NetApp storage solutions for Apache Spark

NetApp artificial intelligence solutions

NetApp
December 04, 2025

Table of Contents

NetApp storage solutions for Apache Spark

TR-4570: NetApp Storage Solutions for Apache Spark: Architecture, Use Cases, and Performance Results

Rick Huang, Karthikeyan Nagalingam, NetApp

This document focuses on the Apache Spark architecture, customer use cases, and the NetApp storage portfolio related to big data analytics and artificial intelligence (AI). It also presents various testing results using industry-standard AI, machine learning (ML), and deep learning (DL) tools against a typical Hadoop system so that you can choose the appropriate Spark solution. To begin, you need a Spark architecture, appropriate components, and two deployment modes (cluster and client).

This document also provides customer use cases to address configuration issues, and it discusses an overview of the NetApp storage portfolio relevant to big data analytics and AI, ML, and DL with Spark. We then finish with testing results derived from Spark-specific use cases and the NetApp Spark solution portfolio.

Customer challenges

This section focuses on customer challenges with big data analytics and AI/ML/DL in data growth industries such as retail, digital marketing, banking, discrete manufacturing, process manufacturing, government, and professional services.

Unpredictable performance

Traditional Hadoop deployments typically use commodity hardware. To improve performance, you must tune the network, operating system, Hadoop cluster, ecosystem components such as Spark, and hardware. Even if you tune each layer, it can be difficult to achieve desired performance levels because Hadoop is running on commodity hardware that was not designed for high performance in your environment.

Media and node failures

Even under normal conditions, commodity hardware is prone to failure. If one disk on a data node fails, the Hadoop master by default considers that node to be unhealthy. It then copies specific data from that node over the network from replicas to a healthy node. This process slows down the network packets for any Hadoop jobs. The cluster must then copy the data back again and remove the over- replicated data when the unhealthy node returns to a healthy state.

Hadoop vendor lock-in

Hadoop distributors have their own Hadoop distribution with their own versioning, which locks in the customer to those distributions. However, many customers require support for in-memory analytics that does not tie the customer to specific Hadoop distributions. They need the freedom to change distributions and still bring their analytics with them.

Lack of support for more than one language

Customers often require support for multiple languages in addition to MapReduce Java programs to run their jobs. Options such as SQL and scripts provide more flexibility for getting answers, more options for organizing and retrieving data, and faster ways of moving data into an analytics framework.

Difficulty of use

For some time, people have complained that Hadoop is difficult to use. Even though Hadoop has become simpler and more powerful with each new version, this critique has persisted. Hadoop requires that you understand Java and MapReduce programming patterns, a challenge for database administrators and people with traditional scripting skill sets.

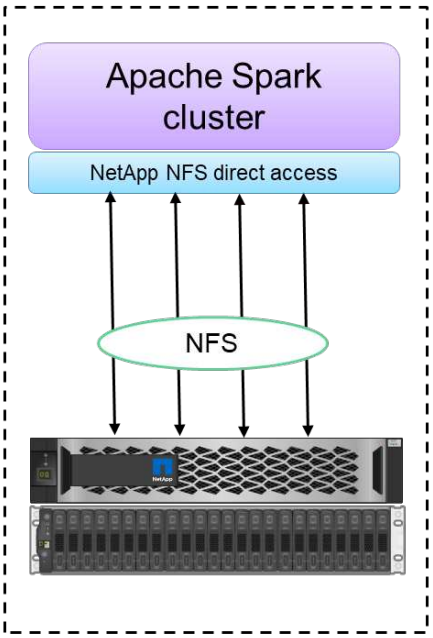
Complicated frameworks and tools

Enterprises AI teams face multiple challenges. Even with expert data science knowledge, tools and frameworks for different deployment ecosystems and applications might not translate simply from one to another. A data science platform should integrate seamlessly with corresponding big data platforms built on Spark with ease of data movement, reusable models, code out of the box, and tools that support best practices for prototyping, validating, versioning, sharing, reusing, and quickly deploying models to production.

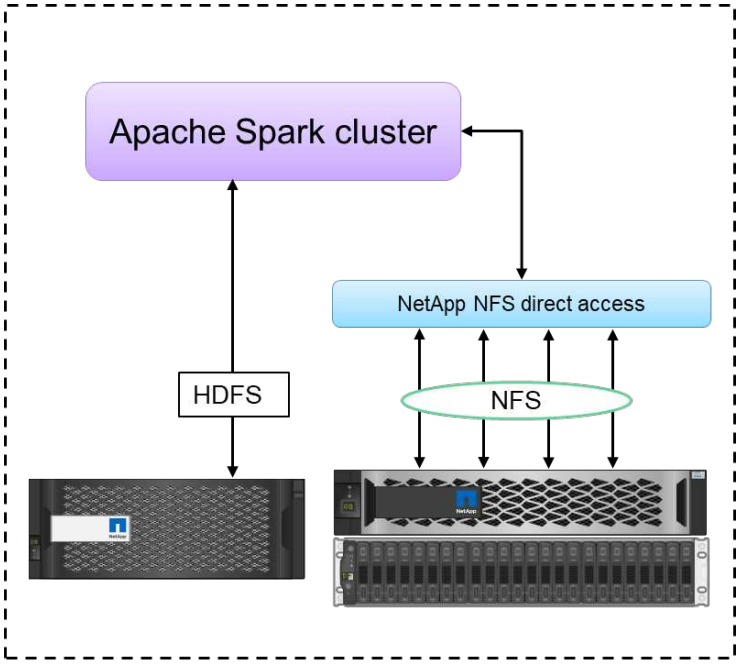
Why choose NetApp?

NetApp can improve your Spark experience in the following ways:

- NetApp NFS direct access (shown in the figure below) allows customers to run big-data-analytics jobs on their existing or new NFSv3 or NFSv4 data without moving or copying the data. It prevents multiple copies of data and eliminates the need to sync the data with a source.
- More efficient storage and less server replication. For example, the NetApp E-Series Hadoop solution requires two rather than three replicas of the data, and the FAS Hadoop solution requires a data source but no replication or copies of data. NetApp storage solutions also produce less server-to-server traffic.
- Better Hadoop job and cluster behavior during drive and node failure.
- Better data-ingest performance.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

For example, in the financial and healthcare sector, the movement of data from one place to another must meet legal obligations, which is not an easy task. In this scenario, NetApp NFS direct access analyzes the financial and healthcare data from its original location. Another key benefit is that using NetApp NFS direct

access simplifies protecting Hadoop data by using native Hadoop commands and enabling data protection workflows with the rich data management portfolio from NetApp.

NetApp NFS direct access provides two kinds of deployment options for Hadoop/Spark clusters:

- By default, Hadoop or Spark clusters use the Hadoop Distributed File System (HDFS) for data storage and the default file system. NetApp NFS direct access can replace the default HDFS with NFS storage as the default file system, enabling direct analytics on NFS data.
- In another deployment option, NetApp NFS direct access supports configuring NFS as additional storage along with HDFS in a single Hadoop or Spark cluster. In this case, the customer can share data through NFS exports and access it from the same cluster along with HDFS data.

The key benefits of using NetApp NFS direct access include the following:

- Analyzing the data from its current location, which prevents the time- and performance-consuming task of moving analytics data to a Hadoop infrastructure such as HDFS.
- Reducing the number of replicas from three to one.
- Enabling users to decouple compute and storage to scale them independently.
- Providing enterprise data protection by leveraging the rich data management capabilities of ONTAP.
- Certification with the Hortonworks data platform.
- Enabling hybrid data analytics deployments.
- Reducing backup time by leveraging dynamic multithread capability.

See TR-4657: NetApp hybrid cloud data solutions - Spark and Hadoop based on customer use cases for backing up Hadoop data, backup and disaster recovery from the cloud to on-premises, enabling DevTest on existing Hadoop data, data protection and multicloud connectivity, and accelerating analytics workloads.

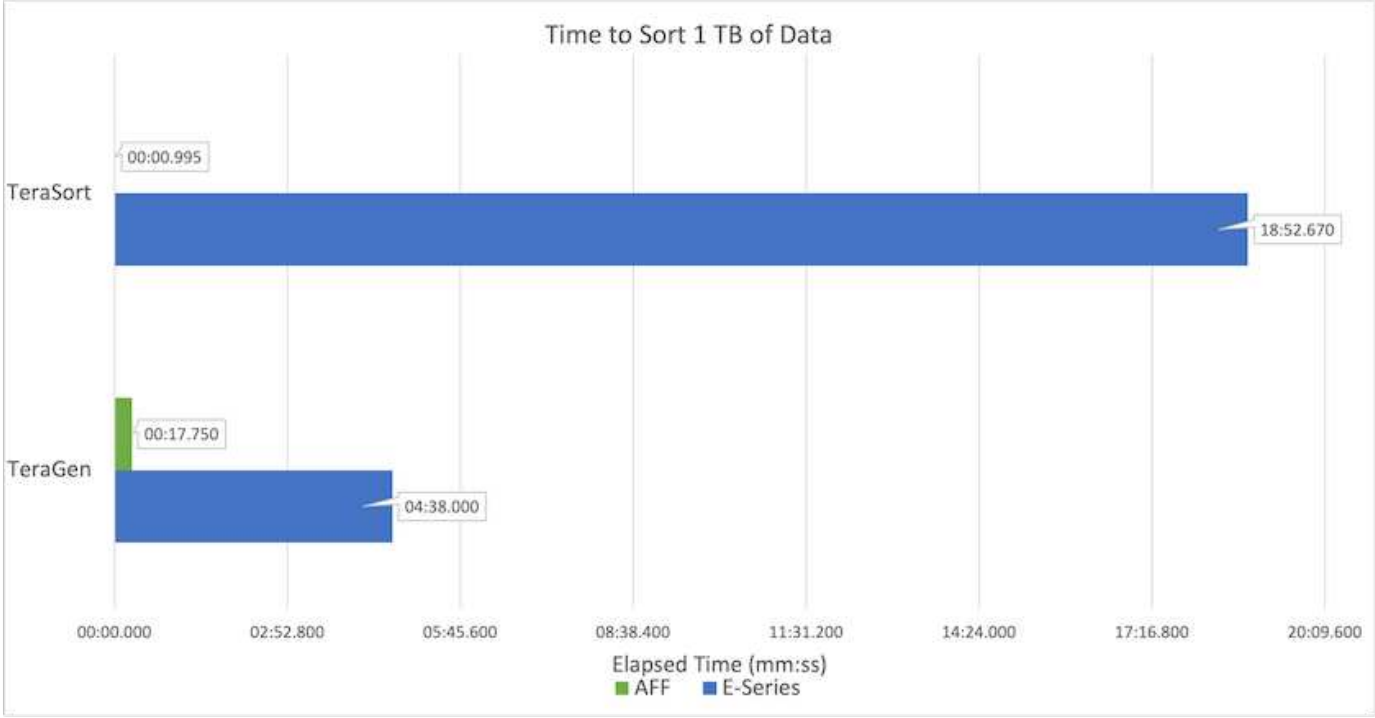
The following sections describe storage capabilities that are important for Spark customers.

Storage tiering

With Hadoop storage tiering, you can store files with different storage types in accordance with a storage policy. Storage types include `hot`, `cold`, `warm`, `all_ssd`, `one_ssd`, and `lazy_persist`.

We performed validation of Hadoop storage tiering on a NetApp AFF storage controller and an E-Series storage controller with SSD and SAS drives with different storage policies. The Spark cluster with AFF-A800 has four compute worker nodes, whereas the cluster with E-Series has eight. This is mainly to compare the performance of solid-state drives (SSDs) versus hard-drive disks (HDDs).

The following figure shows the performance of NetApp solutions for a Hadoop SSD.



- The baseline NL-SAS configuration used eight compute nodes and 96 NL-SAS drives. This configuration generated 1TB of data in 4 minutes and 38 seconds. See TR-3969 NetApp E-Series Solution for Hadoop for details on the cluster and storage configuration.
- Using TeraGen, the SSD configuration generated 1TB of data 15.66x faster than the NL-SAS configuration. Moreover, the SSD configuration used half the number of compute nodes and half the number of disk drives (24 SSd drives in total). Based on the job completion time, it was almost twice as fast as the NL-SAS configuration.
- Using TeraSort, the SSD configuration sorted 1TB of data 1138.36 times more quickly than the NL-SAS configuration. Moreover, the SSD configuration used half the number of compute nodes and half the number of disk drives (24 SSd drives in total). Therefore, per drive, it was approximately three times faster than the NL-SAS configuration.
- The takeaway is transitioning from spinning disks to all-flash improves performance. The number of compute nodes was not the bottleneck. With NetApp's all-flash storage, runtime performance scales well.
- With NFS, the data was functionally equivalent to being pooled all together, which can reduce the number of compute nodes depending on your workload. The Apache Spark cluster users do not have to manually rebalance data when changing number of compute nodes.

Performance scaling - Scale out

When you need more computation power from a Hadoop cluster in an AFF solution, you can add data nodes with an appropriate number of storage controllers. NetApp recommends starting with four data nodes per storage controller array and increasing the number to eight data nodes per storage controller, depending on workload characteristics.

AFF and FAS are perfect for in-place analytics. Based on computation requirements, you can add node managers, and non-disruptive operations allow you to add a storage controller on demand without downtime. We offer rich features with AFF and FAS, such as NVME media support, guaranteed efficiency, data reduction, QOS, predictive analytics, cloud tiering, replication, cloud deployment, and security. To help customers meet their requirements, NetApp offers features such as file system analytics, quotas, and on-box load balancing with no additional license costs. NetApp has better performance in the number of concurrent jobs, lower latency, simpler operations, and higher gigabytes per second throughput than our competitors. Furthermore,

NetApp Cloud Volumes ONTAP runs on all three major cloud providers.

Performance scaling - Scale up

Scale-up features allow you to add disk drives to AFF, FAS, and E-Series systems when you need additional storage capacity. With Cloud Volumes ONTAP, scaling storage to the PB level is a combination of two factors: tiering infrequently used data to object storage from block storage and stacking Cloud Volumes ONTAP licenses without additional compute.

Multiple protocols

NetApp systems support most protocols for Hadoop deployments, including SAS, iSCSI, FCP, InfiniBand, and NFS.

Operational and supported solutions

The Hadoop solutions described in this document are supported by NetApp. These solutions are also certified with major Hadoop distributors. For information, see the Hortonworks site, and the Cloudera certification and partner sites.

Target audience

The world of analytics and data science touches multiple disciplines in IT and business:

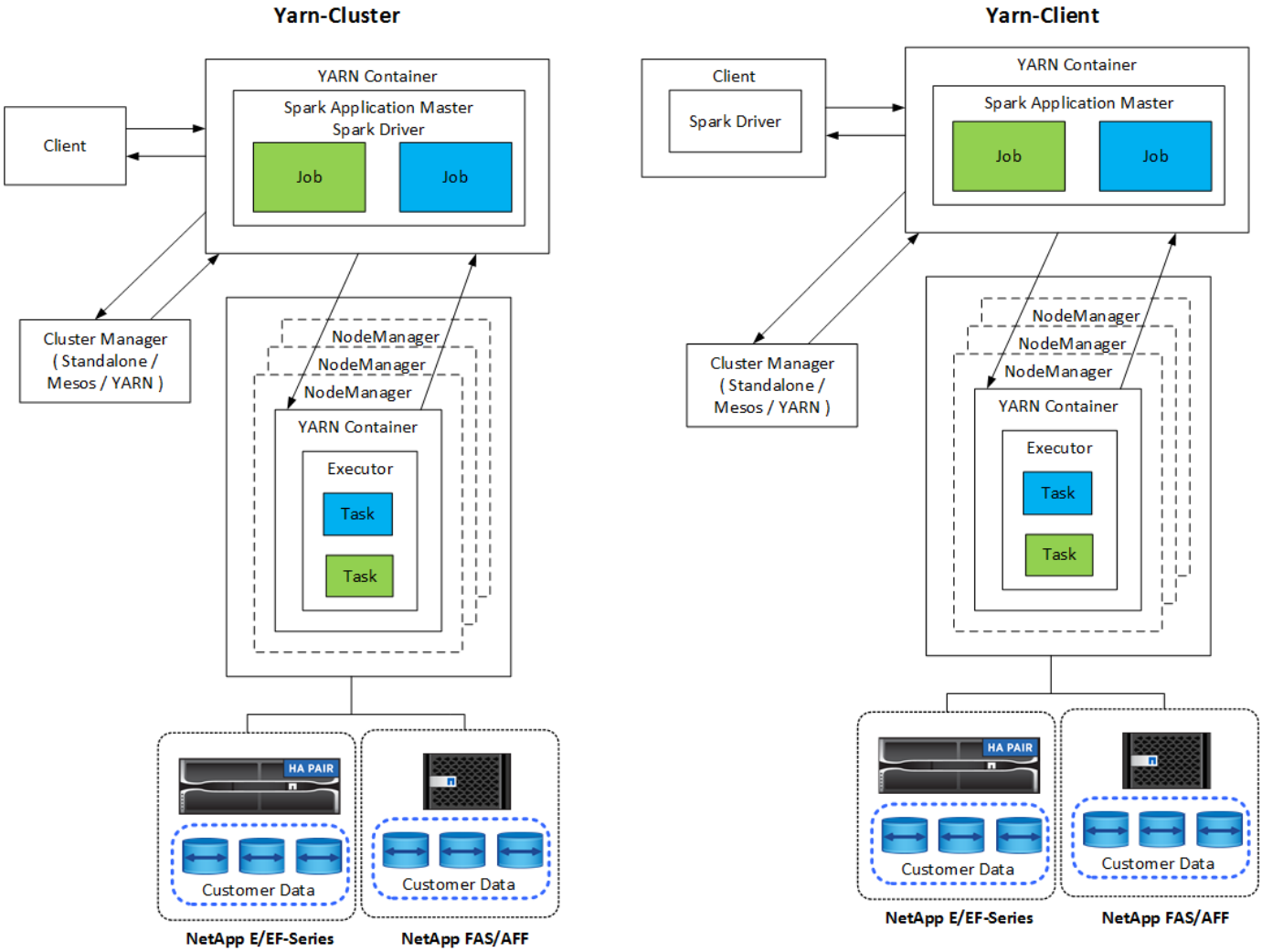
- The data scientist needs the flexibility to use their tools and libraries of choice.
- The data engineer needs to know how the data flows and where it resides.
- A DevOps engineer needs the tools to integrate new AI and ML applications into their CI and CD pipelines.
- Cloud administrators and architects must be able to set up and manage hybrid cloud resources.
- Business users want to have access to analytics, AI, ML, and DL applications.

In this technical report, we describe how NetApp AFF, E-Series, StorageGRID, NFS direct access, Apache Spark, Horovod, and Keras help each of these roles bring value to business.

Solution technology

Apache Spark is a popular programming framework for writing Hadoop applications that works directly with the Hadoop Distributed File System (HDFS). Spark is production ready, supports processing of streaming data, and is faster than MapReduce. Spark has configurable in-memory data caching for efficient iteration, and the Spark shell is interactive for learning and exploring data. With Spark, you can create applications in Python, Scala, or Java. Spark applications consist of one or more jobs that have one or more tasks.

Every Spark application has a Spark driver. In YARN-Client mode, the driver runs on the client locally. In YARN-Cluster mode, the driver runs in the cluster on the application master. In the cluster mode, the application continues to run even if the client disconnects.



There are three cluster managers:

- **Standalone.** This manager is a part of Spark, which makes it easy to set up a cluster.
- **Apache Mesos.** This is a general cluster manager that also runs MapReduce and other applications.
- **Hadoop YARN.** This is a resource manager in Hadoop 3.

The resilient distributed dataset (RDD) is the primary component of Spark. RDD recreates the lost and missing data from data stored in memory in the cluster and stores the initial data that comes from a file or is created programmatically. RDDs are created from files, data in memory, or another RDD. Spark programming performs two operations: transformation and actions. Transformation creates a new RDD based on an existing one. Actions return a value from an RDD.

Transformations and actions also apply to Spark Datasets and DataFrames. A dataset is a distributed collection of data that provides the benefits of RDDs (strong typing, use of lambda functions) with the benefits of Spark SQL's optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, and so on.). A DataFrame is a dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python. DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive/HBase, external databases on-premises or in the cloud, or existing RDDs.

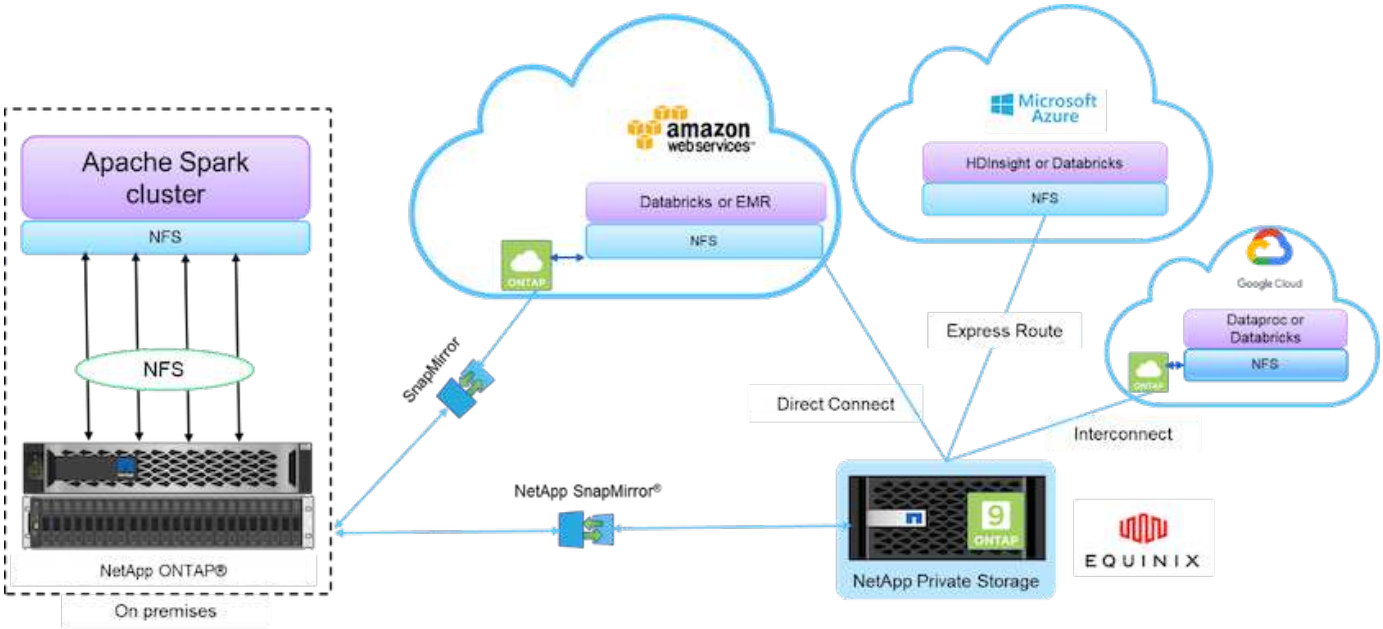
Spark applications include one or more Spark jobs. Jobs run tasks in executors, and executors run in YARN containers. Each executor runs in a single container, and executors exist throughout the life of an application.

An executor is fixed after the application starts, and YARN does not resize the already allocated container. An executor can run tasks concurrently on in-memory data.

NetApp Spark solutions overview

NetApp has three storage portfolios: FAS/AFF, E-Series, and Cloud Volumes ONTAP. We have validated AFF and the E-Series with ONTAP storage system for Hadoop solutions with Apache Spark.

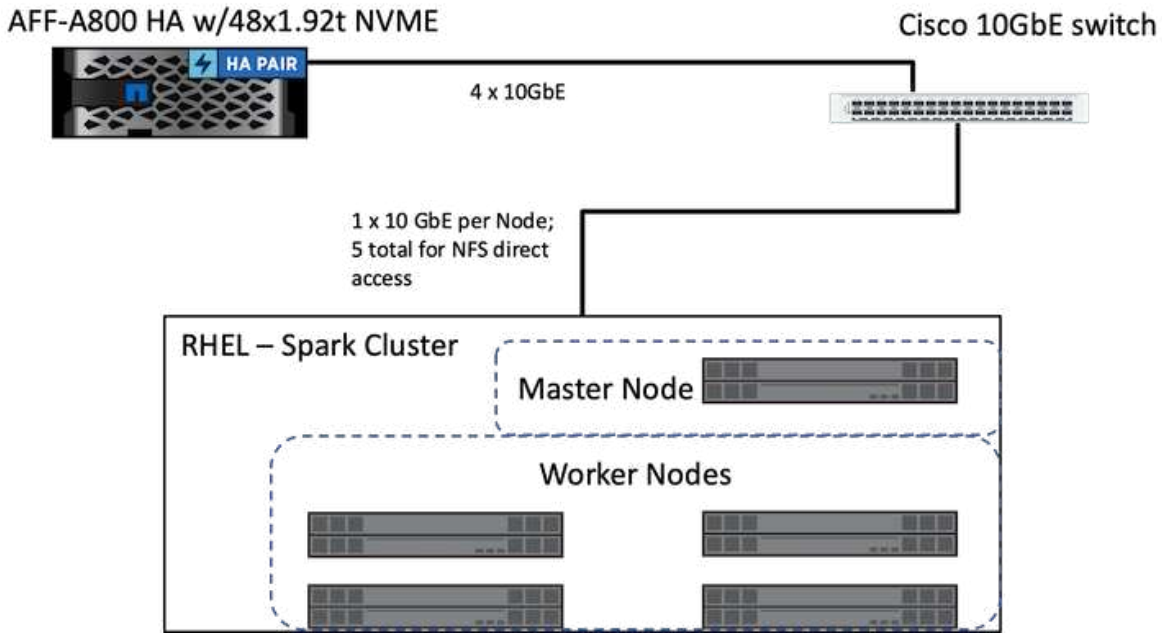
The data fabric powered by NetApp integrates data management services and applications (building blocks) for data access, control, protection, and security, as shown in the figure below.



The building blocks in the figure above include:

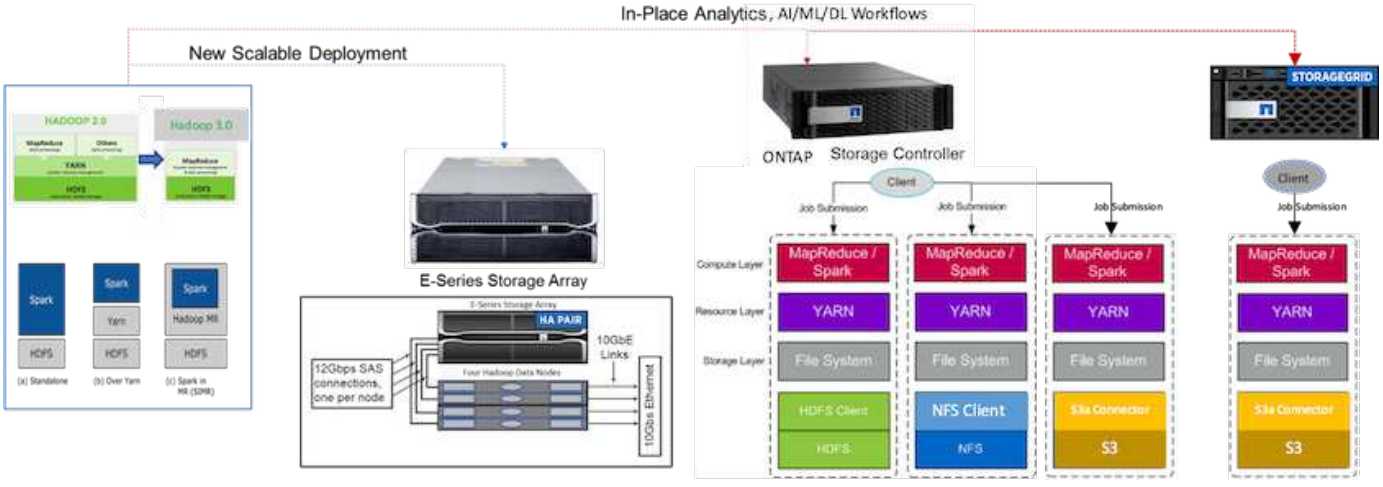
- **NetApp NFS direct access.** Provides the latest Hadoop and Spark clusters with direct access to NetApp NFS volumes without additional software or driver requirements.
- **NetApp Cloud Volumes ONTAP and Google Cloud NetApp Volumes.** Software-defined connected storage based on ONTAP running in Amazon Web Services (AWS) or Azure NetApp Files (ANF) in Microsoft Azure cloud services.
- **NetApp SnapMirror technology.** Provides data protection capabilities between on-premises and ONTAP Cloud or NPS instances.
- **Cloud service providers.** These providers include AWS, Microsoft Azure, Google Cloud, and IBM Cloud.
- **PaaS.** Cloud-based analytics services such as Amazon Elastic MapReduce (EMR) and Databricks in AWS as well as Microsoft Azure HDInsight and Azure Databricks.

The following figure depicts the Spark solution with NetApp storage.

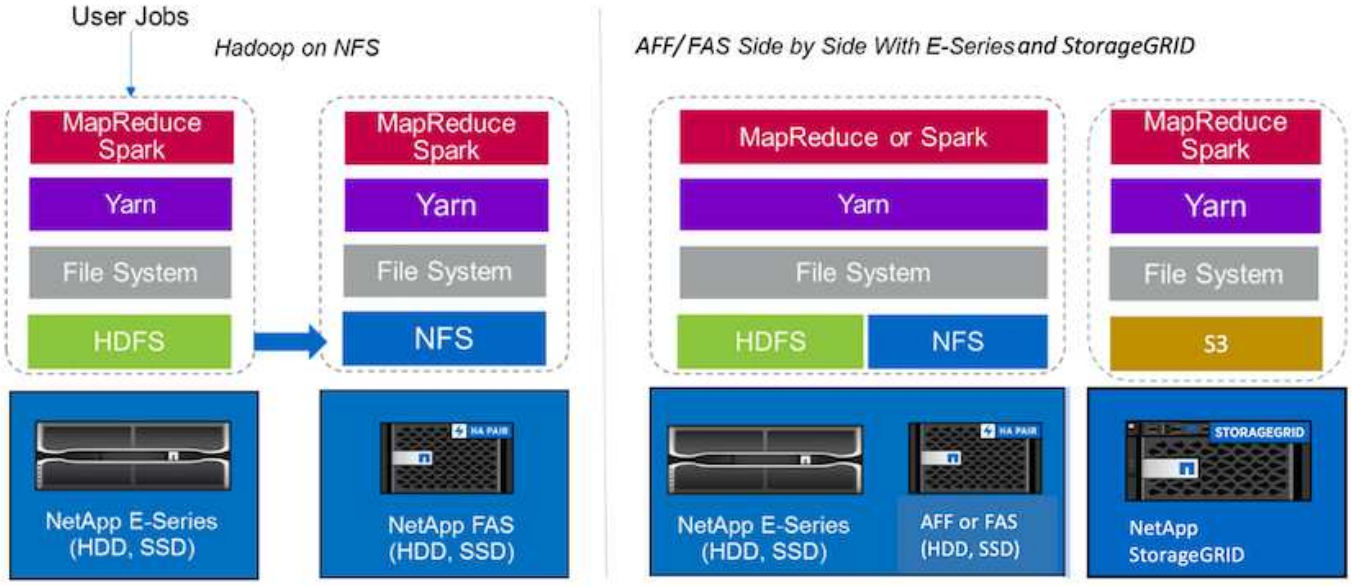


The ONTAP Spark solution uses the NetApp NFS direct access protocol for in-place analytics and AI, ML, and DL workflows using access to existing production data. Production data available to Hadoop nodes is exported to perform in-place analytical and AI, ML, and DL jobs. You can access data to process in Hadoop nodes either with NetApp NFS direct access or without it. In Spark with the standalone or `yarn` cluster manager, you can configure an NFS volume by using `file://<target_volume>`. We validated three use cases with different datasets. The details of these validations are presented in the section "Testing Results." (xref)

The following figure depicts NetApp Apache Spark/Hadoop storage positioning.



We identified the unique features of the E-Series Spark solution, the AFF/FAS ONTAP Spark solution, and the StorageGRID Spark solution, and performed detailed validation and testing. Based upon our observations, NetApp recommends the E-Series solution for greenfield installations and new scalable deployments and the AFF/FAS solution for in-place analytics, AI, ML, and DL workloads using existing NFS data, and StorageGRID for AI, ML, and DL and modern data analytics when object storage is required.



A data lake is a storage repository for large datasets in native form that can be used for analytics, AI, ML, and DL jobs. We built a data lake repository for the E-Series, AFF/FAS, and StorageGRID SG6060 Spark solutions. The E-Series system provides HDFS access to the Hadoop Spark cluster, whereas existing production data is accessed through the NFS direct access protocol to the Hadoop cluster. For datasets that reside in object storage, NetApp StorageGRID provides S3 and S3a secure access.

Use case summary

This page describes the different areas in which this solution can be used.

Streaming data

Apache Spark can process streaming data, which is used for streaming extract, transform, and load (ETL) processes; data enrichment; triggering event detection; and complex session analysis:

- **Streaming ETL.** Data is continually cleaned and aggregated before it is pushed into datastores. Netflix uses Kafka and Spark streaming to build a real-time online movie recommendation and data monitoring solution that can process billions of events per day from different data sources. Traditional ETL for batch processing is treated differently, however. This data is read first, and then it is converted into a database format before being written to the database.
- **Data enrichment.** Spark streaming enriches the live data with static data to enable more real-time data analysis. For example, online advertisers can deliver personalized, targeted ads directed by information about customer behavior.
- **Trigger event detection.** Spark streaming allows you to detect and respond quickly to unusual behavior that could indicate potentially serious problems. For example, financial institutions use triggers to detect and stop fraud transactions, and hospitals use triggers to detect dangerous health changes detected in a patient's vital signs.
- **Complex session analysis.** Spark streaming collects events such as user activity after logging in to a website or application, which are then grouped and analyzed. For example, Netflix uses this functionality to provide real-time movie recommendations.

For more streaming data configuration, Confluent Kafka verification, and performance tests, see TR-4912: Best practice guidelines for Confluent Kafka tiered storage with NetApp.

Machine learning

The Spark integrated framework helps you run repeated queries on datasets using the machine learning library (MLlib). MLlib is used in areas such as clustering, classification, and dimensionality reduction for some common big data functions such as predictive intelligence, customer segmentation for marketing purposes, and sentiment analysis. MLlib is used in network security to conduct real-time inspections of data packets for indications of malicious activity. It helps security providers learn about new threats and stay ahead of hackers while protecting their clients in real time.

Deep learning

TensorFlow is a popular deep learning framework used across the industry. TensorFlow supports the distributed training on a CPU or GPU cluster. This distributed training allows users to run it on a large amount of data with lot of deep layers.

Until fair recently, if we wanted to use TensorFlow with Apache Spark, we needed to perform all necessary ETL for TensorFlow in PySpark and then write data to intermediate storage. That data would then be loaded onto the TensorFlow cluster for the actual training process. This workflow required the user to maintain two different clusters, one for ETL and one for distributed training of TensorFlow. Running and maintaining multiple clusters was typically tedious and time consuming.

DataFrames and RDD in earlier Spark versions were not well-suited for deep learning because random access was limited. In Spark 3.0 with project hydrogen, native support for the deep learning frameworks is added. This approach allows non-MapReduce-based scheduling on the Spark cluster.

Interactive analysis

Apache Spark is fast enough to perform exploratory queries without sampling with development languages other than Spark, including SQL, R, and Python. Spark uses visualization tools to process complex data and visualize it interactively. Spark with structured streaming performs interactive queries against live data in web analytics that enable you to run interactive queries against a web visitor’s current session.

Recommender system

Over the years, recommender systems have brought tremendous changes to our lives, as businesses and consumers have responded to dramatic changes in online shopping, online entertainment, and many other industries. Indeed, these systems are among the most evident success stories of AI in production. In many practical use cases, recommender systems are combined with conversational AI or chatbots interfaced with an NLP backend to obtain relevant information and produce useful inferences.

Today, many retailers are adopting newer business models like buying online and picking up in store, curbside pickup, self-checkout, scan-and-go, and more. These models have become prominent during the COVID-19 pandemic by making shopping safer and more convenient for consumers. AI is crucial for these growing digital trends, which are influenced by consumer behavior and vice versa. To meet the growing demands of consumers, to augment the customer experience, to improve operational efficiency, and to grow revenue, NetApp helps its enterprise customers and businesses use machine- learning and deep- learning algorithms to design faster and more accurate recommender systems.

There are several popular techniques used for providing recommendations, including collaborative filtering, content-based systems, the deep learning recommender model (DLRM), and hybrid techniques. Customers previously utilized PySpark to implement collaborative filtering for creating recommendation systems. Spark MLlib implements alternating least squares (ALS) for collaborative filtering, a very popular algorithm among enterprises before the rise of DLRM.

Natural language processing

Conversational AI, made possible by natural language processing (NLP), is the branch of AI helping computers communicate with humans. NLP is prevalent in every industry vertical and many use cases, from smart assistants and chatbots to Google search and predictive text. According to a Gartner prediction, by 2022, 70% of people will be interacting with conversational AI platforms on a daily basis. For a high-quality conversation between a human and a machine, responses must be rapid, intelligent, and natural sounding.

Customers need a large amount of data to process and train their NLP and automatic speech recognition (ASR) models. They also need to move data across the edge, core, and cloud, and they need the power to perform inference in milliseconds to establish natural communication with humans. NetApp AI and Apache Spark is an ideal combination for compute, storage, data processing, model training, fine-tuning, and deployment.

Sentiment analysis is a field of study within NLP in which positive, negative, or neutral sentiments are extracted from text. Sentiment analysis has a variety of use cases, from determining support center employee performance in conversations with callers to providing appropriate automated chatbot responses. It has also been used to predict a firm's stock price based on the interactions between firm representatives and the audience at quarterly earnings calls. Furthermore, sentiment analysis can be used to determine a customer's view on the products, services, or support provided by the brand.

We used the Spark NLP library from John Snow Labs to load pretrained pipelines and Bidirectional Encoder Representations from Transformers (BERT) models including financial news sentiment and FinBERT, performing tokenization, named entity recognition, model training, fitting and sentiment analysis at scale. Spark NLP is the only open-source NLP library in production that offers state-of-the-art transformers such as BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM- RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT, and GPT2. The library works not only in Python and R, but also in the JVM ecosystem (Java, Scala, and Kotlin) at scale by extending Apache Spark natively.

Major AI, ML, and DL use cases and architectures

Major AI, ML, and DL use cases and methodology can be divided into the following sections:

Spark NLP pipelines and TensorFlow distributed inferencing

The following list contains the most popular open-source NLP libraries that have been adopted by the data science community under different levels of development:

- Natural Language Toolkit (NLTK). The complete toolkit for all NLP techniques. It has been maintained since the early 2000s.
- TextBlob. An easy-to-use NLP tools Python API built on top of NLTK and Pattern.
- Stanford Core NLP. NLP services and packages in Java developed by the Stanford NLP Group.
- Gensim. Topic Modelling for Humans started off as a collection of Python scripts for the Czech Digital Mathematics Library project.
- SpaCy. End-to-end industrial NLP workflows with Python and Cython with GPU acceleration for transformers.
- Fasttext. A free, lightweight, open-source NLP library for the learning-of-word embeddings and sentence classification created by Facebook's AI Research (FAIR) lab.

Spark NLP is a single, unified solution for all NLP tasks and requirements that enables scalable, high-

performance, and high-accuracy NLP-powered software for real production use cases. It leverages transfer learning and implements the latest state-of-the-art algorithms and models in research and across industries. Due to the lack of full support by Spark for the above libraries, Spark NLP was built on top of Spark ML to take advantage of Spark's general-purpose in-memory distributed data processing engine as an enterprise-grade NLP library for mission-critical production workflows. Its annotators utilize rule-based algorithms, machine learning, and TensorFlow to power deep learning implementations. This covers common NLP tasks including but not limited to tokenization, lemmatization, stemming, part-of-speech tagging, named-entity recognition, spell checking, and sentiment analysis.

Bidirectional Encoder Representations from Transformers (BERT) is a transformer-based machine learning technique for NLP. It popularized the concept of pretraining and fine tuning. The transformer architecture in BERT originated from machine translation, which models long-term dependencies better than Recurrent Neural Network (RNN)-based language models. It also introduced the Masked Language Modelling (MLM) task, where a random 15% of all tokens are masked and the model predicts them, enabling true bidirectionality.

Financial sentiment analysis is challenging due to the specialized language and lack of labeled data in that domain. FinBERT, a language model based on pretrained BERT, was domain adapted on Reuters TRC2, a financial corpus, and fine-tuned with labeled data (Financial PhraseBank) for financial sentiment classification. Researchers extracted 4, 500 sentences from news articles with financial terms. Then 16 experts and masters students with finance backgrounds labeled the sentences as positive, neutral, and negative. We built an end-to-end Spark workflow to analyze sentiment for Top-10 NASDAQ company earnings call transcripts from 2016 to 2020 using FinBERT and two other pre-trained pipelines, Explain Document DL) from Spark NLP.

The underlying deep learning engine for Spark NLP is TensorFlow, an end-to-end, open-source platform for machine learning that enables easy model building, robust ML production anywhere, and powerful experimentation for research. Therefore, when executing our pipelines in Spark `yarn cluster` mode, we were essentially running distributed TensorFlow with data and model parallelization across one master and multiple worker nodes, as well as network- attached storage mounted on the cluster.

Horovod distributed training

The core Hadoop validation for MapReduce-related performance is performed with TeraGen, TeraSort, TeraValidate, and DFSIO (read and write). The TeraGen and TeraSort validation results are presented in NetApp E-Series Solution for Hadoop and in the section "Storage Tiering" for AFF.

Based upon customer requests, we consider distributed training with Spark to be one of the most important of the various use cases. In this document, we used the Hovorod on Spark to validate Spark performance with NetApp on-premises, cloud-native, and hybrid cloud solutions using NetApp All Flash FAS (AFF) storage controllers, Azure NetApp Files, and StorageGRID.

The Horovod on Spark package provides a convenient wrapper around Horovod that makes running distributed training workloads in Spark clusters simple, enabling a tight model design loop in which data processing, model training, and model evaluation are all done in Spark where training and inferencing data resides.

There are two APIs for running Horovod on Spark: a high-level Estimator API and a lower-level Run API. Although both use the same underlying mechanism to launch Horovod on Spark executors, the Estimator API abstracts the data processing, model training loop, model checkpointing, metrics collection, and distributed training. We used Horovod Spark Estimators, TensorFlow, and Keras for an end-to-end data preparation and distributed training workflow based on the Kaggle Rossmann Store Sales competition.

The script `keras_spark_horovod_rossmann_estimator.py` can be found in the section "Python scripts for each major use case." It contains three parts:

- The first part performs various data preprocessing steps over an initial set of CSV files provided by Kaggle and gathered by the community. The input data is separated into a training set with a `Validation` subset, and a testing dataset.
- The second part defines a Keras Deep Neural Network (DNN) model with logarithmic sigmoid activation function and an Adam optimizer, and it performs distributed training of the model using Horovod on Spark.
- The third part performs prediction on the testing dataset using the best model that minimizes the validation set overall mean absolute error. It then creates an output CSV file.

See the section "Machine Learning" for various runtime comparison results.

Multi-worker deep learning using Keras for CTR prediction

With the recent advances in ML platforms and applications, a lot of attention is now on learning at scale. The click-through rate (CTR) is defined as the average number of click-throughs per hundred online ad impressions (expressed as a percentage). It is widely adopted as a key metric in various industry verticals and use cases, including digital marketing, retail, e-commerce, and service providers. For more detail on the applications of CTR and distributed training performance results, see the Deep learning models for CTR prediction performance section.

In this technical report we used a variation of the Criteo Terabyte Click Logs dataset (see TR-4904) for multi-worker distributed deep learning using Keras to build a Spark workflow with Deep and Cross Network (DCN) models, comparing its performance in terms of log loss error function with a baseline Spark ML Logistic Regression model. DCN efficiently captures effective feature interactions of bounded degrees, learns highly nonlinear interactions, requires no manual feature engineering or exhaustive searching, and has low computational cost.

Data for web-scale recommender systems is mostly discrete and categorical, leading to a large and sparse feature space that is challenging for feature exploration. This has limited most large-scale systems to linear models such as logistic regression. However, identifying frequently predictive features and at the same time exploring unseen or rare cross features is the key to making good predictions. Linear models are simple, interpretable, and easy to scale, but they are limited in their expressive power.

Cross features, on the other hand, have been shown to be significant in improving the models' expressiveness. Unfortunately, it often requires manual feature engineering or exhaustive search to identify such features. Generalizing to unseen feature interactions is often difficult. Using a cross neural network like DCN avoids task-specific feature engineering by explicitly applying feature crossing in an automatic fashion. The cross network consists of multiple layers, where the highest degree of interactions is provably determined by layer depth. Each layer produces higher-order interactions based on existing ones and keeps the interactions from previous layers.

A deep neural network (DNN) has the promise to capture very complex interactions across features. However, compared to DCN, it requires nearly an order of magnitude more parameters, is unable to form cross features explicitly, and may fail to efficiently learn some types of feature interactions. The cross network is memory efficient and easy to implement. Jointly training the cross and DNN components together efficiently captures predictive feature interactions and delivers state-of-the-art performance on the Criteo CTR dataset.

A DCN model starts with an embedding and stacking layer, followed by a cross network and a deep network in parallel. These in turn are followed by a final combination layer which combines the outputs from the two networks. Your input data can be a vector with sparse and dense features. In Spark, the libraries contain the type `SparseVector`. It is therefore important for users to distinguish between the two and be mindful when calling their respective functions and methods. In web-scale recommender systems such as CTR prediction, the inputs are mostly categorical features, for example `'country=usa'`. Such features are often encoded as one-hot vectors, for example, `'[0,1,0, …]'`. One-hot-encoding (OHE) with `SparseVector` is useful when

dealing with real-world datasets with ever-changing and growing vocabularies. We modified examples in DeepCTR to process large vocabularies, creating embedding vectors in the embedding and stacking layer of our DCN.

The Criteo Display Ads dataset predicts the ads click-through rate. It has 13 integer features and 26 categorical features in which each category has a high cardinality. For this dataset, an improvement of 0.001 in logloss is practically significant due to the large input size. A small improvement in prediction accuracy for a large user base can potentially lead to a large increase in a company's revenue. The dataset contains 11GB of user logs from a period of 7 days, which equates to around 41 million records. We used Spark `dataFrame.randomSplit()function` to randomly split the data for training (80%), cross-validation (10%), and the remaining 10% for testing.

DCN was implemented on TensorFlow with Keras. There are four main components in implementing the model training process with DCN:

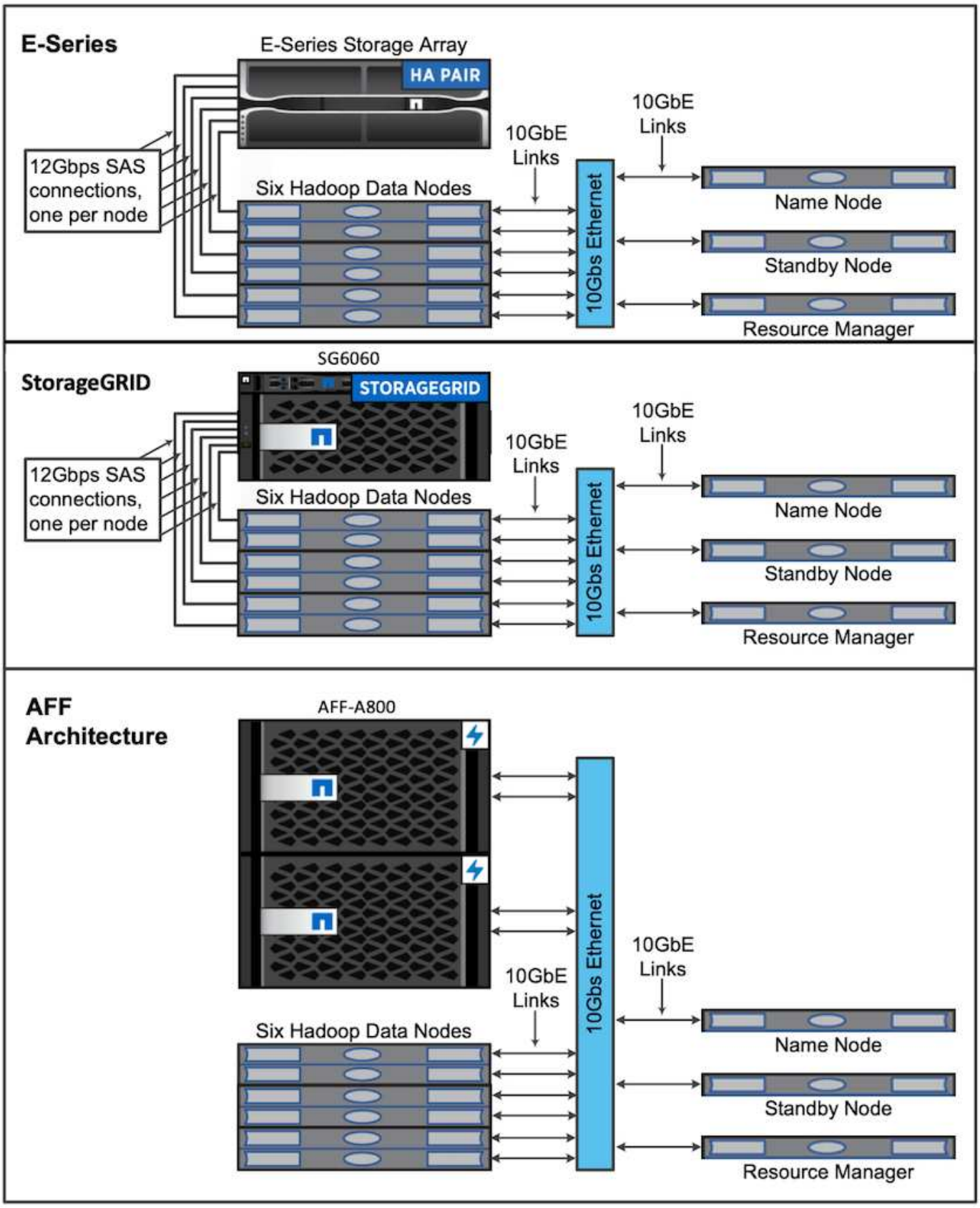
- **Data processing and embedding.** Real-valued features are normalized by applying a log transform. For categorical features, we embed the features in dense vectors of dimension 6×(category cardinality)1/4. Concatenating all embeddings results in a vector of dimension 1026.
- **Optimization.** We applied mini-batch stochastic optimization with the Adam optimizer. The batch size was set to 512. Batch normalization was applied to the deep network and the gradient clip norm was set at 100.
- **Regularization.** We used early stopping, as L2 regularization or dropout was not found to be effective.
- **Hyperparameters.** We report results based on a grid search over the number of hidden layers, the hidden layer size, the initial learning rate, and the number of cross layers. The number of hidden layers ranged from 2 to 5, with hidden layer sizes ranging from 32 to 1024. For DCN, the number of cross layers was from 1 to 6. The initial learning rate was tuned from 0.0001 to 0.001 with increments of 0.0001. All experiments applied early stopping at training step 150,000, beyond which overfitting started to occur.

In addition to DCN, we also tested other popular deep-learning models for CTR prediction, including DeepFM, AutoInt, and DCN v2.

Architectures used for validation

For this validation, we used four worker nodes and one master nodes with an AFF-A800 HA pair. All cluster members were connected through 10GbE network switches.

For this NetApp Spark solution validation, we used three different storage controllers: the E5760, the E5724, and the AFF-A800. The E-Series storage controllers were connected to five data nodes with 12Gbps SAS connections. The AFF HA-pair storage controller provides exported NFS volumes through 10GbE connections to Hadoop worker nodes. The Hadoop cluster members were connected through 10GbE connections in the E-Series, AFF, and StorageGRID Hadoop solutions.



Testing results

We used the TeraSort and TeraValidate scripts in the TeraGen benchmarking tool to

measure the Spark performance validation with E5760, E5724, and AFF-A800 configurations. In addition, three major use cases were tested: Spark NLP pipelines and TensorFlow distributed training, Horovod distributed training, and multi-worker deep learning using Keras for CTR Prediction with DeepFM.

For both E-Series and StorageGRID validation, we used Hadoop replication factor 2. For AFF validation, we only used one source of data.

The following table lists the hardware configuration for the Spark performance validation.

Type	Hadoop worker nodes	Drive type	Drives per node	Storage controller
SG6060	4	SAS	12	Single high-availability (HA) pair
E5760	4	SAS	60	Single HA pair
E5724	4	SAS	24	Single HA pair
AFF800	4	SSD	6	Single HA pair

The following table lists software requirements.

Software	Version
RHEL	7.9
OpenJDK Runtime Environment	1.8.0
OpenJDK 64-Bit Server VM	25.302
Git	2.24.1
GCC/G++	11.2.1
Spark	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
Keras	2.9.0
Horovod	0.24.3

Financial sentiment analysis

We published TR-4910: Sentiment Analysis from Customer Communications with NetApp AI, in which an end-to-end conversational AI pipeline was built using the NetApp DataOps Toolkit, AFF storage, and NVIDIA DGX System. The pipeline performs batch audio signal processing, automatic speech recognition (ASR), transfer learning, and sentiment analysis leveraging the DataOps Toolkit, NVIDIA Riva SDK, and the Tao framework. Expanding the sentiment analysis use case to the financial services industry, we built a SparkNLP workflow, loaded three BERT models for various NLP tasks, such as named entity recognition, and obtained sentence-level sentiment for NASDAQ Top 10 companies' quarterly earnings calls.

The following script `sentiment_analysis_spark. py` uses the FinBERT model to process transcripts in

HDFS and produce positive, neutral, and negative sentiment counts, as shown in the following table:

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
hdfs:///data1/Transcripts/
> ./sentiment_analysis_hdfs.log 2>&1
real13m14.300s
user557m11.319s
sys4m47.676s
```

The following table lists the earnings-call, sentence-level sentiment analysis for NASDAQ Top 10 companies from 2016 to 2020.

Sentiment counts and percentage	All 10 Companies	AAPL	AMD	AMZN	CSCO	GOOGL	INTC	MSFT	NVDA
Positive counts	7447	1567	743	290	682	826	824	904	417
Neutral counts	64067	6856	7596	5086	6650	5914	6099	5715	6189
Negative counts	1787	253	213	84	189	97	282	202	89
Uncategorized counts	196	0	0	76	0	0	0	1	0
(total counts)	73497	8676	8552	5536	7521	6837	7205	6822	6695

In terms of percentages, most sentences spoken by the CEOs and CFOs are factual and therefore carry neutral sentiment. During an earnings call, analysts ask questions which might convey positive or negative sentiment. It is worth further investigating quantitatively how negative or positive sentiment affect stock prices on the same or next day of trading.

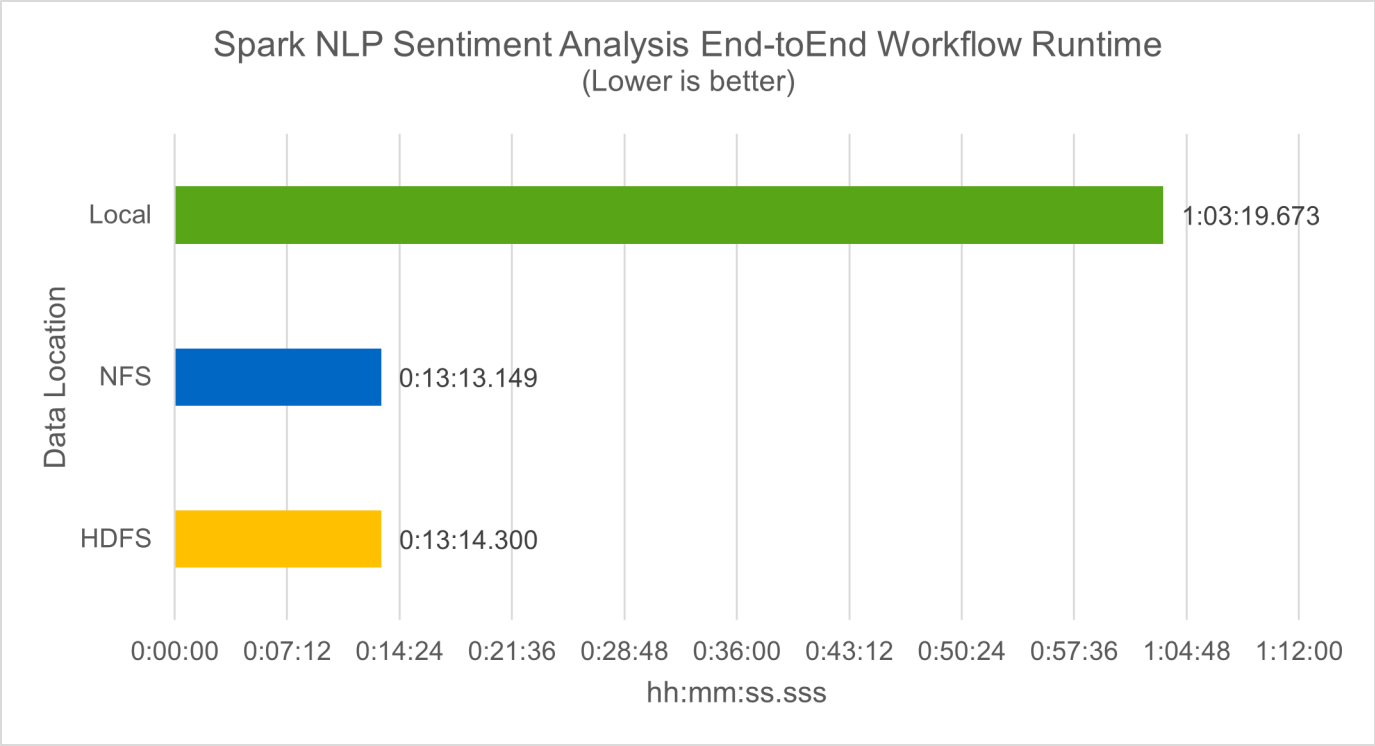
The following table lists the sentence-level sentiment analysis for NASDAQ Top 10 companies, expressed in percentage.

Sentiment percentage	All 10 Companies	AAPL	AMD	AMZN	CSCO	GOOGL	INTC	MSFT	NVDA
Positive	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
Neutral	87.17%	79.02%	88.82%	91.87%	88.42%	86.50%	84.65%	83.77%	92.44%
Negative	2.43%	2.92%	2.49%	1.52%	2.51%	1.42%	3.91%	2.96%	1.33%
Uncategorized	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

In terms of the workflow runtime, we saw a significant 4.78x improvement from `local` mode to a distributed environment in HDFS, and a further 0.14% improvement by leveraging NFS.

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
file:///sparkdemo/sparknlp/Transcripts/
> ./sentiment_analysis_nfs.log 2>&1
real13m13.149s
user537m50.148s
sys4m46.173s
```

As the following figure shows, data and model parallelism improved the data processing and distributed TensorFlow model inferencing speed. Data location in NFS yielded a slightly better runtime because the workflow bottleneck is the downloading of pretrained models. If we increase the transcripts dataset size, the advantage of NFS is more obvious.



Distributed training with Horovod performance

The following command produced runtime information and a log file in our Spark cluster using a single `master` node with 160 executors each with one core. The executor memory was limited to 5GB to avoid out-of-memory error. See the section "Python scripts for each major use case" for more detail regarding the data processing, model training, and model accuracy calculation in `keras_spark_horovod_rossmann_estimator.py`.

```
(base) [root@n138 horovod]# time spark-submit
--master local
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkusecase/horovod
--local-submission-csv /tmp/submission_0.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_local. log 2>&1
```

The resulting runtime with ten training epochs was as follows:

```
real43m34.608s
user12m22.057s
sys2m30.127s
```

It took more than 43 minutes to process input data, train a DNN model, calculate accuracy, and produce

TensorFlow checkpoints and a CSV file for prediction results. We limited the number of training epochs to 10, which in practice is often set to 100 to ensure satisfactory model accuracy. The training time typically scales linearly with the number of epochs.

We next used the four worker nodes available in the cluster and executed the same script in `yarn` mode with data in HDFS:

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir hdfs:///user/hdfs/tr-4570/experiments/horovod
--local-submission-csv /tmp/submission_1.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_yarn.log 2>&1
```

The resulting runtime was improved as follows:

```
real8m13.728s
user7m48.421s
sys1m26.063s
```

With Horovod's model and data parallelism in Spark, we saw a 5.29x runtime speedup of `yarn` versus `local` mode with ten training epochs. This is shown in the following figure with the legends `HDFS` and `Local`. The underlying TensorFlow DNN model training can be further accelerated with GPUs if available. We plan to conduct this testing and publish results in a future technical report.

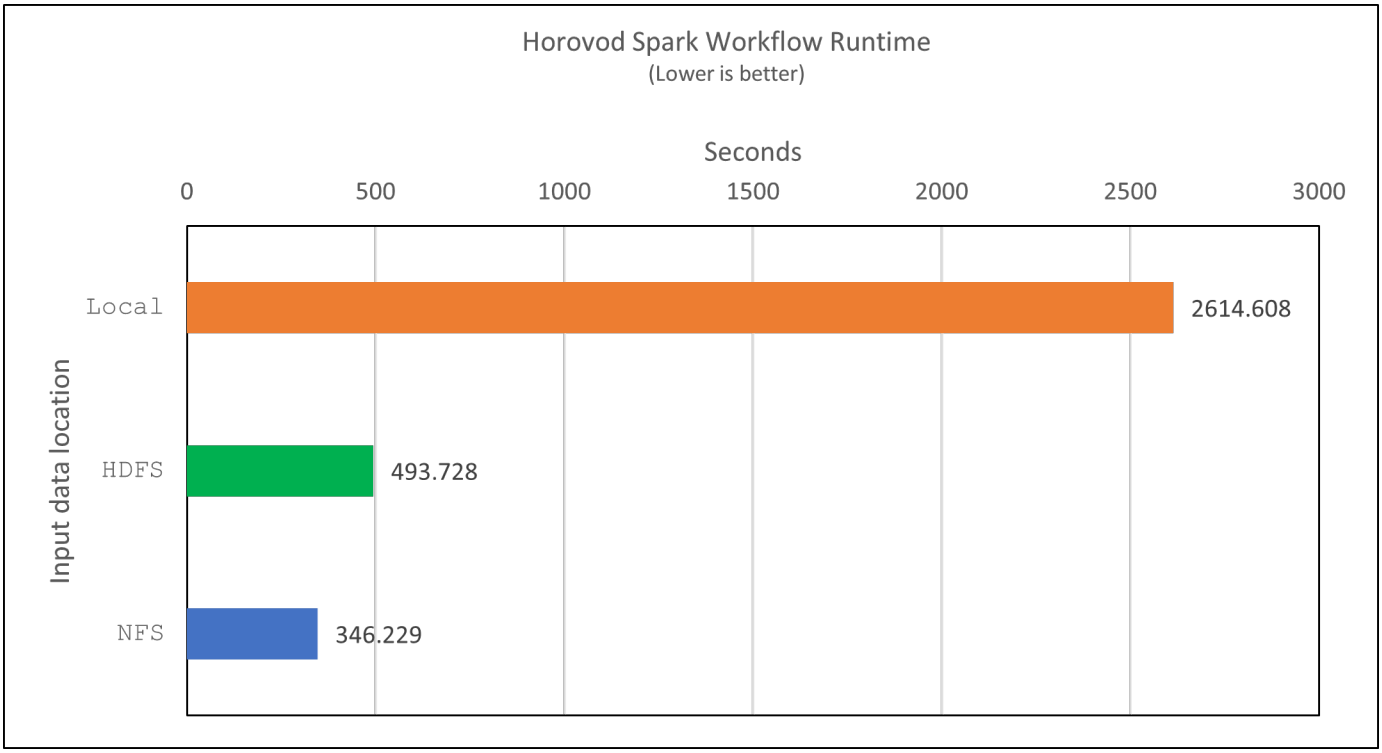
Our next test compared the runtimes with input data residing in NFS versus HDFS. The NFS volume on the AFF A800 was mounted on `/sparkdemo/horovod` across the five nodes (one master, four workers) in our Spark cluster. We ran a similar command as for previous tests, with the `--data- dir` parameter now pointing to the NFS mount:

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkdemo/horovod
--local-submission-csv /tmp/submission_2.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_nfs.log 2>&1
```

The resulting runtime with NFS was as follows:

```
real 5m46.229s
user 5m35.693s
sys  1m5.615s
```

There was a further 1.43x speedup, as shown in the following figure. Therefore, with a NetApp all-flash storage connected to their cluster, customers enjoy the benefits of fast data transfer and distribution for Horovod Spark workflows, achieving 7.55x speedup versus running on a single node.



Deep learning models for CTR prediction performance

For recommender systems designed to maximize CTR, you must learn sophisticated feature interactions behind user behaviors that can be mathematically calculated from low order to high order. Both low-order and high-order feature interactions should be equally important for a good deep learning model without biasing towards one or the other. Deep Factorization Machine (DeepFM), a factorization machine-based neural network, combines factorization machines for recommendation and deep learning for feature learning in a new neural network architecture.

Although conventional factorization machines model pairwise feature interactions as an inner product of latent vectors between features and can theoretically capture high-order information, in practice, machine learning practitioners usually only use second- order feature interactions due to the high computation and storage complexity. Deep neural network variants like Google's Wide & Deep Models on the other hand learns sophisticated feature interactions in a hybrid network structure by combining a linear wide model and a deep model.

There are two inputs to this Wide & Deep Model, one for the underlying wide model and the other for the deep, the latter part of which still requires expert feature engineering and thus renders the technique less generalizable to other domains. Unlike the Wide & Deep Model, DeepFM can be efficiently trained with raw

features without any feature engineering because its wide part and deep part share the same input and the embedding vector.

We first processed the Criteo `train.txt` (11GB) file into a CSV file named `ctr_train.csv` stored in an NFS mount `/sparkdemo/tr-4570-data` using `run_classification_criteo_spark.py` from the section "Python scripts for each major use case." Within this script, the function `process_input_file` performs several string methods to remove tabs and insert `','` as the delimiter and `'\n'` as newline. Note that you only need to process the original `train.txt` once, so that the code block is shown as comments.

For the following testing of different DL models, we used `ctr_train.csv` as the input file. In subsequent testing runs, the input CSV file was read into a Spark DataFrame with schema containing a field of `'label'`, integer dense features `['I1', 'I2', 'I3', …, 'I13']`, and sparse features `['C1', 'C2', 'C3', …, 'C26']`. The following `spark-submit` command takes in an input CSV, trains DeepFM models with 20% split for cross validation, and picks the best model after ten training epochs to calculate prediction accuracy on the testing set:

```
(base) [root@n138 ~]# time spark-submit --master yarn --executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py --data
-dir file:///sparkdemo/tr-4570-data >
/tmp/run_classification_criteo_spark_local.log 2>&1
```

Note that since the data file `ctr_train.csv` is over 11GB, you must set a sufficient `spark.driver.maxResultSize` greater than the dataset size to avoid error.

```
 spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
```

In the above `SparkSession.builder` configuration we also enabled Apache Arrow, which converts a Spark DataFrame into a Pandas DataFrame with the `df.toPandas()` method.

```
22/06/17 15:56:21 INFO scheduler.DAGScheduler: Job 2 finished: toPandas at
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py:96, took
627.126487 s
Obtained Spark DF and transformed to Pandas DF using Arrow.
```

After random splitting, there are over 36M rows in the training dataset and 9M samples in the testing set:

```
Training dataset size =  36672493
Testing dataset size =  9168124
```

Because this technical report is focused on CPU testing without using any GPUs, it is imperative that you build TensorFlow with appropriate compiler flags. This step avoids invoking any GPU-accelerated libraries and takes full advantage of TensorFlow's Advanced Vector Extensions (AVX) and AVX2 instructions. These features are designed for linear algebraic computations like vectorized addition, matrix multiplications inside a feed-forward, or back-propagation DNN training. Fused Multiply Add (FMA) instruction available with AVX2 using 256-bit floating point (FP) registers is ideal for integer code and data types, resulting in up to a 2x speedup. For FP code and data types, AVX2 achieves 8% speedup over AVX.

```
2022-06-18 07:19:20.101478: I
tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary
is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the
appropriate compiler flags.
```

To build TensorFlow from source, NetApp recommends using Bazel. For our environment, we executed the following commands in the shell prompt to install `dnf`, `dnf-plugins`, and Bazel.

```
yum install dnf
dnf install 'dnf-command(copr)'
dnf copr enable vbatts/bazel
dnf install bazel5
```

You must enable GCC 5 or newer to use C++17 features during the build process, which is provided by RHEL with Software Collections Library (SCL). The following commands install `devtoolset` and GCC 11.2.1 on our RHEL 7.9 cluster:

```
subscription-manager repos --enable rhel-server-rhscl-7-rpms
yum install devtoolset-11-toolchain
yum install devtoolset-11-gcc-c++
yum update
scl enable devtoolset-11 bash
. /opt/rh/devtoolset-11/enable
```

Note that the last two commands enable `devtoolset-11`, which uses `/opt/rh/devtoolset-11/root/usr/bin/gcc` (GCC 11.2.1). Also, make sure your `git` version is greater than 1.8.3 (this comes with RHEL 7.9). Refer to this article for updating `git` to 2.24.1.

We assume that you have already cloned the latest TensorFlow master repo. Then create a `workspace` directory with a `WORKSPACE` file to build TensorFlow from source with AVX, AVX2, and FMA. Run the `configure` file and specify the correct Python binary location. CUDA is disabled for our testing because we did not use a GPU. A `.bazelrc` file is generated according to your settings. Further, we edited the file and set `build --define=no_hdfs_support=false` to enable HDFS support. Refer to `.bazelrc` in the section "Python scripts for each major use case," for a complete list of settings and flags.

```
./configure
bazel build -c opt --copt=-mavx --copt=-mavx2 --copt=-mfma --copt=
-mfpmath=both -k //tensorflow/tools/pip_package:build_pip_package
```

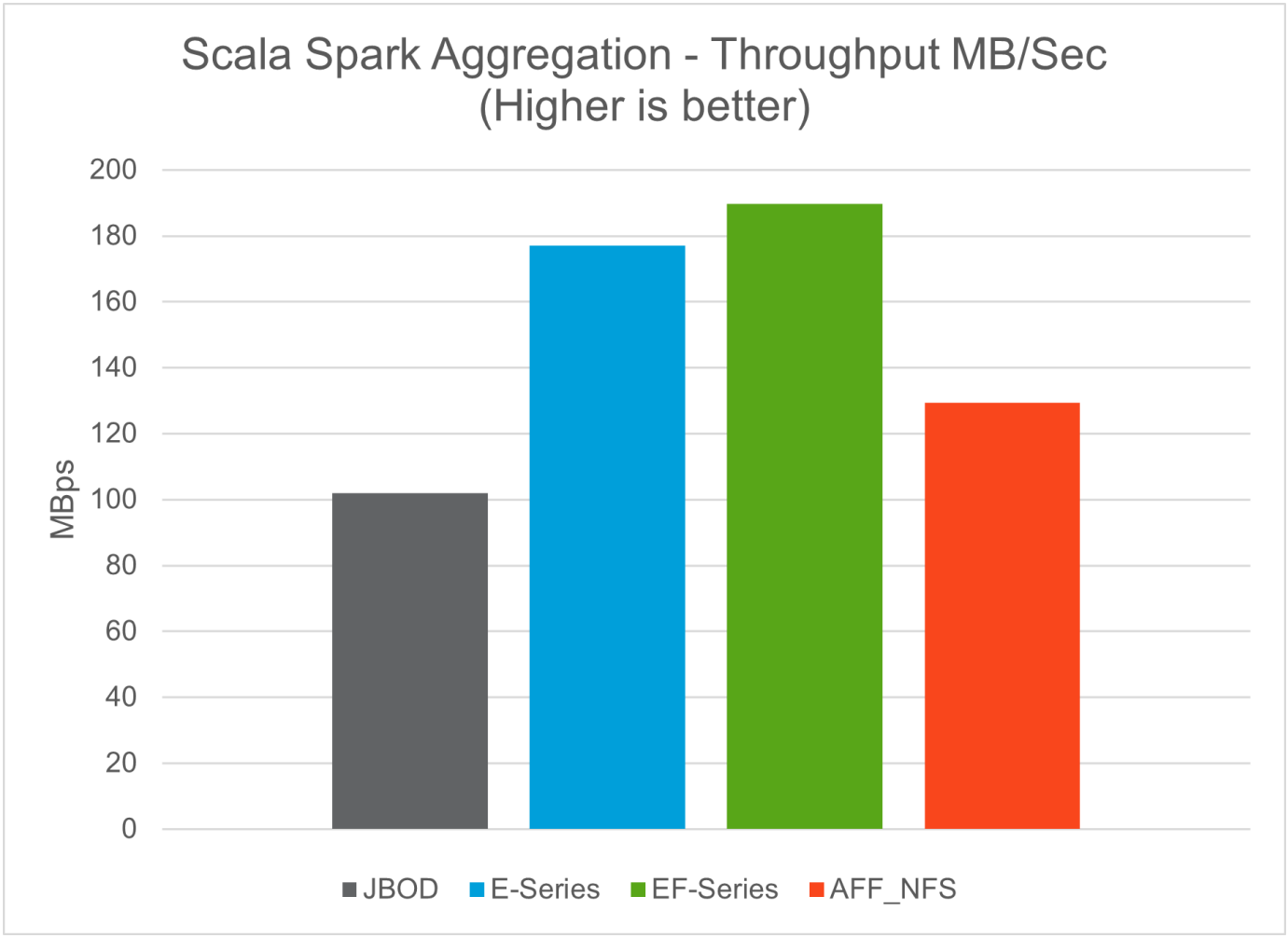
After you build TensorFlow with the correct flags, run the following script to process the Criteo Display Ads dataset, train a DeepFM model, and calculate the Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

After ten training epochs, we obtained the AUC score on the testing dataset:

```
Epoch 1/10
125/125 - 7s - loss: 0.4976 - binary_crossentropy: 0.4974 - val_loss:
0.4629 - val_binary_crossentropy: 0.4624
Epoch 2/10
125/125 - 1s - loss: 0.3281 - binary_crossentropy: 0.3271 - val_loss:
0.5146 - val_binary_crossentropy: 0.5130
Epoch 3/10
125/125 - 1s - loss: 0.1948 - binary_crossentropy: 0.1928 - val_loss:
0.6166 - val_binary_crossentropy: 0.6144
Epoch 4/10
125/125 - 1s - loss: 0.1408 - binary_crossentropy: 0.1383 - val_loss:
0.7261 - val_binary_crossentropy: 0.7235
Epoch 5/10
125/125 - 1s - loss: 0.1129 - binary_crossentropy: 0.1102 - val_loss:
0.7961 - val_binary_crossentropy: 0.7934
Epoch 6/10
125/125 - 1s - loss: 0.0949 - binary_crossentropy: 0.0921 - val_loss:
0.9502 - val_binary_crossentropy: 0.9474
Epoch 7/10
125/125 - 1s - loss: 0.0778 - binary_crossentropy: 0.0750 - val_loss:
1.1329 - val_binary_crossentropy: 1.1301
Epoch 8/10
125/125 - 1s - loss: 0.0651 - binary_crossentropy: 0.0622 - val_loss:
1.3794 - val_binary_crossentropy: 1.3766
Epoch 9/10
125/125 - 1s - loss: 0.0555 - binary_crossentropy: 0.0527 - val_loss:
1.6115 - val_binary_crossentropy: 1.6087
Epoch 10/10
125/125 - 1s - loss: 0.0470 - binary_crossentropy: 0.0442 - val_loss:
1.6768 - val_binary_crossentropy: 1.6740
test AUC 0.6337
```

In a manner similar to previous use cases, we compared the Spark workflow runtime with data residing in different locations. The following figure shows a comparison of the deep learning CTR prediction for a Spark workflows runtime.

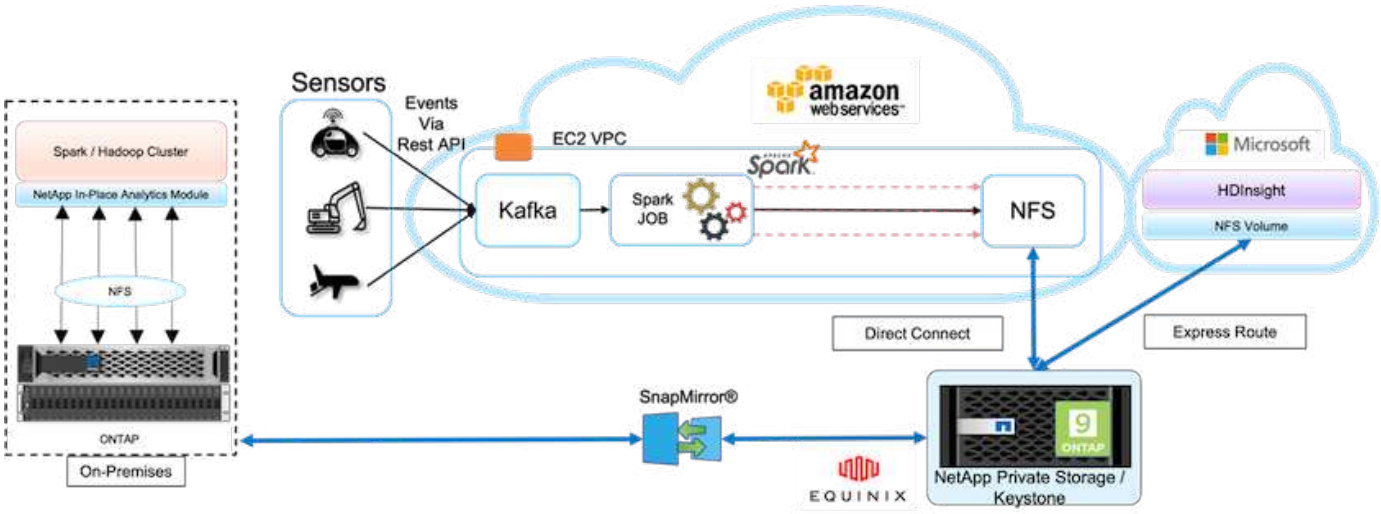


Hybrid cloud solution

A modern enterprise data center is a hybrid cloud that connects multiple distributed infrastructure environments through a continuous data management plane with a consistent operating model, on premises and/or in multiple public clouds. To get the most out of a hybrid cloud, you must be able to seamlessly move data between your on-premises and multi-cloud environments without the need for any data conversions or application refactoring.

Customers have indicated that they start their hybrid cloud journey either by moving secondary storage to the cloud for use cases such as data protection or by moving less business-critical workloads such as application development and DevOps to the cloud. They then move on to more critical workloads. Web and content hosting, DevOps and application development, databases, analytics, and containerized apps are among the most popular hybrid-cloud workloads. The complexity, cost, and risks of enterprise AI projects have historically hindered AI adoption from experimental stage to production.

With a NetApp hybrid-cloud solution, customers benefit from integrated security, data governance, and compliance tools with a single control panel for data and workflow management across distributed environments, while optimizing the total cost of ownership based on their consumption. The following figure is an example solution of a cloud service partner tasked with providing multi-cloud connectivity for customers' big-data-analytics data.



In this scenario, IoT data received in AWS from different sources is stored in a central location in NetApp Private Storage (NPS). The NPS storage is connected to Spark or Hadoop clusters located in AWS and Azure enabling big-data-analytics applications running in multiple clouds accessing the same data. The main requirements and challenges for this use case include the following:

- Customers want to run analytics jobs on the same data using multiple clouds.
- Data must be received from different sources such as on-premises and cloud environments through different sensors and hubs.
- The solution must be efficient and cost effective.
- The main challenge is to build a cost-effective and efficient solution that delivers hybrid analytics services between different on-premises and cloud environments.

Our data protection and multicloud connectivity solution resolves the pain point of having cloud analytics applications across multiple hyperscalers. As shown in the figure above, data from sensors is streamed and ingested into the AWS Spark cluster through Kafka. The data is stored in an NFS share residing in NPS, which is located outside of the cloud provider within an Equinix data center.

Because NetApp NPS is connected to Amazon AWS and Microsoft Azure through Direct Connect and Express Route connections respectively, customers can leverage the In-Place Analytics Module to access the data from both Amazon and AWS analytics clusters. Consequently, because both on-premises and NPS storage runs ONTAP software, SnapMirror can mirror the NPS data into the on-premises cluster, providing hybrid cloud analytics across on-premises and multiple clouds.

For the best performance, NetApp typically recommends using multiple network interfaces and direct connection or express routes to access the data from cloud instances. We have other data mover solutions including XCP and BlueXP Copy and Sync to help customers build application-aware, secure, and cost-effective hybrid-cloud Spark clusters.

Python scripts for each major use case

The following three Python scripts correspond to the three major use cases tested. First is `sentiment_analysis_sparknlp.py`.

```
# TR-4570 Refresh NLP testing by Rick Huang
from sys import argv
```

```
import os
import sparknlp
import pyspark.sql.functions as F
from sparknlp import Finisher
from pyspark.ml import Pipeline
from sparknlp.base import *
from sparknlp.annotator import *
from sparknlp.pretrained import PretrainedPipeline
from sparknlp import Finisher
# Start Spark Session with Spark NLP
spark = sparknlp.start()
print("Spark NLP version:")
print(sparknlp.version())
print("Apache Spark version:")
print(spark.version)
spark = sparknlp.SparkSession.builder \
     .master("yarn") \
     .appName("test_hdfs_read_write") \
     .config("spark.executor.cores", "1") \
     .config("spark.jars.packages", "com.johnsnowlabs.nlp:spark-
nlp_2.12:3.4.3")\
     .config('spark.executor.memory', '5gb') \
     .config('spark.executor.memoryOverhead','1000')\
     .config('spark.driver.memoryOverhead','1000')\
     .config("spark.sql.shuffle.partitions", "480")\
     .getOrCreate()
sc = spark.sparkContext
from pyspark.sql import SQLContext
sql = SQLContext(sc)
sqlContext = SQLContext(sc)
# Download pre-trained pipelines & sequence classifier
explain_pipeline_model = PretrainedPipeline('explain_document_dl',
lang='en').model#pipeline_sa =
PretrainedPipeline("classifierdl_bertwiki_finance_sentiment_pipeline",
lang="en")
# pipeline_finbert =
BertForSequenceClassification.loadSavedModel('/sparkusecase/bert_sequence_
classifier_finbert_en_3', spark)
sequenceClassifier = BertForSequenceClassification \
        .pretrained('bert_sequence_classifier_finbert', 'en') \
        .setInputCols(['token', 'document']) \
        .setOutputCol('class') \
        .setCaseSensitive(True) \
        .setMaxSentenceLength(512)
def process_sentence_df(data):
    # Pre-process: begin
```

```
    print("1. Begin DataFrame pre-processing...\n")
    print(f"\n\t2. Attaching DocumentAssembler Transformer to the
pipeline")
    documentAssembler = DocumentAssembler() \
        .setInputCol("text") \
        .setOutputCol("document") \
        .setCleanupMode("inplace_full")
        #.setCleanupMode("shrink", "inplace_full")
    doc_df = documentAssembler.transform(data)
    doc_df.printSchema()
    doc_df.show(truncate=50)
    # Pre-process: get rid of  blank lines
    clean_df = doc_df.withColumn("tmp", F.explode("document")) \
        .select("tmp.result").where("tmp.end !=
-1").withColumnRenamed("result", "text").dropna()
    print("[OK!] DataFrame after initial cleanup:\n")
    clean_df.printSchema()
    clean_df.show(truncate=80)
    # for FinBERT
    tokenizer = Tokenizer() \
        .setInputCols(['document']) \
        .setOutputCol('token')
    print(f"\n\t3. Attaching Tokenizer Annotator to the pipeline")
    pipeline_finbert = Pipeline(stages=[
        documentAssembler,
        tokenizer,
        sequenceClassifier
        ])
    # Use Finisher() & construct PySpark ML pipeline
    finisher = Finisher().setInputCols(["token", "lemma", "pos",
"entities"])
    print(f"\n\t4. Attaching Finisher Transformer to the pipeline")
    pipeline_ex = Pipeline() \
        .setStages([
           explain_pipeline_model,
           finisher
           ])
    print("\n\t\t\t ---- Pipeline Built Successfully ----")
    # Loading pipelines to annotate
    #result_ex_df = pipeline_ex.transform(clean_df)
    ex_model = pipeline_ex.fit(clean_df)
    annotations_finished_ex_df = ex_model.transform(clean_df)
    # result_sa_df = pipeline_sa.transform(clean_df)
    result_finbert_df = pipeline_finbert.fit(clean_df).transform(clean_df)
    print("\n\t\t\t ----Document Explain, Sentiment Analysis & FinBERT
Pipeline Fitted Successfully ----")
```

```
    # Check the result entities
    print("[OK!] Simple explain ML pipeline result:\n")
    annotations_finished_ex_df.printSchema()
    annotations_finished_ex_df.select('text',
'finished_entities').show(truncate=False)
    # Check the result sentiment from FinBERT
    print("[OK!] Sentiment Analysis FinBERT pipeline result:\n")
    result_finbert_df.printSchema()
    result_finbert_df.select('text', 'class.result').show(80, False)
    sentiment_stats(result_finbert_df)
    return
def sentiment_stats(finbert_df):
    result_df = finbert_df.select('text', 'class.result')
    sa_df = result_df.select('result')
    sa_df.groupBy('result').count().show()
    # total_lines = result_clean_df.count()
    # num_neutral = result_clean_df.where(result_clean_df.result ==
['neutral']).count()
    # num_positive = result_clean_df.where(result_clean_df.result ==
['positive']).count()
    # num_negative = result_clean_df.where(result_clean_df.result ==
['negative']).count()
    # print(f"\nRatio of neutral sentiment = {num_neutral/total_lines}")
    # print(f"Ratio of positive sentiment = {num_positive / total_lines}")
    # print(f"Ratio of negative sentiment = {num_negative /
total_lines}\n")
    return
def process_input_file(file_name):
    # Turn input file to Spark DataFrame
    print("START processing input file...")
    data_df = spark.read.text(file_name)
    data_df.show()
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    output_df.printSchema()
    return output_dfdef process_local_dir(directory):
    filelist = []
    for subdir, dirs, files in os.walk(directory):
        for filename in files:
            filepath = subdir + os.sep + filename
            print("[OK!] Will process the following files:")
            if filepath.endswith(".txt"):
                print(filepath)
                filelist.append(filepath)
    return filelist
def process_local_dir_or_file(dir_or_file):
```

```
    numfiles = 0
    if os.path.isfile(dir_or_file):
        input_df = process_input_file(dir_or_file)
        print("Obtained input_df.")
        process_sentence_df(input_df)
        print("Processed input_df")
        numfiles += 1
    else:
        filelist = process_local_dir(dir_or_file)
        for file in filelist:
            input_df = process_input_file(file)
            process_sentence_df(input_df)
            numfiles += 1
    return numfiles
def process_hdfs_dir(dir_name):
    # Turn input files to Spark DataFrame
    print("START processing input HDFS directory...")
    data_df = spark.read.option("recursiveFileLookup",
"true").text(dir_name)
    data_df.show()
    print("[DEBUG] total lines in data_df = ", data_df.count())
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    print("[DEBUG] output_df looks like: \n")
    output_df.show(40, False)
    print("[DEBUG] HDFS dir resulting data_df schema: \n")
    output_df.printSchema()
    process_sentence_df(output_df)
    print("Processed HDFS directory: ", dir_name)
    returnif __name__ == '__main__':
    try:
        if len(argv) == 2:
            print("Start processing input...\n")
    except:
        print("[ERROR] Please enter input text file or path to
process!\n")
        exit(1)
    # This is for local file, not hdfs:
    numfiles = process_local_dir_or_file(str(argv[1]))
    # For HDFS single file & directory:
    input_df = process_input_file(str(argv[1]))
    print("Obtained input_df.")
    process_sentence_df(input_df)
    print("Processed input_df")
    numfiles += 1
    # For HDFS directory of subdirectories of files:
```

```
    input_parse_list = str(argv[1]).split('/')
    print(input_parse_list)
    if input_parse_list[-2:-1] == ['Transcripts']:
        print("Start processing HDFS directory: ", str(argv[1]))
        process_hdfs_dir(str(argv[1]))
    print(f"[OK!] All done. Number of files processed = {numfiles}")
```

The second script is `keras_spark_horovod_rossmann_estimator.py`.

```
# Copyright 2022 NetApp, Inc.
# Authored by Rick Huang
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
==========================================================================
====
# The below code was modified from: https://www.kaggle.com/c/rossmann-
store-sales
import argparse
import datetime
import os
import sys
from distutils.version import LooseVersion
import pyspark.sql.types as T
import pyspark.sql.functions as F
from pyspark import SparkConf, Row
from pyspark.sql import SparkSession
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Input, Embedding, Concatenate, Dense,
Flatten, Reshape, BatchNormalization, Dropout
import horovod.spark.keras as hvd
from horovod.spark.common.backend import SparkBackend
from horovod.spark.common.store import Store
from horovod.tensorflow.keras.callbacks import BestModelCheckpoint
```

```
parser = argparse.ArgumentParser(description='Horovod Keras Spark Rossmann
Estimator Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--master',
                    help='spark cluster to use for training. If set to
None, uses current default cluster. Cluster'
                         'should be set up to provide a Spark task per
multiple CPU cores, or per GPU, e.g. by'
                         'supplying `-c <NUM_GPUS>` in Spark Standalone
mode')
parser.add_argument('--num-proc', type=int,
                    help='number of worker processes for training,
default: `spark.default.parallelism`')
parser.add_argument('--learning_rate', type=float, default=0.0001,
                    help='initial learning rate')
parser.add_argument('--batch-size', type=int, default=100,
                    help='batch size')
parser.add_argument('--epochs', type=int, default=100,
                    help='number of epochs to train')
parser.add_argument('--sample-rate', type=float,
                    help='desired sampling rate. Useful to set to low
number (e.g. 0.01) to make sure that '
                         'end-to-end process works')
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
parser.add_argument('--local-submission-csv', default='submission.csv',
                    help='output submission predictions CSV')
parser.add_argument('--local-checkpoint-file', default='checkpoint',
                    help='model checkpoint')
parser.add_argument('--work-dir', default='/tmp',
                    help='temporary working directory to write
intermediate files (prefix with hdfs:// to use HDFS)')
if __name__ == '__main__':
    args = parser.parse_args()
    # ================ #
    # DATA PREPARATION #
    # ================ #
    print('================')
    print('Data preparation')
    print('================')
    # Create Spark session for data preparation.
    conf = SparkConf() \
        .setAppName('Keras Spark Rossmann Estimator Example') \
        .set('spark.sql.shuffle.partitions', '480') \
```

```
        .set("spark.executor.cores", "1") \
        .set('spark.executor.memory', '5gb') \
        .set('spark.executor.memoryOverhead','1000')\
        .set('spark.driver.memoryOverhead','1000')
    if args.master:
        conf.setMaster(args.master)
    elif args.num_proc:
        conf.setMaster('local[{}]'.format(args.num_proc))
    spark = SparkSession.builder.config(conf=conf).getOrCreate()
    train_csv = spark.read.csv('%s/train.csv' % args.data_dir,
header=True)
    test_csv = spark.read.csv('%s/test.csv' % args.data_dir, header=True)
    store_csv = spark.read.csv('%s/store.csv' % args.data_dir,
header=True)
    store_states_csv = spark.read.csv('%s/store_states.csv' %
args.data_dir, header=True)
    state_names_csv = spark.read.csv('%s/state_names.csv' % args.data_dir,
header=True)
    google_trend_csv = spark.read.csv('%s/googletrend.csv' %
args.data_dir, header=True)
    weather_csv = spark.read.csv('%s/weather.csv' % args.data_dir,
header=True)
    def expand_date(df):
        df = df.withColumn('Date', df.Date.cast(T.DateType()))
        return df \
            .withColumn('Year', F.year(df.Date)) \
            .withColumn('Month', F.month(df.Date)) \
            .withColumn('Week', F.weekofyear(df.Date)) \
            .withColumn('Day', F.dayofmonth(df.Date))
    def prepare_google_trend():
        # Extract week start date and state.
        google_trend_all = google_trend_csv \
            .withColumn('Date', F.regexp_extract(google_trend_csv.week,
'(.*?) -', 1)) \
            .withColumn('State', F.regexp_extract(google_trend_csv.file,
'Rossmann_DE_(.*)', 1))
        # Map state NI -> HB,NI to align with other data sources.
        google_trend_all = google_trend_all \
            .withColumn('State', F.when(google_trend_all.State == 'NI',
'HB,NI').otherwise(google_trend_all.State))
        # Expand dates.
        return expand_date(google_trend_all)
    def add_elapsed(df, cols):
        def add_elapsed_column(col, asc):
            def fn(rows):
                last_store, last_date = None, None
```

```
                for r in rows:
                    if last_store != r.Store:
                        last_store = r.Store
                        last_date = r.Date
                    if r[col]:
                        last_date = r.Date
                    fields = r.asDict().copy()
                    fields[('After' if asc else 'Before') + col] = (r.Date
- last_date).days
                    yield Row(**fields)
            return fn
        df = df.repartition(df.Store)
        for asc in [False, True]:
            sort_col = df.Date.asc() if asc else df.Date.desc()
            rdd = df.sortWithinPartitions(df.Store.asc(), sort_col).rdd
            for col in cols:
                rdd = rdd.mapPartitions(add_elapsed_column(col, asc))
            df = rdd.toDF()
        return df
    def prepare_df(df):
        num_rows = df.count()
        # Expand dates.
        df = expand_date(df)
        df = df \
            .withColumn('Open', df.Open != '0') \
            .withColumn('Promo', df.Promo != '0') \
            .withColumn('StateHoliday', df.StateHoliday != '0') \
            .withColumn('SchoolHoliday', df.SchoolHoliday != '0')
        # Merge in store information.
        store = store_csv.join(store_states_csv, 'Store')
        df = df.join(store, 'Store')
        # Merge in Google Trend information.
        google_trend_all = prepare_google_trend()
        df = df.join(google_trend_all, ['State', 'Year',
'Week']).select(df['*'], google_trend_all.trend)
        # Merge in Google Trend for whole Germany.
        google_trend_de = google_trend_all[google_trend_all.file ==
'Rossmann_DE'].withColumnRenamed('trend', 'trend_de')
        df = df.join(google_trend_de, ['Year', 'Week']).select(df['*'],
google_trend_de.trend_de)
        # Merge in weather.
        weather = weather_csv.join(state_names_csv, weather_csv.file ==
state_names_csv.StateName)
        df = df.join(weather, ['State', 'Date'])
        # Fix null values.
        df = df \
```

```
            .withColumn('CompetitionOpenSinceYear',
F.coalesce(df.CompetitionOpenSinceYear, F.lit(1900))) \
            .withColumn('CompetitionOpenSinceMonth',
F.coalesce(df.CompetitionOpenSinceMonth, F.lit(1))) \
            .withColumn('Promo2SinceYear', F.coalesce(df.Promo2SinceYear,
F.lit(1900))) \
            .withColumn('Promo2SinceWeek', F.coalesce(df.Promo2SinceWeek,
F.lit(1)))
        # Days & months competition was open, cap to 2 years.
        df = df.withColumn('CompetitionOpenSince',
                           F.to_date(F.format_string('%s-%s-15',
df.CompetitionOpenSinceYear,

df.CompetitionOpenSinceMonth)))
        df = df.withColumn('CompetitionDaysOpen',
                           F.when(df.CompetitionOpenSinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(360 *
2), F.datediff(df.Date, df.CompetitionOpenSince))))
                           .otherwise(0))
        df = df.withColumn('CompetitionMonthsOpen',
(df.CompetitionDaysOpen / 30).cast(T.IntegerType()))
        # Days & weeks of promotion, cap to 25 weeks.
        df = df.withColumn('Promo2Since',
                           F.expr('date_add(format_string("%s-01-01",
Promo2SinceYear), (cast(Promo2SinceWeek as int) - 1) * 7)'))
        df = df.withColumn('Promo2Days',
                           F.when(df.Promo2SinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(25 *
7), F.datediff(df.Date, df.Promo2Since))))
                           .otherwise(0))
        df = df.withColumn('Promo2Weeks', (df.Promo2Days /
7).cast(T.IntegerType()))
        # Check that we did not lose any rows through inner joins.
        assert num_rows == df.count(), 'lost rows in joins'
        return df
    def build_vocabulary(df, cols):
        vocab = {}
        for col in cols:
            values = [r[0] for r in df.select(col).distinct().collect()]
            col_type = type([x for x in values if x is not None][0])
            default_value = col_type()
            vocab[col] = sorted(values, key=lambda x: x or default_value)
        return vocab
    def cast_columns(df, cols):
        for col in cols:
            df = df.withColumn(col,
```

```
F.coalesce(df[col].cast(T.FloatType()), F.lit(0.0)))
        return df
    def lookup_columns(df, vocab):
        def lookup(mapping):
            def fn(v):
                return mapping.index(v)
            return F.udf(fn, returnType=T.IntegerType())
        for col, mapping in vocab.items():
            df = df.withColumn(col, lookup(mapping)(df[col]))
        return df
    if args.sample_rate:
        train_csv = train_csv.sample(withReplacement=False,
fraction=args.sample_rate)
        test_csv = test_csv.sample(withReplacement=False,
fraction=args.sample_rate)
    # Prepare data frames from CSV files.
    train_df = prepare_df(train_csv).cache()
    test_df = prepare_df(test_csv).cache()
    # Add elapsed times from holidays & promos, the data spanning training
& test datasets.
    elapsed_cols = ['Promo', 'StateHoliday', 'SchoolHoliday']
    elapsed = add_elapsed(train_df.select('Date', 'Store', *elapsed_cols)
                          .unionAll(test_df.select('Date', 'Store',
*elapsed_cols)),
                          elapsed_cols)
    # Join with elapsed times.
    train_df = train_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(train_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    test_df = test_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(test_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    # Filter out zero sales.
    train_df = train_df.filter(train_df.Sales > 0)
    print('====================')
    print('Prepared data frame')
    print('====================')
    train_df.show()
    categorical_cols = [
        'Store', 'State', 'DayOfWeek', 'Year', 'Month', 'Day', 'Week',
'CompetitionMonthsOpen', 'Promo2Weeks', 'StoreType',
        'Assortment', 'PromoInterval', 'CompetitionOpenSinceYear',
'Promo2SinceYear', 'Events', 'Promo',
        'StateHoliday', 'SchoolHoliday'
```

```
    ]
    continuous_cols = [
        'CompetitionDistance', 'Max_TemperatureC', 'Mean_TemperatureC',
'Min_TemperatureC', 'Max_Humidity',
        'Mean_Humidity', 'Min_Humidity', 'Max_Wind_SpeedKm_h',
'Mean_Wind_SpeedKm_h', 'CloudCover', 'trend', 'trend_de',
        'BeforePromo', 'AfterPromo', 'AfterStateHoliday',
'BeforeStateHoliday', 'BeforeSchoolHoliday', 'AfterSchoolHoliday'
    ]
    all_cols = categorical_cols + continuous_cols
    # Select features.
    train_df = train_df.select(*(all_cols + ['Sales', 'Date'])).cache()
    test_df = test_df.select(*(all_cols + ['Id', 'Date'])).cache()
    # Build vocabulary of categorical columns.
    vocab = build_vocabulary(train_df.select(*categorical_cols)

.unionAll(test_df.select(*categorical_cols)).cache(),
                             categorical_cols)
    # Cast continuous columns to float & lookup categorical columns.
    train_df = cast_columns(train_df, continuous_cols + ['Sales'])
    train_df = lookup_columns(train_df, vocab)
    test_df = cast_columns(test_df, continuous_cols)
    test_df = lookup_columns(test_df, vocab)
    # Split into training & validation.
    # Test set is in 2015, use the same period in 2014 from the training
set as a validation set.
    test_min_date = test_df.agg(F.min(test_df.Date)).collect()[0][0]
    test_max_date = test_df.agg(F.max(test_df.Date)).collect()[0][0]
    one_year = datetime.timedelta(365)
    train_df = train_df.withColumn('Validation',
                                   (train_df.Date > test_min_date -
one_year) & (train_df.Date <= test_max_date - one_year))
    # Determine max Sales number.
    max_sales = train_df.agg(F.max(train_df.Sales)).collect()[0][0]
    # Convert Sales to log domain
    train_df = train_df.withColumn('Sales', F.log(train_df.Sales))
    print('====================================')
    print('Data frame with transformed columns')
    print('====================================')
    train_df.show()
    print('================')
    print('Data frame sizes')
    print('================')
    train_rows = train_df.filter(~train_df.Validation).count()
    val_rows = train_df.filter(train_df.Validation).count()
    test_rows = test_df.count()
```

```
    print('Training: %d' % train_rows)
    print('Validation: %d' % val_rows)
    print('Test: %d' % test_rows)
    # ============== #
    # MODEL TRAINING #
    # ============== #
    print('==============')
    print('Model training')
    print('==============')
    def exp_rmspe(y_true, y_pred):
        """Competition evaluation metric, expects logarithic inputs."""
        pct = tf.square((tf.exp(y_true) - tf.exp(y_pred)) /
tf.exp(y_true))
        # Compute mean excluding stores with zero denominator.
        x = tf.reduce_sum(tf.where(y_true > 0.001, pct,
tf.zeros_like(pct)))
        y = tf.reduce_sum(tf.where(y_true > 0.001, tf.ones_like(pct),
tf.zeros_like(pct)))
        return tf.sqrt(x / y)
    def act_sigmoid_scaled(x):
        """Sigmoid scaled to logarithm of maximum sales scaled by 20%."""
        return tf.nn.sigmoid(x) * tf.math.log(max_sales) * 1.2
    CUSTOM_OBJECTS = {'exp_rmspe': exp_rmspe,
                      'act_sigmoid_scaled': act_sigmoid_scaled}
    # Disable GPUs when building the model to prevent memory leaks
    if LooseVersion(tf.__version__) >= LooseVersion('2.0.0'):
        # See https://github.com/tensorflow/tensorflow/issues/33168
        os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    else:

K.set_session(tf.Session(config=tf.ConfigProto(device_count={'GPU': 0})))
    # Build the model.
    inputs = {col: Input(shape=(1,), name=col) for col in all_cols}
    embeddings = [Embedding(len(vocab[col]), 10, input_length=1,
name='emb_' + col)(inputs[col])
                  for col in categorical_cols]
    continuous_bn = Concatenate()([Reshape((1, 1), name='reshape_' +
col)(inputs[col])
                                   for col in continuous_cols])
    continuous_bn = BatchNormalization()(continuous_bn)
    x = Concatenate()(embeddings + [continuous_bn])
    x = Flatten()(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
```

```
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(500, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dropout(0.5)(x)
    output = Dense(1, activation=act_sigmoid_scaled)(x)
    model = tf.keras.Model([inputs[f] for f in all_cols], output)
    model.summary()
    opt = tf.keras.optimizers.Adam(lr=args.learning_rate, epsilon=1e-3)
    # Checkpoint callback to specify options for the returned Keras model
    ckpt_callback = BestModelCheckpoint(monitor='val_loss', mode='auto',
save_freq='epoch')
    # Horovod: run training.
    store = Store.create(args.work_dir)
    backend = SparkBackend(num_proc=args.num_proc,
                           stdout=sys.stdout, stderr=sys.stderr,
                           prefix_output_with_timestamp=True)
    keras_estimator = hvd.KerasEstimator(backend=backend,
                                         store=store,
                                         model=model,
                                         optimizer=opt,
                                         loss='mae',
                                         metrics=[exp_rmspe],
                                         custom_objects=CUSTOM_OBJECTS,
                                         feature_cols=all_cols,
                                         label_cols=['Sales'],
                                         validation='Validation',
                                         batch_size=args.batch_size,
                                         epochs=args.epochs,
                                         verbose=2,

checkpoint_callback=ckpt_callback)
    keras_model =
keras_estimator.fit(train_df).setOutputCols(['Sales_output'])
    history = keras_model.getHistory()
    best_val_rmspe = min(history['val_exp_rmspe'])
    print('Best RMSPE: %f' % best_val_rmspe)
    # Save the trained model.
    keras_model.save(args.local_checkpoint_file)
    print('Written checkpoint to %s' % args.local_checkpoint_file)
    # ================ #
    # FINAL PREDICTION #
    # ================ #
    print('================')
    print('Final prediction')
    print('================')
```

```
    pred_df=keras_model.transform(test_df)
    pred_df.printSchema()
    pred_df.show(5)
    # Convert from log domain to real Sales numbers
    pred_df=pred_df.withColumn('Sales_pred', F.exp(pred_df.Sales_output))
    submission_df = pred_df.select(pred_df.Id.cast(T.IntegerType()),
pred_df.Sales_pred).toPandas()
    submission_df.sort_values(by=['Id']).to_csv(args.local_submission_csv,
index=False)
    print('Saved predictions to %s' % args.local_submission_csv)
    spark.stop()
```

The third script is `run_classification_criteo_spark.py`.

```
import tempfile, string, random, os, uuid
import argparse, datetime, sys, shutil
import csv
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from pyspark import SparkContext
from pyspark.sql import SparkSession, SQLContext, Row, DataFrame
from pyspark.mllib import linalg as mllib_linalg
from pyspark.mllib.linalg import SparseVector as mllibSparseVector
from pyspark.mllib.linalg import VectorUDT as mllibVectorUDT
from pyspark.mllib.linalg import Vector as mllibVector, Vectors as
mllibVectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.ml import linalg as ml_linalg
from pyspark.ml.linalg import VectorUDT as mlVectorUDT
from pyspark.ml.linalg import SparseVector as mlSparseVector
from pyspark.ml.linalg import Vector as mlVector, Vectors as mlVectors
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import OneHotEncoder
from math import log
from math import exp  # exp(-t) = e^-t
from operator import add
from pyspark.sql.functions import udf, split, lit
from pyspark.sql.functions import size, sum as sqlsum
import pyspark.sql.functions as F
import pyspark.sql.types as T
from pyspark.sql.types import ArrayType, StructType, StructField,
LongType, StringType, IntegerType, FloatType
from pyspark.sql.functions import explode, col, log, when
```

```
from collections import defaultdict
import pandas as pd
import pyspark.pandas as ps
from sklearn.metrics import log_loss, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, DenseFeat,
get_feature_names
spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
# spark.conf.set("spark.sql.execution.arrow.enabled", "true") # deprecated
print("Apache Spark version:")
print(spark.version)
sc = spark.sparkContext
sqlContext = SQLContext(sc)
parser = argparse.ArgumentParser(description='Spark DCN CTR Prediction
Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
def process_input_file(file_name, sparse_feat, dense_feat):
    # Need this preprocessing to turn Criteo raw file into CSV:
    print("START processing input file...")
    # only convert the file ONCE
    # sample = open(file_name)
    # sample = '\n'.join([str(x.replace('\n', '').replace('\t', ',')) for
x in sample])
    # # Add header in data file and save as CSV
    # header = ','.join(str(x) for x in (['label'] + dense_feat +
sparse_feat))
    # with open('/sparkdemo/tr-4570-data/ctr_train.csv', mode='w',
encoding="utf-8") as f:
```

```
    #     f.write(header + '\n' + sample)
    #     f.close()
    # print("Raw training file processed and saved as CSV: ", f.name)
    raw_df = sqlContext.read.option("header", True).csv(file_name)
    raw_df.show(5, False)
    raw_df.printSchema()
    # convert columns I1 to I13 from string to integers
    conv_df = raw_df.select(col('label').cast("double"),
                            *(col(i).cast("float").alias(i) for i in
raw_df.columns if i in dense_feat),
                            *(col(c) for c in raw_df.columns if c in
sparse_feat))
    print("Schema of raw_df with integer columns type changed:")
    conv_df.printSchema()
    # result_pdf = conv_df.select("*").toPandas()
    tmp_df = conv_df.na.fill(0, dense_feat)
    result_df = tmp_df.na.fill('-1', sparse_feat)
    result_df.show()
    return result_df
if __name__ == "__main__":
    args = parser.parse_args()
    # Pandas read CSV
    # data = pd.read_csv('%s/criteo_sample.txt' % args.data_dir)
    # print("Obtained Pandas df.")
    dense_features = ['I' + str(i) for i in range(1, 14)]
    sparse_features = ['C' + str(i) for i in range(1, 27)]
    # Spark read CSV
    # process_input_file('%s/train.txt' % args.data_dir, sparse_features,
dense_features) # run only ONCE
    spark_df = process_input_file('%s/data.txt' % args.data_dir,
sparse_features, dense_features) # sample data
    # spark_df = process_input_file('%s/ctr_train.csv' % args.data_dir,
sparse_features, dense_features)
    print("Obtained Spark df and filled in missing features.")
    data = spark_df
    # Pandas
    #data[sparse_features] = data[sparse_features].fillna('-1', )
    #data[dense_features] = data[dense_features].fillna(0, )
    target = ['label']
    label_npa = data.select("label").toPandas().to_numpy()
    print("label numPy array has length = ", len(label_npa)) # 45,840,617
w/ 11GB dataset
    label_npa.ravel()
    label_npa.reshape(len(label_npa), )
    # 1.Label Encoding for sparse features,and do simple Transformation
for dense features
```

```
    print("Before LabelEncoder():")
    data.printSchema()  # label: float (nullable = true)
    for feat in sparse_features:
        lbe = LabelEncoder()
        tmp_pdf = data.select(feat).toPandas().to_numpy()
        tmp_ndarray = lbe.fit_transform(tmp_pdf)
        print("After LabelEncoder(), tmp_ndarray[0] =", tmp_ndarray[0])
        # print("Data tmp PDF after lbe transformation, the output ndarray
has length = ", len(tmp_ndarray)) # 45,840,617 for 11GB dataset
        tmp_ndarray.ravel()
        tmp_ndarray.reshape(len(tmp_ndarray), )
        out_ndarray = np.column_stack([label_npa, tmp_ndarray])
        pdf = pd.DataFrame(out_ndarray, columns=['label', feat])
        s_df = spark.createDataFrame(pdf)
        s_df.printSchema() # label: double (nullable = true)
        print("Before joining data df with s_df, s_df example rows:")
        s_df.show(1, False)
        data = data.drop(feat).join(s_df, 'label').drop('label')
        print("After LabelEncoder(), data df example rows:")
        data.show(1, False)
        print("Finished processing sparse_features: ", feat)
    print("Data DF after label encoding: ")
    data.show()
    data.printSchema()
    mms = MinMaxScaler(feature_range=(0, 1))
    # data[dense_features] = mms.fit_transform(data[dense_features]) # for
Pandas df
    tmp_pdf = data.select(dense_features).toPandas().to_numpy()
    tmp_ndarray = mms.fit_transform(tmp_pdf)
    tmp_ndarray.ravel()
    tmp_ndarray.reshape(len(tmp_ndarray), len(tmp_ndarray[0]))
    out_ndarray = np.column_stack([label_npa, tmp_ndarray])
    pdf = pd.DataFrame(out_ndarray, columns=['label'] + dense_features)
    s_df = spark.createDataFrame(pdf)
    s_df.printSchema()
    data.drop(*dense_features).join(s_df, 'label').drop('label')
    print("Finished processing dense_features: ", dense_features)
    print("Data DF after MinMaxScaler: ")
    data.show()

    # 2.count #unique features for each sparse field,and record dense
feature field name
    fixlen_feature_columns = [SparseFeat(feat,
vocabulary_size=data.select(feat).distinct().count() + 1, embedding_dim=4)
                              for i, feat in enumerate(sparse_features)] +
\
```

```
                              [DenseFeat(feat, 1, ) for feat in
dense_features]
    dnn_feature_columns = fixlen_feature_columns
    linear_feature_columns = fixlen_feature_columns
    feature_names = get_feature_names(linear_feature_columns +
dnn_feature_columns)
    # 3.generate input data for model
    # train, test = train_test_split(data.toPandas(), test_size=0.2,
random_state=2020) # Pandas; might hang for 11GB data
    train, test = data.randomSplit(weights=[0.8, 0.2], seed=200)
    print("Training dataset size = ", train.count())
    print("Testing dataset size = ", test.count())
    # Pandas:
    # train_model_input = {name: train[name] for name in feature_names}
    # test_model_input = {name: test[name] for name in feature_names}
    # Spark DF:
    train_model_input = {}
    test_model_input = {}
    for name in feature_names:
        if name.startswith('I'):
            tr_pdf = train.select(name).toPandas()
            train_model_input[name] = pd.to_numeric(tr_pdf[name])
            ts_pdf = test.select(name).toPandas()
            test_model_input[name] = pd.to_numeric(ts_pdf[name])
    # 4.Define Model,train,predict and evaluate
    model = DeepFM(linear_feature_columns, dnn_feature_columns,
task='binary')
    model.compile("adam", "binary_crossentropy",
                  metrics=['binary_crossentropy'], )
    lb_pdf = train.select(target).toPandas()
    history = model.fit(train_model_input,
pd.to_numeric(lb_pdf['label']).values,
                        batch_size=256, epochs=10, verbose=2,
validation_split=0.2, )
    pred_ans = model.predict(test_model_input, batch_size=256)
    print("test LogLoss",
round(log_loss(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
    print("test AUC",
round(roc_auc_score(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
```

Conclusion

In this document, we discuss the Apache Spark architecture, customer use cases, and

the NetApp storage portfolio as it relates to big data, modern analytics, and AI, ML, and DL. In our performance validation tests based on industry-standard benchmarking tools and customer demand, the NetApp Spark solutions demonstrated superior performance relative to native Hadoop systems. A combination of the customer use cases and performance results presented in this report can help you to choose an appropriate Spark solution for your deployment.

Where to find additional information

The following references were used in this TR:

- Apache Spark architecture and components

  http://spark.apache.org/docs/latest/cluster-overview.html

- Apache Spark use cases

  https://www.qubole.com/blog/big-data/apache-spark-use-cases/

- Spark NLP

  https://www.johnsnowlabs.com/spark-nlp/

- BERT

  https://arxiv.org/abs/1810.04805

- Deep and Cross Network for Ad Click Predictions

  https://arxiv.org/abs/1708.05123

- FlexGroup

  https://www.netapp.com/pdf.html?item=/media/7337-tr4557pdf.pdf

- Streaming ETL

  https://www.infoq.com/articles/apache-spark-streaming

- NetApp E-Series Solutions for Hadoop

  https://www.netapp.com/media/16420-tr-3969.pdf

- NetApp Modern Data Analytics Solutions

  Data Analytics Solutions

- SnapMirror

  https://docs.netapp.com/us-en/ontap/data-protection/snapmirror-replication-concept.html

- XCP

https://mysupport.netapp.com/documentation/docweb/index.html?productID=63942&language=en-US

- BlueXP Copy and Sync

  https://cloud.netapp.com/cloud-sync-service

- DataOps Toolkit

  https://github.com/NetApp/netapp-dataops-toolkit

Copyright information

Trademark information