remote sensing

MDPI

Technical Note

Scalable Artificial Intelligence for Earth Observation Data Using Hopsworks

Desta Haileselassie Hagos [1,*], Theofilos Kakantousis [2], Sina Sheikholeslami [1], Tianze Wang [1], Vladimir Vlassov [1], Amir Hossein Payberah [1], Moritz Meister [2], Robin Andersson [2] and Jim Dowling [1,2]

[1] Division of Software and Computer Systems, KTH Royal Institute of Technology, 100 44 Stockholm, Sweden; sinash@kth.se (S.S.); tianzew@kth.se (T.W.); vladv@kth.se (V.V.); payberah@kth.se (A.H.P.); jdowling@kth.se or jim@logicalclocks.com (J.D.)

[2] Logical Clocks AB, 118 72 Stockholm, Sweden; theo@logicalclocks.com (T.K.); moritz@logicalclocks.com (M.M.); robin@logicalclocks.com (R.A.)

* Correspondence: destah@kth.se

Abstract: This paper introduces the Hopsworks platform to the entire Earth Observation (EO) data community and the Copernicus programme. Hopsworks is a scalable data-intensive open-source Artificial Intelligence (AI) platform that was jointly developed by Logical Clocks and the KTH Royal Institute of Technology for building end-to-end Machine Learning (ML)/Deep Learning (DL) pipelines for EO data. It provides the full stack of services needed to manage the entire life cycle of data in ML. In particular, Hopsworks supports the development of horizontally scalable DL applications in notebooks and the operation of workflows to support those applications, including parallel data processing, model training, and model deployment at scale. To the best of our knowledge, this is the first work that demonstrates the services and features of the Hopsworks platform, which provide users with the means to build scalable end-to-end ML/DL pipelines for EO data, as well as support for the discovery and search for EO metadata. This paper serves as a demonstration and walkthrough of the stages of building a production-level model that includes data ingestion, data preparation, feature extraction, model training, model serving, and monitoring. To this end, we provide a practical example that demonstrates the aforementioned stages with real-world EO data and includes source code that implements the functionality of the platform. We also perform an experimental evaluation of two frameworks built on top of Hopsworks, namely MAGGY and AUTOABLATION. We show that using MAGGY for hyperparameter tuning results in roughly half the wall-clock time required to execute the same number of hyperparameter tuning trials using Spark while providing linear scalability as more workers are added. Furthermore, we demonstrate how AUTOABLATION facilitates the definition of ablation studies and enables the asynchronous parallel execution of ablation trials.

Keywords: Hopsworks; Copernicus; Earth Observation; machine learning; deep learning; artificial intelligence; model serving; big data; ablation studies; MAGGY; *ExtremeEarth*

Citation: Hagos, D.H.; Kakantousis, T.; Sheikholeslami, S.; Wang, T.; Vlassov, V.; Payberah, A.H.; Meister, M.; Andersson, R.; Dowling, J. Scalable Artificial Intelligence for Earth Observation Data Using Hopsworks. *Remote Sens.* **2022**, *14*, 1889. https://doi.org/10.3390/rs14081889

Academic Editor: Lefei Zhang

Received: 10 February 2022
Accepted: 10 April 2022
Published: 14 April 2022

1. Introduction

In recent years, huge volumes of big data have been generated in various domains. Sentinel satellites (https://www.copernicus.eu/en (accessed on 9 April 2022)), for example, collect more than three petabytes of EO sentinel data for Copernicus yearly; Copernicus is the European Union's flagship EO programme for monitoring the planet Earth and its environment. This enormous amount of data is made readily available to researchers who want to use it, including those who wish to develop AI algorithms employing DL approaches that are more appropriate for big data. However, one of the most significant obstacles that researchers face is a lack of modern and emerging technologies that can assist them in unlocking the potential of this massive data and developing classification

and prediction AI models. As part of addressing this challenge, we presented a software platform architecture in our previous works [1,2]. Furthermore, in [1,2], we demonstrated how the Hopsworks platform is integrated with various services and components to extract meaningful knowledge from AI and build AI-based applications using Copernicus and EO data.

In this work, we introduce the Hopsworks (https://www.hopsworks.ai/ (accessed on 9 April 2022)) platform, an open-source AI platform jointly developed by Logical Clocks (https://www.logicalclocks.com/ (accessed on 9 April 2022)) and the KTH Royal Institute of Technology (https://www.kth.se/en (accessed on 9 April 2022)). We describe in detail how Hopsworks can be used to enable massive-scale AI for EO data and other tasks, such as data-parallel and distributed DL by employing features that enhance its scalability, including a Feature Store [3] and the MAGGY framework [4]. Hopsworks (https://github.com/logicalclocks/hopsworks (accessed on 9 April 2022)) is an open-source AI platform that provides users with an execution environment for designing, distributing training for, and running end-to-end ML/DL pipelines at scale. This work describes how the features of the Hopsworks platform are applied to EO data. Hopsworks also has the most-scalable distributed hierarchical file system—which enables users to reduce the storage cost of EO data—named HopsFS. HopsFS stores EO data in the native back-end object-storage infrastructure of Data and Information Access Services (DIAS); the back-end is accessed via the S3 or Swift protocols (https://creodias.eu/data-access-interfaces (accessed on 9 April 2022)). These features provide excellent support for implementing scalable DL models to process enormous volumes of EO data generated by various sources.

To this end, this paper serves as a demonstration and walkthrough of the stages of building a production-level end-to-end ML/DL pipeline with a main focus on EO data utilising Hopsworks, including data ingestion, data preparation, feature extraction, model training, model serving, and monitoring. This demonstration is developed and presented in the context of the *ExtremeEarth* project (http://earthanalytics.eu/ (accessed on 9 April 2022)). Our experience of using Hopsworks in the project is explained below. Two applications, sea ice classification and crop-type mapping and classification, were developed using the software platform architecture mentioned above by utilising the petabytes of Copernicus satellite big data made available through the Copernicus EO programme and infrastructure [1].

Experience of using Hopsworks in *ExtremeEarth* Project. One of the primary factors that differentiates the *ExtremeEarth* project from other Earth analytics methodologies and technologies is the use of Hopsworks. Hopsworks has been used for developing DL models for the use cases considered in the *ExtremeEarth* project, namely a Polar use case that uses the Sentinel-1 SAR images for snow parameters and a Food Security use case that uses Sentinel-2 images. The overall *ExtremeEarth* infrastructure and its multi-layer architecture were presented in our previous work [2]. In this paper, we focus only on the processing layer of the *ExtremeEarth* software infrastructure. The layer provides the Hopsworks data-intensive AI platform that presents scalable AI support for EO data. For more detailed information about the different architecture layers and the integration of associated components, including Hopsworks and the detailed flow of events of the *ExtremeEarth* infrastructure, refer to our previous works [1,2].

The work presented in this paper demonstrates in detail the scalability services and features of Hopsworks that provide users with the means to build scalable ML/DL pipelines for EO data as well as support for the discovery and search for EO metadata. Hopsworks is a horizontally scalable platform for big data and AI applications [5]. It provides first-class support for both data analytics and data science at scale. In particular, Hopsworks supports the development of ML and DL applications in notebooks and the operation of workflows to support those applications, including parallel data processing at scale, model training at scale, and model deployment. A data science application, especially in the realm of big data, typically comprises a set of essential stages that form an ML/DL pipeline. This

data pipeline is responsible for transforming data and serving it as knowledge by using data-engineering processes and by employing ML and DL techniques.

In the context of the EO data domain, these end-to-end ML/DL pipelines must scale to the petabyte-sized data and datasets available within the Copernicus programme. A typical ML/DL pipeline consists of stages: data ingestion, data preparation and validation, feature extraction, model building and validation (training), and model serving and monitoring. The first two stages, namely data ingestion and preparation, can also be described as data pipelines. Figure 1 illustrates the horizontally scalable infrastructure that enables developers to manage the lifecycle of EO ML applications. The feature extraction stage is facilitated by the Feature Store service, presented in Section 2.

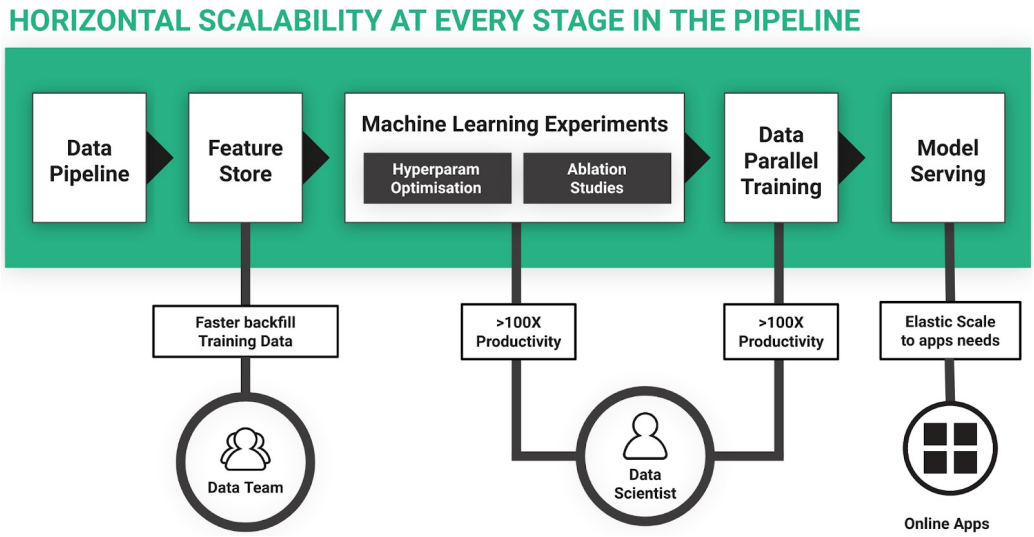

Figure 1. Horizontally scalable end-to-end ML/DL pipeline stages.

The Hopsworks platform is a data-intensive AI platform that presents scalable AI support for Copernicus big data. It provides data engineers and data scientists with all the necessary technical tools that can be used to build and manage each stage of the end-to-end ML/DL pipelines, as shown in Figure 1. Once data goes through all the stages and the ML/DL pipeline model is served, the stages may be executed again to consider new data that has arrived in the meantime and to further fine-tune the pipeline stages, leading to more accurate models and results. Therefore, a data-science lifecycle is formed that continuously iterates the above ML pipeline. As a result, data scientists are faced with the highly complex task of developing DL workflows that utilise each stage of the ML/DL pipeline. The complexities of such pipelines can grow as input data increase in volume, which, in the case of EO data, means that a robust and flexible architecture needs to be in place to assist data engineers and scientists in developing these pipelines. Figure 2 depicts the overall architecture and lifecycle of an end-to-end ML/DL pipeline, along with the technologies that are used to implement it and are demonstrated in the rest of the paper.

The stages of ingesting data, pre-processing, and managing a service to store curated feature data and to compute features can be considered part of the data-engineering lifecycle. The Feature Store is the service used in this ML/DL pipeline to manage curated feature data. The second step of the pipeline, the actual ML training and model development, starts by fetching feature data in appropriate file formats to be used as inputs for training, with each file format depending on the ML framework that is used. This step can be considered the data-science lifecycle, in which new feature data are fetched and new models are iteratively developed and pushed to production.

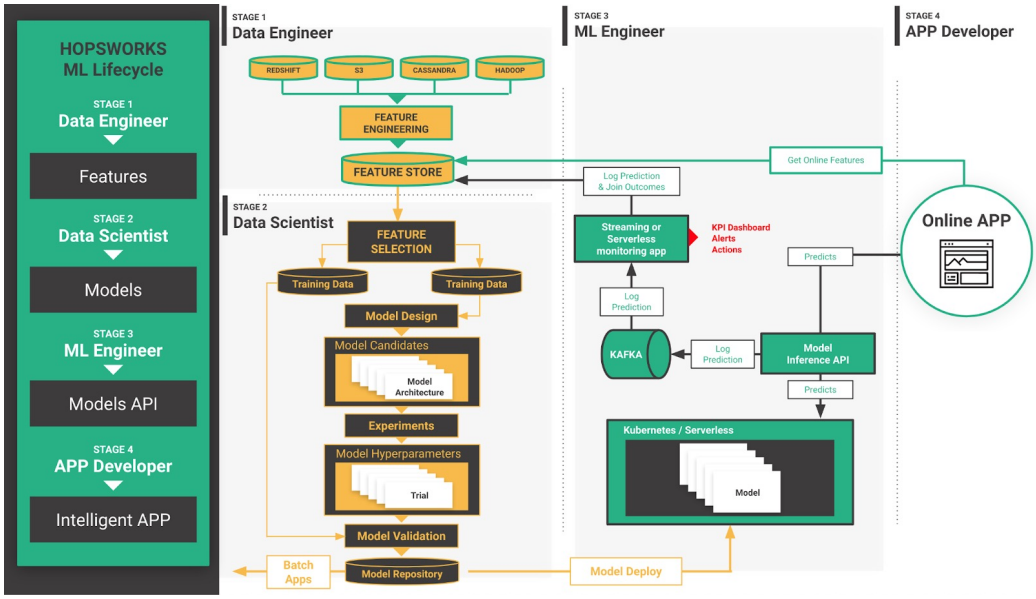

Figure 2. Hopsworks lifecycle of an end-to-end ML/DL pipeline.

One of the main goals of an end-to-end ML/DL pipeline is to continuously improve output models by utilising user-defined metrics. To detect when an ML/DL pipeline should be triggered to update a DL model served in production, there needs to be a mechanism that logs all inference requests and monitors how the model is performing over time. Model serving and monitoring in Hopsworks provide these capabilities to developers of pipelines and are further demonstrated in Section 2. Figure 3 depicts the Hopsworks services stack. HopsFS [6] and RonDB (NDB/MySQL Cluster) provide horizontally scalable data and metadata storage. Apache Hadoop YARN and Kubernetes are the resource management frameworks on the upper layer. The Hopsworks platform utilises the flavour of YARN that is developed within Hopsworks as a resource management service for deploying distributed applications on a cluster of servers. In Hopsworks, YARN supports scheduling applications with resource constraints, such as CPU, main memory, and GPU [7] constraints. Therefore, Spark is deployed on top of YARN in Hopsworks, and users developing ML/DL pipelines can easily, from within the Hopsworks UI, request these three resources to be allocated to their job or notebook. This is particularly important for allocating GPUs when the Spark program needs access to a GPU's computing power.

The Hopsworks platform has also been extended with an API that allows clients to submit Spark applications on a cluster easily. Hopsworks supports Apache Spark as a Service, which automatically sets up default Spark configuration parameters. It also provides a flexible way for users to provide additional configurations with their Spark application via the UI or the RESTful and client APIs for their applications. These services provide resources to the distributed processing framework in Hopsworks, Apache Spark, and Hopsworks itself to provide EO data pre-processing with an arbitrary programming language functionality and to run Python jobs and notebooks. Additional services, such as providing tools for debugging logs and metrics monitoring, are part of this layer. The next layer comprises Hopsworks itself, the webapp with the RESTful API that provides client applications and users connectivity to the entire Hopsworks cluster.

Deploying and running an end-to-end ML/DL pipeline can be a repetitive task as most (if not all) stages need to run when new input data are ingested into the system. In case of a failure, orchestrating the order of stage execution, monitoring progress, and putting a retry mechanism in place are essential for making an EO data pipeline production ready. To this end, Hopsworks integrates Apache Airflow (https://airflow.apache.org/ (accessed on 9 April 2022)) as an orchestration engine.

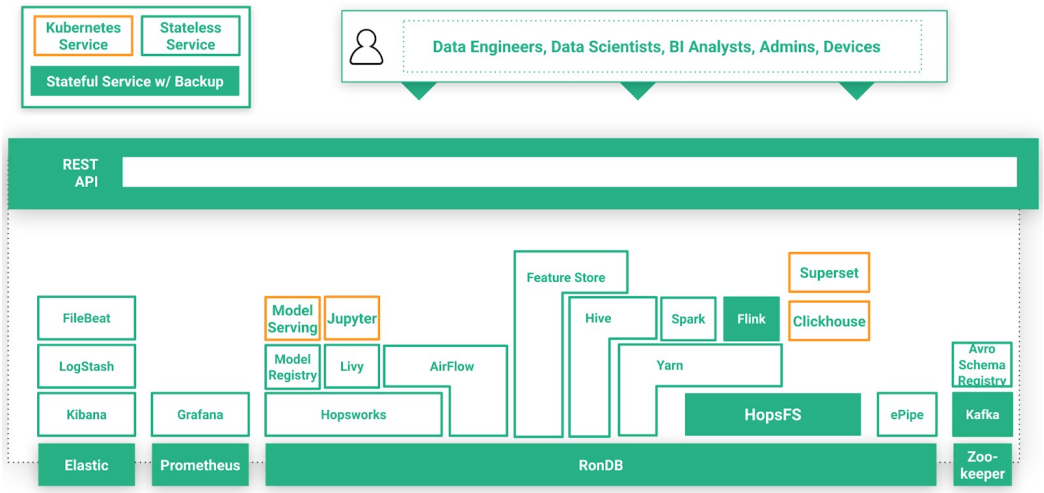

Figure 3. Hopsworks Services Stack.

Application areas of the Hopsworks platform. In prior work, we showed how the Hopsworks platform was used to engineer the features of, train, and serve DL models using many terabytes of Copernicus and EO data [1,2]. While Hopsworks is a horizontal platform for developing and operating AI applications at scale, it has been customised for remote sensing and the EO community. Within the context of *ExtremeEarth*, the platform has already been used to develop two use cases: sea ice classification for the Polar Thematic Exploitation Platform (TEPs) and crop-type mapping and classification for the Food Security TEPs [1]. Hopsworks is currently being used to train and operate machine-learning models at scale in other domains, such as finance, healthcare, and natural language processing. Although Hopsworks has not yet been used as a platform for other Copernicus TEPs, such as the Marine Environment Monitoring Service, we believe the platform can be used in a manner similar to the Polar and Food Security TEPs, that is, for scalable feature engineering, scale-out deep learning, and online model serving. In Section 2, we demonstrate how the end-to-end DL pipeline for the ship–iceberg classification model is developed.

Our work presented in this paper is an early study showing the promise of the Hopsworks platform and advanced techniques targeting the EO domain. Although there are many different application areas in EO, we cover only the Polar and Food Security use cases, but we believe that the techniques discussed in this paper can be generalised to cover and make contributions to different EO application areas. The results presented in this paper are compelling enough to make a case for this platform to be applied to a variety of other application domains.

Scalable AI. Scalable storage, scale-out feature engineering, scale-out training, and scale-out online model serving are the main components of a scalable AI platform. There are two main parts to scalable AI. First, the platform has to be able to store increasing amounts of data by adding resources (storage servers) when needed. Hopsworks provides HopsFS [6] as a scale-out hierarchical distributed file system that can store data at a low cost on local disks or on top of the Object Store. Second, the platform must support scale-out computing (feature engineering with CPUs and training deep-learning models with GPUs). The results of independently scalable storage and scalable computing are that as the data volume grows, the platform can store the increasing volumes of data and process the data into features in a similar time by adding more resources (CPU and memory), and the time required to train a deep-learning model can be reduced by adding more GPUs in what is known as data-parallel model training. Similarly, if the inference load on our online models increases, we should be able to add more resources to handle the increased model-serving load while keeping response latencies below the level stipulated in a service level agreement. To summarise, scalable AI means that we can go beyond the capacity of a single server, and we can use many servers to reduce training and inference times reasonably and

are able to train larger models. As presented in Section 3, our work introduces a parallel model training approach with which the model training is partitioned over many servers where the models do not fit in a single server.

Contributions. In summary, the main contributions of this work are as follows.

- We introduce the Hopsworks platform, which brings scalable AI support to the EO data community and the Copernicus programme.
- We describe in detail how Hopsworks can be used to enable massive-scale AI for EO data and other tasks.
- We present EO data pipelines with enhanced and newly developed features to enable and improve support for data parallelism and distributed DL.
- We demonstrate the scalability services and features of the Hopsworks platform, which provide users with the means to build scalable end-to-end ML/DL pipelines for EO data as well as support for the discovery and search for EO metadata.
- We provide a practical example that demonstrates the stages of building a production-level model with real-world EO data and source code that implements the functionality of the platform.
- We perform an experimental evaluation of two frameworks built on top of Hopsworks: the MAGGY framework for hyperparameter tuning and parallel ML experiments and the AUTOABLATION framework for automated ablation studies.

Outline. The remainder of the paper is organised as follows. The scalable AI capabilities of the Hopsworks platform and its features for the scalable end-to-end ML/DL pipeline stages for EO data are explained in sufficient detail in Section 2. The experimental settings and evaluation results of the frameworks discussed in this paper are presented in Section 3. In Section 4, we summarise and discuss the key findings and implications of our work and highlight directions for future research work. Finally, Section 5 concludes our paper.

2. Materials and Methods

This section presents in detail the end-to-end ML/DL pipeline stages, the scalable AI capabilities of the Hopsworks frameworks and Hopsworks' features for scalable ML.

2.1. ML/DL Pipeline Stages

2.1.1. Data Ingestion

Locating and identifying the sources of input data for the AI platform is the first step in building a scalable end-to-end ML/DL pipeline. The next stage is developing advanced methods for ingesting data from the input sources into the underlying AI platform, which runs the ML/DL pipeline. The format in which the input data are stored and the implemented protocols to send data to other systems can vary widely among these input-data sources. Raw data from Internet of Things (IoT) devices, image data from satellites, financial transactions from real-time systems, structured data from data warehouses, social media, etc. are some examples of such sources. The data in *ExtremeEarth* is normally stored in the DIAS, which can be accessed directly from the Hopsworks platform. Some of the data ingestion sources for an end-to-end ML/DL pipeline, where the external systems reside, are shown in Figure 4. We show several ways in which the Hopsworks platform has been enhanced and extended to make satellite-image data for both Polar and Food Security use cases easily ingested in the platform for subsequent processing in the context of the *ExtremeEarth* project.

Accessing EO data from Hopsworks. In the context of the *ExtremeEarth* project, the Hopsworks platform is deployed on the CREODIAS, one of the DIASs of the Copernicus programme environment where EO data needed for this project resides. In the CREODIAS environment, EO data are made available via the Object Store, where it can be accessed via the S3 protocols implemented by OpenStack Swift (https://docs.openstack.org/swift/latest/s3_compat.html (accessed on 9 April 2022)) or via standardised web services, such as WMS, WMTS, WCS, and WFS (https://creodias.eu/data-access-interfaces (accessed on 9 April 2022)).

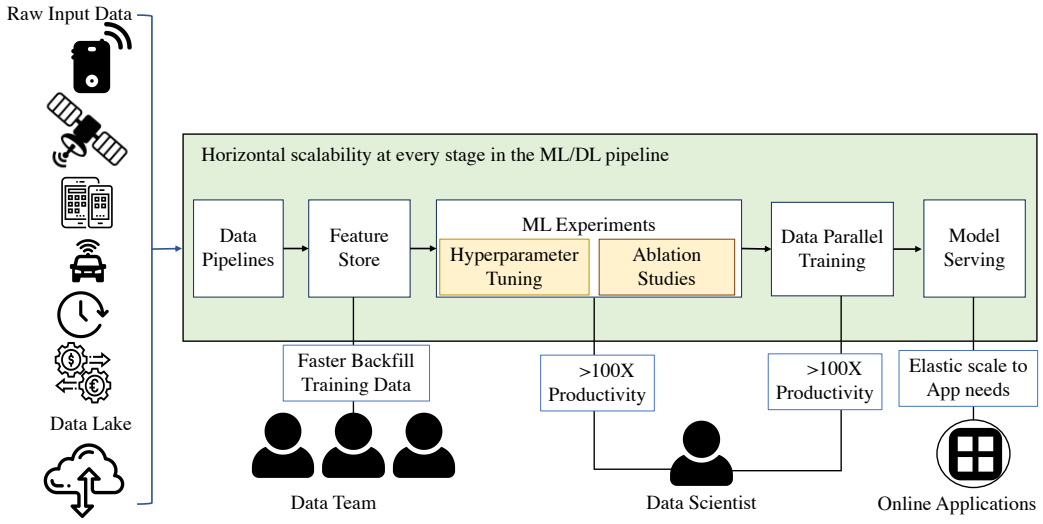

Figure 4. Data ingestion sources for an ML/DL pipeline for EO data in the Hopsworks platform [2].

2.1.2. EO Data Pre-Processing

EO data pre-processing with Python. Satellite data often need to be processed before being provided as inputs into ML/DL algorithms. By utilising the Python support the Hopsworks platform provides, data scientists in *ExtremeEarth* can utilise utility programs, such as GDAL (https://gdal.org/ (accessed on 9 April 2022)), to process EO data.

EO data pre-processing with Docker and Kubernetes. The Hopsworks platform has been extended in *ExtremeEarth* to provide efficient and flexible support for working with arbitrary programming languages and frameworks when processing EO data. This enables users to utilise the tools and programming languages of their choice to process EO data by running arbitrary Docker containers on Kubernetes via the Hopsworks jobs service. The main motivation behind this functionality is that users might need to utilise user-friendly tools and frameworks that are not necessarily available in the Python Anaconda environment of the project, such as Java or C++ tools related to remote sensing and EO data. Figure 3 displays the software stack integrated into Hopsworks that enables users to run jobs and notebooks, including the Docker job type used for the EO data pre-processing.

2.1.3. Feature Engineering and Data Validation

Feature Store. The process of applying domain knowledge to create features used during ML/DL pipeline stages is referred to as feature engineering. With the constant growth in input data, there is a greater need for an efficient framework that allows for feature engineering and decreases the complexity of managing features. As a result, the need has also increased the complexity of ML/DL pipelines.

In this work, the Hopsworks platform has been enhanced and extended with a new framework named Feature Store [3] to allow data scientists and engineers working with EO to organise their ML assets and curate the EO data features to enhance the management of curated feature data.

The Feature Store is a data-science and data-engineering interface that acts as an enterprise's central data management system across different cloud services and containerised applications (https://www.hopsworks.ai/feature-store (accessed on 9 April 2022)). The generated features that can be utilised to create ML/DL models must be defined, computed, and persisted before using the validated data to develop them. In the context of using the *ExtremeEarth* software architecture on EO data, the underlying service that data engineers and data scientists utilise for such tasks is the Hopsworks Feature Store. In order to deal with large amounts of data and complex data types and relationships, the Feature Store provides comprehensive APIs, scalability, and elasticity. Utilising such services, users can, for example, create groups of features or compute new features, such as aggregations of existing ones.

Models are trained using sets of features. Generating reusable features that can be distributed across various teams in an organisation that can effectively facilitate the development of new ML models and pipelines, as shown in Figure 5, is the primary motivation for feature engineering. Feature discoverability with free-text searches across an organisation's feature data, reusing generated features across different pipelines, and applying software engineering principles to ML features with versioning, documentation, access control tools, and time-travel by fetching previous feature data used for training a particular model are just a few of the Feature Store's main benefits. In addition, it enables data scientists to collect and manage numerous petabytes of feature datasets with scalability, allowing them to acquire valuable insights into data distribution and correlation.

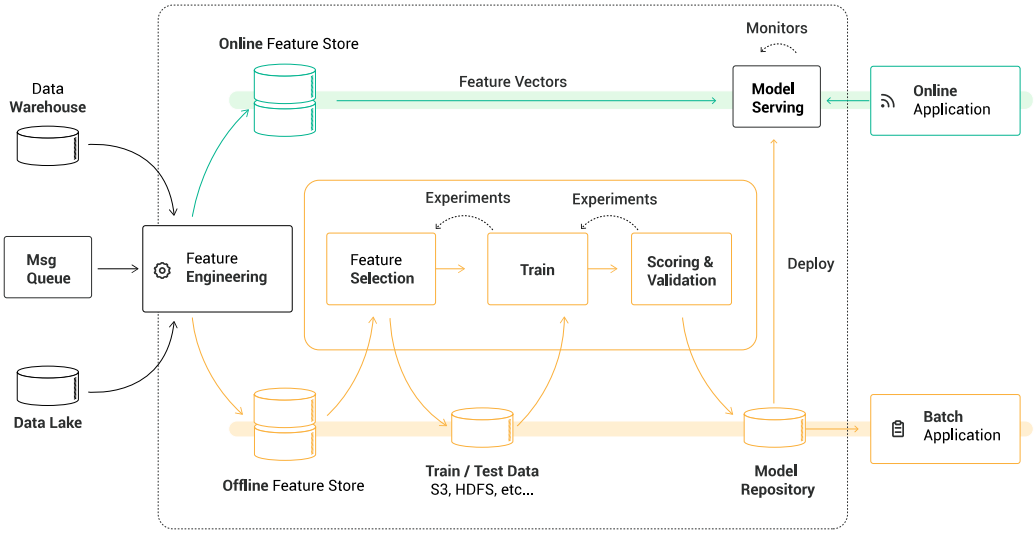

Figure 5. The Feature Store interface of Hopsworks and its core components.

The Feature Store [3] is built on fault-tolerant, distributed, and scalable services to achieve all of the main properties mentioned above. Online data are saved in RonDB (NDB/MySQL Cluster), while Apache Hive [8] stores large volumes of offline data. Offline features can be employed for training and are typically utilised in batch-oriented data processing, with which previous feature data can be evaluated to provide meaningful statistics. For pipelines that require data at a prediction time, online features must be available in real time. In addition to storing data, the Feature Store uses a Hopsworks Spark integration to compute and analyse features. The key components of the Hopsworks Feature Store are shown in Figure 6.

Feature validation. In this work, the Feature Store is further extended to support the automated computation of ingested data statistics and a feature validation framework that enables users to define data quality rules and constraints to be applied to ingested data. Feature validation is the process of checking and cleansing data to be used as features in ML models to ensure that their correctness and quality are sufficiently reasonable to be processed by the subsequent stages of the ML/DL pipeline. The process of performing feature validation can significantly vary in implementation among ML/DL pipelines or data engineers and scientists. This is because data validation is not a strict set of rules that need to be applied to ingested data. Instead, it is a set of best practices and some common rules derived typically from the domain of statistics. Some of the data validation processes used when inserting data into the Feature Store are the validation of data types and structures, statistical properties, and values (such as an accepted range of values).

In the context of *ExtremeEarth*, there is one more constraint that must be considered when establishing a feature validation process: feature validation needs to be applied to large volumes of data in a distributed storage and processing environment. In addition to this, feature validation in the ML/DL pipeline context is applied to the feature data

that resides in the Feature Store. Then, these features are extracted in the form of training or test datasets to be served as inputs in the training stage. To achieve feature validation at scale, the Hopsworks Feature Store has been extended to support feature validation by introducing the concepts of feature expectations and validation rules (https://docs.hopsworks.ai/latest/generated/feature_validation/ (accessed on 9 April 2022)). This validation framework is built on Apache Spark and Deequ (https://github.com/awslabs/deequ (accessed on 9 April 2022)).

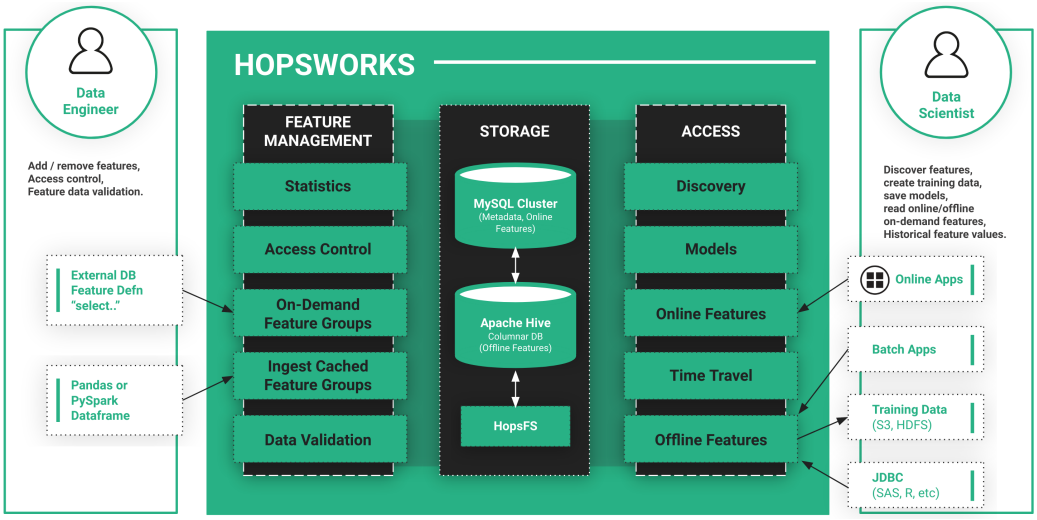

Figure 6. The Hopsworks platform's Feature Store architecture [2].

2.1.4. Model Analysis and Training

Model analysis is an analytical method of examining how a ML model responds to various assumptions. The analysis is essential for users to efficiently solve real-world AI and data-mining problems [9]. Here, we evaluate and compare ML models to each other using slices of data or data points. Building ML models is an iterative process; note that the quality of training data is as important as the underlying ML model. This is because the underlying model must be built using either a new training dataset or relying on feedback from a model validation. Subsequently, distributing training over enormous datasets to develop and deploy ML models is not easy. It is often necessary to validate the data input into the training algorithm and the model developed from these data. Hence, Hopsworks has been extended to include the *What-If Tool* (https://github.com/pair-code/what-if-tool (accessed on 9 April 2022)), a simple interactive visual interface for expanding understandings of a black-box regression or classification ML model.

Model interpretability is concerned with understanding what an ML model does, including during the pre- and post-processing stages and its behaviours for various inputs. To facilitate this functionality, the *What-If* model analysis framework is shipped with Hopsworks as part of the default Python environments that Hopsworks projects come with. It makes it easier for data scientists and other users to evaluate the underlying ML model's predictions. Therefore, users do not need to install it along with its dependencies, avoiding any risks of Python library dependency conflicts as well. Listing 1 shows the code snippet used to perform a model analysis for the ship–iceberg classification model developed to demonstrate this functionality. Users can set the number of data points to be displayed, the test dataset location to be used to analyse the model, and the features to be used.

2.1.5. Model Serving and Monitoring

Exporting, serving for inference, and monitoring the models developed in the previous stages are threefolds of the last stage of the DL workflow. After a model has been generated and exported from the Hopsworks platform utilising the previous stages in the ML/DL pipeline, it must be served to be used for inferences by external applications, such as

iceberg detection or water availability detection for food crops. Hopsworks' built-in elastic model-serving infrastructure allows users to submit inference requests for TensorFlow and scikit-learn. After the model is deployed, its performance must be monitored continuously in real time so that users can determine when the best time to start the training stage is. High-performance serving systems and other inference pipelines for ML/DL models have also been added to the Hopsworks platform as extensions [2]. Moreover, KFServing (https://hopsworks.readthedocs.io/en/latest/hopsml/kfserving.html (accessed on 9 April 2022)) has been integrated into Hopsworks for model serving. Users can use docker containers provided by either Hopsworks or KFServing.

Listing 1. Code snippet for model analysis with the *What-If* Tool.

```
# Invoke What-If Tool for test data and the trained model {display-mode: "form"}
num_datapoints = 2000  #@param {type: "number"}
tool_height_in_px = 1000  #@param {type: "number"}

from witwidget.notebook.visualization import WitConfigBuilder
from witwidget.notebook.visualization import WitWidget

test_examples = df_to_examples(test_df, features_and_labels)

# Setup the tool with the test examples and the trained classifier
config_builder = WitConfigBuilder(test_examples)
    .set_estimator_and_feature_spec(classifier, feature_spec)
    .set_label_vocab(['not iceberg', 'is iceberg'])
WitWidget(config_builder, height=tool_height_in_px)
```

As explained above, the Hopsworks platform also hides the complexity of managing the lifecycle of docker containers, EO data access, and logs. Furthermore, users can choose the smallest number of samples required for the underlying model to serve in a runtime, which gives Hopsworks users the vital property of elasticity. Additionally, the Hopsworks platform's model monitoring infrastructure continuously monitors the incoming requests sent to a model and its responses. Users can then apply their business logic to determine which actions to take based on how the monitoring metrics output changes over time. Hopsworks also introduces a novel security model based on TLS certificates that allow users to save sensitive data on the platform. Hopsworks logs all the inference requests in Apache Kafka [10]. The model serving and monitoring architecture of the Hopsworks platform is shown in Figure 7. The code snippets presented in Listings 2–4 show end-to-end examples of models serving on the Hopsworks platform using the TensorFlow framework. The helper library in Hopsworks that enables development by hiding the complexities of running applications and interfacing with the underlying services is *hops-util-py*.

2.2. Hopsworks Frameworks for Scalable ML

In this section, we will explain and summarise the scalability services and features that Hopsworks provides for ML and DL. Figure 8 shows the challenges in scaling out ML and DL. Hopsworks makes use of PySpark as an orchestration layer that is transparent to the users to scale out both the inner loop and the outer loop of distributed ML and DL. The inner loop is where model development (training) is done. In the inner loop, the current best practice for reducing the time required to train models is data-parallel training using multiple GPUs; hence, scaling out in this context means making use of more GPUs, which may be distributed across multiple machines, to make data-parallel training go faster. The outer loop is where we run as many experiments as needed to establish good hyperparameters of the model we are going to train. We typically run many experiments as we need to search for good values of the hyperparameters, since hyperparameters, as the name implies, are not learned during the training phase (i.e., in the inner loop).

Hopsworks supports the execution of hyperparameter tuning experiments in two ways: synchronous parallel executions through the Experiment API as well as a new framework for the asynchronous parallel execution of trials, called MAGGY [4]. To optimise the hyperparameters for DL, the out-of-the-box use of state-of-the-art directed search algorithms that work better (e.g., genetic algorithms, Bayesian optimisation [12], Hyperopt [13], and ASHA [14]) is provided. In the following subsections, we describe the Experiment API,

distributed training techniques available in Hopsworks, hyperparameter tuning with the MAGGY framework, and ablation studies with the AUTOABLATION framework.

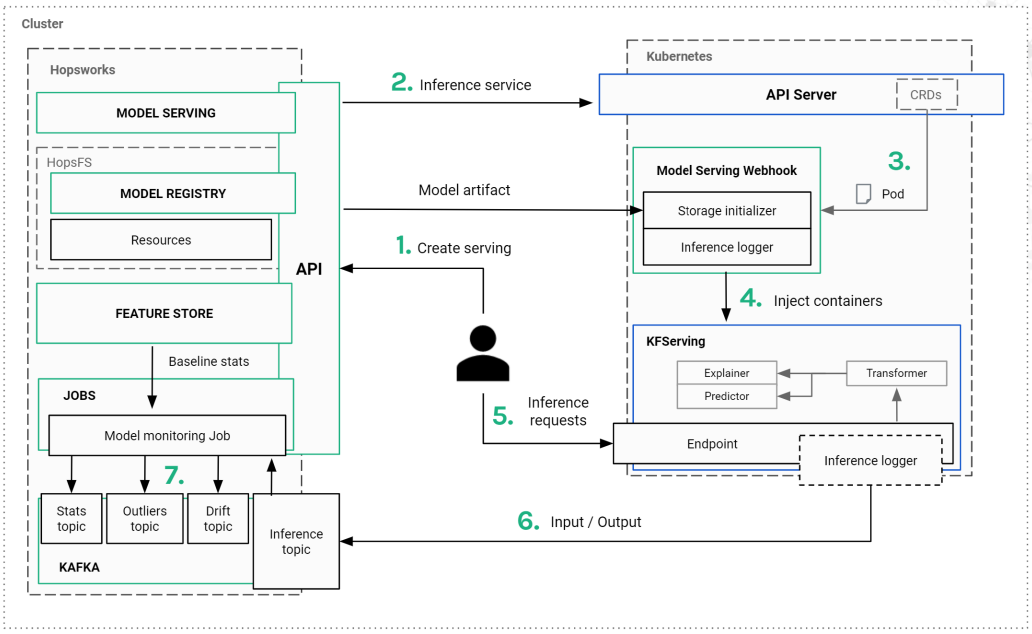

Figure 7. The architecture for model monitoring and serving in the Hopsworks platform [11].

Listing 2. Querying the model repository for the best MNIST model.

```
from hops import model
from hops.model import Metric
MODEL_NAME="mnist"
EVALUATION_METRIC="accuracy"

best_model = model.get_best_model(MODEL_NAME, EVALUATION_METRIC, Metric.MAX)

print('Model name: ' + best_model['name'])
print('Model version: ' + str(best_model['version']))
print(best_model['metrics'])
```

Listing 3. Creating model serving.

```
from hops import serving

# Create serving. Optionally, add the kfserving flag to deploy the model server
# using this serving tool. If not specified, it is deployed using the serving tool by default
# on the current Hopsworks version (docker or kubernetes)
serving_name = MODEL_NAME
model_path="/Models/" + best_model['name']
response = serving.create_or_update(serving_name, model_path,
                      model_version=best_model['version'],
                      model_server="TENSORFLOW_SERVING", kfserving=False)

# List all available servings in the project
for s in serving.get_all():
    print(s.name)
```

Listing 4. Sending prediction requests to the served model using the Hopsworks RESTful API.

```
import numpy as np
import json

TOPIC_NAME = serving.get_kafka_topic(serving_name)
NUM_FEATURES=784

for i in range(20):
    data = {
        "signature_name": "serving_default", "instances": [np.random.rand(NUM_FEATURES).tolist()]
            }
    response = serving.make_inference_request(serving_name, data)
    print(response)
```

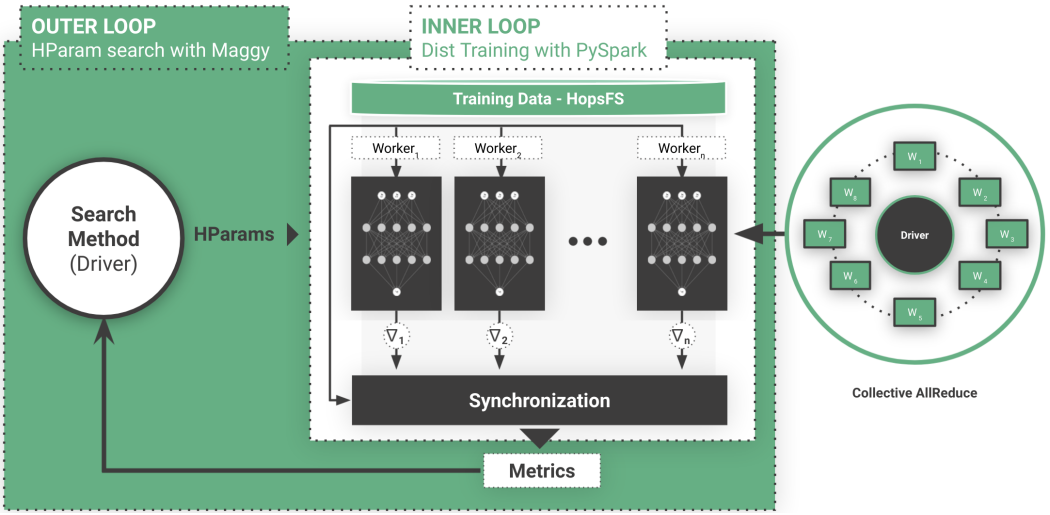

Figure 8. Scaling out distributed training of ML and DL.

2.2.1. The Experiment API

An experiment can be defined as the process of finding the optimal hyperparameters for model training by using training samples and building a model. Data scientists conduct multiple ML experiments on samples of their training datasets and build several models deployed, evaluated, and enhanced later on. The Hopsworks platform is shipped with a novel Experiments service (a.k.a., the Experiment API) that has first-class support for writing programs in Python and integrates popular open-source ML libraries and frameworks, such as TensorFlow, Keras, scikit-learn, TensorBoard, PyTorch, and any other framework that provides a Python API. Furthermore, it allows for managing the histories of ML experiments as well as monitoring during training (https://hopsworks.readthedocs.io/en/latest/hopsml/hopsML.html#experimentation (accessed on 9 April 2022)).

It is useful to have a common abstraction that defines the type of training, the configuration parameters, the input dataset, and the infrastructure environment that a ML program runs in to carry out ML training. In Hopsworks, the abstraction of an experiment is used to encapsulate the aforementioned properties. To productionise ML models, it is important to easily run a past experiment in case a software bug is discovered and the models need to be developed again based on previously seen data. A repeatable experiment is an abstraction that enables users to rerun a past experiment by managing to reproduce an execution environment, fetch the exact same data the original experiment runs on, and set the same configuration properties.

Hyperparameter tuning. Parameters that define and control a model architecture are called hyperparameters. Hyperparameter tuning is critical to achieving the best accuracy for ML and DL models. In hyperparameter tuning, a trial is an experiment with a given set of hyperparameters that returns its result as a metric. In synchronous hyperparameter tuning (using the Experiment API), the results of trials (metrics) are written to HopsFS [6], with which the driver reads the results and can then issue new jobs with new trials as Spark tasks to executors, iterating until hyperparameter optimisation is finished. Executing the code shown in Listing 5, six trials will be run with all possible combinations of learning rates and dropouts (using the grid search approach). Executors will run these trials in parallel, so if we run this grid search code with six executors, it is expected to complete six times faster than running the trials sequentially. HopsFS [6] is used to store results of the trials, logs, models trained, code, and any visualisation data for TensorBoard.

2.2.2. Parallel and Distributed Training Techniques

DL can greatly benefit in performance from executing training tasks on GPU-equipped hardware. In addition to this, model training can be accelerated even more by distributing

tasks on multiple GPUs of a cluster. The Hopsworks platform provides GPUs as managed resources as users can request from one to potentially all the GPUs of the cluster and Hopsworks will allocate the resources to applications. Grid search and random search are two examples of hyperparameter tuning techniques that are parallelisable by nature, which means that various executors will perform different hyperparameter combination evaluations. If a particular executor sits idle, it will be reclaimed by the cluster, which also means that GPUs will be optimally used in the cluster. This is made possible by using Dynamic Spark Executors (https://hopsworks.readthedocs.io/en/latest/hopsml/experiment.html (accessed on 9 April 2022)).

Listing 5. Grid search for hyperparameter values using the Experiment API.

```
# RUNS ON THE EXECUTORS
def train(lr, dropout):
    def input_fn(): # treturn dataset
    optimizer = ...
    model = ...
    model.add(Conv2D(...))
    model.compile(...)
    model.fit(...)
    model.evaluate(...)

# RUNS ON THE DRIVER
Hparams = {'lr':[0.001, 0.0001], 'dropout': [0.25, 0.5, 0.75]}
experiment.grid_search(train, Hparams)
```

The Hopsworks platform provides parallelised versions of the grid search, random search, and state-of-the-art evolutionary optimisation algorithms that will automatically search for hyperparameters to iteratively enhance evaluation metrics for models, such as model accuracy. The pseudo-code snippet shown in Listing 6 demonstrates how to run a single experiment abstraction. Listing 7 shows how we can run parallel experiments. Note that the only difference between the two code snippets is that in Listing 7, we provide a dictionary of parameters as an argument for the `launch` function, and the Experiment API will take care of the parallel execution of each trial.

Listing 6. Single experiment.

```
def training_function():
    import tensorflow as tf
    # Import hops helper modules
    from hops import hdfs
    from hops import tensorboard
    dropout = 0.5
    learning_rate = 0.001
    # define the model...

    # Point to tfrecords dataset in your project
    dataset = tf.data.TFRecordDataset(hdfs.project_path() + '/Resources/train.tfrecords')
    logdir = tensorboard.logdir()
    metric = model.train(learning_rate, dropout, logdir)
    return metric

from hops import experiment
experiment.launch(training_function)
```

Listing 7. Parallel experiment.

```
args_dict = {'learning_rate': [0.001, 0.0005, 0.0001], 'dropout': [0.45, 0.7]}

def training_function(learning_rate, dropout):
    # Training code - similar to the previous listing
    metric = model.train(learning_rate, dropout)
    return metric

from hops import experiment
experiment.launch(training_function, args_dict)
```

MultiWorkerMirroredStrategy. Once good hyperparameters have been found and a good model architecture has been designed, a model can be trained on a full dataset. If the training is slow, it can be sped up by adding more GPUs, potentially across multiple machines, to train in parallel using the data-parallel training approach. In data-parallel training, each worker (executor) trains on different shards of the training data. This type

of distributed training benefits significantly from having a distributed file system (shown in Figure 9—HopsFS [6] in Hopsworks) where workers can read the same training data and write to the same directories containing logs for all the workers, checkpoints for recovery if training crashes for some reason, TensorBoard logs, and any models that are produced at the end of the training.

Synchronous Stochastic Gradient Descent (SGD) is the state-of-the-art algorithm for the updating of weights in DL models, and it maps well to Spark's stage-based execution model. MultiWorkerMirroredStrategy is the state-of-the-art implementation of synchronous SGD as it is bandwidth optimal (using both upload and download bandwidths for all workers) compared to the parameter server model, which can be I/O bound at the parameter server(s). In the MultiWorkerMirroredStrategy, within a stage, each worker will read its share of a mini-batch, then send its gradients (changes to its weights as a result of the learning algorithm) to its successor on a ring while receiving gradients from its predecessor on the ring in parallel. Assuming all workers train on similar batch sizes per iteration and there are no stragglers, this approach can result in the near-optimal utilisation of GPUs. The MultiWorkerMirroredStrategy shown in Listing 8 demonstrates how the Experiment API is used for distributed training.

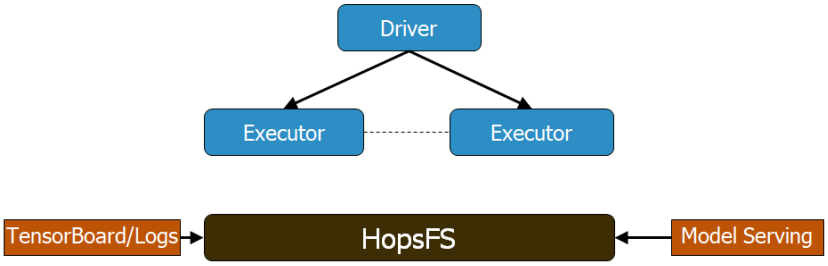

Figure 9. Distributed file system.

ParameterServerStrategy. Distributed training with parameter servers is another strategy supported by the Hopsworks platform. It is a common data-parallel approach in which, in addition to the workers, one or more parameter servers receive gradient or model parameter updates from the workers at each iteration, aggregate them, and send a new model replica or gradients to all the workers. The Experiment API also supports the ParameterServerStrategy as a distributed training approach. Listing 9 shows the code needed to use this strategy through the Experiment API.

2.2.3. Hyperparameter Tuning with MAGGY

MAGGY (source code: https://github.com/logicalclocks/maggy (accessed on 9 April 2022)) is a unified programming framework developed for the efficient asynchronous parallel execution of ML and DL experiments that supports the early stopping of under-performing trials by exploiting global knowledge, guided by an optimiser [4]. The programming model of MAGGY is based on distribution-oblivious training functions [15] with which users have to divide dataset creation and model creation code into their distinct functions and pass them as parameters to the training function. This enables the resulting parameterised code to be utilised and launched for a variety of experiment types and distribution settings (e.g., single-host or several workers) without requiring further code changes. For hyperparameter tuning tasks, the MAGGY framework currently includes algorithms for the existing implementations of random searches and Bayesian optimisations (with tree Parzen estimators [16] and Gaussian processes [17]), as well as Hyperband [18] and ASHA [14] as optimisers and a median early stopping rule for the early stopping of under-performing trials [19]. In addition, MAGGY provides a developer API that includes base classes for both the optimisers and the early stopping rule, allowing users and developers to implement and utilise their own optimisers or early stopping rules.

Listing 8. Using the MultiWorkerMirroredStrategy for distributed training in Hopsworks.

```
def multi_worker_mirrored_training():
    import sys
    import numpy as np
    import tensorflow as tf
    from hops import tensorboard
    from hops import devices
    from hops import hdfs
    import pydoop.hdfs as pydoop
    log_dir = tensorboard.logdir()
    # Define distribution strategy
    strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
    batch_size_per_replica = 8
    # Define global batch size
    batch_size = batch_size_per_replica * strategy.num_replicas_in_sync
    # Define model hyper parameters here

    # Input image dimensions
    img_rows, img_cols = 28, 28
    input_shape = (28, 28, 1)
    train_filenames = [hdfs.project_path() + "TourData/mnist/train/train.tfrecords"]
    validation_filenames = [hdfs.project_path() + "TourData/mnist/validation/validation.tfrecords"]

    # Construct model under distribution strategy scope
    with strategy.scope():
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Conv2D(...))
        model.add(tf.keras.layers.Conv2D(...))
        model.add(tf.keras.layers.MaxPooling2D(...))
        model.add(tf.keras.layers.Dropout(...))
        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(...))
        model.add(tf.keras.layers.Dropout(...))
        model.add(tf.keras.layers.Dense(...))
        opt = tf.keras.optimizers.Adadelta(...)
        model.compile(...)

# To finish the experiment
from hops import experiment
experiment.mirrored(multi_worker_mirrored_training, name='mnist model', metric_key='accuracy')
```

To optimise and tune hyperparameters, users must specify the hyperparameter tuning algorithm and the early-stopping rule (a no-stop rule, which does not early stop trials, is also implemented), as well as the search spaces of the hyperparameters (an example definition is shown in Listing 10). In the MAGGY framework, the Spark driver and executors communicate with each other via Remote Procedure Calls (RPCs). The flow of data for the underlying communication protocol, as well as the associated runtime behavior, is shown in Figure 10. The optimiser that guides effective hyperparameter searches is located on the Spark driver, and it assigns trials to the Spark executors. The Spark executors run a single long-running task and receive commands from the driver (optimiser) for trials to execute. Executors also periodically send metrics to the driver to enable the optimiser to make global early stopping decisions. Because of the impedance mismatch between trials and the stage- or task-based execution model of Spark, we block the executors with long-running tasks to run multiple trials per task. In this way, the Spark executors are always kept busy running trials (see Figure 11), and the global information needed for an efficient early stopping is aggregated in the optimiser. This results in improving the overall resource utilisation and the execution speeds of the experiments.

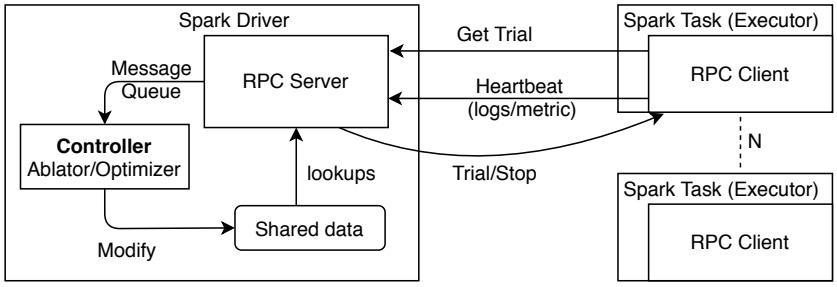

Figure 10. The MAGGY framework's runtime behaviour and data flow for the RPC protocol [4].

Listing 9. Using the ParameterServerStrategy for distributed training in Hopsworks.

```
def parameter_server_training():
    import sys
    import numpy as np
    import tensorflow as tf
    from hops import tensorboard
    from hops import devices
    from hops import hdfs
    import pydoop.hdfs as pydoop
    log_dir = tensorboard.logdir()
    # Define distribution strategy
    strategy = tf.distribute.experimental.ParameterServerStrategy(
        num_gpus_per_worker=devices.get_num_gpus()
        )
    batch_size_per_replica = 8
    # Define global batch size
    batch_size = batch_size_per_replica * strategy.num_replicas_in_sync
    # Define model hyper parameters here

    # Input image dimensions
    img_rows, img_cols = 28, 28
    input_shape = (28, 28, 1)
    train_filenames = [hdfs.project_path() + "TourData/mnist/train/train.tfrecords"]
    validation_filenames = [hdfs.project_path() + "TourData/mnist/validation/validation.tfrecords"]

    # Construct model under distribution strategy scope
    with strategy.scope():
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Conv2D(...))
        model.add(tf.keras.layers.Conv2D(...))
        model.add(tf.keras.layers.MaxPooling2D(...))
        model.add(tf.keras.layers.Dropout(...))
        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(...))
        model.add(tf.keras.layers.Dropout(...))
        model.add(tf.keras.layers.Dense(...))
        opt = tf.keras.optimizers.Adadelta(...)
        model.compile(...)

# To finish the experiment
from hops import experiment
experiment.parameter_server(parameter_server_training, name='mnist model', metric_key='accuracy')
```

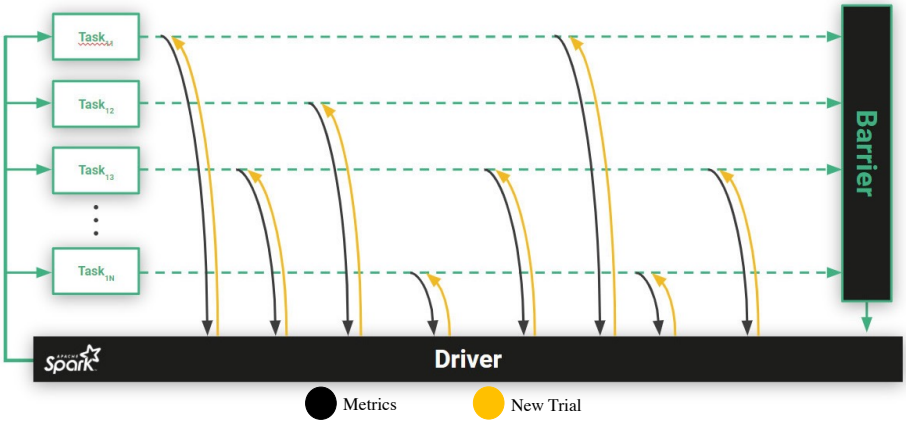

Figure 11. Asynchronous search using the MAGGY framework.

2.2.4. Ablation Studies

Ablation studies provide insights into the relative contributions of various architectural and regularisation components to the performance of ML models. These components include dataset features and model layers, although an ablation study might also include anything from a design choice to a system module. By removing each building block (e.g., a particular layer of the network architecture or a set of features of a training dataset), retraining, and observing the resulting performance, we can gain insights into the relative contributions of each of these building blocks. For ablation studies, the Hopsworks platform has been extended with AUTOABLATION, a framework for automated parallel ablation studies [20]. AUTOABLATION runs on top of the MAGGY framework and includes a Leave One Component Out (LOCO) ablator that removes a certain component from the training

process at a time (i.e., in each trial). A component can be considered one or a group of network architecture layers, one or a group of dataset features, and modules of network architecture, such as Inception v3 inception modules [21].

An ablation study, similar to the concept of search spaces in hyperparameter optimisation experiments, is defined by a list of components that are to be ablated (i.e., excluded) and an ablation policy (e.g., LOCO) that dictates how the given components should be removed for each trial. Once the study is defined, AUTOABLATION will automatically create the corresponding relevant trials for the ablation study and run them in parallel. An ablation study can be thought of as an experiment consisting of a series of trials. As shown in Figure 12, each model ablation trial, for example, involves training a model with one or more of its layers (e.g., a component) removed. However, a feature ablation trial involves training a model with various dataset features and observing the corresponding results. In Section 3 we provide code snippets as examples for defining and running typical ablation experiments.

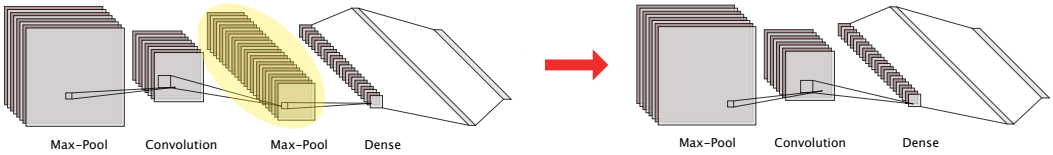

Figure 12. Example of a model (layer) ablation trial. The layer highlighted in yellow is removed from the base model. The subsequent model is trained on the same training data to determine the relative contribution of the ablated (removed) layer to the model's performance.

3. Results

In this section, we present the experimental evaluation results of the frameworks presented in Section 2, and in particular, the MAGGY and AUTOABLATION frameworks. We first present results for the task of hyperparameter tuning using the Experiment API as well as MAGGY and compare the two frameworks in terms of scalability and performance. Then, using AUTOABLATION, we demonstrate Hopsworks' support for automated ablation studies for DL training workloads.

3.1. Hyperparameter Tuning

For the task of hyperparameter tuning, we trained a three-layer Convolutional Neural Network (CNN) with a fully connected layer using the Fashion-MNIST dataset [22]. We compare the performance of a random search [23] using the MAGGY (asynchronous parallel execution of trials over Spark) framework to the Experiment API (synchronous parallel execution of trials over Spark). We run a fixed number of trials (N = 100) over 4, 8, 16, and 32 workers. The hyperparameter search space for this experiment is shown in Listing 10.

Listing 10. Defining hyperparameter search space for the Fashion-MNIST dataset.

```
sp = Searchspace(kernel=('INTEGER', [2, 8]),
    pool=('INTEGER', [2, 8]),
    dropout=('DOUBLE', [0.01, 0.99]),
    learning_rate=('DOUBLE', [0.000001, 0.99]))
```

As shown in Figures 13 and 14, the asynchronous execution of trials in MAGGY combined with the early stopping of under-performing trials using the median early stopping rule [19] reduced the wall-clock time by roughly half compared to using Spark or the Experiment API without compromising the final accuracies of the best trials. We can also see that increasing the number of workers linearly decreased the total execution time of the experiment for both MAGGY and Spark. The final best accuracies, after 100 trials, for both MAGGY and Spark are presented in both Table 1 as well as Figure 13. We can also see that both MAGGY and Spark converge to a comparable level of accuracy.

The effect of the early stopping of under-performing trials can be further emphasised by looking at Table 2, which consists of the total experiment running time in seconds

(wall-clock time) for MAGGY and Spark with a various number of workers as well as the number of early trials stopped by MAGGY. The median early stopping rule used for this experiment comes into effect after the first four trials are completed and stops trials that perform worse than the median at the same point in time (the stochastic-gradient descent optimisation step) during training. We can observe that when MAGGY uses the median early stopping rule, half of the trials are stopped on average, resulting in the reduction of the total wall-clock time by approximately half.

Table 1. The final accuracies of the MAGGY framework and Spark after 100 trials [4].

Number of Workers	MAGGY Accuracy	Spark Accuracy
4	0.915	0.905
8	0.909	0.912
16	0.909	0.913
32	0.913	0.909

Table 2. Relative speedup of MAGGY over the general Spark implementation (the Experiment API), total experiment runtime in seconds, and number of early trials stopped by MAGGY [4].

Number of Workers	MAGGY/Spark	MAGGY (s)	Spark (s)	Number of Early Stopped Trials
4	0.41	16,284	40,051	54
8	0.33	9828	29,511	52
16	0.47	6486	13,745	47
32	0.58	3804	6474	44

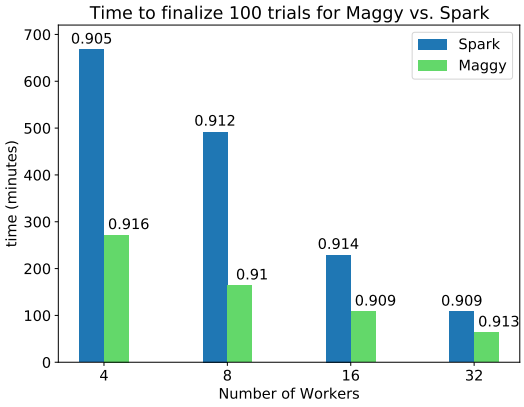

Figure 13. When compared to Spark (the Experiment API), the asynchronous execution of trials and the median stopping rule in MAGGY allow 100 hyperparameter trials to be executed in a lower wall-clock time (lower is better) without any loss in accuracy (denoted on top of the bars). We can also see that both MAGGY and Spark exhibit linear scalabilities (linear reductions in the experiment's wall-clock time) when more workers are added [4].

3.2. Ablation Studies

We now demonstrate how AUTOABLATION facilitates ablation experiments for DL through three common settings in which ablation studies are performed. The first two experiments focus on results from feature ablation and layer ablation experiments. The third experiment demonstrates the near-linear scalability of AUTOABLATION as more workers are added to the execution environment.

Experiment 1: Feature ablation of the Titanic dataset. Here we perform a feature ablation experiment on a modified version of the famous Titanic dataset (https://www.kaggle.com/c/titanic/data (accessed on 9 April 2022). The training dataset contains six features apart from the label, so we will have six trials where we exclude one training feature and one base trial containing all the features (i.e., seven trials in total). For this experiment, we use a simple Keras sequential model with two hidden `Dense` layers. To train

the model for 10 epochs, we divide the input dataset into training and testing sets with 80% and 20% of the data, respectively. As can be seen in Listing 11, defining this experiment in AUTOABLATION requires only a few lines of Python code.

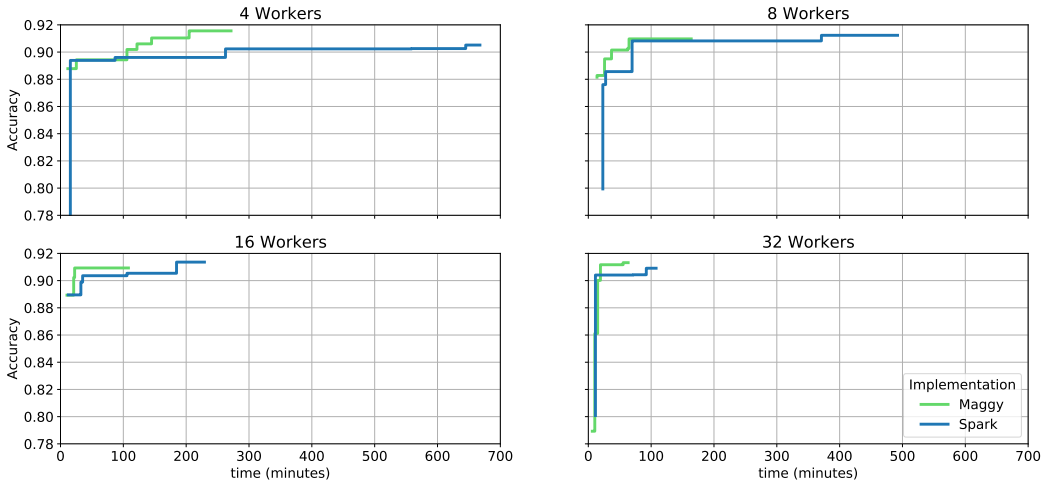

Figure 14. MAGGY finds better configurations faster through the asynchronous parallel execution and early stopping of under-performing trials. Due to shorter trials, MAGGY concludes experiments with the same number of trials in a shorter wall-clock time. Each trial in Spark, the Experiment API, is run to completion with no early stopping, producing a similar accuracy as expected and resulting in a higher wall-clock time needed to execute 100 trials than MAGGY [4].

Listing 11. Code snippet of defining feature ablation experiment in AUTOABLATION.

```
from maggy.ablation import AblationStudy
study = AblationStudy('titanic_train_dataset', label_name='survived')
list_of_features = ['pclass', 'fare', 'sibsp', 'sex', 'parch', 'age']
study.features.include(list_of_features)
```

This experiment is repeated five times, after which the training features are ranked according to their average effect on test accuracy achieved, as shown in Table 3. We can see from the results that training the model with all the dataset's features (none ablated) resulted in the worst test accuracy. The model trained on the dataset obtained by removing the `fare` feature, however, has the highest test accuracy.

Table 3. A ranking of the average accuracies in the test set in ascending order after ablating each feature from the training set [20].

Ablated Features	Test Accuracy
None (base trial)	0.583
pclass	0.596
sex	0.609
sibsp	0.616
age	0.667
parch	0.672
fare	0.695

Experiment 2: Model ablation of a Keras sequential CNN model. Here, we perform a layer ablation study on a CNN classifier model for the MNIST dataset [24]. The underlying CNN model consists of two `Conv2D` layers followed by a `MaxPooling2D` layer, one `Dropout` layer, a `Flatten` layer, one `Dense` layer, and another `Dropout` layer before the output layer of the network. In our evaluation, we are mainly interested in the relative contributions of

the second `Conv2D` layer, the `Dense` layer, and the two `Dropout` layers to the performance of the model. Listing 12 shows the AUTOABLATION code for defining the layer ablation study. To effectively evaluate the model, we repeat the experiment five times and then rank the selected layers according to their average effects on the test accuracy, as shown in Table 4. The results show that for this network dataset pair, removing the second `Conv2D` layer has the lowest effect on the test accuracy. However, the model produced from removing the `Dropout` layers performs better than the base model.

Listing 12. Code for the ablation experiment of the CNN classifier model.

```
from maggy.ablation import AblationStudy
study = AblationStudy("mnist", 1, "number",)
study.model.layers.include('second_conv', 'first_dropout', 'dense_layer', 'second_dropout')
```

Table 4. A ranking of the average accuracies of the test set in ascending order resulting from excluding layers of interest from the base model [20].

Ablated Layer	Test Accuracy
second_conv	0.913
dense_layer	0.954
None (base trial)	0.969
second_dropout	0.982
first_dropout	0.988

Experiment 3: Model ablation of Inception v3. This experiment demonstrates how the parallel execution of ablation trials using AUTOABLATION can provide a near-linear scalability. We perform an ablation study on seven modules of the Inception v3 network [21] on a subset of the TenGeoPSAR dataset [25] split into training, validation, and testing sets with 3200, 800, and 1000 images, respectively. Each image is labeled with one out of ten classes that correspond to different geophysical phenomena.

In this experiment, we use an Inception v3 network that has been pre-trained on ImageNet [26] and then change the output layer to suit our 10-class classification task. We perform a module ablation study on the first seven blocks of this network, which consists of 11 blocks known as inception modules. Here, note that the inception network is pre-defined, and we load it from the DL framework (TensorFlow in this case); thus, we do not explicitly define how the layers and modules are structured. Hence, we first compile the network to learn about the names of different layers and identify the entrance and endpoints of each module. This can be done using the Keras library by plotting the architecture or simply observing the output of `model.summary()`. We define the ablation study using the code snippet shown in Listing 13 after we have identified the layers.

Listing 13. Module ablation experiment of the Inception v3 network.

```
from maggy.ablation import AblationStudy
study = AblationStudy("TenGeoPSARwv", 1, "type",)
study.model.add_module('max_pooling2d_1', 'mixed0')
study.model.add_module('mixed0', 'mixed1')
study.model.add_module('mixed1', 'mixed2')
...
study.model.add_module('mixed5', 'mixed6')
```

Each ablation trial consists of fine-tuning the network for 40 epochs on the TenGeoPSAR dataset. To demonstrate the scalability of our approach, we perform this experiment in three different settings: (*i*) a single executor without parallelisation, (*ii*) two executors, and (*iii*) four executors. Figure 15 shows the wall-clock time for each setting. To approximate the linear scalability, we take the wall-clock time of the sequential run as the baseline; however, keep in mind that since each trial trains a different model, the trials differ in terms of their wall-clock times. Figure 15 shows how AUTOABLATION achieves a near-linear scalability by running ablation trials in an asynchronous parallel fashion.

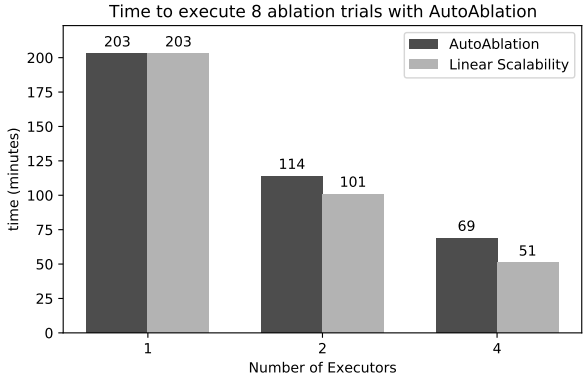

Figure 15. AUTOABLATION's near-linear scalability [20].

4. Discussion

The work we present in this paper is an early study showing the promise of the Hopsworks platform and advanced techniques targeting the EO and remote-sensing domains. Although there are many different application areas in EO, we covered only the Polar and Food Security use cases—but we believe that the techniques discussed in this paper can be generalised to cover and make contributions to different EO application areas. The results presented in this paper are compelling enough to make a case for this platform to be applied to various other application domains. This study explains the different components and features available in Hopsworks that the remote sensing and EO community can use. In addition to this, through experimental evaluations, we showed that using the MAGGY framework for hyperparameter tuning results in roughly half the wall-clock time required to execute the same number of hyperparameter tuning trials using Spark while providing a linear scalability as more workers are added. The work presented in this paper also demonstrated how AUTOABLATION facilitates the definition of ablation studies and enables the asynchronous parallel execution of ablation trials.

To the best of our knowledge, this is the first work that demonstrates the services and features of the Hopsworks platform, which provides users with the means to build scalable ML/DL pipelines for EO data as well as support for the discovery and search for EO metadata. While Hopsworks is a horizontal platform for developing and operating AI applications at scale, it has been customised for remote sensing and the EO community. Hopsworks is currently being used to train and operate machine-learning models at scale in other domains, such as finance, healthcare, and natural language processing. Although Hopsworks has not yet been used as a platform for other Copernicus TEPs, such as the Marine Environment Monitoring Service, we believe the platform can be used in a manner similar to the Polar and Food Security TEPs, that is, for scalable feature engineering, scale-out deep learning, and online model serving.

As future work, we will keep developing the Hopsworks platform to make it even more compatible with the advanced tools and methods used by researchers across the entire remote-sensing and EO communities. We will also continue the development of our use cases with more sophisticated DL models using even more advanced distributed DL training techniques. Note that this paper focuses the work of distributed learning on data parallelism. However, we believe our approach presented in this paper is applicable to broader tasks. Hence, as a natural extension of this work, it would be interesting to explore how other parallelisation methods (e.g., model parallelism and pipeline parallelism), could further improve distributed training speeds and resource utilisations.

5. Conclusions

In this paper, we introduce the Hopsworks platform and describe in detail how it can be used to enable massive-scale AI for EO data and other tasks, such as data-parallel and distributed DL by employing features that enhance its scalability, including the MAGGY

framework and the Feature Store. This work describes how the features of the Hopsworks platform are applied to EO data. To this end, this paper serves as a demonstration and walkthrough of the stages of building a production-level end-to-end ML/DL pipeline with a main focus on EO data utilising Hopsworks, which includes data ingestion, data preparation, feature extraction, model training, model serving, and monitoring. This demonstration is developed and presented in the context of the *ExtremeEarth* project. Within the context of *ExtremeEarth*, the Hopsworks platform has already been used to develop two use cases: sea ice classification for Polar TEPs and crop-type mapping and classification for Food Security TEPs.

Funding: This work was supported by the *ExtremeEarth* project (project website: http://earthanalytics.eu/ (accessed on 9 April 2022)), funded by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 825258.

Data Availability Statement: Everything about our work can be found here: https://www.hopsworks.ai/ (accessed on 9 April 2022).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hagos, D.H.; Kakantousis, T.; Vlassov, V.; Sheikholeslami, S.; Wang, T.; Dowling, J.; Fleming, A.; Cziferszky, A.; Muerth, M.; Appel, F.; et al. The ExtremeEarth Software Architecture for Copernicus Earth Observation Data. In Proceedings of the 2021 Conference on Big Data from Space, Publications Office of the European Union, Virtual Event, 18–20 May 2021.
2. Hagos, D.H.; Kakantousis, T.; Vlassov, V.; Sheikholeslami, S.; Wang, T.; Dowling, J.; Paris, C.; Marinelli, D.; Weikmann, G.; Bruzzone, L.; et al. ExtremeEarth Meets Satellite Data From Space. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2021**, *14*, 9038–9063. [CrossRef]
3. Kakantousis, T.; Kouzoupis, A.; Buso, F.; Berthou, G.; Dowling, J.; Haridi, S. Horizontally Scalable ML Pipelines with a Feature Store. In Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA, 31 March–2 April 2019.
4. Meister, M.; Sheikholeslami, S.; Payberah, A.H.; Vlassov, V.; Dowling, J. Maggy: Scalable Asynchronous Parallel Hyperparameter Search. In Proceedings of the 1st Workshop on Distributed Machine Learning, Barcelona, Spain, 1 December 2020; pp. 28–33.
5. Ismail, M.; Gebremeskel, E.; Kakantousis, T.; Berthou, G.; Dowling, J. Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; pp. 2525–2528.
6. Niazi, S.; Ismail, M.; Haridi, S.; Dowling, J.; Grohsschmiedt, S.; Ronström, M. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17), Santa Clara, CA, USA, 27 February–2 March 2017; pp. 89–104.
7. Andersson, R. GPU integration for Deep Learning on YARN. Master's Thesis, KTH Royal Institute of Technology, 21 August 2017.
8. Robbie, G.; Owen, C.; Yevgeni, L. Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning. 2018. Available online: https://eng.uber.com/petastorm/ (accessed on 9 April 2022).
9. Liu, S.; Wang, X.; Liu, M.; Zhu, J. Towards better analysis of machine learning models: A visual analytics perspective. *Vis. Inform.* **2017**, *1*, 48–56. [CrossRef]
10. Garg, N. *Apache Kafka*; Packt Publishing Ltd.: Birmingham, UK, 2013.
11. De la Rúa Martínez, J. Scalable Architecture for Automating Machine Learning Model Monitoring. Master's Thesis, KTH Royal Institute of Technology, 7 September 2020.
12. Wu, J.; Chen, X.Y.; Zhang, H.; Xiong, L.D.; Lei, H.; Deng, S.H. Hyperparameter optimization for machine learning models based on Bayesian optimization. *J. Electron. Sci. Technol.* **2019**, *17*, 26–40.
13. Bergstra, J.; Yamins, D.; Cox, D.D.; et al. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In Proceedings of the 12th Python in Science Conference, Austin, TX, USA, 24–29 June 2013; Volume 13, p. 20.
14. Li, L.; Jamieson, K.; Rostamizadeh, A.; Gonina, E.; Hardt, M.; Recht, B.; Talwalkar, A. A system for massively parallel hyperparameter tuning. *arXiv* **2018**, arXiv:1810.05934.
15. Meister, M.; Sheikholeslami, S.; Andersson, R.; Ormenisan, A.A.; Dowling, J. Towards Distribution Transparency for Supervised ML with Oblivious Training Functions. In Proceedings of the Workshop on MLOps Systems, Austin, TX, USA, 2–4 March 2020.
16. Bergstra, J.; Yamins, D.; Cox, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In Proceedings of the International Conference on Machine Learning, Atlanta, GA, USA, 17–19 June 2013; pp. 115–123.
17. Ginsbourger, D.; Janusevskis, J.; Le Riche, R. Dealing with Asynchronicity in Parallel Gaussian Process Based Global Optimization. Ph.D. Thesis, Mines Saint-Etienne, Saint-Étienne, France, 2011.
18. Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.* **2017**, *18*, 6765–6816.

19. Prechelt, L. Early stopping-but when? In *Neural Networks: Tricks of the Trade*; Springer: Berlin/Heidelberg, Germany, 1998; pp. 55–69.
20. Sheikholeslami, S.; Meister, M.; Wang, T.; Payberah, A.H.; Vlassov, V.; Dowling, J. AutoAblation: Automated Parallel Ablation Studies for Deep Learning. In Proceedings of the 1st Workshop on Machine Learning and Systems, Online, 26 April 2021; pp. 55–61.
21. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 2818–2826.
22. Xiao, H.; Rasul, K.; Vollgraf, R. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv* **2017**, arXiv:1708.07747.
23. Bergstra, J.; Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* **2012**, *13*, 281–305.
24. LeCun, Y. The MNIST Database of Handwritten Digits. 1998. Available online: http://yann.lecun.com/exdb/mnist/ (accessed on 9 April 2022).
25. Wang, C.; Mouche, A.; Tandeo, P.; Stopa, J.E.; Longépé, N.; Erhard, G.; Foster, R.C.; Vandemark, D.; Chapron, B. A labelled ocean SAR imagery dataset of ten geophysical phenomena from Sentinel-1 wave mode. *Geosci. Data J.* **2019**, *6*, 105–115. [CrossRef]
26. Deng, J.; Dong, W.; Socher, R.; Li, L.; Li, K.; Fei-Fei, L. Imagenet: A Large-Scale Hierarchical Image Database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255.