適用於 **Apache Spark** 的**NetApp**儲存解決方案

NetApp artificial intelligence solutions

NetApp
February 12, 2026

目錄

適用於 **Apache Spark** 的**NetApp**儲存解決方案

TR-4570：適用於 **Apache Spark** 的**NetApp**儲存解決方案：架構、用例和效能結果

Rick Huang，Karthikeyan Nagalingam， NetApp

本文檔重點介紹 Apache Spark 架構、客戶用例以及與大數據分析和人工智慧 (AI) 相關的NetApp儲存產品組合。它還展示了使用業界標準 AI、機器學習 (ML) 和深度學習 (DL) 工具針對典型 Hadoop 系統進行的各種測試結果，以便您可以選擇合適的 Spark 解決方案。首先，您需要一個 Spark 架構、適當的元件和兩種部署模式（叢集和用戶端）。

該文件還提供了解決配置問題的客戶用例，並討論了與大數據分析以及 Spark 的 AI、ML 和 DL 相關的NetApp儲存產品組合的概述。然後，我們得到來自 Spark 特定用例和NetApp Spark 解決方案組合的測試結果。

客戶挑戰

本節重點在於零售、數位行銷、銀行、離散製造、流程製造、政府和專業服務等資料成長產業中客戶面臨的大數據分析和 AI/ML/DL 挑戰。

不可預測的表現

傳統的 Hadoop 部署通常使用商品硬體。為了提高效能，您必須調整網路、作業系統、Hadoop 叢集、生態系統元件（如 Spark）和硬體。即使您調整每一層，也很難達到所需的效能水平，因為 Hadoop 運行在並非為您的環境的高效能而設計的商用硬體上。

介質和節點故障

即使在正常條件下，商品硬體也容易故障。如果資料節點上的磁碟發生故障，則 Hadoop 主伺服器預設該節點不健康。然後，它透過網路將該節點上的特定資料從副本複製到健康節點。此過程會減慢任何 Hadoop 作業的網路封包速度。當不健康的節點恢復健康狀態時，叢集必須再次複製資料並刪除過度複製的資料。

Hadoop供應商鎖定

Hadoop 經銷商擁有自己的 Hadoop 發行版和版本控制，將客戶鎖定在這些發行版上。然而，許多客戶需要記憶體分析支持，而這種支援不會將客戶綁定到特定的 Hadoop 發行版。他們需要自由地改變分佈，同時仍保留他們的分析能力。

缺乏對多種語言的支持

客戶通常需要除了 MapReduce Java 程式之外的多種語言支援來執行他們的作業。 SQL 和腳本等選項為獲取答案提供了更大的靈活性，為組織和檢索資料提供了更多的選項，以及將資料移至分析框架的更快的方式。

使用難度

一段時間以來，人們一直抱怨 Hadoop 難以使用。儘管 Hadoop 的每個新版本都變得更簡單、更強大，但這種批評仍然存在。 Hadoop 要求您了解 Java 和 MapReduce 程式模式，這對資料庫管理員和具有傳統腳本技能的人員來說是一個挑戰。

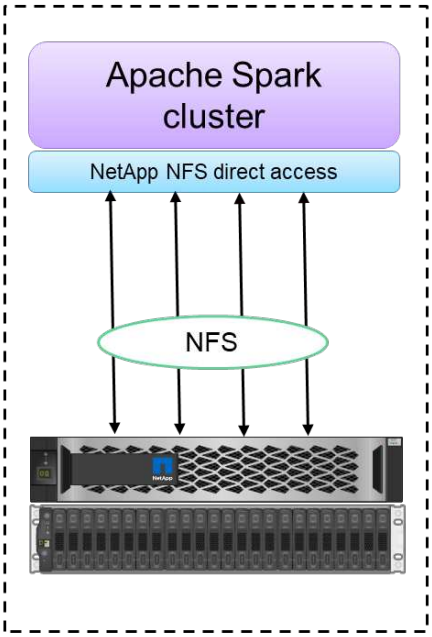
複雜的框架和工具

企業AI團隊面臨多重挑戰。即使擁有專業的數據科學知識，不同部署生態系統和應用程式的工具和框架也可能無法簡單地從一個轉換到另一個。數據科學平台應該與基於 Spark 構建的相應大數據平台無縫集成，易於數據移動、可重複使用的模型、開箱即用的代碼以及支持原型設計、驗證、版本控制、共享、重用和快速將模型部署到生產的最佳實踐的工具。

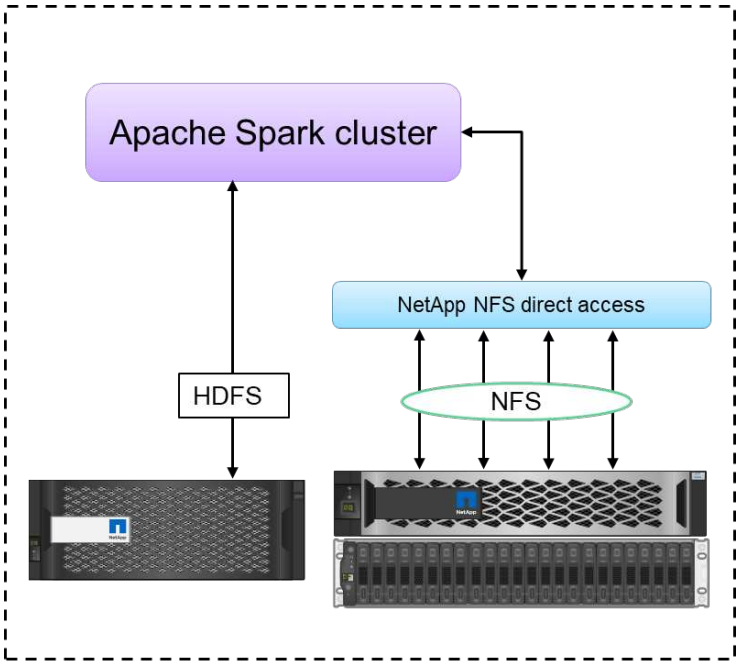
為什麼選擇**NetApp**？

NetApp可以透過以下方式改善您的 Spark 體驗：

- NetApp NFS 直接存取（如下圖所示）允許客戶在其現有或新的 NFSv3 或 NFSv4 資料上執行大數據分析作業，而無需移動或複製資料。它可以防止資料的多次複製，並且無需將資料與來源同步。
- 更有效率的儲存和更少的伺服器複製。例如， NetApp E 系列 Hadoop 解決方案需要兩個而不是三個資料副本，而FAS Hadoop 解決方案需要一個資料來源，但不需要資料複製或副本。 NetApp儲存解決方案還能減少伺服器之間的流量。
- 驅動器和節點故障期間更好的 Hadoop 作業和叢集行為。
- 更好的數據提取性能。



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

例如，在金融和醫療保健領域，資料從一個地方移動到另一個地方必須滿足法律義務，這不是一件容易的事。在這種情況下， NetApp NFS 直接存取會從原始位置分析財務和醫療保健資料。另一個主要優勢是，使用NetApp NFS 直接存取可以透過使用本機 Hadoop 指令簡化 Hadoop 資料的保護，並利用NetApp豐富的資料管理產品組合來實現資料保護工作流程。

NetApp NFS 直接存取為 Hadoop/Spark 叢集提供了兩種部署選項：

- 預設情況下，Hadoop 或 Spark 叢集使用 Hadoop 分散式檔案系統 (HDFS) 進行資料儲存和預設檔案系統。NetApp NFS 直接存取可以用 NFS 儲存取代預設的 HDFS 作為預設檔案系統，從而實現對 NFS 資料的直接分析。

- 在另一個部署選項中， NetApp NFS 直接存取支援在單一 Hadoop 或 Spark 叢集中將 NFS 與 HDFS 一起配置為附加儲存。在這種情況下，客戶可以透過 NFS 匯出共享數據，並從同一集群存取數據以及 HDFS 數據。

使用NetApp NFS 直接存取的主要優勢包括：

- 從目前位置分析數據，從而避免將分析數據移動到 Hadoop 基礎架構（如 HDFS）這一耗時耗能的任務。
- 將副本數量從三個減少到一個。
- 使用戶能夠分離計算和存儲以獨立擴展它們。
- 利用ONTAP豐富的資料管理功能提供企業資料保護。
- 通過 Hortonworks 資料平台認證。
- 支援混合資料分析部署。
- 利用動態多執行緒功能減少備份時間。

看"TR-4657： NetApp混合雲資料解決方案 - 基於客戶使用案例的 Spark 和 Hadoop"用於備份 Hadoop 資料、從雲端到本地的備份和災難復原、對現有 Hadoop 資料進行 DevTest、資料保護和多雲連接以及加速分析工作負載。

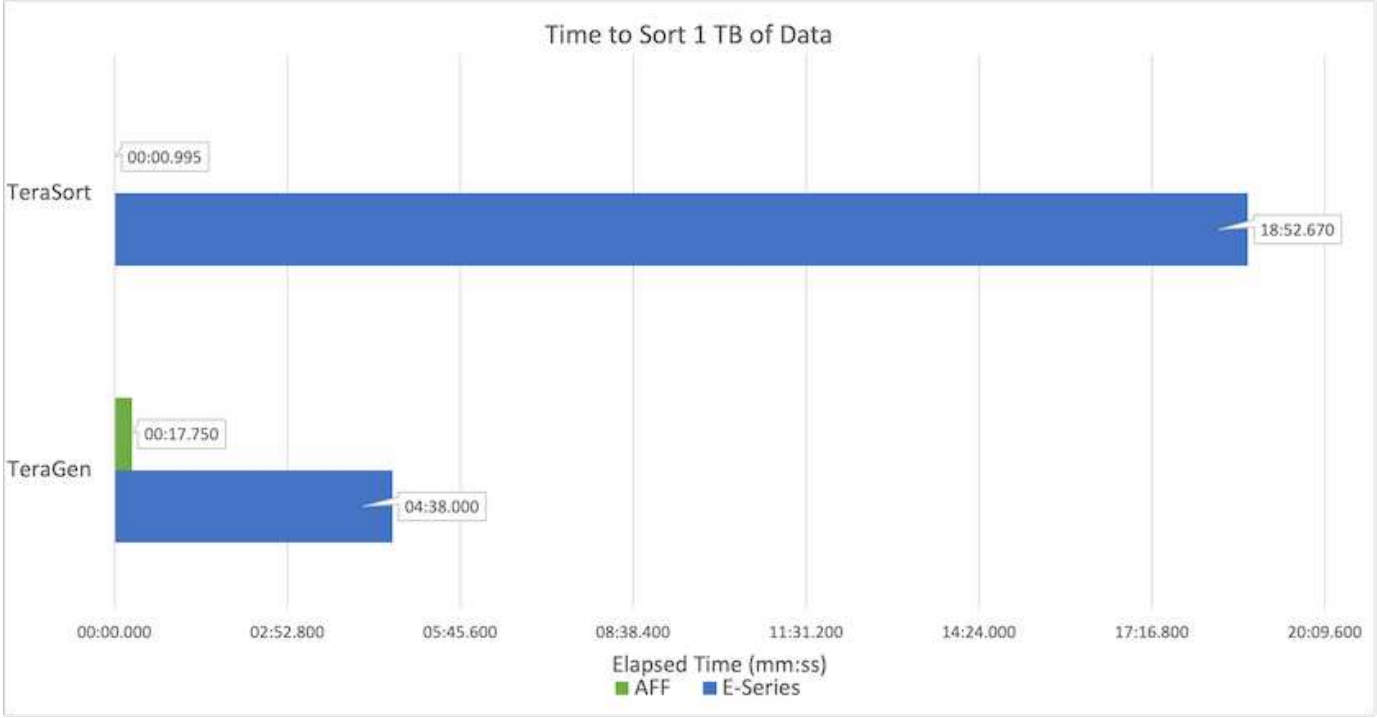
以下部分介紹了對 Spark 客戶來說重要的儲存功能。

儲存分層

透過 Hadoop 儲存分層，您可以根據儲存策略儲存具有不同儲存類型的檔案。儲存類型包括 `hot`， `cold`， `warm`， `all_ssd`， `one_ssd`， 和 `lazy_persist`。

我們在NetApp AFF儲存控制器和具有不同儲存策略的 SSD 和 SAS 磁碟機的 E 系列儲存控制器上對 Hadoop 儲存分層進行了驗證。具有AFF-A800 的 Spark 叢集有四個運算工作節點，而有 E 系列的叢集有八個。這主要是為了比較固態硬碟 (SSD) 與硬碟 (HDD) 的效能。

下圖顯示了NetApp針對 Hadoop SSD 的解決方案的效能。



- 基線 NL-SAS 配置使用八個運算節點和 96 個 NL-SAS 磁碟機。此配置在 4 分 38 秒內產生了 1TB 的資料。看 "TR-3969 NetApp E系列Hadoop解決方案"有關叢集和儲存配置的詳細資訊。
- 使用 TeraGen，SSD 配置產生 1TB 資料的速度比 NL-SAS 配置快 15.66 倍。此外，SSD 配置使用了一半數量的計算節點和一半數量的磁碟機（總共 24 個 SSd 磁碟機）。根據作業完成時間，它幾乎比 NL-SAS 配置快兩倍。
- 使用 TeraSort，SSD 配置對 1TB 資料的排序速度比 NL-SAS 配置快 1138.36 倍。此外，SSD 配置使用了一半數量的計算節點和一半數量的磁碟機（總共 24 個 SSd 磁碟機）。因此，每個驅動器的速度大約比 NL-SAS 配置快三倍。
- 重點是從旋轉磁碟過渡到全快閃記憶體可以提高效能。計算節點的數量不是瓶頸。借助 NetApp 的全快閃存儲，運行時性能可以很好地擴展。
- 使用 NFS，資料在功能上相當於被集中在一起，這可以根據您的工作負載減少計算節點的數量。 Apache Spark 叢集使用者在更改運算節點數量時不必手動重新平衡資料。

效能擴展 - 橫向擴展

當您需要從AFF解決方案中的 Hadoop 叢集取得更多運算能力時，您可以新增具有適當數量儲存控制器的資料節點。 NetApp建議從每個儲存控制器陣列 4 個資料節點開始，然後根據工作負載特性將每個儲存控制器的資料節點數量增加到 8 個。

AFF和FAS非常適合就地分析。根據運算需求，您可以新增節點管理器，且無中斷操作可讓您按需新增儲存控制器而無需停機。我們提供AFF和FAS的豐富功能，例如 NVME 媒體支援、保證效率、資料減少、QOS、預測分析、雲端分層、複製、雲端部署和安全性。為了幫助客戶滿足他們的需求， NetApp提供了檔案系統分析、配額和機上負載平衡等功能，無需額外的授權費用。 NetApp在並發作業數量、更低的延遲、更簡單的操作以及更高的每秒千兆位元組吞吐量方面比我們的競爭對手錶現更好。此外， NetApp Cloud Volumes ONTAP可在三大雲端供應商上運作。

效能擴展 - 擴大規模

當您需要額外的儲存容量時，擴充功能可讓您將磁碟機新增至AFF、 FAS和 E 系列系統。使用Cloud Volumes

ONTAP，將儲存擴展到 PB 層級需要結合兩個因素：將不常用的資料從區塊儲存分層到物件存儲，以及堆疊Cloud Volumes ONTAP授權而無需額外的運算。

多種協定

NetApp系統支援大多數 Hadoop 部署協議，包括 SAS、iSCSI、FCP、InfiniBand 和 NFS。

營運和支援的解決方案

本文檔中所述的 Hadoop 解決方案由NetApp支援。這些解決方案也經過了主要 Hadoop 經銷商的認證。欲了解更多信息，請參閱 "Hortonworks"站點和 Cloudera "認證"和 "夥伴"站點。

目標受眾

分析和數據科學領域涉及 IT 和商業的多個學科：

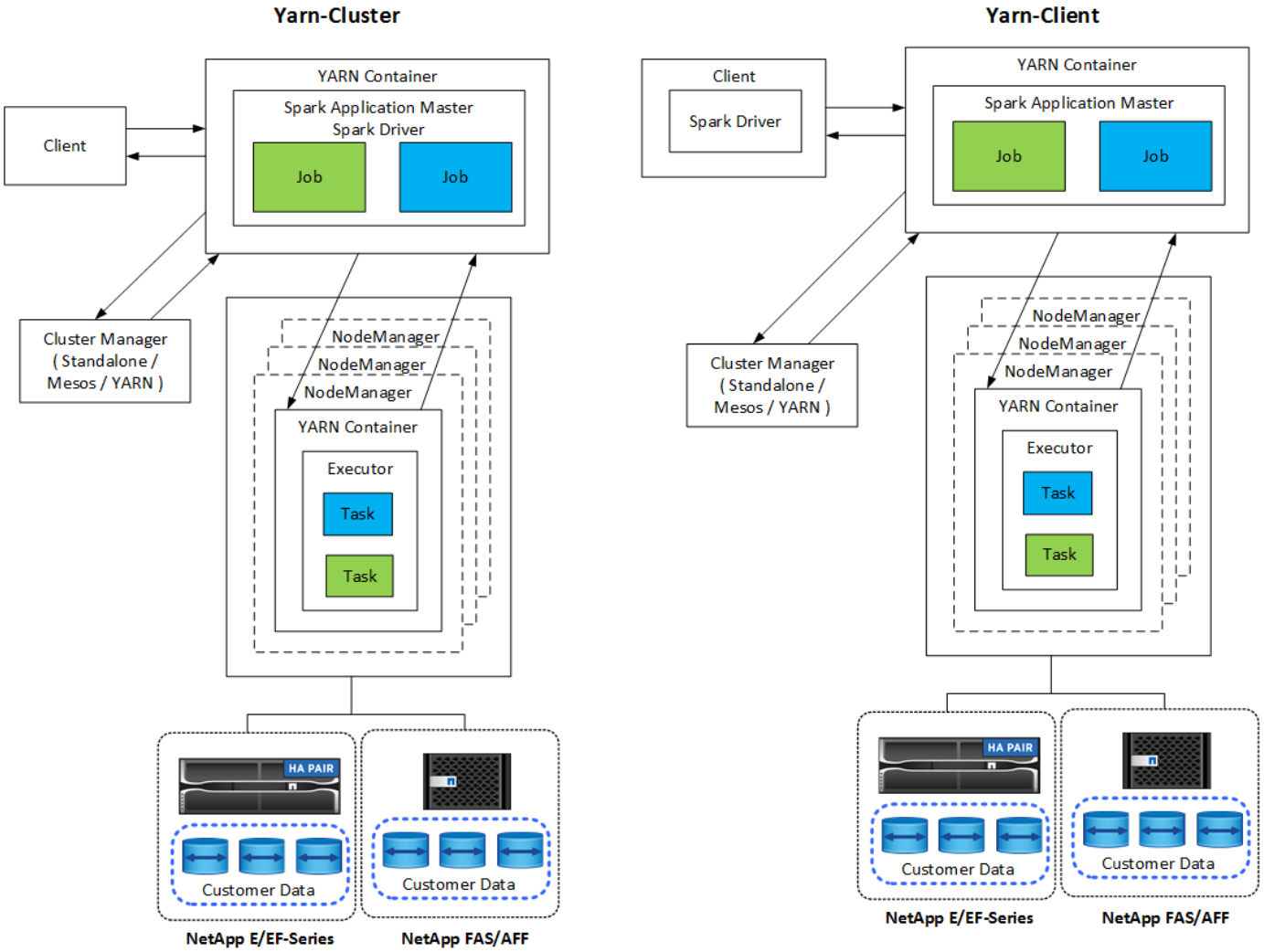
- 資料科學家需要靈活地使用他們選擇的工具和函式庫。
- 資料工程師需要知道資料如何流動以及位於何處。
- DevOps 工程師需要工具將新的 AI 和 ML 應用程式整合到他們的 CI 和 CD 管道中。
- 雲端管理員和架構師必須能夠設定和管理混合雲資源。
- 商業用戶希望能夠存取分析、AI、ML 和 DL 應用程式。

在本技術報告中，我們描述了NetApp AFF、E 系列、 StorageGRID、NFS 直接存取、Apache Spark、Horovod 和 Keras 如何幫助這些角色為企業帶來價值。

解決方案技術

Apache Spark 是一種流行的程式框架，用於編寫可直接與 Hadoop 分散式檔案系統 (HDFS) 協同工作的 Hadoop 應用程式。 Spark 已準備好投入生產，支援串流資料處理，並且比 MapReduce 更快。 Spark 具有可配置的記憶體資料緩存，可實現高效迭代，並且 Spark shell 具有互動性，可用於學習和探索資料。使用 Spark，您可以用 Python、Scala 或 Java 建立應用程式。 Spark 應用程式由一個或多個具有一個或多個任務的作業組成。

每個 Spark 應用程式都有一個 Spark 驅動程式。在 YARN-Client 模式下，驅動程式在客戶端本地運行。在 YARN-Cluster 模式下，驅動程式在應用程式主機上的叢集中運作。在叢集模式下，即使客戶端斷開連接，應用程式仍繼續運作。



有三個集群管理器：

- *獨立。 *此管理器是 Spark 的一部分，可以輕鬆設定叢集。
- Apache Mesos。這是一個通用叢集管理器，也運行 MapReduce 和其他應用程式。
- Hadoop YARN。這是 Hadoop 3 中的資源管理器。

彈性分散式資料集（RDD）是 Spark 的主要元件。 RDD 從叢集記憶體中儲存的資料重新建立遺失和缺少的數據，並儲存來自檔案或以程式設計方式建立的初始資料。 RDD 是從檔案、記憶體中的資料或另一個 RDD 建立的。 Spark 程式設計執行兩種操作：轉換和操作。轉換基於現有 RDD 建立新的 RDD。操作從 RDD 傳回一個值。

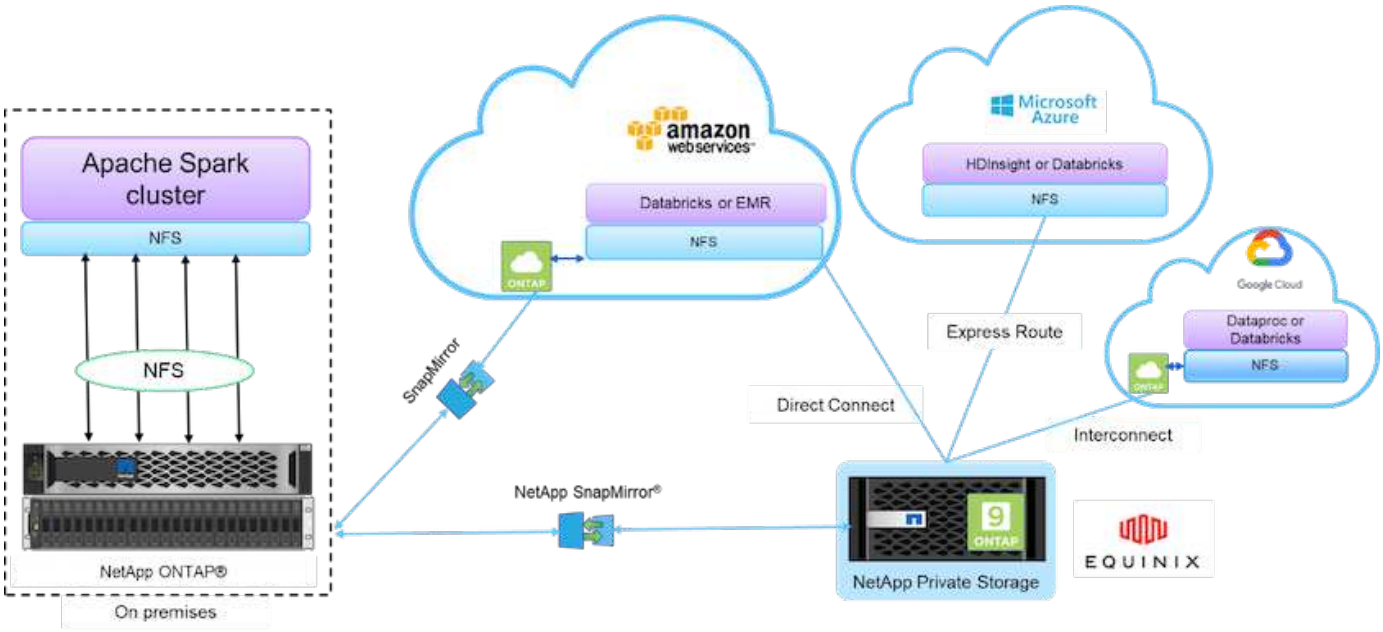
轉換和操作也適用於 Spark 資料集和 DataFrames。資料集是分散式資料集合，它兼具 RDD 的優勢（強型別、使用 lambda 函數）和 Spark SQL 優化執行引擎的優勢。可以從 JVM 物件建立資料集，然後使用功能轉換（map、flatMap、filter 等）進行操作。 DataFrame 是按命名列組織起來的資料集。它在概念上等同於關聯式資料庫中的表或 R/Python 中的資料框。 DataFrames 可以從多種來源構建，例如結構化資料檔案、Hive/HBase 中的表、本地或雲端的外部資料庫或現有的 RDD。

Spark 應用程式包括一個或多個 Spark 作業。作業在執行器中運行任務，執行器在 YARN 容器中運行。每個執行器都在單一容器中運行，並且執行器在應用程式的整個生命週期中都存在。應用程式啟動後，執行器就固定了，YARN 不會調整已指派的容器的大小。執行器可以對記憶體資料同時運行任務。

NetApp Spark 解決方案概述

NetApp有三個儲存產品組合： FAS/ AFF、E 系列和Cloud Volumes ONTAP。我們已經透過 Apache Spark 驗證了適用於 Hadoop 解決方案的AFF和具有ONTAP儲存系統的 E 系列。

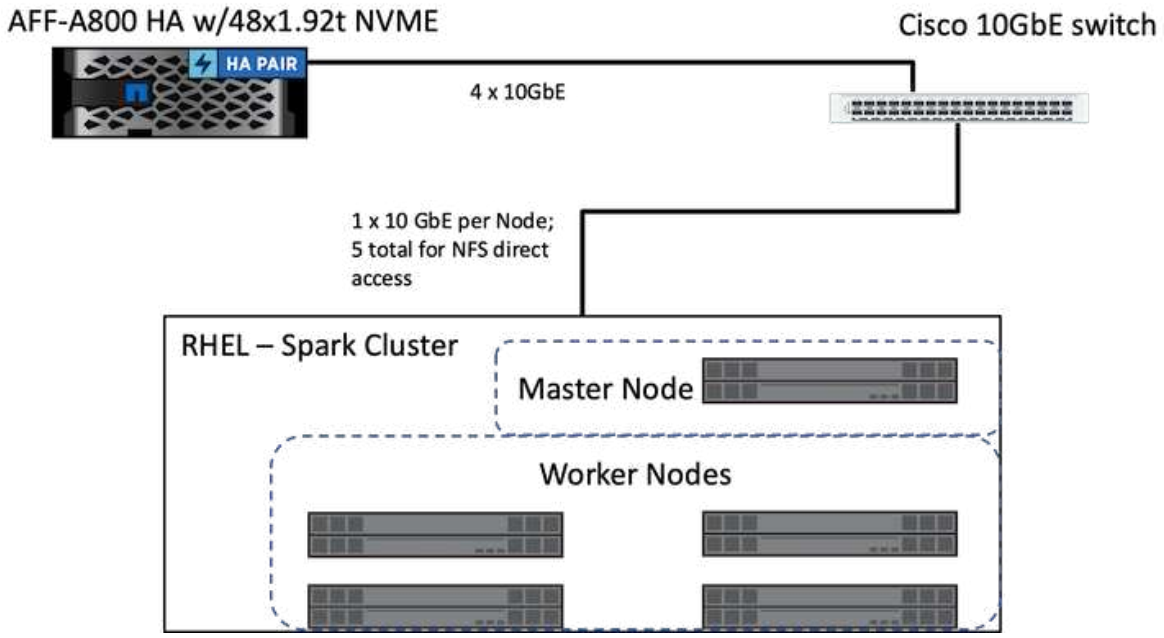
NetApp提供支援的資料結構整合了資料管理服務和應用程式（構建塊），用於資料存取、控制、保護和安全，如下圖所示。



上圖中的構建塊包括：

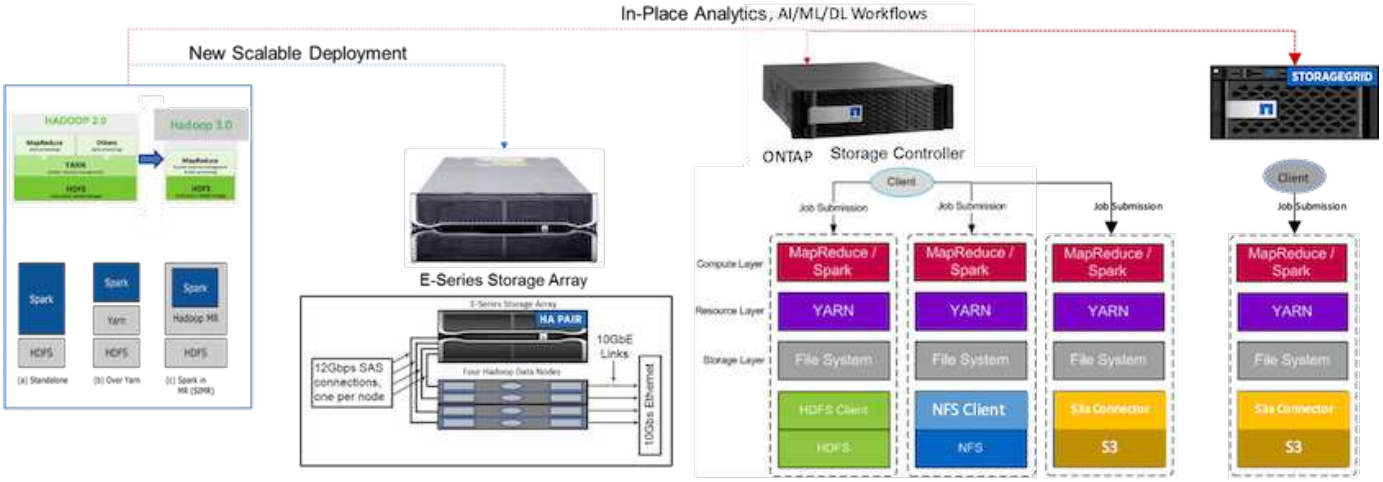
- * NetApp NFS 直接存取。 *為最新的 Hadoop 和 Spark 叢集提供對NetApp NFS 磁碟區的直接訪問，無需額外的軟體或驅動程式要求。
- * NetApp Cloud Volumes ONTAP和Google Cloud NetApp Volumes 。 *基於在 Amazon Web Services (AWS) 或 Microsoft Azure 雲端服務中的Azure NetApp Files (ANF) 中執行的ONTAP的軟體定義連線儲存。
- * NetApp SnapMirror技術。 *在本機和ONTAP Cloud 或 NPS 實例之間提供資料保護功能。
- *雲端服務提供者。 *這些供應商包括 AWS、Microsoft Azure、Google Cloud 和 IBM Cloud。
- 平台即服務。基於雲端的分析服務，例如 AWS 中的 Amazon Elastic MapReduce (EMR) 和 Databricks 以及 Microsoft Azure HDInsight 和 Azure Databricks。

下圖描述了採用NetApp儲存的 Spark 解決方案。

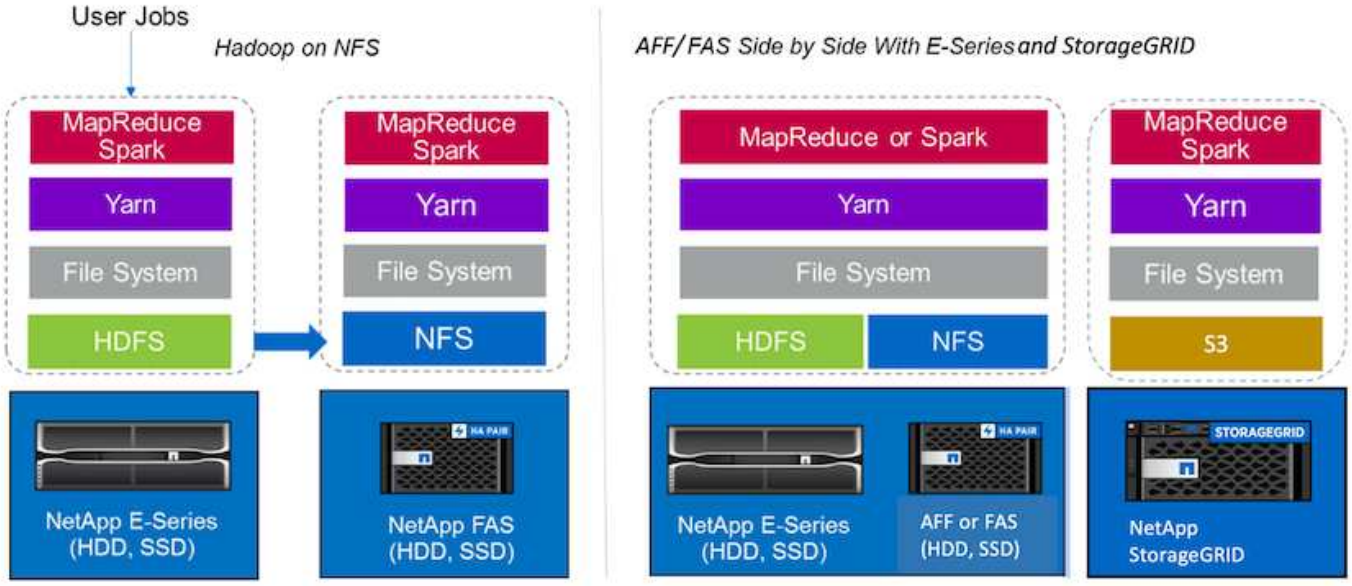


ONTAP Spark 解決方案使用NetApp NFS 直接存取協定進行就地分析以及透過存取現有生產資料來實現 AI、ML 和 DL 工作流程。 Hadoop 節點可用的生產資料被匯出以執行就地分析和 AI、ML 和 DL 作業。您可以使用NetApp NFS 直接存取或不使用 NetApp NFS 直接存取來存取 Hadoop 節點中要處理的資料。在 Spark 中，使用獨立或 `yarn`叢集管理器，您可以使用設定 `NFS 卷 `\file://<target_volume>`。我們用不同的資料集驗證了三個用例。這些驗證的詳細資訊在「測試結果」部分中介紹。 （外部參照）

下圖描述了NetApp Apache Spark/Hadoop 儲存定位。



我們確定了 E 系列 Spark 解決方案、 AFF/ FAS ONTAP Spark 解決方案和StorageGRID Spark 解決方案的獨特功能，並進行了詳細的驗證和測試。根據我們的觀察， NetApp建議對於綠地安裝和新的可擴展部署使用 E 系列解決方案，對於使用現有 NFS 資料的就地分析、AI、ML 和 DL 工作負載使用AFF/ FAS解決方案，對於需要物件儲存時的 AI、ML、DL 和現代資料分析使用StorageGRID 。



資料湖是原生形式的大型資料集的儲存庫，可用於分析、AI、ML 和 DL 作業。我們為 E 系列、 AFF/ FAS 和StorageGRID SG6060 Spark 解決方案建置了一個資料湖儲存庫。 E 系列系統提供對 Hadoop Spark 叢集的 HDFS 訪問，而現有生產資料則透過 NFS 直接存取協定存取 Hadoop 叢集。對於駐留在物件儲存中的資料集，NetApp StorageGRID提供 S3 和 S3a 安全存取。

用例摘要

本頁描述了可以使用該解決方案的不同領域。

串流資料

Apache Spark 可以處理串流數據，用於串流提取、轉換和載入 (ETL) 過程；數據豐富；觸發事件檢測；以及複雜的會話分析：

- 流式 ETL。 *資料在被推送到資料儲存之前會不斷清理和匯總。 Netflix 使用 Kafka 和 Spark 串流建立即時線上電影推薦和資料監控解決方案，每天可以處理來自不同資料來源的數十億個事件。然而，用於批次處理的傳統 ETL 的處理方式有所不同。首先讀取該數據，然後將其轉換為資料庫格式，然後寫入資料庫。
- *數據豐富。 * Spark Streaming 使用靜態資料豐富即時數據，以實現更即時的資料分析。例如，線上廣告主可以根據客戶行為資訊投放個人化、有針對性的廣告。
- *觸發事件檢測。 * Spark 串流可讓您偵測並快速回應可能表示有嚴重問題的異常行為。例如，金融機構使用觸發器來偵測和阻止詐欺交易，醫院使用觸發器來檢測患者生命徵像中檢測到的危險健康變化。
- *複雜的會話分析。 * Spark 流會收集使用者登入網站或應用程式後的活動等事件，然後進行分組和分析。例如，Netflix 使用此功能提供即時電影推薦。

有關串流資料配置、Confluent Kafka 驗證和效能測試的更多內容，請參閱"TR-4912： NetApp Confluent Kafka 分層儲存最佳實務指南"。

機器學習

Spark 整合框架可協助您使用機器學習程式庫 (MLlib) 對資料集執行重複查詢。 MLlib 用於聚類、分類和降維等領域，用於一些常見的大數據功能，例如預測智慧、用於行銷目的的客戶細分和情感分析。 MLlib 用於網路安全，對封包進行即時檢查，以發現惡意活動的跡象。它可以幫助安全提供者了解新的威脅並領先駭客，同時即時保

護他們的客戶。

深度學習

TensorFlow 是業界流行的深度學習框架。 TensorFlow支援在CPU或GPU叢集上進行分散式訓練。這種分散式訓練允許使用者在具有大量深層的資料上運行它。

直到最近，如果我們想將 TensorFlow 與 Apache Spark 一起使用，我們需要在 PySpark 中為 TensorFlow 執行所有必要的 ETL，然後將資料寫入中間儲存。然後，該資料將載入到 TensorFlow 叢集上，用於實際的訓練流程。此工作流程要求使用者維護兩個不同的集群，一個用於 ETL，一個用於 TensorFlow 的分散式訓練。運作和維護多個叢集通常很繁瑣且耗時。

早期 Spark 版本中的 DataFrames 和 RDD 不太適合深度學習，因為隨機存取受到限制。在氫化計畫的 Spark 3.0 中，加入了對深度學習框架的原生支援。這種方法允許在 Spark 叢集上進行非基於 MapReduce 的調度。

互動式分析

Apache Spark 的速度夠快，可以使用 Spark 以外的開發語言（包括 SQL、R 和 Python）執行探索性查詢而無需採樣。 Spark 使用視覺化工具來處理複雜資料並以互動方式進行視覺化。具有結構化流的 Spark 對網路分析中的即時資料執行互動式查詢，使您能夠對網路訪客的當前會話執行互動式查詢。

推薦系統

多年來，隨著企業和消費者對線上購物、線上娛樂和許多其他行業的巨大變化做出了反應，推薦系統為我們的生活帶來了巨大的變化。事實上，這些系統是人工智慧在生產中最明顯的成功案例之一。在許多實際用例中，推薦系統與對話式 AI 或與 NLP 後端互動的聊天機器人結合，以獲取相關資訊並產生有用的推論。

如今，許多零售商正在採用更新的商業模式，例如線上購買、店內取貨、路邊取貨、自助結帳、掃描即走等等。這些模式在新冠疫情期間特別突出，因為它們讓消費者的購物更加安全、更加便利。人工智慧對於這些日益增長的數位趨勢至關重要，這些趨勢受到消費者行為的影響，反之亦然。為了滿足消費者日益增長的需求、增強客戶體驗、提高營運效率和增加收入， NetApp可協助其企業客戶和企業使用機器學習和深度學習演算法來設計更快、更準確的推薦系統。

有幾種流行的技術用於提供推薦，包括協同過濾、基於內容的系統、深度學習推薦模型 (DLRM) 和混合技術。客戶之前利用 PySpark 實現協同過濾來創建推薦系統。 Spark MLlib 實作了用於協同過濾的交替最小二乘法 (ALS)，這是 DLRM 興起之前企業中非常流行的演算法。

自然語言處理

對話式人工智慧是透過自然語言處理 (NLP) 實現的，它是幫助電腦與人類溝通的人工智慧的一個分支。 NLP 在每個垂直行業和許多用例中都很普遍，從智慧助理和聊天機器人到Google搜尋和預測文字。根據 "Gartner"預測到2022年，70%的人將每天與對話式人工智慧平台互動。為了實現人與機器之間的高品質對話，反應必須快速、聰明且聽起來自然。

客戶需要大量資料來處理和訓練他們的 NLP 和自動語音辨識 (ASR) 模型。他們還需要在邊緣、核心和雲端移動數據，並且需要在幾毫秒內進行推理的能力，以與人類建立自然的交流。 NetApp AI 和 Apache Spark 是運算、儲存、資料處理、模型訓練、微調和部署的理想組合。

情緒分析是 NLP 中的一個研究領域，它從文本中提取正面、負面或中性情緒。情緒分析有多種用例，從確定支援中心員工與呼叫者對話的表現到提供適當的自動聊天機器人回應。它也被用來根據公司代表和季度收益電話會議上的聽眾之間的互動來預測公司的股價。此外，情緒分析可用於確定客戶對品牌提供的產品、服務或支援的看法。

我們使用了 "Spark NLP"來自的圖書館 "約翰·斯諾實驗室"載入預訓練管道和 Transformer (BERT) 模型的雙向編碼器表示，包括 "財經新聞情緒"和 "FinBERT"，大規模執行標記化、命名實體識別、模型訓練、擬合和情緒分析。 Spark NLP 是唯一正在生產的開源 NLP 庫，它提供最先進的轉換器，例如 BERT、ALBERT、ELECTRA、XLNet、DistilBERT、RoBERTa、DeBERTa、XLM-RoBERTa、Longformer、ELMO、Universal Sentence Encoder、Google T5、MarianMT 和 GPT2。該程式庫不僅適用於 Python 和 R，還可以透過原生擴充 Apache Spark 在 JVM 生態系統（Java、Scala 和 Kotlin）中大規模運行。

主要的 AI、ML 和 DL 用例和架構

主要的 AI、ML 和 DL 用例和方法可分為以下幾部分：

Spark NLP 管道和 TensorFlow 分散式推理

以下列表包含資料科學界在不同發展層次下採用的最受歡迎的開源 NLP 庫：

- "自然語言工具包（NLTK）"。所有 NLP 技術的完整工具包。它自 21 世紀初以來一直得到維護。
- "文字區塊"。基於 NLTK 和 Pattern 建立的易於使用的 NLP 工具 Python API。
- "史丹佛核心 NLP"。史丹佛 NLP 小組開發的 Java NLP 服務和套件。
- "Gensim"。人類主題建模最初是捷克數位數學圖書館計畫的 Python 腳本集合。
- "SpaCy"。使用 Python 和 Cython 實現端對端工業 NLP 工作流程，並為 Transformer 提供 GPU 加速。
- "快文"。Facebook 的 AI 研究 (FAIR) 實驗室創建的免費、輕量級、開源 NLP 庫，用於學習單字嵌入和句子分類。

Spark NLP 是針對所有 NLP 任務和要求的單一、統一的解決方案，可為實際生產用例提供可擴展、高效能和高精度的 NLP 軟體。它利用遷移學習並在研究和跨行業中實施最新的最先進的演算法和模型。由於 Spark 缺乏對上述函式庫的全面支持，Spark NLP 建立在 "Spark 機器學習"利用 Spark 的通用記憶體分散式資料處理引擎作為關鍵任務生產工作流程的企業級 NLP 庫。它的註釋器利用基於規則的演算法、機器學習和 TensorFlow 來支援深度學習的實作。這涵蓋了常見的 NLP 任務，包括但不限於標記化、詞形還原、詞幹提取、詞性標註、命名實體識別、拼字檢查和情緒分析。

來自 Transformer 的雙向編碼器表示 (BERT) 是一種基於 Transformer 的 NLP 機器學習技術。它推廣了預訓練和微調的概念。 BERT 中的 Transformer 架構源自機器翻譯，它比基於循環神經網路 (RNN) 的語言模型更好地模擬長期依賴關係。它還引入了掩蔽語言建模 (MLM) 任務，其中隨機 15% 的所有標記被掩蔽，並且模型對其進行預測，從而實現真正的雙向性。

由於該領域的專業語言和缺乏標記數據，金融情緒分析具有挑戰性。 FinBERT 是一種基於預訓練 BERT 的語言模型，已在以下領域進行了調整： "路透社 TRC2"，一個金融語料庫，並使用標記資料進行微調（ "金融短語庫") 用於金融情緒分類。研究人員從包含金融術語的新聞文章中提取了 4,500 個句子。然後，16位具有金融背景的專家和碩士生將這些句子標記為肯定、中性和否定。我們建立了一個端到端的 Spark 工作流程，使用 FinBERT 和其他兩個預先訓練的流程來分析 2016 年至 2020 年納斯達克十大公司收益電話會議記錄的情緒， "解釋文檔 DL"）來自 Spark NLP。

Spark NLP 的底層深度學習引擎是 TensorFlow，這是一個端到端的開源機器學習平台，可以輕鬆建立模型、在任何地方進行強大的 ML 生產以及進行強大的研究實驗。因此，在 Spark 中執行管道時 `yarn cluster`模式，我們本質上是在運行分散式 TensorFlow，資料和模型在一個主節點和多個工作節點上並行化，並在叢集上安裝網路附加儲存。

Horovod分散式訓練

與 MapReduce 相關的效能的核心 Hadoop 驗證是使用 TeraGen、TeraSort、TeraValidate 和 DFSIO（讀寫）執行的。 TeraGen 和 TeraSort 驗證結果如下 "NetApp E系列Hadoop解決方案"以及AFF的「儲存分層」部分。

根據客戶要求，我們認為使用 Spark 進行分散式訓練是各種用例中最重要的用例之一。在本文檔中，我們使用了 "Spark 上的 Hovorod"使用NetApp All Flash FAS (AFF) 儲存控制器、 Azure NetApp Files和StorageGRID來驗證 Spark 與NetApp本地端、雲端原生和混合雲端解決方案的效能。

Horovod on Spark 套件為 Horovod 提供了一個便捷的包裝器，使得在 Spark 叢集中運行分散式訓練工作負載變得簡單，從而實現了緊密的模型設計循環，其中資料處理、模型訓練和模型評估都在訓練和推理資料所在的 Spark 中完成。

有兩個用於在 Spark 上運行 Horovod 的 API：高級 Estimator API 和低階 Run API。儘管兩者都使用相同的底層機制在 Spark 執行器上啟動 Horovod，但 Estimator API 抽象化了資料處理、模型訓練循環、模型檢查點、指標收集和分散式訓練。我們使用 Horovod Spark Estimators、TensorFlow 和 Keras 進行端到端資料準備和分散式訓練工作流程，基於 "Kaggle Rossmann 商店銷售"競賽。

腳本 `keras_spark_horovod_rossmann_estimator.py`可以在以下部分找到"每個主要用例的 Python 腳本。"它包含三個部分：

- 第一部分對 Kaggle 提供並由社群收集的一組初始 CSV 檔案執行各種資料預處理步驟。輸入資料被分成一個訓練集， `Validation`子集和測試資料集。
- 第二部分定義了一個具有對數 S 型激活函數和 Adam 優化器的 Keras 深度神經網路 (DNN) 模型，並使用 Spark 上的 Horovod 對模型進行分散式訓練。
- 第三部分使用最小化驗證集總體平均絕對誤差的最佳模型對測試資料集進行預測。然後創建一個輸出 CSV 檔案。

請參閱"機器學習"用於各種運行時比較結果。

使用 **Keras** 進行多任務深度學習以進行 **CTR** 預測

隨著機器學習平台和應用的最新進展，人們將大量注意力放在了大規模學習上。點擊率（CTR）定義為每百次線上廣告展示的平均點擊次數（以百分比表示）。它被廣泛採用為各行業垂直領域和用例的關鍵指標，包括數位行銷、零售、電子商務和服務提供者。有關 CTR 和分佈式訓練表現結果的應用的更多詳細信息，請參閱"CTR預測表現的深度學習模型"部分。

在本技術報告中，我們使用了 "Criteo Terabyte 點選日誌資料集"（參見 TR-4904）用於多工作者分散式深度學習，使用 Keras 建立具有深度和交叉網路 (DCN) 模型的 Spark 工作流程，並將其對數損失誤差函數方面的效能與基線 Spark ML 邏輯回歸模型進行比較。 DCN 有效地捕捉有界度的有效特徵交互，學習高度非線性交互，不需要手動特徵工程或窮舉搜索，且計算成本低。

網路規模推薦系統的資料大多是離散的和分類的，導致特徵空間龐大且稀疏，這對於特徵探索來說是一個挑戰。這使得大多數大型系統僅限於邏輯迴歸等線性模型。然而，識別經常預測的特徵並同時探索看不見的或罕見的交叉特徵是做出良好預測的關鍵。線性模型簡單、可解釋、易於擴展，但其表達能力有限。

另一方面，交叉特徵已被證明對提高模型的表現力具有重要意義。不幸的是，通常需要手動特徵工程或詳盡搜尋來識別這些特徵。推廣到看不見的特徵互動通常很困難。使用像 DCN 這樣的交叉神經網路可以透過以自動方式明確應用特徵交叉來避免特定於任務的特徵工程。交叉網路由多層組成，其中最高程度的交互作用可由層深度決定。每一層都會在現有交互的基礎上產生更高階的交互，並保留前幾層的交互。

深度神經網路 (DNN) 有望捕捉跨特徵的非常複雜的交互作用。然而，與 DCN 相比，它需要的參數幾乎多一個

數量級，無法明確地形成交叉特徵，並且可能無法有效地學習某些類型的特徵交叉。交叉網路記憶體效率高且易於實現。聯合訓練交叉和 DNN 元件可以有效地捕捉預測特徵交互作用並在 Criteo CTR 資料集上提供最先進的效能。

DCN 模型從嵌入和堆疊層開始，然後並行連接交叉網路和深度網路。接下來是最終的組合層，它將兩個網路的輸出組合在一起。您的輸入資料可以是具有稀疏和密集特徵的向量。在 Spark 中，庫包含類型 `SparseVector`。因此，使用者區分兩者並在呼叫各自的函數和方法時要小心，這一點很重要。在 CTR 預測等網路規模推薦系統中，輸入大多是分類特徵，例如 `'country=usa'`。這些特徵通常被編碼為獨熱向量，例如，`'[0,1,0, …]'` 。獨熱編碼（OHE） `SparseVector`在處理詞彙不斷變化和增長的真實世界資料集時很有用。我們修改了範例 "深度點擊率"處理大型詞彙表，在 DCN 的嵌入和堆疊層中建立嵌入向量。

這 "Criteo 展示廣告資料集"預測廣告點擊率。它有 13 個整數特徵和 26 個分類特徵，其中每個類別都有很高的基數。對於該資料集，由於輸入規模較大，對數損失 0.001 的改進實際上具有顯著意義。對於龐大的用戶群，預測準確度的微小提升都可能帶來公司收入的大幅增加。該資料集包含 7 天內 11GB 的使用者日誌，相當於約 4,100 萬筆記錄。我們使用了 Spark `dataFrame.randomSplit()function`隨機分割資料用於訓練（80%）、交叉驗證（10%），剩餘 10% 用於測試。

DCN 是使用 Keras 在 TensorFlow 上實現的。使用DCN實現模型訓練過程主要有四個部分：

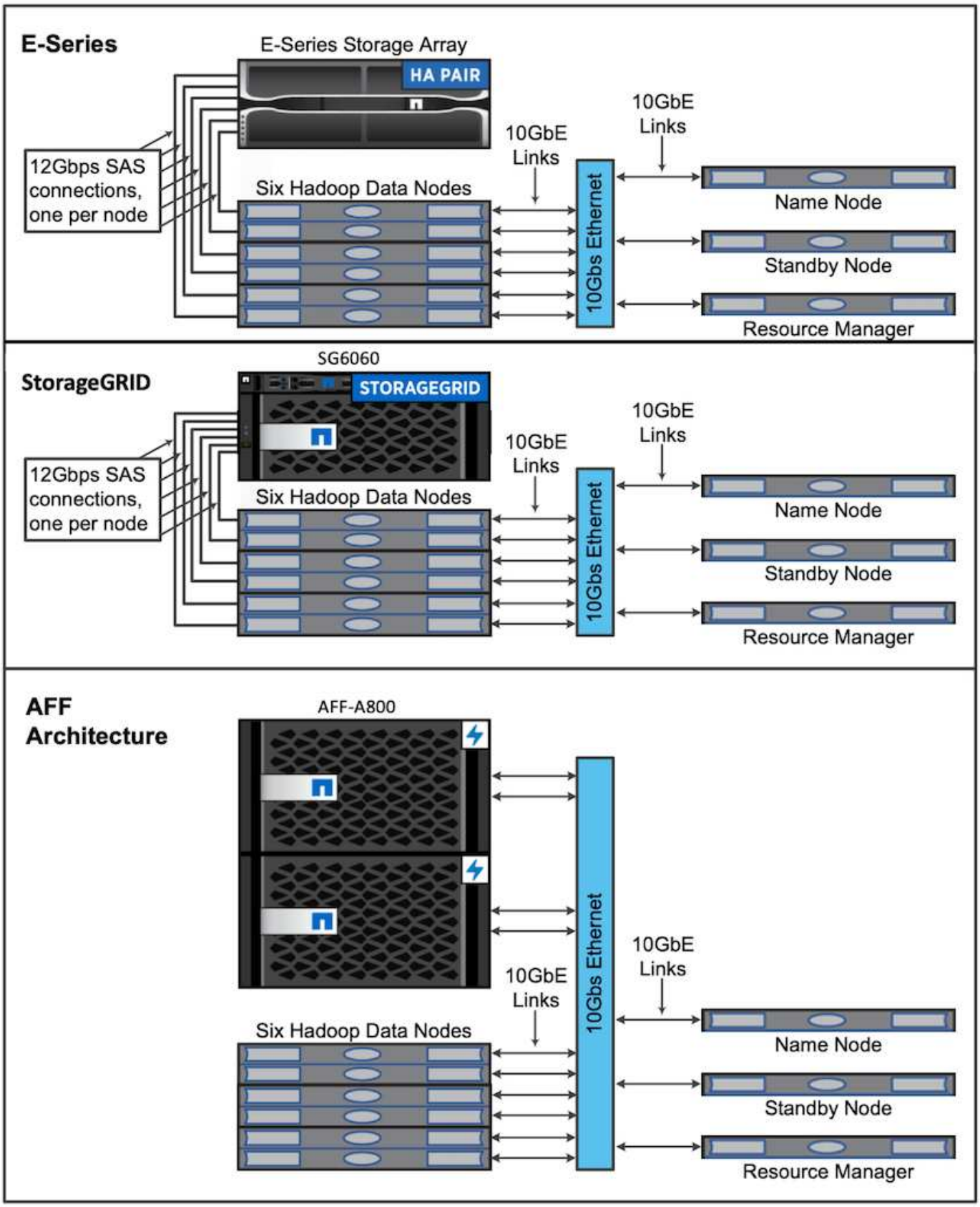
- *資料處理和嵌入。 *透過應用對數變換對實值特徵進行規範化。對於分類特徵，我們將特徵嵌入到維度為 6×(類別基數)1/4 的密集向量中。連接所有嵌入將產生一個維度為 1026 的向量。
- *最佳化.*我們利用 Adam 優化器進行了小批量隨機優化。批次大小設定為 512。將深度網路進行批量歸一化，梯度裁剪範數設為100。
- *正則化。 *我們採用了早期停止的方法，因為 L2 正規化或 dropout 被發現無效。
- 超參數。我們報告基於對隱藏層數量、隱藏層大小、初始學習率和交叉層數量的網格搜尋的結果。隱藏層的數量範圍為 2 至 5，隱藏層大小範圍為 32 至 1024。對於DCN，交叉層的數量為1至6。初始學習率從 0.0001 調整到 0.001，增量為 0.0001。所有實驗均在訓練步驟 150,000 時提前停止，超過該步驟後就會開始出現過度擬合。

除了 DCN 之外，我們還測試了其他流行的深度學習模型來進行 CTR 預估，包括 "DeepFM"， "自動輸入" ， 和 "DCN v2"。

用於驗證的架構

為了進行此驗證，我們使用了四個工作節點和一個主節點以及一個AFF-A800 HA 對。所有群集成員都透過 10GbE 網路交換器連接。

為了驗證NetApp Spark 解決方案，我們使用了三種不同的儲存控制器：E5760、E5724 和AFF-A800。 E系列儲存控制器透過12Gbps SAS連線連接到五個資料節點。 AFF HA 對儲存控制器透過 10GbE 連線向 Hadoop 工作節點提供匯出的 NFS 磁碟區。 Hadoop 叢集成員透過 E 系列、 AFF和StorageGRID Hadoop 解決方案中的 10GbE 連線進行連線。



測試結果

我們使用 TeraGen 基準測試工具中的 TeraSort 和 TeraValidate 腳本來測量 E5760

、E5724 和AFF-A800 配置的 Spark 效能驗證。此外，還測試了三個主要用例：Spark NLP 管道和 TensorFlow 分散式訓練、Horovod 分散式訓練以及使用 Keras 進行 DeepFM CTR 預測的多工深度學習。

對於 E 系列和StorageGRID驗證，我們使用了 Hadoop 複製因子 2。對於AFF驗證，我們只使用一個資料來源。

下表列出了 Spark 效能驗證的硬體配置。

類型	**Hadoop** 工作節點	驅動器類型	每個節點的驅動器	儲存控制器
SG6060	4	SAS	12	單一高可用性 (HA) 對
E5760	4	SAS	60	單一 HA 對
E5724	4	SAS	24	單一 HA 對
AFF800	4	固態硬碟	6	單一 HA 對

下表列出了軟體要求。

軟體	版本
紅帽企業版	7.9
OpenJDK 運作環境	1.8.0
OpenJDK 64 位元伺服器虛擬機	25.302
Git	2.24.1
GCC/G++	11.2.1
火花	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
喀拉拉	2.9.0
霍羅沃德	0.24.3

金融情緒分析

我們發表了"TR-4910：利用NetApp AI 進行客戶溝通情緒分析"，其中使用 "NetApp DataOps 工具包"、 AFF儲存和NVIDIA DGX 系統。此管道利用 DataOps Toolkit 執行批量音訊訊號處理、自動語音辨識 (ASR)、遷移學習和情緒分析， "NVIDIA Riva SDK" ，以及 "道框架"。將情緒分析用例擴展到金融服務業，我們建立了 SparkNLP 工作流程，為各種 NLP 任務（例如命名實體識別）加載了三個 BERT 模型，並獲得了納斯達克十大公司季度收益電話會議的句子級情緒。

以下腳本 `sentiment_analysis_spark. py`使用 FinBERT 模型處理 HDFS 中的轉錄本，並產生正面、中性和負面情緒計數，如下表所示：

```
-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
```

下表列出了 2016 年至 2020 年納斯達克十大公司的收益電話會議句子級情緒分析。

情緒計數和百分比	全部 **10** 家公司	蘋果	**AMD**	亞馬遜	思科	**Google**	國際貿易中心	微軟	**NVDA**
正計數	7447	1567	743	290	682	826	824	904	417
中立計數	64067	6856	7596	5086	6650	5914	6099	5715	6189
負數	1787	253	213	84	189	97	282	202	89
未分類的計數	196	0	0	76	0	0	0	1	0
（總數）	73497	8676	8552	5536	7521	6837	7205	6822	6695

從百分比來看，執行長和財務長所說的大多數句子都是事實，因此帶有中立的情緒。在收益電話會議期間，分析師提出的問題可能會傳達正面或負面的情緒。值得進一步定量研究負面或正面情緒如何影響交易當天或隔天的股票價格。

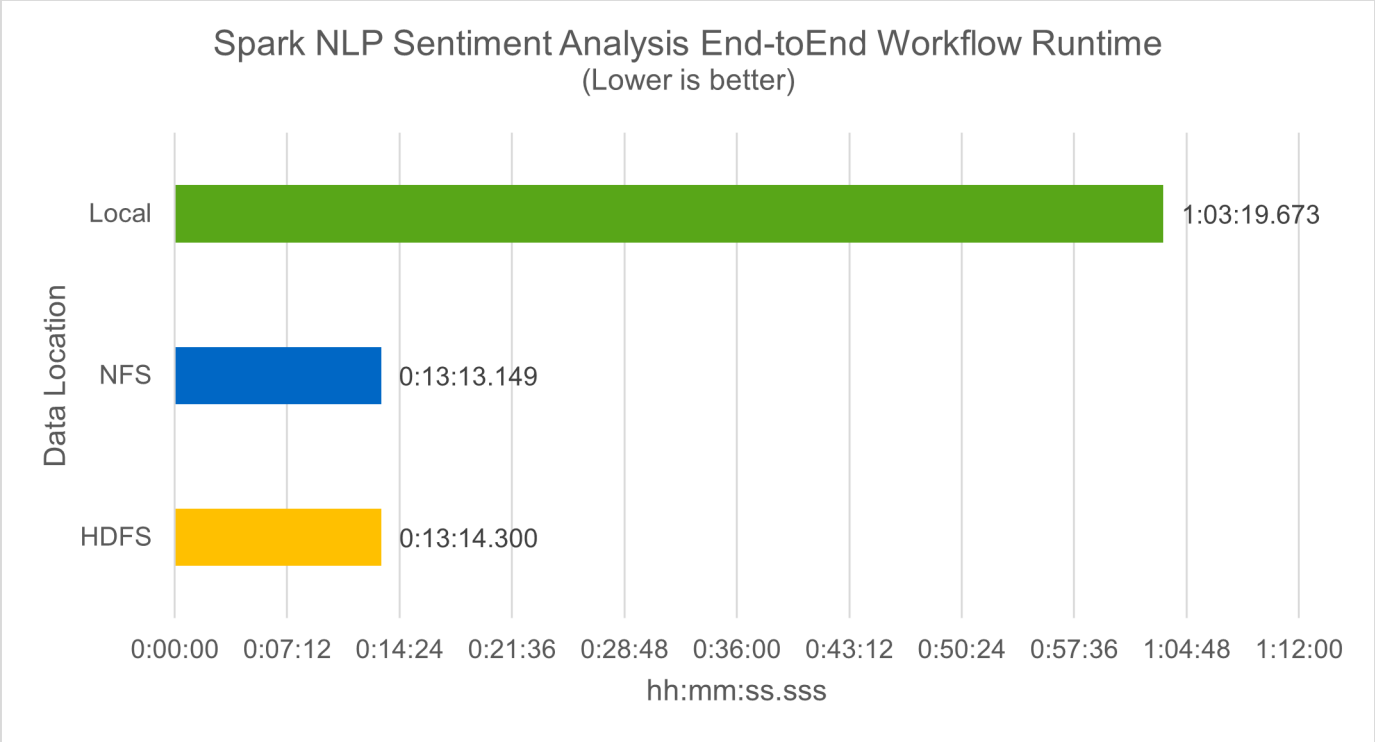
下表列出了納斯達克十大公司的句子級情感分析，以百分比表示。

情緒百分比	全部 **10** 家公司	蘋果	**AMD**	亞馬遜	思科	**Google**	國際貿易中心	微軟	**NVDA**
積極的	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
中性的	87.17%	79.02%	88.82%	91.87%	88.42%	86.50%	84.65%	83.77%	92.44%
消極的	2.43%	2.92%	2.49%	1.52%	2.51%	1.42%	3.91%	2.96%	1.33%
未分類	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

在工作流程運行時方面，我們看到了顯著的 4.78 倍改進 `local`模式到 HDFS 中的分散式環境，並透過利用 NFS 進一步提高 0.14%。

```
-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
```

如下圖所示，資料和模型並行提高了資料處理和分散式 TensorFlow 模型推理的速度。 NFS 中的資料位置產生了稍微更好的運行時間，因為工作流程瓶頸是預訓練模型的下載。如果我們增加成績單資料集的大小，NFS 的優勢就更加明顯。



Horovod 表現的分散式訓練

以下命令使用單一 `master`具有 `160` 個執行器的節點，每個執行器都有一個核心。執行器記憶體限制為 `5GB`，以避免記憶體不足錯誤。請參閱"針對每個主要用例的 `Python` 腳本"有關數據處理、模型訓練和模型準確率計算的更多詳細信息，請參閱 `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
```

經過 10 個訓練週期後，最終的運行時間如下：

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

處理輸入資料、訓練 DNN 模型、計算準確度以及產生 TensorFlow 檢查點和預測結果的 CSV 檔案花費了超過 43 分鐘。我們將訓練週期數限制為 10，在實踐中通常設定為 100，以確保令人滿意的模型準確率。訓練時間通常與訓練次數呈線性關係。

接下來，我們使用叢集中可用的四個工作節點，並在 `yarn`HDFS 中的資料模式：

```
(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
```

最終的運轉時間改進如下：

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

借助 Horovod 模型和 Spark 中的資料並行性，我們看到運行速度提高了 5.29 倍 `yarn`相對 `local`具有十個訓練階段的模式。下圖中圖例顯示了這一點 `HDFS`和 `Local`。如果可用的話，可以使用 GPU 進一步加速

底層 TensorFlow DNN 模型訓練。我們計劃進行此項測試並在未來的技術報告中發布結果。

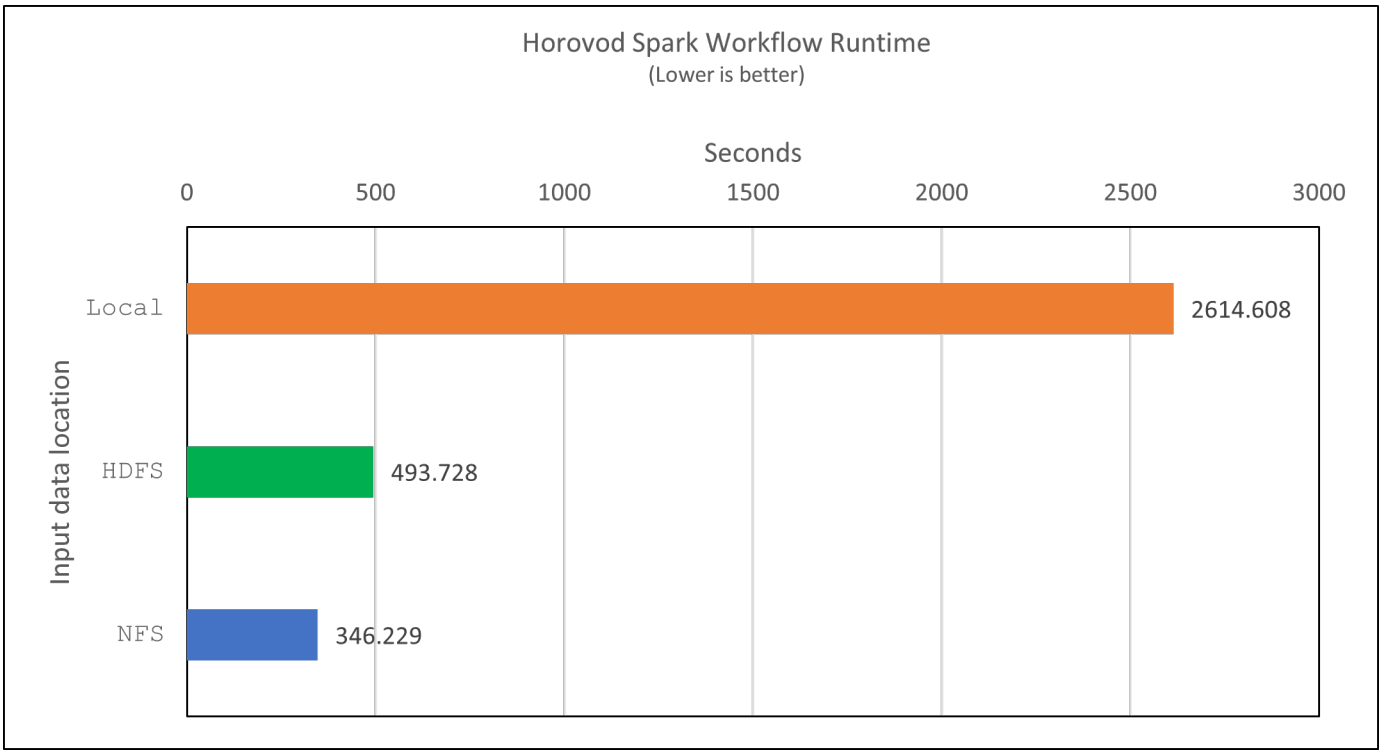
我們的下一個測試比較了 NFS 和 HDFS 中的輸入資料的運行時間。 AFF A800上的 NFS 磁碟區已安裝在 `/sparkdemo/horovod`分佈於 Spark 叢集的五個節點（一個主節點，四個工作節點）上。我們運行了與先前的測試類似的命令， `--data- dir`參數現在指向 NFS 掛載：

```
(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
```

使用 NFS 的運行結果如下：

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

速度又提高了 1.43 倍，如下圖所示。因此，透過將NetApp全快閃儲存連接到其集群，客戶可以享受 Horovod Spark 工作流程的快速資料傳輸和分發優勢，與在單一節點上運行相比，可實現 7.55 倍的加速。



CTR預測表現的深度學習模型

對於旨在最大化點擊率的推薦系統，必須學習使用者行為背後複雜的特徵交互，這些特徵交互可以透過數學方式從低階到高階計算。對於良好的深度學習模型來說，低階和高階特徵交叉應該同等重要，而不應偏向其中任何一方。深度分解機（DeepFM）是一種基於分解機的神經網絡，它將用於推薦的分解機和用於特徵學習的深度學習結合在新的神經網路架構中。

雖然傳統的分解機將成對的特徵交叉建模為特徵之間潛在向量的內積，並且理論上可以捕獲高階信息，但在實踐中，機器學習從業者通常只使用二階特徵交叉，因為計算和存儲複雜度很高。深度神經網路變體，例如Google的 "廣度與深度模型"另一方面，透過結合線性寬模型和深度模型，在混合網路結構中學習複雜的特徵交互作用。

這個 Wide & Deep 模型有兩個輸入，一個用於底層的廣度模型，另一個用於深度模型，後者仍然需要專家的特徵工程，因此該技術不太適用於其他領域。與廣度和深度模型不同，DeepFM 可以使用原始特徵進行有效訓練，而無需任何特徵工程，因為它的廣度部分和深度部分共享相同的輸入和嵌入向量。

我們首先處理了 Criteo `train.txt` （11GB）檔案轉換為名為 `ctr_train.csv`儲存在 NFS 掛載中 `/sparkdemo/tr-4570-data`使用 `run_classification_criteo_spark.py`來自部分"每個主要用例的 Python 腳本。"在此腳本中，函數 `process_input_file`執行幾個字串方法來刪除製表符並插入 ’,‘`作為分隔符號和 ’\n‘`作為換行符。請注意，您只需處理原始 `train.txt`一次，這樣程式碼區塊就顯示為註解。

為了對不同的 DL 模型進行以下測試，我們使用 `ctr_train.csv`作為輸入檔。在後續的測試運行中，輸入的 `CSV` 檔案被讀入 `Spark DataFrame`，其模式包含以下字段 `’label'`，整數密集特徵 `['I1', 'I2', 'I3', …, 'I13']`和稀疏特徵 `['C1', 'C2', 'C3', …, 'C26']`。下列 `spark-submit`指令接受輸入 CSV，以 20% 的比例訓練 DeepFM 模型進行交叉驗證，並在十個訓練週期後選出最佳模型來計算測試集上的預測準確率：

```
(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
```

請注意，由於數據文件 `ctr_train.csv`超過 11GB，則必須設定足夠的 `spark.driver.maxResultSize`大於資料集大小以避免錯誤。

```
 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()
```

在上述 `SparkSession.builder`配置我們還啟用了 "阿帕契箭"，將 Spark DataFrame 轉換為 Pandas DataFrame ， `df.toPandas()`方法。

```
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.
```

隨機分割後，訓練資料集中有超過 3,600 萬行，測試集中有 900 萬個樣本：

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

由於本技術報告專注於不使用任何 GPU 的 CPU 測試，因此必須使用適當的編譯器標誌來建立 TensorFlow。此步驟避免呼叫任何 GPU 加速函式庫，並充分利用 TensorFlow 的高階向量擴充 (AVX) 和 AVX2 指令。這些特徵是為線性代數計算而設計的，例如向量加法、前饋中的矩陣乘法或反向傳播 DNN 訓練。 AVX2 提供的融合乘加 (FMA) 指令使用 256 位元浮點 (FP) 暫存器，非常適合整數程式碼和資料類型，可實現高達 2 倍的加速。對於 FP 程式碼和資料類型，AVX2 比 AVX 實現了 8% 的加速。

```
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.
```

若要從原始碼建置 TensorFlow， NetApp建議使用 "巴澤爾"。對於我們的環境，我們在 shell 提示字元下執行以下命令來安裝 `dnf`， `dnf-plugins` ，以及 Bazel。

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

您必須啟用 GCC 5 或更新版本才能在建置過程中使用 C++17 功能，該功能由 RHEL 透過軟體集合庫 (SCL) 提供。以下命令安裝 `devtoolset`以及 RHEL 7.9 叢集上的 GCC 11.2.1：

```
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
```

請注意，最後兩個命令啟用 `devtoolset-11`，使用 `/opt/rh/devtoolset-11/root/usr/bin/gcc`（GCC 11.2.1）。此外，請確保您的 `git`版本高於 1.8.3（隨 RHEL 7.9 提供）。參考這個 "文章"用於更新 `git`至 2.24.1。

我們假設您已經克隆了最新的 TensorFlow 主倉庫。然後創建一個 `workspace`目錄與 `WORKSPACE`檔案使用 AVX、AVX2 和 FMA 從原始程式碼建置 TensorFlow。運行 `configure`檔案並指定正確的 Python 二進位位置。"CUDA"由於我們沒有使用 GPU，因此在我們的測試中被停用。一個 `.bazelrc`文件根據您的設定產生。此外，我們編輯了文件並設置 `build --define=no_hdfs_support=false`啟用 HDFS 支援。參考 `.bazelrc`在本節中"針對每個主要用例的 Python 腳本，"以獲得完整的設定和標誌清單。

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

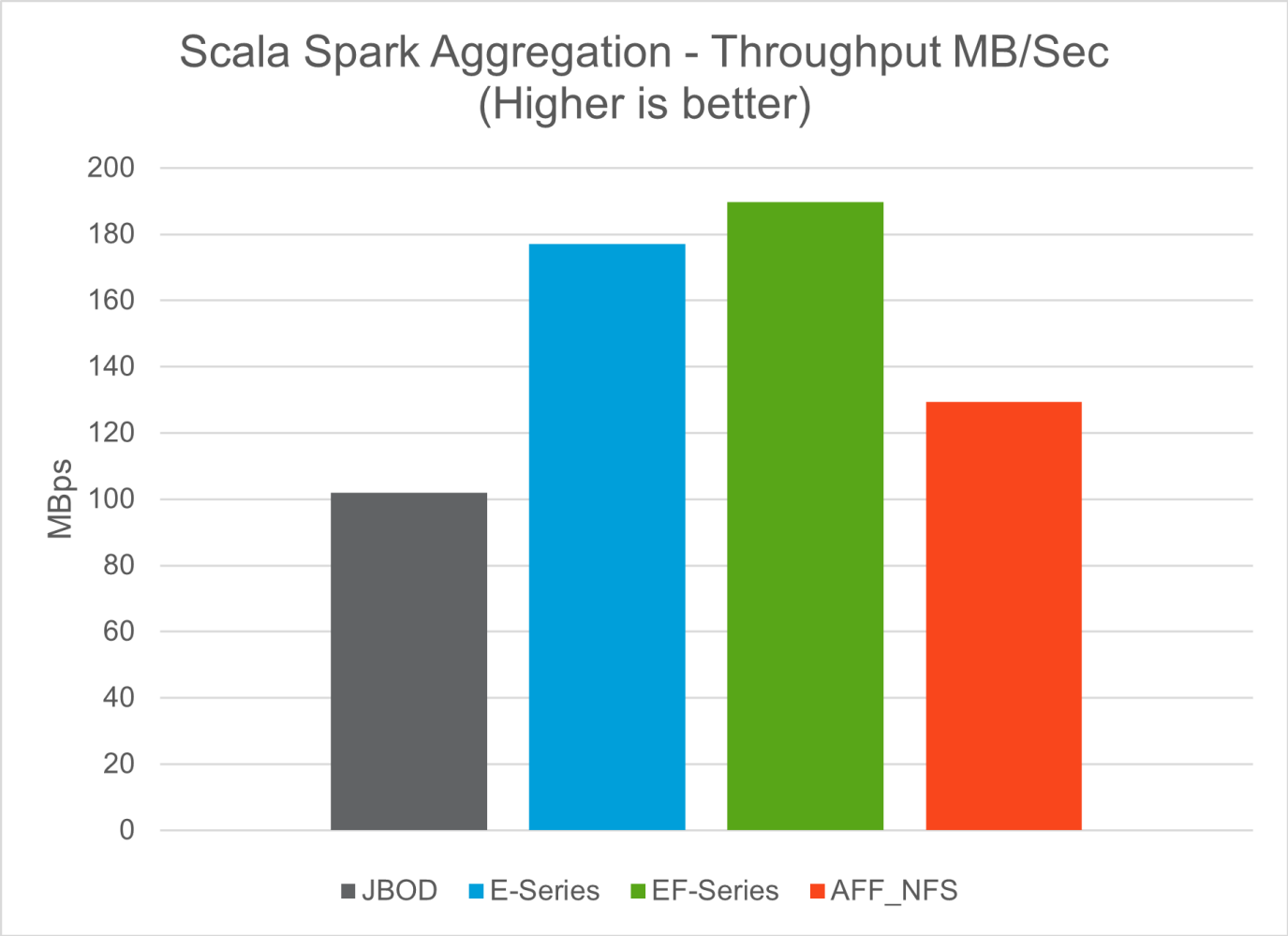
使用正確的標誌建立 TensorFlow 後，執行以下腳本來處理 Criteo Display Ads 資料集，訓練 DeepFM 模型，並根據預測分數計算接收者操作特徵曲線下面積 (ROC AUC)。

```
(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
```

經過十次訓練後，我們獲得了測試資料集上的 AUC 分數：

```
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
```

以與先前的用例類似的方式，我們將 Spark 工作流程運行時與位於不同位置的資料進行了比較。下圖顯示了 Spark 工作流程運行時深度學習 CTR 預測的比較。

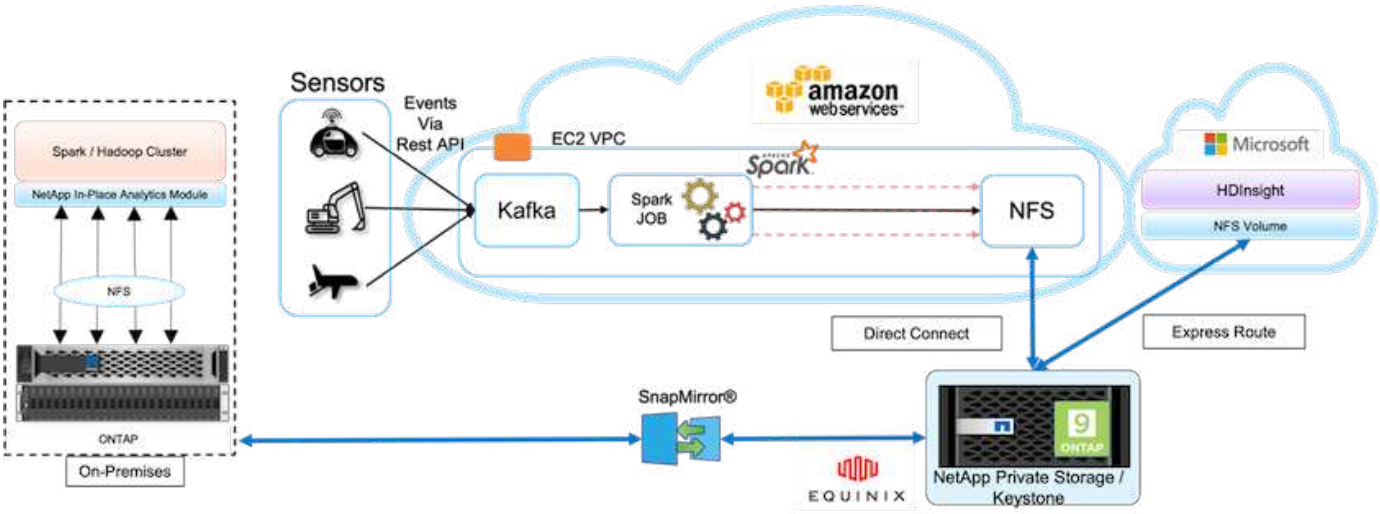


混合雲解決方案

現代企業資料中心是一種混合雲，它透過具有一致操作模式的連續資料管理平面在本地和/或多個公有雲中連接多個分散式基礎架構環境。為了充分利用混合雲，您必須能夠在本地和多雲環境之間無縫移動數據，而無需進行任何數據轉換或應用程式重構。

客戶表示，他們開始混合雲端之旅的方法是將二級儲存遷移到雲端用於資料保護等用例，或將不太重要的業務工作負載（如應用程式開發和 DevOps）遷移到雲端。然後他們轉向處理更重要的工作。 Web 和內容託管、DevOps 和應用程式開發、資料庫、分析和容器化應用程式是最受歡迎的混合雲工作負載。企業 AI 專案的複雜性、成本和風險歷來阻礙 AI 從實驗階段走向生產階段。

透過NetApp混合雲端解決方案，客戶可以透過單一控制面板享受整合的安全性、資料治理和合規性工具，用於跨分散式環境的資料和工作流程管理，同時根據其消費情況優化整體擁有成本。下圖是一個雲端服務合作夥伴的範例解決方案，該合作夥伴負責為客戶的大數據分析資料提供多雲連接。



在這種情況下，AWS 從不同來源接收的 IoT 資料儲存在NetApp私有儲存 (NPS) 的中心位置。 NPS 儲存連接到位於 AWS 和 Azure 中的 Spark 或 Hadoop 集群，使在多個雲端中運行的大數據分析應用程式能夠存取相同的資料。此用例的主要要求和挑戰包括以下內容：

- 客戶希望使用多個雲端對相同資料運行分析作業。
- 必須透過不同的感測器和集線器從不同來源（例如本地和雲端環境）接收資料。
- 該解決方案必須高效且具有成本效益。
- 主要挑戰是建立一個經濟高效的解決方案，在不同的內部部署和雲端環境之間提供混合分析服務。

我們的資料保護和多雲連接解決方案解決了跨多個超大規模運算平台的雲端分析應用程式的痛點。如上圖所示，來自感測器的資料透過 Kafka 串流並輸入到 AWS Spark 叢集中。資料儲存在 NPS 中的 NFS 共用中，NPS 位於 Equinix 資料中心內的雲端供應商之外。

由於NetApp NPS 分別透過 Direct Connect 和 Express Route 連線連接到 Amazon AWS 和 Microsoft Azure，因此客戶可以利用 In-Place Analytics Module 存取來自 Amazon 和 AWS 分析叢集的資料。因此，由於本地儲存和 NPS 儲存都運行ONTAP軟體， "SnapMirror"可以將NPS資料鏡像到本地集群，提供跨本地和多雲的混合雲分析。

為了獲得最佳效能， NetApp通常建議使用多個網路介面和直接連接或快速路由來存取雲端實例的資料。我們還有其他數據移動解決方案，包括 "XCP"和 "BlueXP複製和同步"幫助客戶建立應用程式感知、安全且經濟高效的混合雲 Spark 叢集。

針對每個主要用例的 **Python** 腳本

以下三個 Python 腳本對應測試的三個主要用例。首先是 `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}")
```

第二個腳本是 `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()
```

第三個腳本是 `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))
```

結論

在本文檔中，我們討論了 Apache Spark 架構、客戶用例以及與大數據、現代分析、AI、ML 和 DL 相關的NetApp儲存產品組合。在我們基於行業標準基準測試工具和客戶需求的效能驗證測試中， NetApp Spark 解決方案展現了相對於原生 Hadoop 系統的卓越效能。本報告中提供的客戶使用案例和效能結果的組合可以幫助您為您的部署選擇合適的 Spark 解決方案。

在哪裡可以找到更多信息

本 TR 中使用了以下參考文獻：

- Apache Spark 架構和元件

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

- Apache Spark 用例

  "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"

- 用於廣告點擊預測的深度和交叉網絡

  "https://arxiv.org/abs/1708.05123"

- FlexGroup

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

- 串流 ETL

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

- NetApp E系列Hadoop解決方案

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

- NetApp現代資料分析解決方案

  "數據分析解決方案"

- 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複製和同步

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

- DataOps 工具包

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