Testing results

NetApp Solutions

NetApp

October 20, 2023

Table of Contents

Testing results

Previous: Major AI, ML, and DL use cases and architectures.

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.

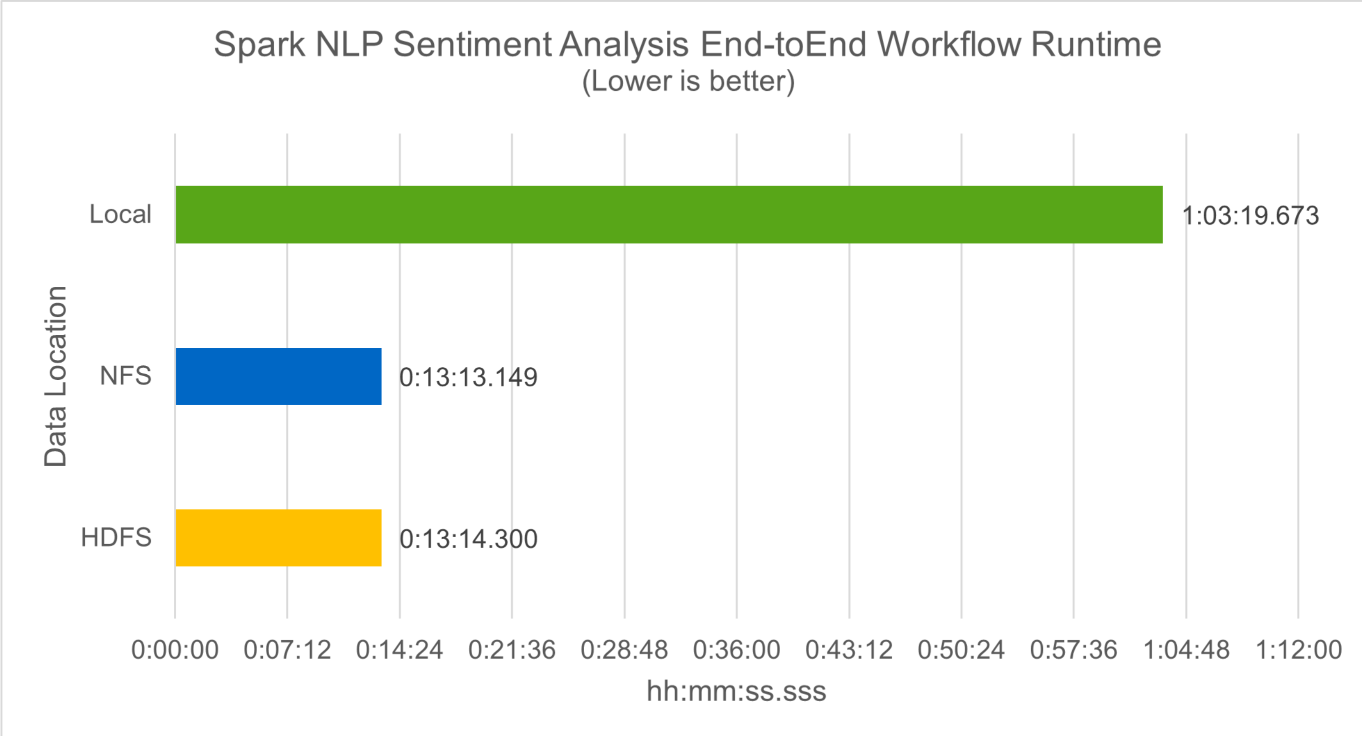
The following table lists the sentence-level sentiment analysis for NASDAQ Top 10 companies, expressed in percentage.

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.

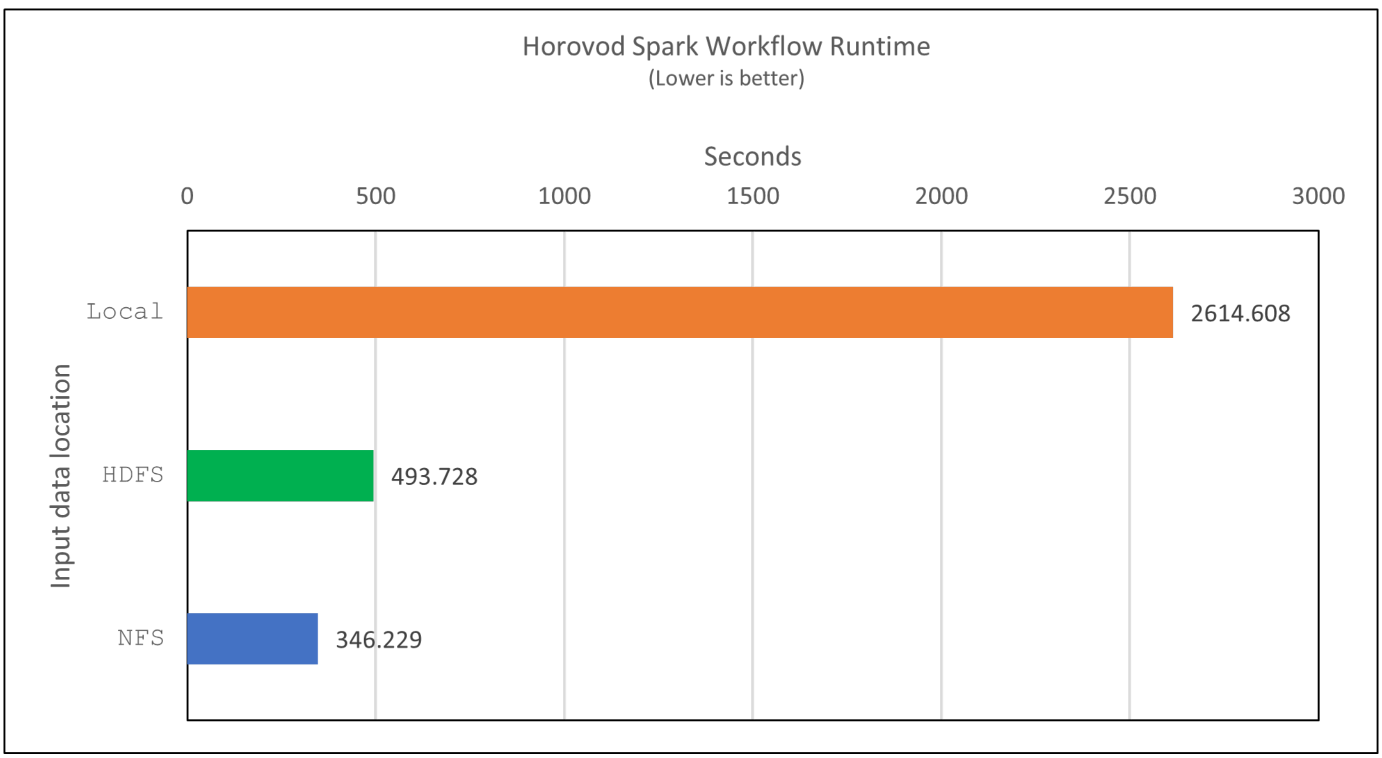
Our next test compared the runtimes with input data residing in NFS versus HDFS. The NFS volume on the AFF A800 was mounted on `/sparkdemo/horovod` across the five nodes (one master, four workers) in our Spark cluster. We ran a similar command as for previous tests, with the `--data- dir` parameter now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

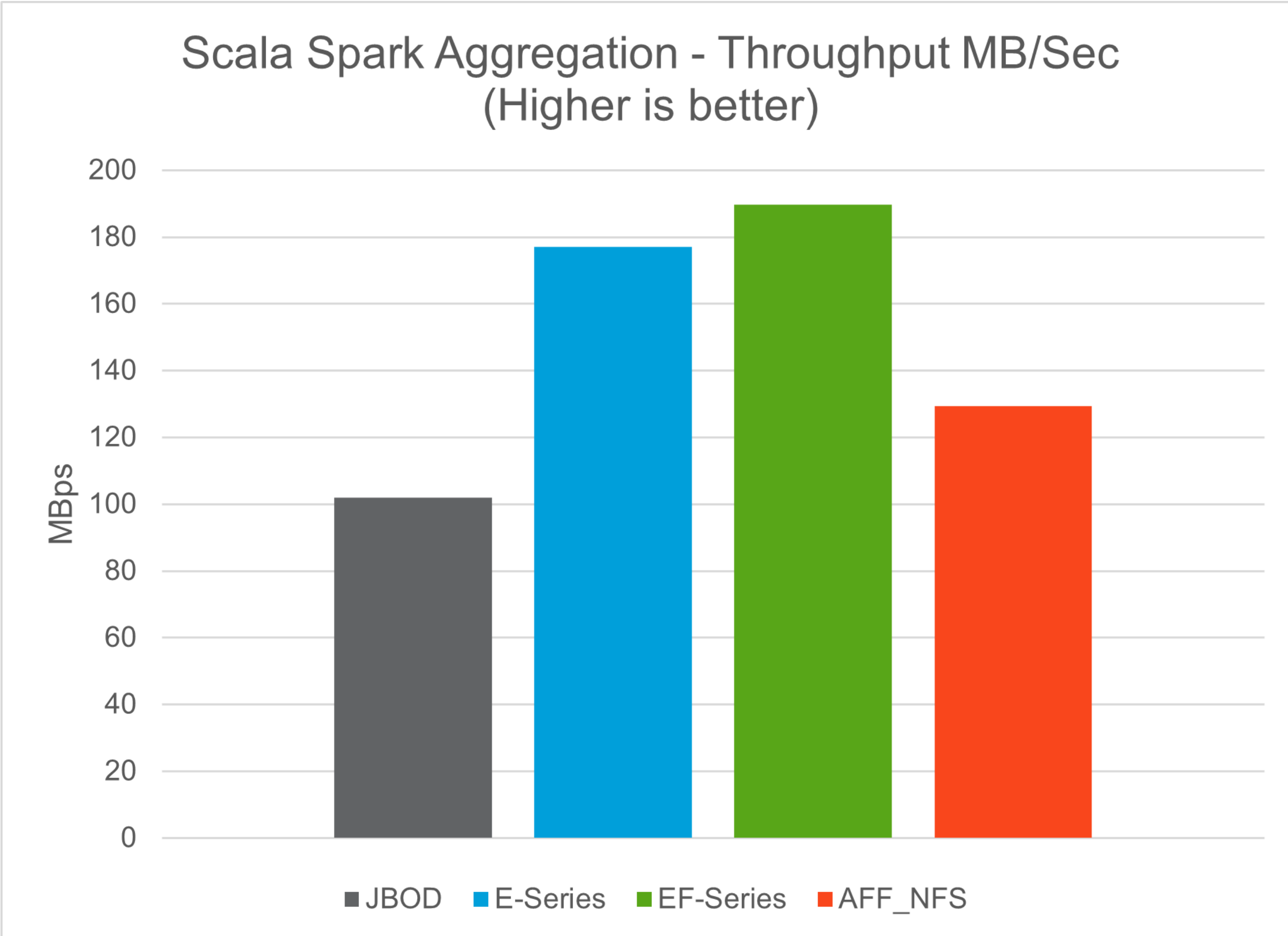
After you build TensorFlow with the correct flags, run the following script to process the Criteo Display Ads dataset, train a DeepFM model, and calculate the Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

```
(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.



Next: Hybrid cloud solution.