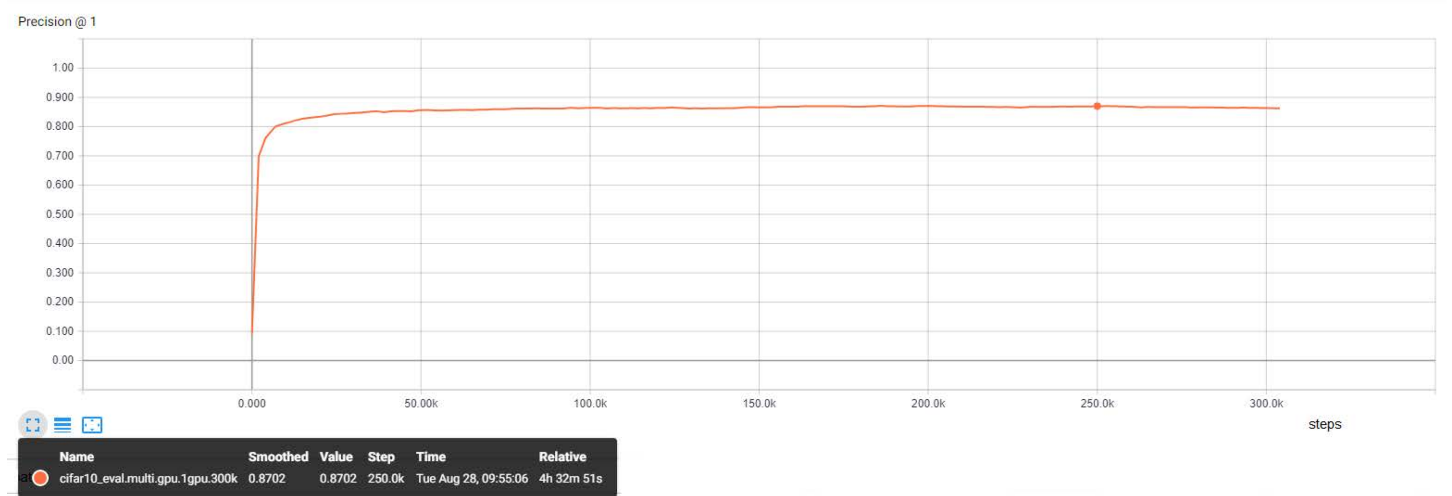


Figure 17. CIFAR-10 training precision progress with 1 GPU node

While training with 2 GPUs, the precision metric value reaches the peak of 0.8703 only after 90,000 steps and 1 hour 45 minutes, as shown in Figure 18.

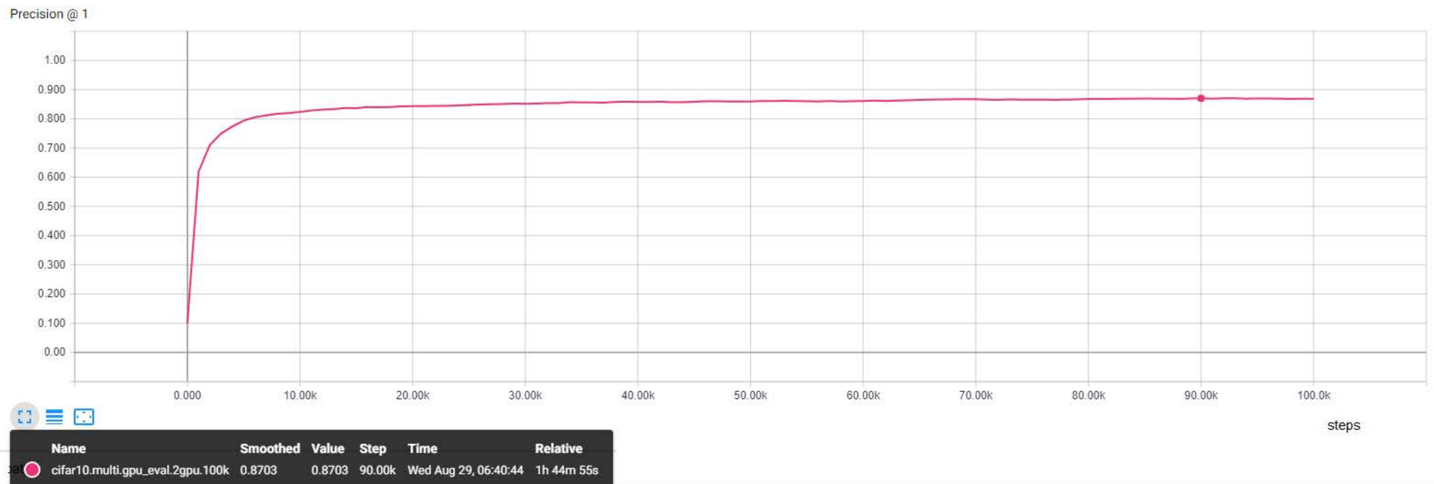


Figure 18. CIFAR-10 training precision progress with 2 GPU nodes

Training Throughput

Figure 19 shows the training throughput with 1 GPU / 2 GPUs.

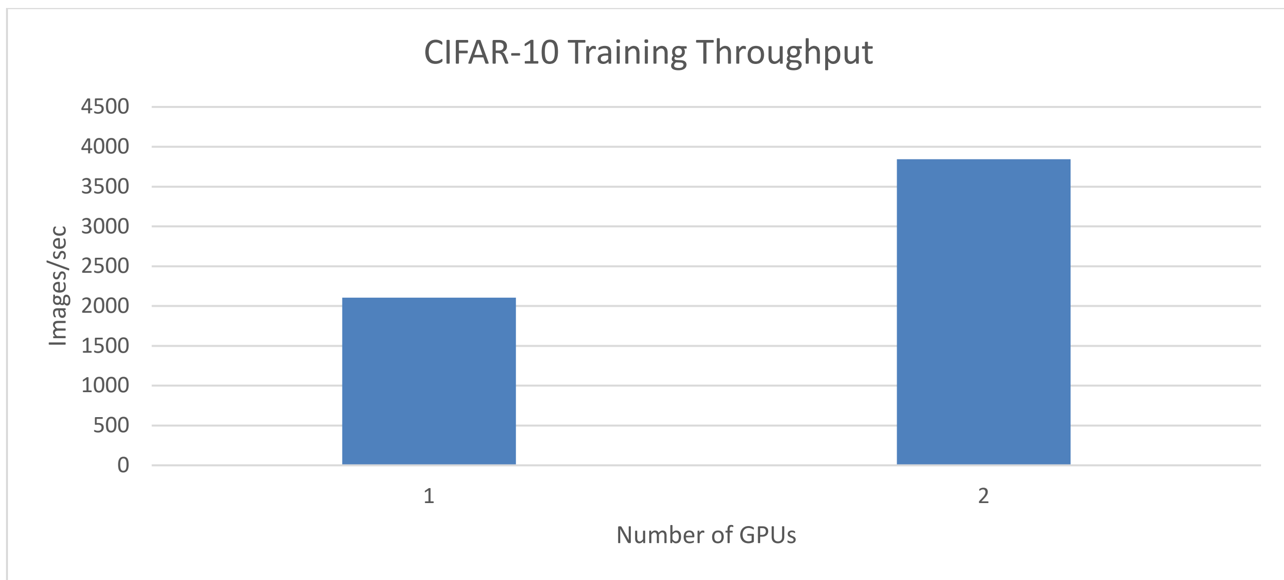


Figure 19. CIFAR-10 batch inference throughput with 1 and 2 GPU nodes

We can see that training throughput scales almost linearly with number of GPUs.

Inference

After the model is trained (with 87% accuracy) on CIFAR-10 dataset, experiments were conducted to evaluate the inference throughput. It was found that CPUs were good enough to provide a good batch inference throughput. Also, the throughput scaled completely in linear fashion as more nodes were used for the inference tasks in parallel.

To measure the inference throughput, following command was used:

```
spark-submit \
--master yarn \
--deploy-mode cluster \
--queue CPU_Q \
--num-executors 10 \
--py-files ${TFoS_HOME}/tfspark.zip,cifar10.zip \
${TFoS_HOME}/examples/cifar10/cifar10_eval.py \
--data_dir ${CIFAR10_DATA} \
--checkpoint_dir ${CIFAR10_TRAIN} \
--eval_dir ${CIFAR10_EVAL}
```

Figure 20 shows the chart of batch inference throughput scaling.

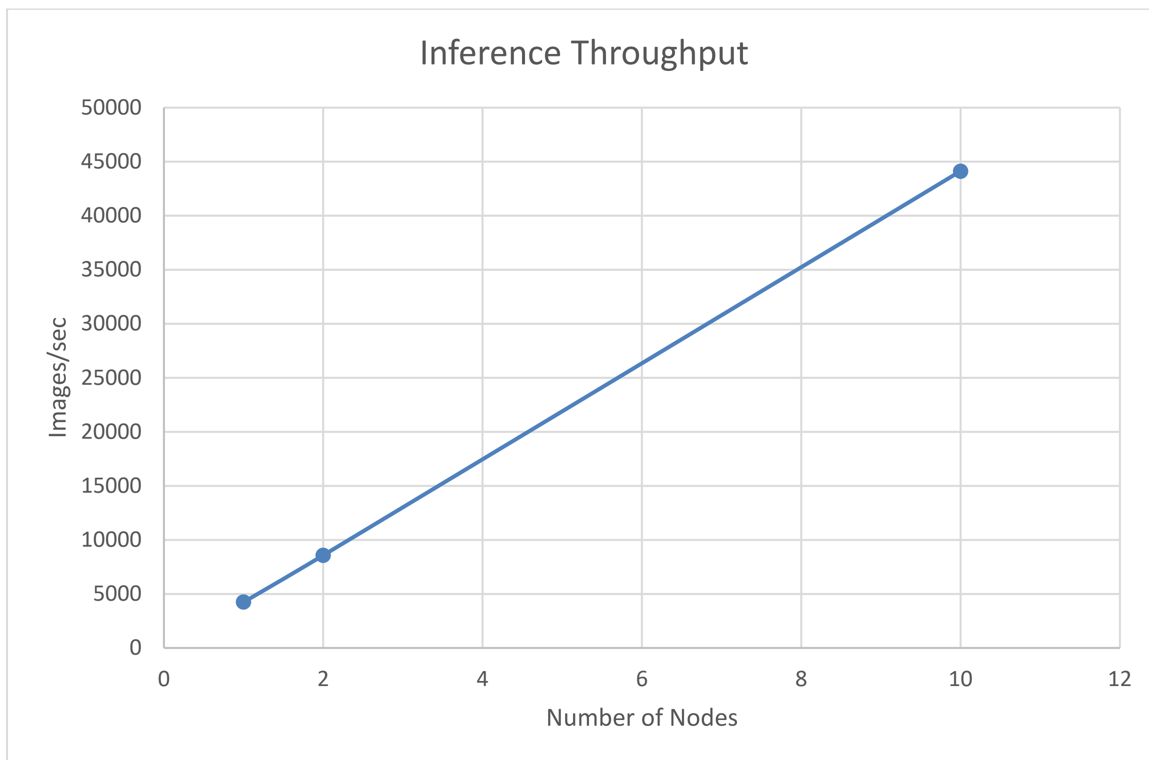


Figure 20. CIFAR-10 batch inference throughput scaling with CPU nodes

ImageNet with Inception v3 model

ImageNet is a large-scale image database organized according in a hierarchical fashion and is part of a research effort inspired by a growing sentiment in the image and vision research field – the need for more data. Images of each concept are quality-controlled and human-annotated. ImageNet contains over 14 million images in 20 thousand categories; a typical category contains several hundred images.

Distributed TensorFlow with TensorFlowOnSpark was used to train and evaluate a convolutional neural network (CNN) model using Google's Inception v3 architecture. Inception v3 is a widely-used image recognition model that has been shown to attain 21.2% top-1 and 5.6% top-5 error on the ImageNet dataset. The model itself is made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max

pooling, concats, dropouts, and fully connected layers. BatchNorm is used extensively throughout the model and applied to activation inputs. Loss is computed via Softmax. This network has a computational cost of 5 billion multiply-adds per inference and has about 25 million parameters. Figure 21 shows a visualization of the model architecture. In this distributed Inception v3 example training was done using synchronous gradient updates.

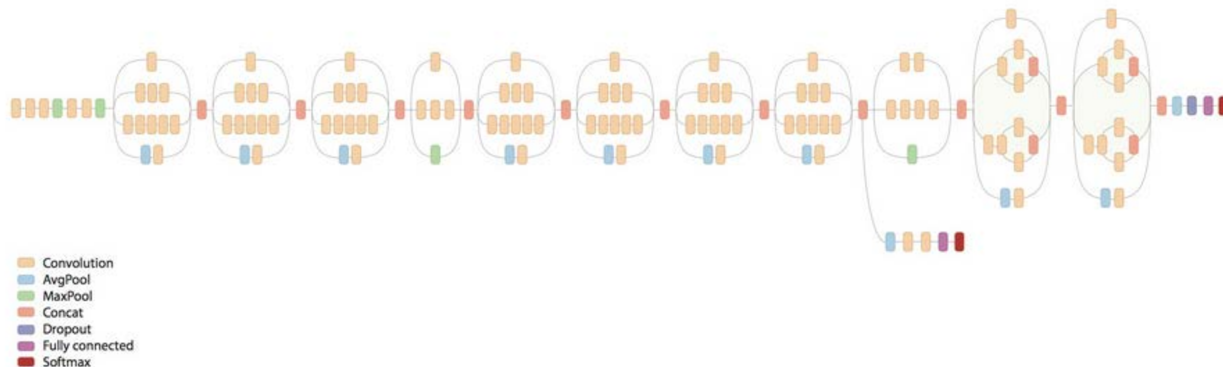


Figure 21. Inception V3 architecture for ImageNet training

Distributed ImageNet training

Before running the training script for the first time, the ImageNet data will need to be downloaded and converted to more efficient TFRecord format. The TFRecord format consists of a set of shared files where each entry contains the ImageNet image as well as metadata such as label etc. All the images are stored as TFRecords in 1024 training files and 128 validation files.

Training an Inception v3 network from scratch is a computationally intensive task and may take several days depending on the resources available.

Experiments were conducted to perform distributed training on MNIST dataset using CPUs and GPUs.

Following command was used for start the distributed training using GPUs:

```
spark-submit    --master yarn --deploy-mode cluster --queue GPU_Q --num-executors 3 \
                --py-files tfspark.zip,inception.zip \
                examples/imagenet/inception/imagenet_distributed_train.py \
                --data_dir hdfs://default/user/root/imagenet-data \
                --train_dir hdfs://default/user/root/imagenet_train \
                --max_steps 100000
```

Following command was used for start the distributed training using CPUs:

```
spark-submit    --master yarn --deploy-mode cluster --queue CPU_Q --num-executors 9 \
                --py-files tfspark.zip,inception.zip \
                examples/imagenet/inception/imagenet_distributed_train.py \
                --data_dir hdfs://default/user/root/imagenet-data \
                --train_dir hdfs://default/user/root/imagenet_train \
                --max_steps 100000
```

While training an Inception v3 model, evaluation of the model can be done using the ImageNet validation data set by running a separate job in parallel. This calculates the Precision @1 and Recall @5 metrics over the entire validation data periodically. The Precision @1 measures how

often the highest scoring prediction from the model matched the ImageNet label. The Recall @5 measures how often the top 5 scoring predictions from the model included the correct ImageNet label.

Following command was used for start the evaluation job:

```
spark-submit     --master yarn --deploy-mode cluster --queue GPU_Q --num-executors 1 \
                 --py-files tfspark.zip,inception.zip  \
                 examples/imagenet/inception/imagenet_eval.py \
                 --data_dir hdfs://default/user/root/imagenet-data \
                 --eval_dir hdfs://default/user/root/imagenet_eval \
                 --checkpoint_dir hdfs://default/user/root/imagenet_train \
                 --subset validation --num_examples 12000
```

Figure 22 shows the accuracy during the training process using 1, 2 worker nodes with GPUs and 8 worker nodes with CPUs for varying amounts of time. We can see that GPUs improve the training process tremendously for Inception v3 model. Also, using 2 GPUs improves the training time and also achieves higher precision in shorter amount of time. For example, to achieve 40% Precision @1 metric, it takes 22 hours for 8 CPU worker nodes, but takes only approximately 11.5 hours for 1 GPU node and 6 hours for 2 GPU worker nodes. Please note that the model was still learning and the experiments had to be terminated due to time constraints.

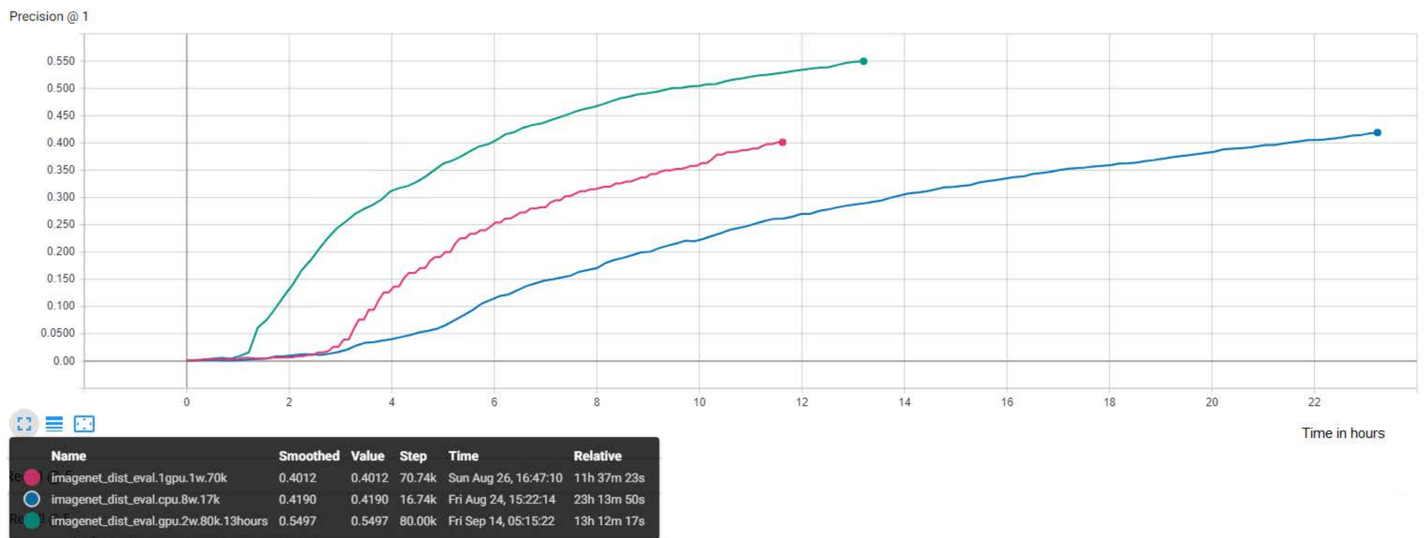


Figure 22. CIFAR-10 training Precision @1 progress with 1 GPU, 2 GPU and 8 CPU nodes

Figure 23 shows the Recall @5 metric during the training process. Again, we can see that to achieve 60% recall, it takes 16 hours for 8 CPU worker nodes, approximately 8.5 hours for 1 GPU node and 4.25 hours for 2 GPU worker nodes.

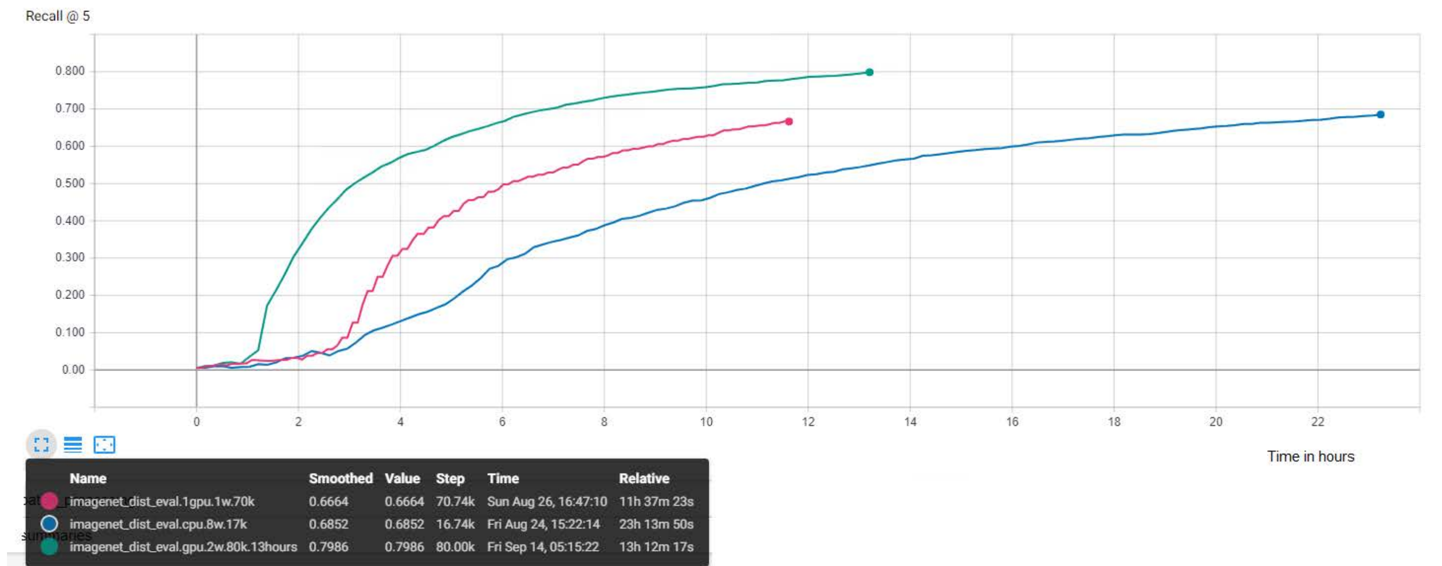


Figure 23. CIFAR-10 training Recall @5 progress with 1 GPU, 2 GPU and 8 CPU nodes

Criteo – Ad Click-Through Rate (CTR) prediction use case

While TensorFlow is an excellent tool for solving image classification type problems, it works equally well on more traditional machine-learning problems such as click prediction for large-scale advertising and recommendation scenarios. To demonstrate this capability, experiments were conducted to train a model to predict display ad click-through rate (CTR) on Criteo Labs clicks logs dataset. A logistic regression model was used for training with features extracted using techniques such as bucketization, hashing and crosses (combinations of two or more columns). Logistic regression models the probability of a click-through event and when working with rare events, probabilistic predictions are useful. Also it scales well to large datasets. Similar technique can be used for use cases like fraud detection and anomaly detection. While the model trained here is not the best CTR predictor, it achieves state-of-the-art performance, i.e. cross-entropy loss of 0.1236 on the validation dataset, as shown in the figure below. It is possible to use a Deep Neural Network model on this data to improve the results, but this could not be done due to time constraints.

The Criteo dataset that is approximately 370GB of gzip compressed TSV files (~1.3TB uncompressed), comprising more than 4.3 billion records. It is taken from 24 days of click data made available by Criteo. These logs contain feature values and click feedback from millions of display ads. Data from the first 23 days is used for training, and the last day serves as validation dataset.

The analytics workflow for data science problems contains three important steps of ML pipeline:

- ETL
- Data Exploration and feature selection, for example, using SQL
- Machine Learning

The example steps for these tasks is described in following sections.

ETL Preprocessing using Apache Pig

First, we download the dataset to HDFS storage. Here, in this exercise, it was stored in the directory hdfs://user/root/criteo-ctr-dataset, with training subdirectory containing first 23 days of data and validation subdirectory containing last day of data. Once we have the data in HDFS storage, we can preprocess the data using batch processing tools like Pig or MapReduce for efficient data storage, feature extraction and data augmentation etc.

In our case, we make use of Apache Pig to divide the training data in 10000 chunks, which will allow TensorFlowOnSpark to reduce its memory usage while training.

- Here are the Pig commands for training data preprocessing :

```
 grunt> SET mapred.output.compress 'true';
 grunt> SET mapred.output.compression.codec 'org.apache.hadoop.io.compress.BZip2Codec';
 grunt>train_data = LOAD  'criteo-ctr-dataset/day_{[0-9],1[0-9],2[0-2]}.gz ';
 grunt>train_data = FOREACH (GROUP train_data BY ROUND(10000* RANDOM()) PARALLEL 10000) GENERATE FLATTEN(train_data);
 grunt> STORE train_data INTO 'criteo-display-ctr-dataset/data/training/' USING PigStorage();
Vertex Stats:
VertexId Parallelism TotalTasks   InputRecords   ReduceInputRecords  OutputRecords  FileBytesRead FileBytesWritten  HdfsBytesRead
HdfsBytesWritten Alias        Feature Outputs
scope-28         23         23     4195197692                    0     4195197692   861128825016    1359110599803   352515279569
0 1-21,train_data
scope-29      10000      10000              0           4195197692     4195197692   687150522451     624606988355             0
353207046525 train_data   GROUP_BY       hdfs://m16hn01.localdomain:8020/user/root/criteo-display-ctr-dataset/data/training,

Input(s):
Successfully read 4195197692 records (352515279569 bytes) from: "hdfs://m16hn01.localdomain:8020/user/root/criteo-ctr-dataset/day_{[0-
9],1[0-9],2[0-2]}.gz"

Output(s):
Successfully stored 4195197692 records (353207046525 bytes) in: "hdfs://m16hn01.localdomain:8020/user/root/criteo-display-ctr-
dataset/data/training"
```

- Pig commands for validation data preprocessing:

```
 grunt> SET mapred.output.compress 'true';
 grunt> SET mapred.output.compression.codec 'org.apache.hadoop.io.compress.BZip2Codec';
 grunt> val_data = load 'criteo-ctr-dataset/day_23.gz ';
 grunt>val_data = FOREACH (GROUP val_data BY ROUND(100* RANDOM()) PARALLEL 100) GENERATE FLATTEN(val_data);
 grunt> STORE val_data INTO 'criteo-display-ctr-dataset/data/validation/' USING PigStorage();
 Vertex Stats:
VertexId Parallelism TotalTasks   InputRecords   ReduceInputRecords  OutputRecords  FileBytesRead FileBytesWritten  HdfsBytesRead
HdfsBytesWritten Alias        Feature Outputs
scope-43          1          1      178274637                    0      178274637    29052326407      57726483454    14941987580
0 1-73,val_data
scope-44        100        100              0            178274637      178274637    28863261074      26842900410             0
14966171565 val_data     GROUP_BY       hdfs://m16hn01.localdomain:8020/user/root/criteo-display-ctr-dataset/data/validation,

Input(s):
Successfully read 178274637 records (14941987580 bytes) from: "hdfs://m16hn01.localdomain:8020/user/root/criteo-ctr-dataset/day_23.gz"

Output(s):
Successfully stored 178274637 records (14966171565 bytes) in: "hdfs://m16hn01.localdomain:8020/user/root/criteo-display-ctr-
dataset/data/validation"
```

Exploratory Data Analysis (EDA) with Hive/Spark SQL

Before applying any machine learning technique, it is very important to analyze the data and to formulate hypotheses about which features and algorithms would be useful to tackle the problem. Based on this analysis, the features can be selected or derived for subsequent machine learning stage.

The Exploratory Data Analysis (EDA) can be done using any number of tools available on EPA platform e.g. PySpark, Hive, Spark SQL etc. In our example, Hive and Spark SQL are used for EDA task. In interest of time, data exploration was done on one day (Day 0) training data set only. All of these explorations can be done on the entire data set to get insights on the numeric and categorical variables prior to getting data ready for building models using TensorFlowOnSpark.

To analyze the data using Hive/Spark SQL, first a table is created in Hive using the following commands:

```
hive> CREATE DATABASE IF NOT EXISTS criteo;
OK
Time taken: 0.606 seconds

hive> CREATE TABLE criteo.criteo_train_day0 (
         col1 string,col2 double,col3 double,col4 double,col5 double,col6 double,col7 double,col8 double,col9 double,col10 double,col11
         double,col12 double,col13 double,col14 double,col15 string,col16 string,col17 string,col18 string,col19 string,col20
         string,col21 string,col22 string,col23 string,col24 string,col25 string,col26 string,col27 string,col28 string,col29
         string,col30 string,col31 string,col32 string,col33 string,col34 string,col35 string,col36 string,col37 string,col38
         string,col39 string,col40 string)
          ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
           LINES TERMINATED BY '\n'
           STORED AS TEXTFILE LOCATION '/user/root//criteo-ctr-dataset.copy/training-day0';
```

Make sure the hive table is created.

```
hive> show tables in criteo;
criteo_train_day0
```

Look at the first five rows to get a feel for the data. First column is the label, if the ad was clicked or not. Next 13 columns are numerical features and the rest are categorical features.

```
hive> SELECT *  FROM criteo.criteo_train_day0   LIMIT 5;
1        5.0     110.0   NULL    16.0    NULL    1.0     0.0     14.0    7.0     1.0     NULL    306.0   NULL    62770d79
          e21f5d58        afea442f     945c7fcf 38b02748     6fcd6dcb 3580aa21        28808903        46dedfa6        2e027dc1
          0c7c4231        95981d1f        00c5ffb7         be4ee537        8a0b74cc    4cdc3efa        d20856aa        b8170bba
          9512c20b        c38e2f28        14f65a5d        25b1b089        d7c1fc0b
          7caf609c        30436bfc     ed10571d
0       32.0    3.0     5.0     NULL    1.0     0.0     0.0     61.0    5.0     0.0     1.0     3157.0  5.0
          e5f3fd8d        a0aaffa6        6faa15d5        da8a3421   3cd69f23   6fcd6dcb ab16ed81        43426c29        1df5e154
          7de9c0a9        6652dc64        99eb4e27        00c5ffb7        be4ee537
        f3bbfe99        4cdc3efa  d20856aa        a1eb1511      9512c20b        febfd863        a3323ca1
          c8e1ee56        1752e9e8        75350c8a      991321ea        b757e957
0       NULL    233.0   1.0     146.0   1.0     0.0     0.0     99.0    7.0     0.0     1.0     3101.0  1.0
          62770d79        ad984203        62bec60d      386c49ee  e755064d     6fcd6dcb b5f5eb62        d1f2cc8b        2e4e821f
          2e027dc1        0c7c4231        12716184        00c5ffb7        be4ee537
         f70f0d0b       4cdc3efa        d20856aa        628f1b8d      9512c20b        c38e2f28        14f65a5d        25b1b089
         d7c1fc0b        34a9b905     ff654802     ed10571d
0       NULL    24.0    NULL    11.0    24.0    NULL    0.0     56.0    3.0     NULL    2.0     20456.0 NULL
          710103fd        c73d2eb5        0c758dfb         f1738f48        6fcd6dcb     e824fc11        09f8a09d        e25a4c11
          12716184        d49eb1df        b96f9e1a        2b083b96    10dd3744   1f7fc70b        a1eb1511        9512c20b
          dc209cd3        b8a81fb0        30436bfc        b757e957
0       60.0    223.0   6.0     15.0    5.0     0.0     0.0     1.0     8.0     0.0     2.0     1582.0  6.0
          02e197c5        c2ced437        a2427619        3f85ecae        b8c51ab7     6fcd6dcb 26d0f5bb        337bf7a5        e25a4c11
          6da2367e        bf624fa3        ec982ce0        a77a4a56        be4ee537
        eb24f585     4cdc3efa        d20856aa        d9f758ff      9512c20b        c709ec07        2b07677e        a89a92a5
        aa137169        e619743b        cdc3217e     ed10571d
Time taken: 0.041 seconds, Fetched: 5 row(s)
```

Continuing the EDA in Spark-SQL

Aggregates are useful in feature selection. To count the number of examples/rows in the training dataset:

```
spark-sql> SELECT COUNT(*) FROM criteo.criteo_train_day0;
195841983
Time taken: 295.565 seconds, Fetched 1 row(s)
```

It is always interesting to see the label distribution in the training dataset.

```
spark-sql> SELECT Col1, COUNT(*)  FROM criteo.criteo_train_day0 GROUP BY Col1;
0       189555458
1       6286525
Time taken: 420.608 seconds, Fetched 2 row(s)
```

We can notice that CTR is only 0.0331645686509328 or 3.33%.

For categorical columns, it is interesting to see the number of unique values they take. To check it for some categorical columns:

```
spark-sql> SELECT COUNT(DISTINCT(Col15)), COUNT(DISTINCT(Col16)), COUNT(DISTINCT(Col17)),
                  COUNT(DISTINCT(Col18)), COUNT(DISTINCT(Col19)), COUNT(DISTINCT(Col20))
                  FROM criteo.criteo_train_day0;
18576837        29427   15127   7295    19901   3
Time taken: 664.959 seconds, Fetched 1 row(s)
```

We can see that Column 15 has around 1.86 million unique values while column 20 has only 3 unique values.

It is also interesting to see what the distribution of the numeric variables looks like.

```
spark-sql> SELECT CAST(hist.x as int) as bin_center, CAST(hist.y as bigint) as bin_height FROM
          (SELECT histogram_numeric(col2, 20) as col2_hist FROM criteo.criteo_train_day0 ) a
          LATERAL VIEW explode(col2_hist) exploded_table as hist;
25      159326959
2505    166138
6419    37161
10367   15618
14302   8626
18073   5478
21715   3458
25028   2674
```

```
28128   2048
31350   1529
34426   1273
37468   1231
40211   758
42903   650
46126   594
49154   637
53262   501
57344   572
61313   476
65501   3127
Time taken: 553.872 seconds, Fetched 20 row(s)
```

We can see that most of the column values are around bin center 25.

The co-occurrence counts of pairs of categorical variables is also of interest to see which pairs are related and is useful in feature selection. To check it for some pairs:

```
spark-sql> SELECT Col15, Col16, COUNT(*) AS coocc_count FROM criteo.criteo_train_day0 GROUP BY Col15, Col16 ORDER BY coocc_count DESC
LIMIT 20;
ad98e872        cea68cd3        10542770
ad98e872        3dbb483e        7858349
ad98e872        43ced263        3466318
ad98e872        fb1e95da        2870464
ad98e872        ac4c5591        2731358
ad98e872        420acc05        2093945
ad98e872        e56937ee        2035590
265366bf        6f5c7c41        1643635
ad98e872        8af1edc8        1390916
e5f3fd8d        77cf2e67        1252089
ad98e872        977b4431        1170012
e5f3fd8d        a15d1051        1067649
ad98e872        d1fade1c        964227
265366bf        20fd45c1        954834
62770d79        e21f5d58        895531
91c3c8e0        63bf29ab        879714
ad98e872        dd86c04a        867037
e5f3fd8d        a0aaffa6        859049
62770d79        1a05a2a2        788271
ad98e872        107444b0        770826
Time taken: 507.116 seconds, Fetched 20 row(s)

spark-sql> SELECT Col15, Col17, COUNT(*) AS coocc_count FROM criteo.criteo_train_day0 GROUP BY Col15, Col17 ORDER BY coocc_count DESC
LIMIT 20;
ad98e872        417e6103        2128686
265366bf        bafcb47b        1152837
ad98e872        f19df729        837469
265366bf        fe54dcfd        788563
ad98e872        3eafa839        704636
ad98e872        4c64221a        618790
ad98e872        b6f8803d        461252
265366bf        17f8a1e6        440978
ad98e872        93e42a93        405389
265366bf        2e276d86        405243
ad98e872        f2485bfb        372877
ad98e872        8a734adc        342185
ad98e872        7ad6521e        311084
ad98e872        5bd4f067        305659
265366bf        94aa4398        303212
ad98e872        8cd20e71        292555
ad98e872        f4703a4d        274335
265366bf        471b4db9        268312
ad98e872        c3d2d035        244597
265366bf        bda1af36        241536
Time taken: 586.333 seconds, Fetched 20 row(s)
```

Distributed Training with TensorFlowOnSpark

Having explored the datasets and demonstrated how to do this type of exploration for any variables including combinations, we can use TensorFlowOnSpark to generate a model to predict the ad clicks. As mentioned above, a logistic regression model is used for training with features extracted using techniques such as bucketization, hashing and crosses.

Following command was used for start the distributed training.

```
#set environment variables to point to the HDFS locations of training, validation datasets and the location for   storing the trained
model
export TRAINING_DATA=hdfs://default/user/root/criteo-display-ctr-dataset/data/training
export VALIDATION_DATA=hdfs://default/user/root/criteo-display-ctr-dataset/data/validation
export MODEL_OUTPUT=hdfs://default/user/root/criteo_ctr_prediction

spark-submit --master yarn --deploy-mode cluster -queue CPU_Q --num-executors 9 \
                       --py-files tfspark.zip,examples/criteo/spark/criteo_dist.py \
                         examples/criteo/spark/criteo_spark.py \
                       --mode train \
                       --data ${TRAINING_DATA} \
                        --validation ${VALIDATION_DATA}\
                        --model ${MODEL_OUTPUT} --steps 100000
```

Experiments were conducted varying the number of worker nodes from 1 to 8 while using one PS (Parameter Server). The following are the results of some of these experiments.

Figure 24 shows the cross entropy loss during the training process using 1 worker node and 1 PS for 4 hours. We can see that the loss improves from 0.7 to 0.1363 on validation data in 4 hours of training and it takes 238 steps of training to achieve this loss.

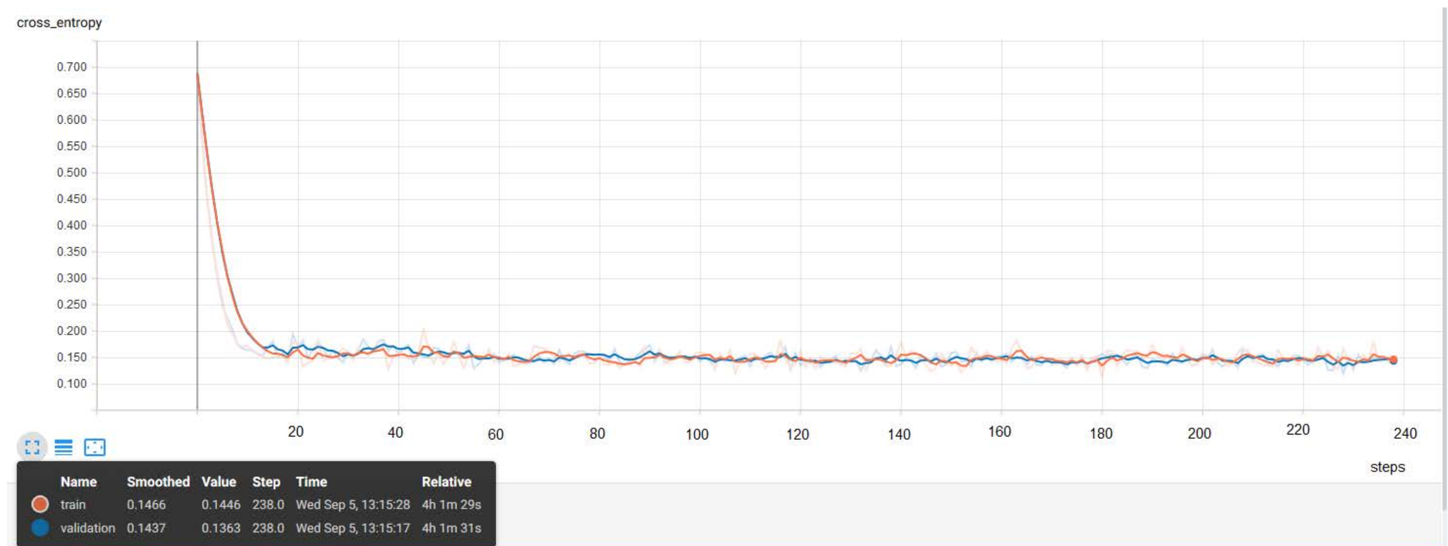


Figure 24. Criteo 1 TB dataset training loss progress with 1 worker node

Figure 25 shows the cross entropy loss during the training process using 4 worker nodes and 1 PS for 4 hours. We can see that the loss improves to 0.1469 on validation data and in about an hour of training. So for the same amount of loss/accuracy, we cut training time by 4 times by scaling worker nodes from 1 to 4. We also get more training steps (403) steps in this time.

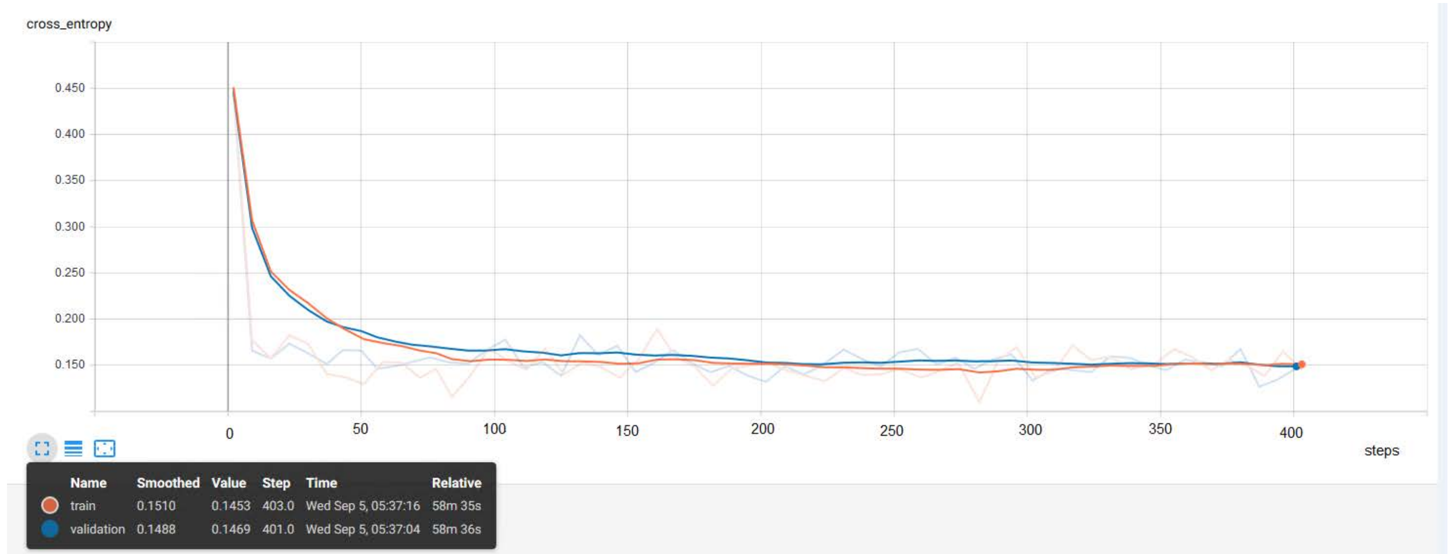


Figure 25. Criteo 1 TB dataset training loss progress with 4 worker nodes

Figure 26 shows the cross entropy loss during the training process using 8 worker nodes and 1 PS for 34 minutes. We achieve a loss of 0.1557 on validation data in just about 34 minutes of training. Again we cut training time and get more training steps done by scaling worker nodes from to 8.

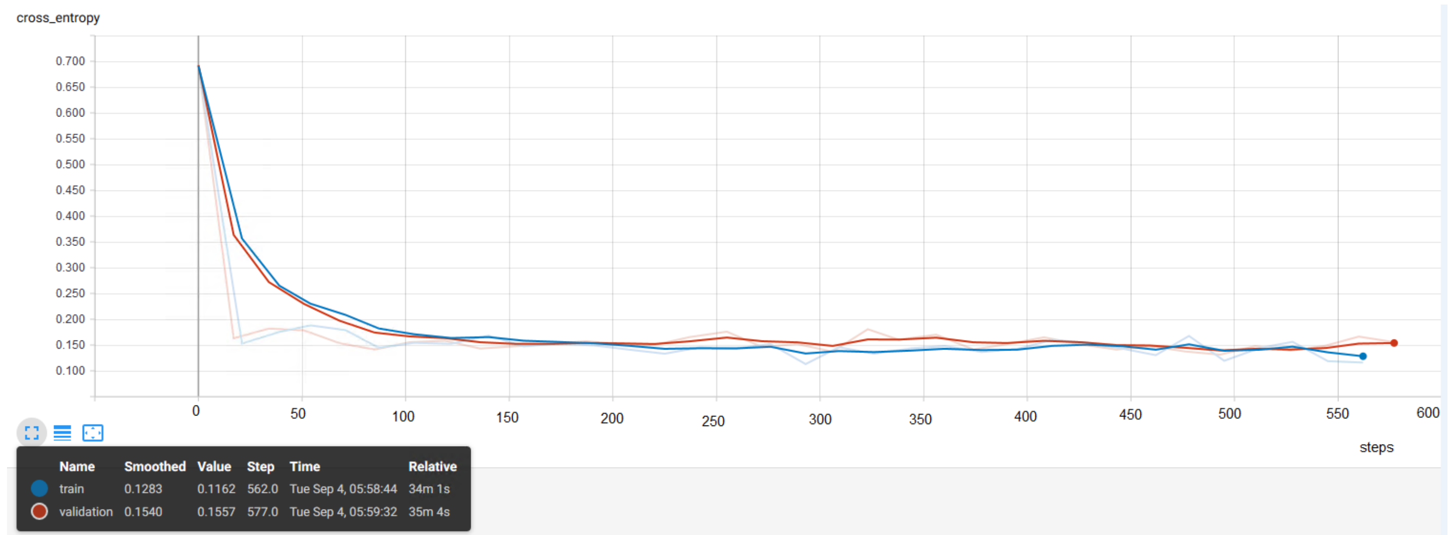


Figure 26. Criteo 1 TB dataset training loss progress with 8 worker nodes

Using 8 worker nodes and 1 PS for training, the model takes about 2 hours to train and reach an evaluation loss of 0.1236 which is on par with the state of the art results published on this dataset. Figure 27 shows the cross entropy loss during this training process.

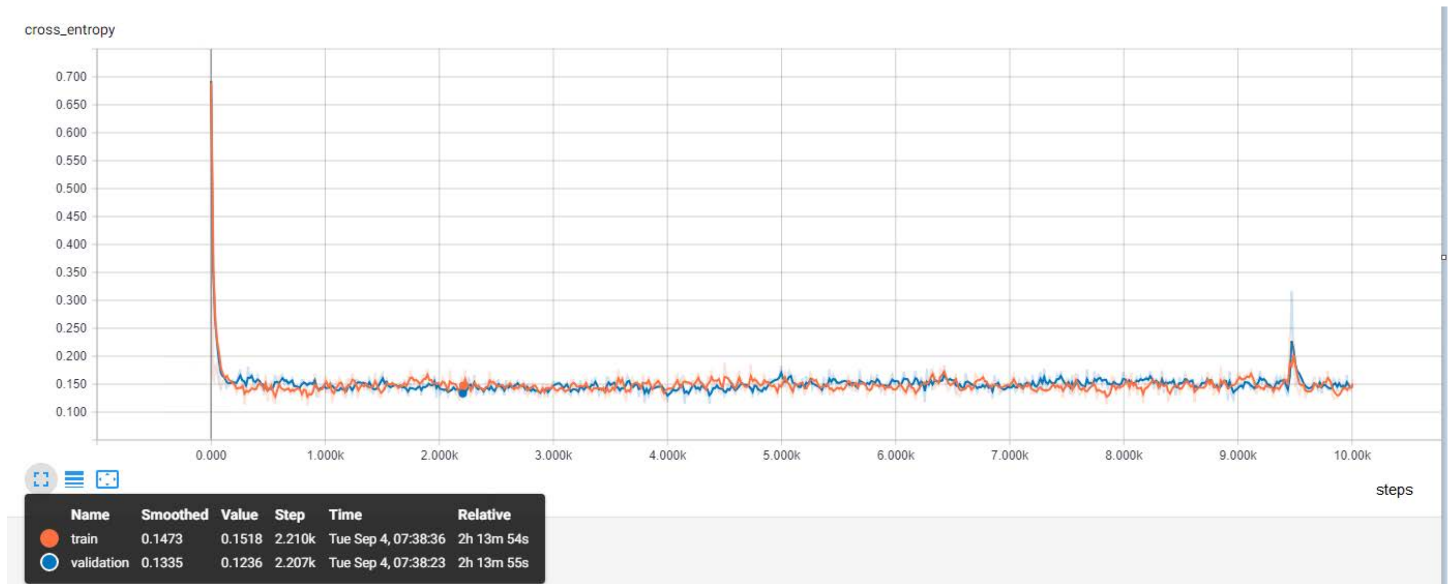


Figure 27. Criteo 1 TB dataset training loss progress for 2 hours with 8 worker nodes

Summary

Enterprises today see AI as a strategic priority to succeed but face significant barriers to AI realization, with lack of IT infrastructure as one of the top challenges. Only a very small fraction of real-world AI systems is the AI code, and the rest is the infrastructure/ecosystem surrounding it to get the AI pipeline working. Most of the AI effort is spent in configuring the ecosystem, collecting the data, verifying the data, and extracting the features from the raw data to feed to the AI algorithms.

The HPE EPA architecture enables flexible scale-out of compute and storage independent of each other, and is well suited for deploying and consolidating diverse workloads, including ML/DL workloads on a multi-tenant analytics platform. Spark, with memory accelerated HPE Apollo 2000 compute block, provides a very good solution to the ecosystem needs surrounding AI as it can connect to all the AI ecosystem components and can be used in ingesting and preprocessing the data for feature selection before AI model training. In addition, using YARN's multi-tenant capabilities along with YARN labels, it is possible to deploy deep learning workload-optimized HPE Apollo 2000 AI compute blocks with GPUs along with data processing workload-optimized HPE Apollo 2000 compute blocks. For the entry-level DL workloads, enterprises can get started with HPE Apollo 2000 AI compute block and can slowly graduate to HPE Apollo 6500 AI compute block for demanding DL workloads.

Distributed training using multiple GPUs can speed up the ML/DL training times but is hard and needs ecosystem support of resource management and distributed file system for input data and saving model checkpoints. For enterprises with existing big data infrastructure, YARN is an ideal choice as the resource manager and, as demonstrated in the proof-of-concept section, HDFS using HPE Apollo 4200 storage block provides a good solution to the distributed storage requirement of distributed training. TensorFlowOnSpark makes it smooth for enterprises to get started on AI projects because of its ability to migrate existing TensorFlow programs with few lines of change yet supporting all TensorFlow functionalities.

By extending the existing investments in big data infrastructure like HPE Elastic Platform for Big Data Analytics with AI accelerators and data processing frameworks like Hadoop, YARN, and Spark with ML/DL frameworks like TensorFlow, enterprises can get started with AI and leverage the full potential of this revolutionary technology.

Resources and additional links

Apache Spark, http://spark.apache.org

TensorFlow, http://www.tensorflow.org/

TensorFlowOnSpark, https://github.com/yahoo/TensorFlowOnSpark

HPE Servers for Big Data Analytics and Hadoop, hpe.com/info/hadoop

HPE Artificial Intelligence and Deep Learning Solutions, https://www.hpe.com/us/en/solutions/hpc-high-performance-computing/deep-learning.html

HPE Deep Learning Cookbook, https://developer.hpe.com/platform/hpe-deep-learning-cookbook/home

HPE Insight Cluster Management Utility (CMU), hpe.com/info/cmu

HPE FlexFabric 5900 switch series, hpe.com/us/en/product-catalog/networking/networking-switches/pip.fixed-port-l3-managed-ethernet-switches.5221896.html

HPE FlexFabric 5950 switch series, hpe.com/us/en/product-catalog/networking/networking-switches/pip.hpe-flexfabric-5950-switch-series.1008901775.html

HPE ProLiant servers, hpe.com/info/proliant

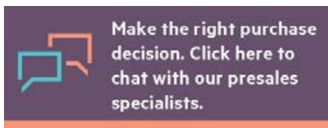
HPE Networking, hpe.com/networking

HPE Pointnext, hpe.com/services

HPE Sizer for the Elastic Platform for Big Data Analytics (HPE EPA Sizing Tool), https://h20195.www2.hpe.com/v2/GetDocument.aspx?docname=a00005868enw

HPE Education Services, http://h10076.www1.hpe.com/ww/en/training/portfolio/bigdata.html

To help us improve our documents, please provide feedback at hpe.com/contact/feedback.



Sign up for updates

© Copyright 2018 Hewlett Packard Enterprise Development LP. The information contained herein is subject to change without notice. The only warranties for Hewlett Packard Enterprise products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Hewlett Packard Enterprise shall not be liable for technical or editorial errors or omissions contained herein.

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc. Intel, Xeon, and Intel Xeon, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Microsoft is a registered trademark of Microsoft Corporation in the United States and/or other countries. NVIDIA, the NVIDIA logo, and [insert any other Nvidia marks used by agreement] are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries.

a00060456enw, November 2018