MDPI

Article

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 H. Payberah[1], Moritz Meister[2], Robin Andersson[2], and Jim Dowling[1,2]

[1] KTH Royal Institute of Technology, Division of Software and Computer Systems, Stockholm, Sweden; e-mail: {destah, sinash, tianzew, vladv, payberah, jdowling}@kth.se).
[2] Logical Clocks AB, Stockholm, Sweden; e-mail: {theo, moritz, robin, jim}@logicalclocks.com).
* 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 an open-source data-intensive and scalable Artificial Intelligence (AI) platform jointly developed by Logical Clocks and KTH Royal Institute of Technology for building end-to-end Machine Learning (ML)/Deep Learning (DL) pipelines for EO data. It provides a 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 that provide users with the means of building scalable end-to-end ML/DL pipelines for EO data, as well as support for discovery and search for EO metadata. This paper serves as a demonstrator 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 in 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 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**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

1. Introduction

In recent years, huge volumes of big data have been generated in various domains. Sentinel satellites[1], for example, collect more than three petabytes of sentinel EO data yearly for Copernicus, the European Union's flagship EO programme for monitoring the planet earth and its environment. This enormous 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 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

[1] https://www.copernicus.eu/en

platform is integrated with the 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[2] platform, an open-source AI platform jointly developed by Logical Clocks[3] and KTH Royal Institute of Technology[4], and describe in detail how it can be used to enable massive-scale AI for EO data and other tasks like data parallel and distributed DL by employing features that enhance its scalability such as Feature Store [3] and MAGGY framework [4]. Hopsworks is an open-source[5] AI platform that provides users with an execution environment for designing, distributing training, and running end-to-end ML/DL pipelines at scale. This work describes how the features of the Hopsworks platform are applied for EO data. Hopsworks also has the most-scalable distributed hierarchical filesystem that enables users to reduce the storage cost for EO data named HopsFS. HopsFS stores EO data in the native back-end object storage infrastructure of Data and Information Access Services (DIAS) that is accessed via the S3 or Swift protocols[6]. These features provide excellent support for implementing scalable DL models to process enormous volumes of EO data generated from various sources.

To this end, this paper serves as a demonstrator and walk-through of the stages of building a production-level end-to-end ML/DL pipelines with the main focus on EO data utilizing Hopsworks, which includes data ingestion, data preparation, feature extraction, model training, model serving, and monitoring. This demonstrator is developed and presented in the context of the *ExtremeEarth* project[7]. Our experience of using Hopsworks in the project is explained below. Two applications, sea ice classification and crop type mapping and classification have already been developed using the software platform architecture mentioned above by utilizing the petabytes of big Copernicus satellite data made available through the Copernicus EO programme and infrastructure [1].

Experience of using Hopsworks in *ExtremeEarth* project. One of the primary factors that differentiate 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 Polar use case that uses the Sentinel-1 SAR images for snow parameters and Food Security use case that uses Sentinel-2 images. The overall *ExtremeEarth* infrastructure and its multi-layer architecture has been presented in our previous work [2]. In this paper, we focus only on the processing layer of the *ExtremeEarth* software infrastructure. The processing layer of the *ExtremeEarth* software infrastructure provides the Hopsworks data-intensive AI platform that brings scalable AI support for EO data. For more detailed information about the different architecture layers and integration of the associated components, including Hopsworks and the detailed flow of events of the *ExtremeEarth* infrastructure, we refer the readers 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 of building scalable ML/DL pipelines for EO data, as well as support for 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.

[2] https://www.hopsworks.ai/
[3] https://www.logicalclocks.com/
[4] https://www.kth.se/en
[5] https://github.com/logicalclocks/hopsworks
[6] https://creodias.eu/data-access-interfaces
[7] http://earthanalytics.eu/

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*, *build and validate the model (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.

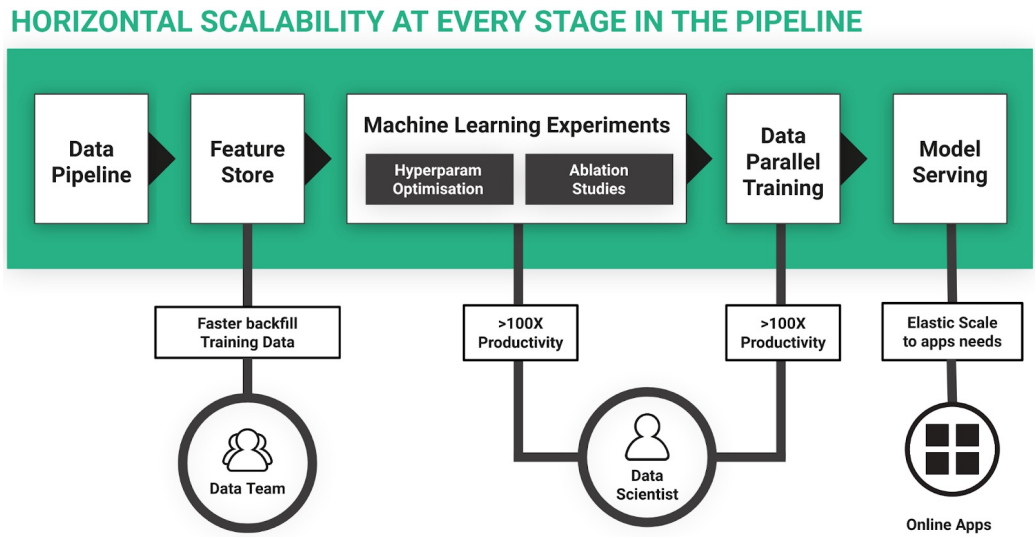

Figure 1. Horizontally scalable end-to-end ML/DL pipeline stages.

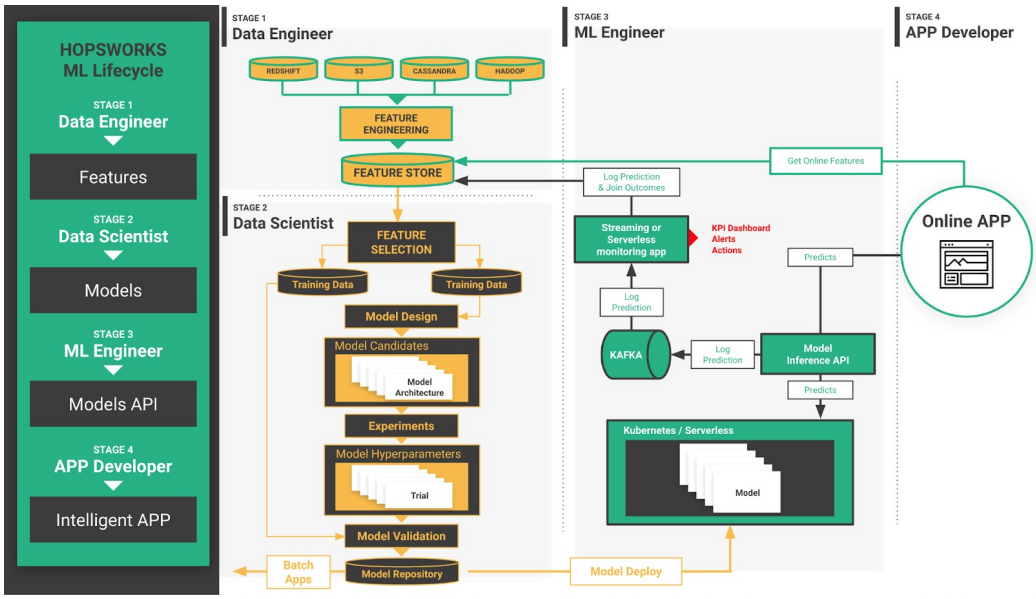

Figure 2. Hopsworks lifecycle of an end-to-end ML/DL pipeline.

The Hopsworks platform is a data-intensive AI platform that brings scalable AI support for big Copernicus 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 life cycle is formed, which continuously iterates the above ML pipeline. As a result, data scientists are faced with the highly complex task of developing DL workflows that utilize each stage of the ML/DL pipeline. The complexity of such pipelines can grow as the input data increases 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 used to implement it and demonstrated in the rest of the paper.

The stages of data ingestion, pre-processing, and management of a service to store curated feature data and compute features can be considered to be a 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 input for training, with the file format depending on the ML framework that is used. This step can be considered as the data science lifecycle, where new feature data is fetched, and new models are iteratively developed and pushed to production.

One of the main goals of an end-to-end ML/DL pipeline is to continuously improve the output models by using user-defined metrics. To detect when the 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 utilizes the flavor of YARN that is developed within Hopsworks as the resource management service for deploying distributed applications on a cluster of servers. YARN in Hopsworks supports scheduling applications with resource constraints such as CPU, main memory, and GPU [7]. Therefore, Spark in Hopsworks is deployed on top of YARN, 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. That is particularly important for allocating GPUs when the Spark program needs access to GPU compute power.

The Hopsworks platform has also been extended with an API that allows clients to submit Spark applications on the cluster easily. Hopsworks supports Apache Spark-as-a-Service that automatically sets up default Spark configuration parameters. It also provides a flexible way for users to provide additional ones 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 arbitrary programming languages functionality and run Python jobs and notebooks. Additional services like providing tools for debugging logs and metrics monitoring are part of this layer. The next layer comprises Hopsworks itself, the Webapp with the REST API that provides client applications and users connectivity to the entire Hopsworks cluster.

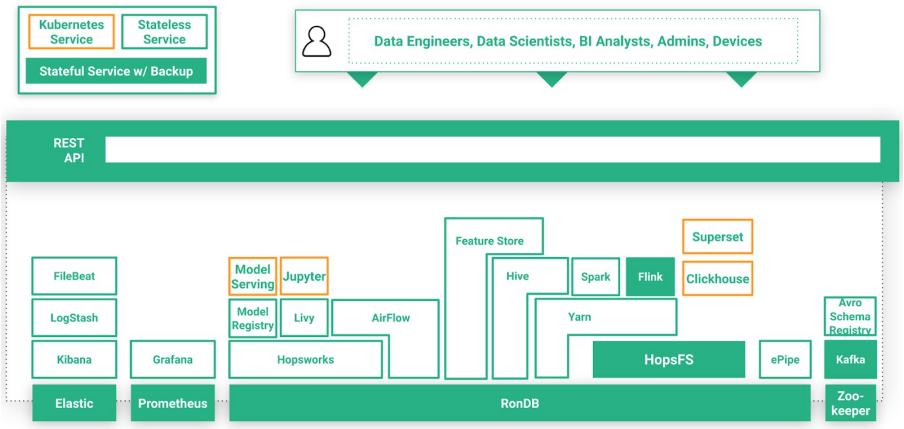

Figure 3. Hopsworks Services Stack.

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 is ingested into the system. In case of failures, orchestrating the order of stage execution, monitoring progress, and putting a retry mechanism in place is essential for making an EO data pipeline production-ready. To this end, Hopsworks integrates Apache Airflow[8] as an orchestration engine.

Application areas of the Hopsworks platform. In prior work, we showed how the Hopsworks platform was used to engineer features, train, and serve DL models on many terabytes of Copernicus data EO data [1,2]. While Hopsworks is a horizontal platform for developing and operating AI applications at scale, it has been customized 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 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 maritime 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 only covered the Polar and Food Security use cases, but we believe that the techniques discussed in this paper can generalize 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 in 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. Firstly, 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 low cost on local disks or on top of an object store. Secondly, the platform must support scaling out compute (feature engineering with CPUs and training deep learning models with GPUs). The result of independently scalable storage and scalable compute is that as the data volume grows, the platform can store the increased volumes of data, process the data into features in a similar time by adding more resources (CPU and memory) and the time required to

[8] https://airflow.apache.org/

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 summarize, scalable AI means that we can go beyond the capacity of a single server, and we can use many servers to reduce the training and inference times reasonably and be able to train larger models. As presented in Section 3, our work has introduced a model parallel training approach where the model training is partitioned over many servers where the models do not fit in a single server.

Contributions. In summary, our main contributions in this work are presented as follows.

- We introduce the Hopsworks platform that 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 that provide users with the means of building scalable end-to-end ML/DL pipelines for EO data, as well as support for 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 includes source code that implements the functionality in the platform.
- We also 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 organized as follows. The scalable AI capabilities of the Hopsworks platform and its features for scalable end-to-end ML/DL pipeline stages for EO data are explained in sufficient detail in Section 2. Experimental settings and evaluation results of the frameworks discussed in this paper are presented in Section 3. In Section 4, we summarize 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 Hopsworks frameworks and its features for scalable ML.

2.1. ML/DL Pipeline Stages

2.1.1. Data Ingestion

Locating and identifying the sources of input data to the AI platform is the first step in building a scalable end-to-end ML/DL pipeline. The next stage is to develop advanced methods for ingesting data from the input sources into the underlying AI platform, which runs the ML/DL pipelines. The format in which the input data is 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 on 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 images data for both Polar and Food Security use cases easily ingested in the platform for subsequent processing in the context of the *ExtremeEarth* project.

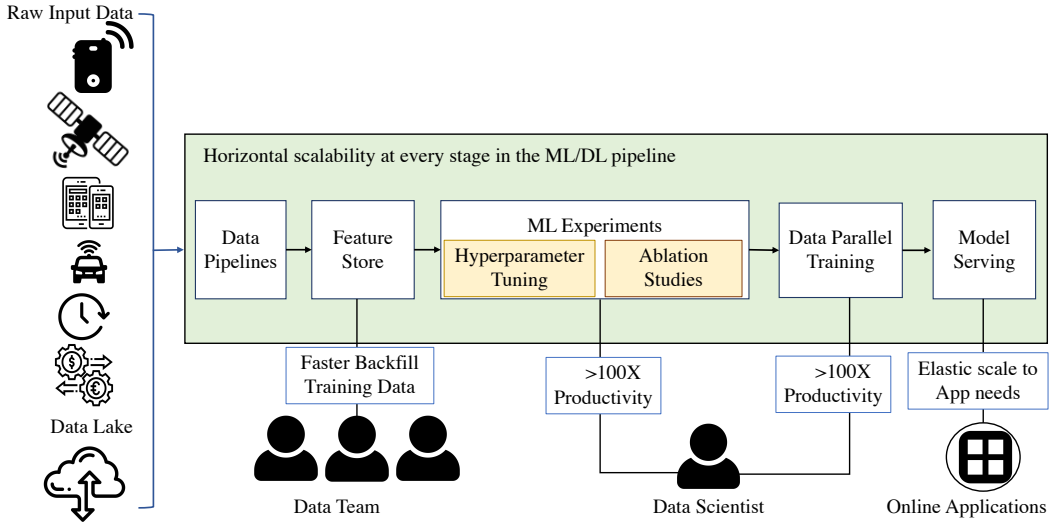

Figure 4. Data ingestion sources for an ML/DL pipeline for EO data in the Hopsworks platform [2].

Accessing EO Data from Hopsworks. In the context of the *ExtremeEarth* project, the Hopsworks platform is deployed on the CREODIAS, one of the DIAS of the Copernicus programme, environment where EO data is needed for this project resides. In the CREODIAS environment, EO data is made available via an object store where it can be accessed via the S3 protocol implemented by OpenStack Swift[9], or it can be accessed via standardized web services such as WMS/WMTS/WCS/WFS[10].

2.1.2. EO Data Pre-Processing

EO data pre-processing with Python. Satellite data often needs to be processed before being provided as input to ML/DL algorithms. By utilizing the Python support the Hopsworks platform provides, data scientists in *ExtremeEarth* can use utility programs such as GDAL[11] 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 use 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 utilize 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 feature engineering and decreases the complexity of managing features. As a result, it 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 organize their ML assets and curate the EO data features to enhance the management of curated feature data.

[9] https://docs.openstack.org/swift/latest/s3_compat.html
[10] https://creodias.eu/data-access-interfaces
[11] https://gdal.org/

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 containerized applications[12]. The generated features that can be utilized to create ML/DL models must be defined, computed, and persisted before using the validated data to develop them. In the context of the *ExtremeEarth* software architecture on EO data, the underlying service that data engineers and data scientists utilize for such tasks is the Hopsworks Feature Store. In order to deal with large amounts of data and complex data types and relations, the Feature Store provides comprehensive APIs, scalability, and elasticity. Utilizing 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 organization 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 organization's feature data, reusing generated features across different pipelines, applying software engineering principles to ML features with versioning, documentation, access control tools, and time-travel by fetching previous features data used for training a particular model are just a few of the Feature Store's main benefits. In addition to this, it also 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.

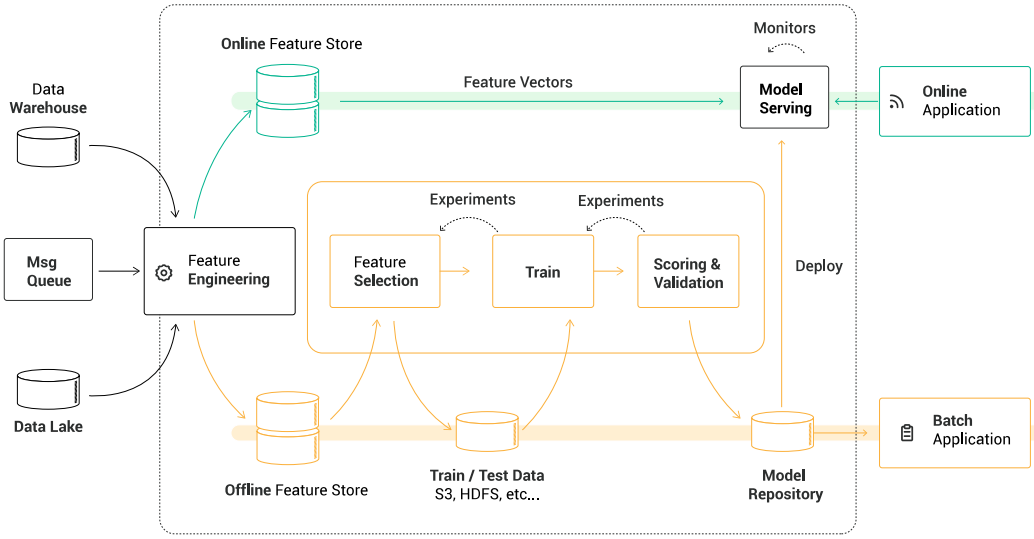

Figure 5. The Feature Store interface of Hopsworks and its core components.

The Feature Store [3] is built on fault-tolerant, distributed and scalable services to achieve all of the main properties mentioned above. Online data is 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 utilized in batch-oriented data processing, where previous feature data can be evaluated to provide meaningful statistics. For pipelines that require data at prediction time, online features must be available in real-time. In addition to storing data, the Feature Store uses Hopsworks Spark integration to compute and analyze features. The key components of the Hopsworks Feature Store are shown in Figure 6.

12 https://www.hopsworks.ai/feature-store

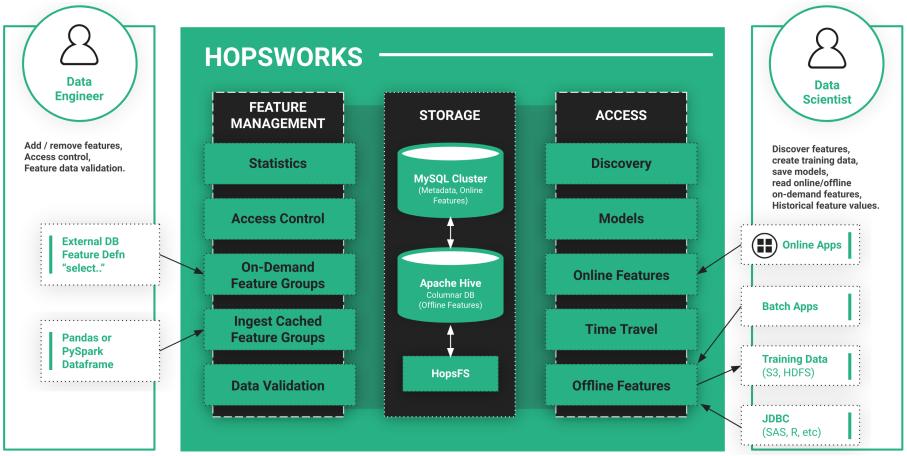

Figure 6. The Hopsworks platform Feature Store architecture [2].

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 type and structure, statistical properties, and values (such as the 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 input 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[13]. This validation framework is built on Apache Spark and Deequ[14].

2.1.4. Model Analysis and Training

Model analysis is an analytical method for 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 between each other against slices of data or data points. Building ML models is an iterative process, and note that the quality of the 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 the feedback from model validation. Subsequently, distributing training over enormous datasets to develop and deploy ML models is not easy. It is often necessary to validate the input data to the training algorithm and the model developed from this data. Hence, Hopsworks has been extended to include the *What-If Tool*[15], a simple interactive visual interface for expanding the understanding of a black-box regression or classification ML model.

[13] https://docs.hopsworks.ai/latest/generated/feature_validation/
[14] https://github.com/awslabs/deequ
[15] https://github.com/pair-code/what-if-tool

Model interpretability is concerned with understanding what an ML model performs, including pre-and post-processing stages and its behavior 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 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 analyze the model, and the features to be used.

```
# 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)
```

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

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 utilizing the previous stages in the ML/DL pipeline, it must be served to be used for inference by external applications such as iceberg detection or water availability detection in 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 an extension [2]. KFServing[16] has been integrated to Hopsworks for model serving. Users can use docker containers provided by either Hopsworks or KFServing.

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 runtime, which gives Hopsworks users the vital property of elasticity. Besides, the Hopsworks platform model monitoring infrastructure continuously monitors the incoming requests sent to the 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*.

[16] https://hopsworks.readthedocs.io/en/latest/hopsml/kfserving.html

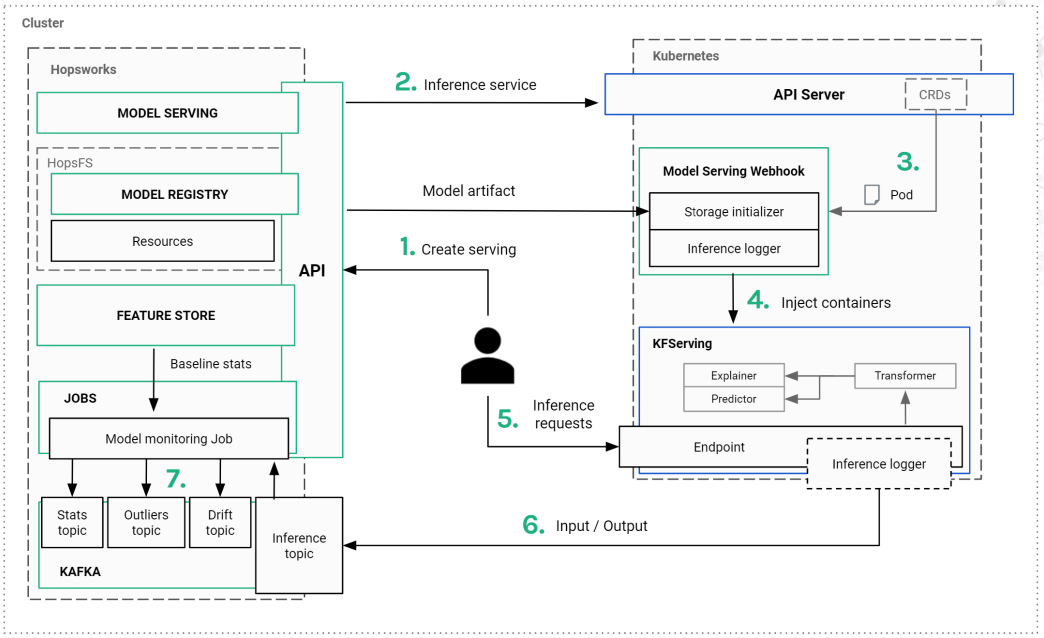

Figure 7. The architecture for model monitoring and serving in the Hopsworks platform [11].

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

```
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 3. Creating model serving.

```
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)
```

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

2.2. Hopsworks Frameworks for Scalable ML

In this section, we will explain and summarize 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 here 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 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 execution through the Experiment API, as well as a new framework for asynchronous parallel execution of trials, called MAGGY [4]. To optimize the hyperparameters for DL, out-of-the-box use of *state-of-the-art* directed search algorithms that work better (e.g., genetic algorithms, Bayesian optimization [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.

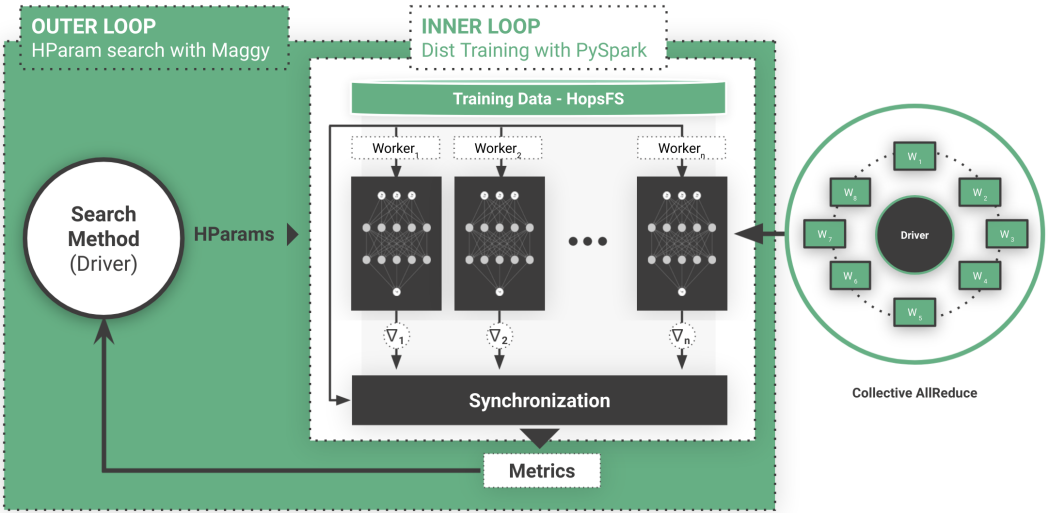

Figure 8. Scaling out distributed training of ML and DL.

2.2.1. The Experiment API

An experiment can be defined as the process of finding the optimal hyperparameters for model training, 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. Hopsworks 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, or any other framework that provides a Python API. Furthermore, it allows for managing the history of ML experiments, as well as monitoring during training[17].

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 the ML program runs in to carry out ML training. In Hopsworks, the abstraction of an *Experiment* is used to encapsulate the aforementioned properties. To productionize ML models, it is important to easily run a past experiment in case a software bug was discovered and the models need to be developed again based on previously seen data. A repeatable experiment

17 https://hopsworks.readthedocs.io/en/latest/hopsml/hopsML.html#experimentation

is an abstraction that enables users to rerun a past experiment by managing to reproduce the 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 the 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], where the *Driver* reads the results and can then issue new jobs with new trials as Spark tasks to *Executors*, iterating until hyperparameter optimization is finished. Executing the code shown on Listing 5, six trials will be run with all possible combinations of learning rate and dropout (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 visualization data for TensorBoard.

```
# 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)
```

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

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 a managed resource, as users can request from one to potentially all the GPUs of the cluster, and Hopsworks will allocate the resource to applications. Grid search and random search are two examples of hyperparameter tuning techniques that are parallelizable by nature, which means that various *Executors* will perform different hyperparameter combinations 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 Dynamic Spark executors[18].

Hopsworks provides parallelized versions of the grid search, random search, or *state-of-the-art* evolutionary optimization algorithms that will automatically search for hyperparameters to iteratively enhance evaluation metrics for our 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 to the `launch` function, and the Experiment API will take care of the parallel execution of each trial.

[18] https://hopsworks.readthedocs.io/en/latest/hopsml/experiment.html

```
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 6. Single 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)
```

Listing 7. Parallel Experiment.

MultiWorkerMirroredStrategy. Once good hyperparameters are found, and a good model architecture has been designed, a model can be trained on the full dataset. If 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 hugely 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 training.

Synchronous Stochastic Gradient Descent (SGD) is the current *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 current *state-of-the-art* implementation of synchronous SGD, as it is bandwidth-optimal (using both upload and download bandwidth for all Workers) compared to the Parameter Server model, which can be I/O bound at the Parameter Server(s). In *MultiWorkerMirroredStrategy*, within a stage, each worker will read its share of the mini-batch, then sends its gradients (changes to its weights as a result of the learning algorithm) to its successor on the 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 near-optimal utilization of GPUs. The *MultiWorkerMirroredStrategy* shown in Listing 8 demonstrates how the Experiment API is used for distributed training.

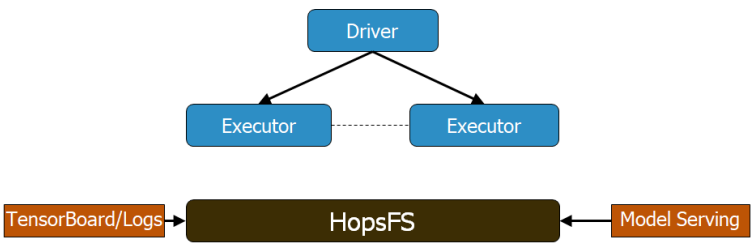

Figure 9. Distributed file system.

```
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')
```

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

ParameterServerStrategy. Distributed training with parameter server(s) 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 the new model replica or gradients to all the workers. The Experiment API also supports 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[19] is a unified programming framework developed for efficient asynchronous parallel execution of ML and DL experiments that supports early stopping of under-performing trials by exploiting global knowledge, guided by an optimizer [4]. The programming model of MAGGY is based on distribution oblivious training functions [15], in which the 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 parameterized code to be utilized and launched for a variety of experiment types and distribution settings (e.g., single-host or on several workers) without requiring further code changes. For hyperparameter tuning tasks, the MAGGY framework currently includes algorithms for the existing implementations of Random Search, Bayesian Optimization (with Tree Parzen Estimators [16] and Gaussian Processes [17]), as well as HyperBand [18] and ASHA [19] as optimizers, and a median early stopping rule for early stopping of under-performing trials [20]. In addition, MAGGY also provides a developer API that includes base classes for both the optimizers and the early stopping rules, allowing users and developers to implement and utilize their own optimizers or early stopping rules.

[19] Source Code. https://github.com/logicalclocks/maggy

```
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')
```

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

To optimize and tune hyperparameters, the 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 space 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 optimizer that guides for effective hyperparameter search 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* (optimizer) for trials to execute. *Executors* also periodically send metrics to the *Driver* to enable the optimizer to take global early stopping decisions. Because of the impedance mismatch between trials and the stage or task-based execution model of Spark, we are blocking *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 global information needed for efficient early stopping is aggregated in the optimizer. This results in improving the overall resource utilization and the execution speed of the experiments.

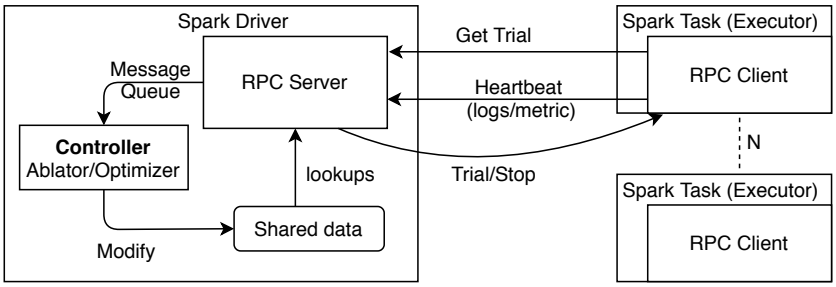

Figure 10. The MAGGY framework's runtime behavior and data flow for the RPC protocol [4].

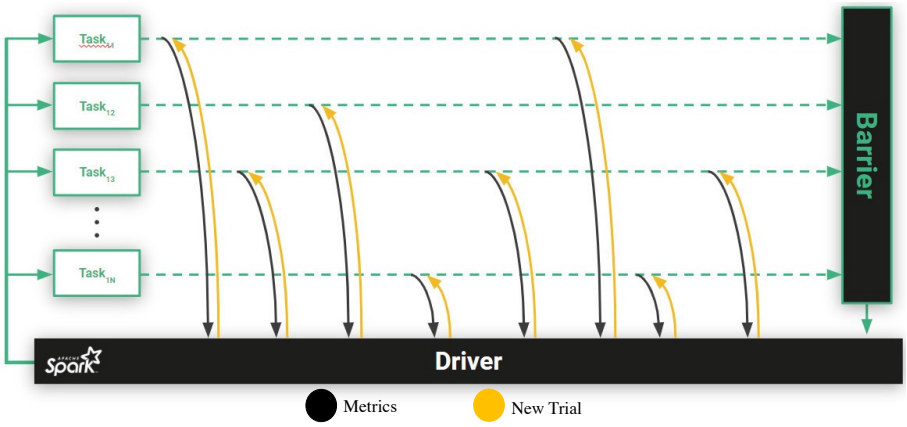

Figure 11. Directed Asynchronous Search using the MAGGY framework.

2.2.4. Ablation Studies

Ablation studies provide insights into the relative contribution of various architectural and regularization 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 the 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 [21]. AUTOABLATION runs on top of the MAGGY framework and includes a Leave-One-Component-Out (LOCO) ablator that removes a certain component out of the training process at a time (i.e., in each trial). A component can be considered as 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 [22].

An ablation study, similar to the concept of search space in hyperparameter optimization experiments, is defined by a list of the 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 trials. As shown in Figure12, each model ablation trial, for example, involves training a model with one or more of its layers (e.g., a component) removed. On the other hand, 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 study experiments.

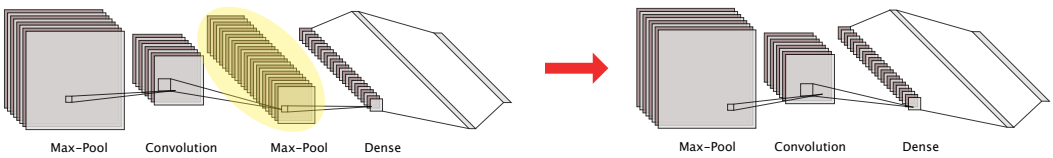

Figure 12. Example of a model (layer) ablation trial. The layer highlighted in yellow is removed from the base model. The subsequent model is trained on the same training data to determine the relative contribution of the ablated (removed) layer to the model's performance.

3. Results

In this section, we present 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 Experiments API as well as MAGGY, and compare the two frameworks in terms of their scalability and performance.

Table 1. The final accuracy 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 the number of early stopped trials by MAGGY [4].

Number of Workers	MAGGY/Spark	MAGGY (s)	Spark (s)	# Early Stopped
4	0.41	16284	40051	54
8	0.33	9828	29511	52
16	0.47	6486	13745	47
32	0.58	3804	6474	44

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 [23]. We compare the performance of random search [24] using the MAGGY (asynchronous parallel execution of trials over Spark) framework as opposed to using 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.

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

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

As shown in Figure 13 and 14 we can see that the asynchronous execution of trials in MAGGY combined with the early stopping of under-performing trials using the median early stopping rule [20] reduces the wall-clock time by roughly half compared to using Spark or the Experiment API, without compromising the final accuracy of the best trials. We can also see that increasing the number of workers linearly decreases the total execution time of the experiment for both MAGGY and Spark. The final best accuracy after 100 trials for both MAGGY and Spark is 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 early stopping of under-performing trials can be further emphasized 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 stopped trials 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 (stochastic gradient descent optimization 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 total wall-clock time by approximately half.

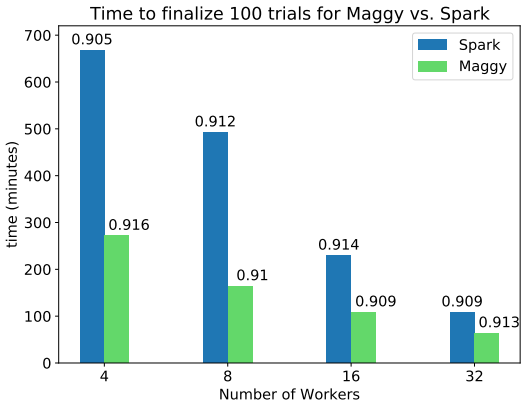

Figure 13. When compared to Spark (the Experiment API), asynchronous execution of trials and the median stopping rule in MAGGY allow 100 hyperparameter trials to be executed in lower wall-clock time (lower is better) without any loss in accuracy (denoted on top of the bars). We can also see that both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

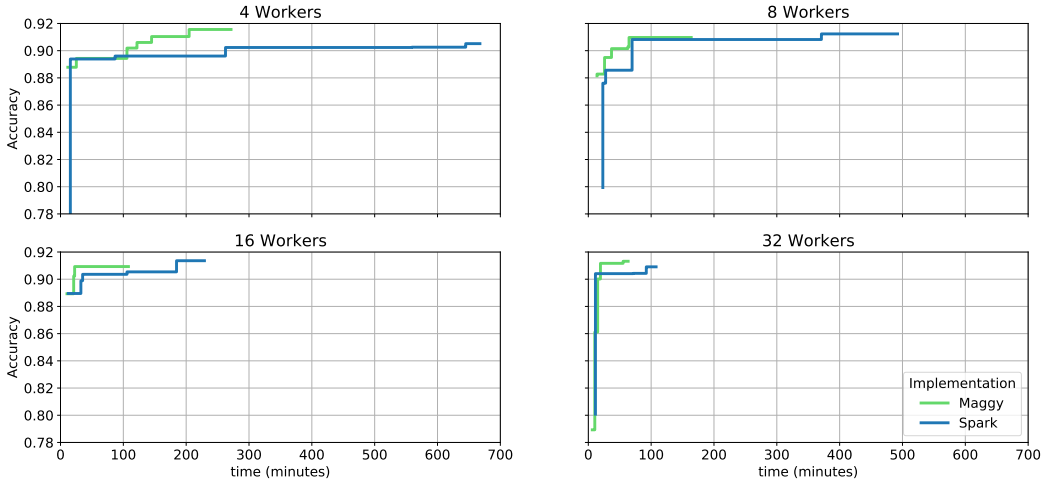

Figure 14. MAGGY finds better configurations faster through asynchronous parallel execution and early stopping of under-performing trials. Due to shorter trials, MAGGY concludes experiments with the same number of trials in shorter wall-clock time. Each trial in Spark, Experiment API, is run to completion, with no early stopping, producing similar accuracy as expected, resulting in a higher wall-clock time to execute 100 trials than MAGGY [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 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 the modified version of the famous Titanic dataset[20]. 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., thus seven trials in total. For this experiment, we use a simple Keras Sequential model with two hidden `Dense` layers.

[20] https://www.kaggle.com/c/titanic/data

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

```
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)
```

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

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 of the dataset's features (`None` ablated) resulted in the worst test accuracy. The model trained on the dataset obtained by removing the `fare` feature, on the other hand, has the highest test accuracy.

Table 3. The ranking of the average accuracy on the test set in ascending order after ablating each feature from the training set [21].

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 [25]. 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 contribution 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 effect on 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.

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

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

Table 4. The ranking of the average accuracy on the test set in ascending order resulting from excluding layers of interest from the base model [21].

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 near-linear scalability. We perform an ablation study on seven modules of the Inception-v3 network [26] on a subset of the TenGeoPSAR dataset [27] split into training, validation, and testing sets with 3200, 800, and 1000 images, respectively. Each image is labeled with one out of 10 classes that correspond to different geophysical phenomena.

In this experiment, we use an Inception-v3 network that has been pre-trained on ImageNet [28] 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 predefined 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 find out 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.

```
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')
```

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

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

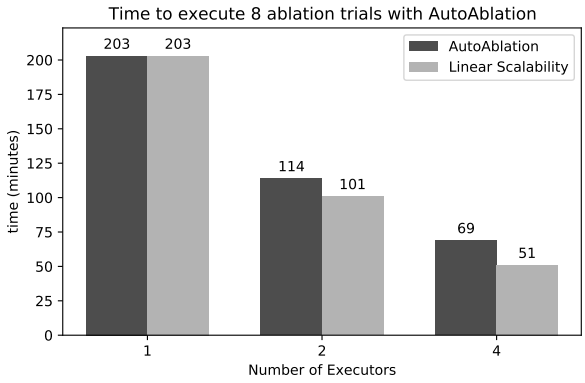

Figure 15. AUTOABLATION's near-linear scalability [21].

4. Discussion

The work we presented in this paper is an early study showing the promise of the Hopsworks platform and advanced techniques targeting the EO and remote sensing domain. Although there are many different application areas in EO, we only covered the Polar and Food Security use cases, but we believe that the techniques discussed in this paper can generalize 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 in 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 evaluation, 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 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 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 that provide users with the means of building scalable ML/DL pipelines for EO data, as well as support for discovery and search for EO metadata. While Hopsworks is a horizontal platform for developing and operating AI applications at scale, it has been customized for remote sensing and the EO community. Hopsworks is currently 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 maritime 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 community. 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 parallelization methods, e.g., model parallelism and pipeline parallelism, could further improve distributed training speed and resource utilization.

5. Conclusion

In this paper, we introduced the Hopsworks platform and described in detail how it can be used to enable massive-scale AI for EO data and other tasks like data parallel and distributed DL by employing features that enhance its scalability such as the MAGGY framework and Feature Store. This work describes how the features of the Hopsworks platform are applied for EO data. To this end, this paper serves as a demonstrator and walk-through of the stages of building a production-level end-to-end ML/DL pipelines with the main focus on EO data utilizing Hopsworks, which includes data ingestion, data preparation, feature extraction, model training, model serving, and monitoring. This demonstrator 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 the Polar TEPs and crop type mapping and classification for the Food Security TEPs.

Funding: This work is supported by the ExtremeEarth project[21] funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258.

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

[21] Project website: http://earthanalytics.eu/

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. Proceedings of the 2021 conference on Big Data from Space. Publications Office of the European Union, 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 Journal of Selected Topics in Applied Earth Observations and Remote Sensing* **2021**, *14*, 9038–9063.
3. Kakantousis, T.; Kouzoupis, A.; Buso, F.; Berthou, G.; Dowling, J.; Haridi, S. Horizontally Scalable ML Pipelines with a Feature Store. Proc. 2nd SysML Conf., Palo Alto, USA, 2019.
4. Meister, M.; Sheikholeslami, S.; Payberah, A.H.; Vlassov, V.; Dowling, J. Maggy: Scalable Asynchronous Parallel Hyperparameter Search. Proceedings of the 1st Workshop on Distributed Machine Learning, 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. 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, 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. 15th USENIX Conference on File and Storage Technologies (FAST 17), 2017, pp. 89–104.
7. Andersson, R. GPU integration for Deep Learning on YARN, 2017.
8. Robbie, G.; Owen, C.; Yevgeni, L. Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning. https://eng.uber.com/petastorm/, 2018.
9. Liu, S.; Wang, X.; Liu, M.; Zhu, J. Towards better analysis of machine learning models: A visual analytics perspective. *Visual Informatics* **2017**, *1*, 48–56.
10. Garg, N. *Apache kafka*; Packt Publishing Ltd, 2013.
11. de la Rúa Martínez, J. Scalable Architecture for Automating Machine Learning Model Monitoring, 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. *Journal of Electronic Science and Technology* **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. Proceedings of the 12th Python in science conference. Citeseer, 2013, Vol. 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 preprint arXiv:1810.05934* **2018**.
15. Meister, M.; Sheikholeslami, S.; Andersson, R.; Ormenisan, A.A.; Dowling, J. Towards Distribution Transparency for Supervised ML With Oblivious Training Functions. Workshop on MLOps Systems, 2020.
16. Bergstra, J.; Yamins, D.; Cox, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. International conference on machine learning. PMLR, 2013, pp. 115–123.
17. Ginsbourger, D.; Janusevskis, J.; Le Riche, R. Dealing with asynchronicity in parallel Gaussian process based global optimization. PhD thesis, Mines Saint-Etienne, 2011.
18. Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* **2017**, *18*, 6765–6816.
19. Li, L.; Jamieson, K.; Rostamizadeh, A.; Gonina, E.; Hardt, M.; Recht, B.; Talwalkar, A. Massively parallel hyperparameter tuning **2018**.
20. Prechelt, L. Early stopping-but when? In *Neural Networks: Tricks of the trade*; Springer, 1998; pp. 55–69.
21. Sheikholeslami, S.; Meister, M.; Wang, T.; Payberah, A.H.; Vlassov, V.; Dowling, J. AutoAblation: Automated Parallel Ablation Studies for Deep Learning. Proceedings of the 1st Workshop on Machine Learning and Systems, 2021, pp. 55–61.
22. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the inception architecture for computer vision. Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 2818–2826.
23. Xiao, H.; Rasul, K.; Vollgraf, R. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747* **2017**.
24. Bergstra, J.; Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research* **2012**, *13*.
25. LeCun, Y. The MNIST Database of Handwritten Digits, 1998. http://yann.lecun.com/exdb/mnist/.
26. Szegedy et al., C. Rethinking the Inception Architecture for Computer Vision. IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2818–2826.
27. Wang et al., C. A labelled ocean SAR imagery dataset of ten geophysical phenomena from Sentinel-1 wave mode. *Geoscience Data Journal* **2019**, *6*, 105–115.
28. Deng et al., J. Imagenet: A Large-Scale Hierarchical Image Database. 2009 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, 2009, pp. 248–255.