Scalable Artificial Intelligence for Earth Observation Data Using Hopsworks

Desta Haileselassie Hagos, Theofilos Kakantousis, Sina Sheikholeslami, Tianze Wang, Vladimir Vlassov, Amir H. Payberah, Moritz Meister, Robin Andersson, and Jim Dowling

***Abstract*—This paper introduces an extended version of Hopsworks to the Earth Observation (EO) data community and the Copernicus programme. Hopsworks is a data-intensive and scalable Artificial Intelligence (AI) platform for building end-to-end Machine Learning (ML)/Deep Learning (DL) 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 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 new 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. 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 provides 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. In addition to this, 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, ExtremeEarth, Food Security, Polar Regions**

I. INTRODUCTION

In recent years, unprecedented amounts of data are generated in diverse domains. Copernicus, a European Union's EO flagship programme for monitoring the planet earth and its environment, produces more than three petabytes of EO data annually from Sentinel satellites[1]. This data is made easily available to researchers that are using it, including to develop AI algorithms, in particular, using DL techniques that are appropriate for big data. However, one of the critical challenges that researchers are facing is the lack of advanced and emerging tools that can help them unlock the potential of this data flood and develop predictive and classification AI models. In our previous works [1, 2], as part of addressing this challenge, we present the *ExtremeEarth* software infrastructure that builds on seamless integration of both existing and novel software platforms and tools for storing, accessing, processing, analyzing, and visualizing large amounts of Copernicus EO data. Moreover, in [1, 2], we introduce Hopsworks[2]. and show how it is integrated with various services and platforms to extract knowledge from AI and build AI-based applications using Copernicus and EO data.

In this paper, we introduce the extended version of Hopsworks with EO data support by adding new features, such as the Feature Store and MAGGY framework, to enable data parallel distributed DL and to enhance Hopsworks scalability. To this end, this paper serves as a demonstrator and walk-through of the stages of building a production-level EO ML/DL pipeline using 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. Using the information provided by the Copernicus satellite data, *ExtremeEarth* brings together the food security and polar communities and is demonstrated in two use cases with societal, environmental, and financial values, namely, the Food Security and the Polar use cases. Two applications, sea ice classification (Polar use case) and crop type mapping and classification (Food Security use case), have already been developed using the architecture mentioned above by utilizing the petabytes of data made available through the Copernicus programme and infrastructure [1].

One of the main factors that differentiate the *ExtremeEarth* project from other Earth analytics techniques and technologies is the use of Hopsworks. The *ExtremeEarth* software infrastructure builds on the integration of European Space Agency (ESA) Thematic Exploitation Platforms (TEPs), Data and Information Access Services (DIAS), and Hopsworks. The *ExtremeEarth* infrastructure, as shown in Figure 1, consists of *four* layers with the TEPs in the *product layer*, Hopsworks in the *processing layer*, CREODIAS in the *data layer*, and Infrastructure as a Service (IaaS) in the *physical layer*. In this paper, we focus only on the processing layer of the *ExtremeEarth* software infrastructure. For more detailed

Desta Haileselassie Hagos, Sina Sheikholeslami, Tianze Wang, Vladimir Vlassov, Amir H. Payberah, and Jim Dowling are with the KTH Royal Institute of Technology, Stockholm, Sweden (e-mail: {destah, sinash, tianzew, vladv, payberah, jdowling}@kth.se).

Theofilos Kakantousis, Moritz Meister, Robin Andersson, and Jim Dowling are with Logical Clocks AB, Stockholm, Sweden (e-mail: {theo, moritz, robin, jim}@logicalclocks.com).

This is a technical report finalized in January 2022.

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

[2] Source Code: https://github.com/logicalclocks/hopsworks

information about the other layers and to see how the *ExtremeEarth* infrastructure has been architectured around Hopsworks to tackle large-scale ML and DL challenges, we refer the readers to our previous works [1, 2].

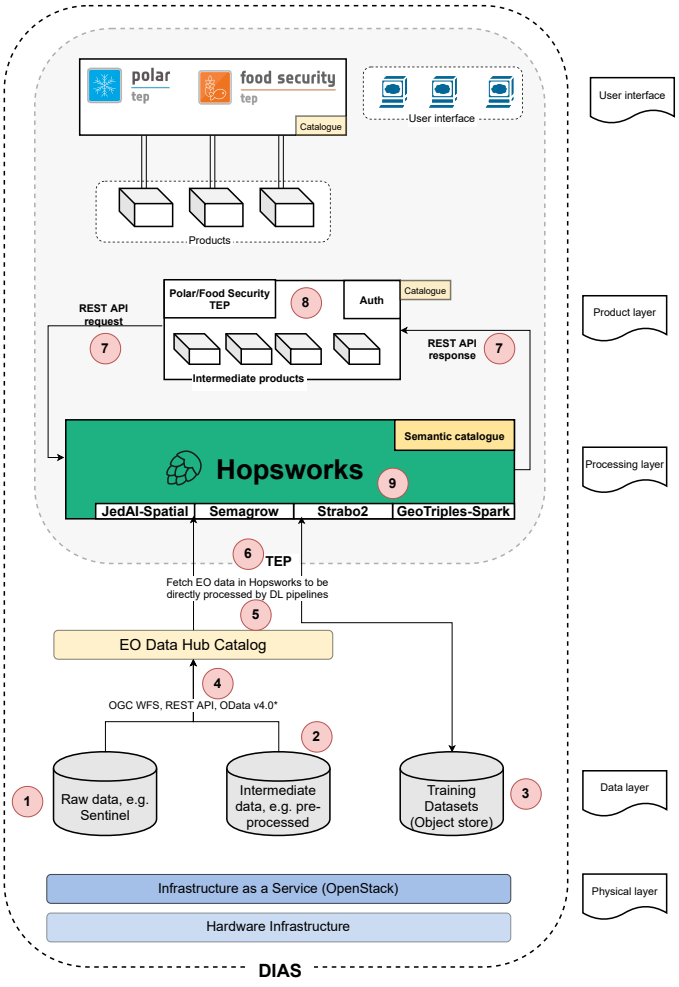


Fig. 1: An overview of *ExtremeEarth* platform infrastructure.

This work 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 [3]. 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 data processing at scale, model training at scale, and model deployment. A data science application, particularly in the domain of big data, typically consists of a set of essential stages that form a ML/DL pipeline. This data pipeline is responsible for transforming data and serving it as knowledge by using data engineering processes and by applying ML and DL techniques. In the context of the EO data domain, these pipelines need to scale to the petabyte-scale data 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 steps, data ingestion and preparation can also be described as data pipelines. Figure 2 illustrates the ML/DL pipeline stages along with the stakeholders of these stages. The feature extraction stage is facilitated by the feature store service, presented in Section II.

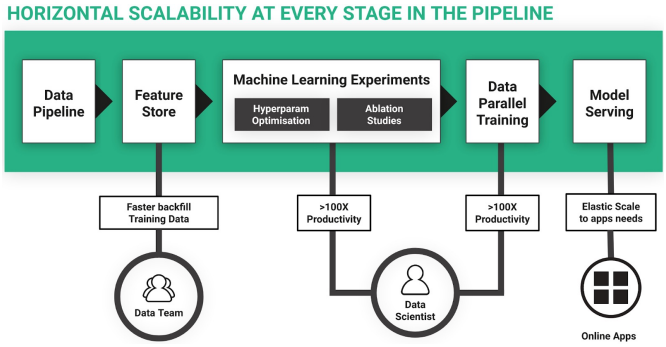


Fig. 2: Horizontally Scalable ML/DL Pipelines.

Hopsworks provides data scientists and engineers with all the necessary tools to build and orchestrate each stage of the pipeline as depicted in Figure 2. 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 3 depicts the overall architecture and lifecycle of a 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 a ML/DL pipeline is to continuously improve

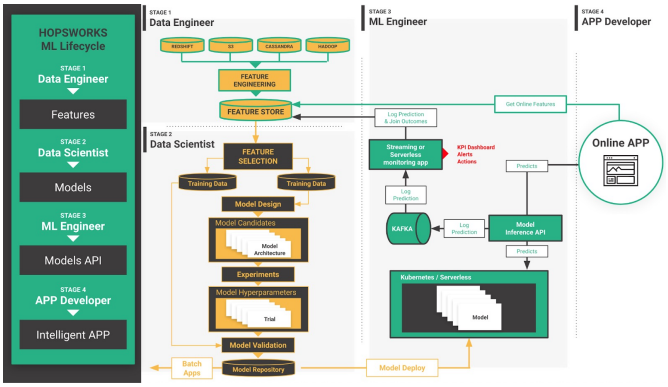


Fig. 3: Hopsworks ML Lifecycle.

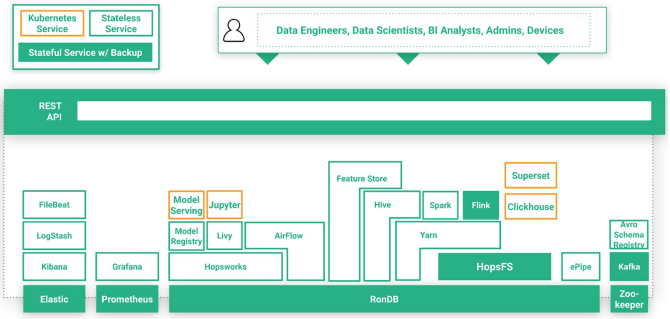


Fig. 4: Hopsworks Services Stack.

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 II-E.

Figure 4 depicts the Hopsworks services stack. HopsFS [4] 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. Hopsworks 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 [5]. 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. Hopsworks has also been extended with an API that allows clients to to submit Spark applications on the cluster easily. Hopsworks 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 application. 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. Auxiliary services like providing logging 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.

Running a ML/DL pipeline can be a repetitive task, as most (if not all) stages need to run when new 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[3] as an orchestration engine.

Contributions: Our main contributions are summarized below.

- We present a scalable AI platform, Hopsworks, to the EO data community and the Copernicus programme.
- We present the extended version of Hopsworks frameworks for scalable ML.
- We present extended EO data pipelines with enhanced and newly developed features to enable and improve support for data parallelism, distributed DL.
- We demonstrate 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.
- 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, namely MAGGY (for hyperparameter tuning and parallel ML experiments) and AUTOABLATION (for automated ablation studies).

Outline: The rest of the paper is organized as follows. Section II presents the ML/DL pipeline stages. The Hopsworks frameworks and its features for scalable ML are explained in detail in Section III. Experimental evaluations are presented in Section IV. Finally, Section V concludes our paper and suggests directions for future research work.

II. ML/DL PIPELINE STAGES

A. Data Ingestion

The first step in building a scalable ML/DL pipeline is to locate the sources where the input data to the AI platform reside. Furthermore, processes need to be established that ingest data from the sources into the AI platform where the ML/DL pipeline runs. These sources can be pretty diverse in the format they use to store data and the protocols they implement to deliver data to other systems. Such sources

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

include raw data which can come from devices connected to the Internet of Things (IoT), images from satellites, structured data from data warehouses, financial transactions from real-time systems, social media, etc. In *ExtremeEarth*, the data typically resides on the DIAS which can be directly accessed from Hopsworks. Figure 5 illustrates wherein the ML/DL pipeline the external systems reside. In the context of *ExtremeEarth*, we show the different ways in which Hopsworks has been extended to make satellite images data easily ingested in the platform for further processing.

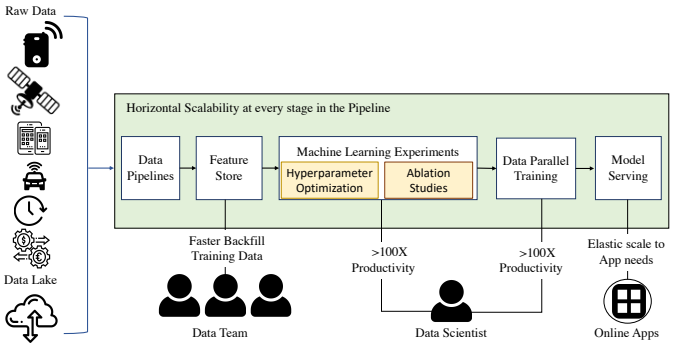


Fig. 5: Data Ingestion sources for a ML/DL pipeline for EO data with Hopsworks.

Accessing EO Data from Hopsworks: In the context of *ExtremeEarth*, Hopsworks is deployed on the CREODIAS 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[4], or it can be accessed via standardized web services such as WMS/WMTS/WCS/WFS[5].

B. EO Data Pre-Processing

EO data pre-processing with Python: Satellite data often needs to be processed before being provided as input to ML algorithms. By utilizing the Python support Hopsworks provides, data scientists in *ExtremeEarth* can use utility programs such as GDAL[6] to process EO data. GDAL is a translator library for raster and vector geospatial data formats.

EO data pre-processing with Docker and Kubernetes: Hopsworks has been extended in *ExtremeEarth* to provide support for working with arbitrary programming languages and frameworks when processing EO data by enabling users to run arbitrary Docker containers on Kubernetes via the Hopsworks jobs service. The motivation behind this functionality is that users might need to use 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 4 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.

[4]https://docs.openstack.org/swift/latest/s3_compat.html
[5]https://creodias.eu/data-access-interfaces
[6]https://gdal.org/

C. Feature Engineering and Data Validation

Feature Store. The process of applying domain knowledge to create features used in further ML/DL pipeline stages is referred to as feature engineering. The need for a framework that enables feature engineering and reduces the complexity of managing features has increased with the continuous growth in input data and increased complexity of ML/DL pipelines. In order to enhance the management of curated feature data, Hopsworks has been extended with a new framework named Feature Store [6] which helps data scientists working with EO to organize their ML assets and curate the EO data features.

The Feature Store acts as the central management layer for curated data in an organization, and it acts as the interface between data engineering and data science teams. Before using the validated data to develop DL models, the features that will be used to develop such models need to be defined, computed, and persisted. Hopsworks Feature Store is the service that data engineers and data scientists use for such tasks. It provides rich APIs, scalability, and elasticity to cope with varying data volumes and complex data types and relations. For example, users can create groups of features or compute new features such as aggregations of existing ones.

Generating reusable features that can be shared across different teams in an organization that can facilitate developing new ML models, as shown in Figure 6 is the primary motivation for feature engineering. The benefits of the Feature Store include reusing features across pipelines, feature discoverability with free-text search across an organization's feature data, applying software engineering principles onto ML features with versioning, documentation, and access control, and time-travel by fetching past feature data that were used for training a particular model. It also brings scalability in managing multiple petabytes of feature datasets so that data scientists can collect valuable insights regarding data distribution and correlation.

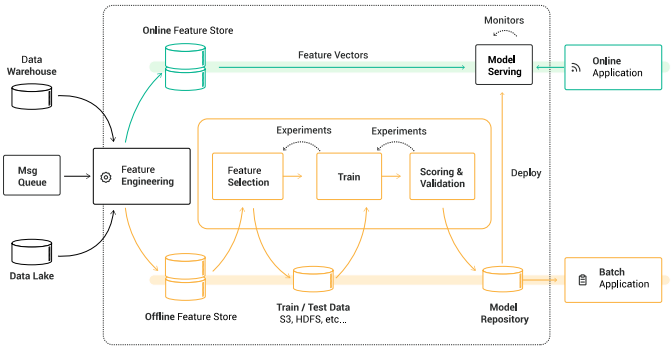


Fig. 6: The Feature Store serves as the link between Feature Engineering and Training.

The Feature Store is implemented on fault-tolerant and scalable services to achieve all the properties mentioned above.

Offline data is stored in Apache Hive [7], a scalable data warehouse built on top of Apache Hadoop, and online data is stored in RonDB (NDB/MySQL Cluster). Offline features can be employed for training and used mainly in batch-oriented use cases where past feature data can be fetched or vast volumes of feature data can be analyzed to generate statistics. Online features need to be accessible in real-time for pipelines that need to get data at prediction time. Besides storing data, the Feature Store utilizes the Spark integration in Hopsworks to compute and analyze features. Figure 7 shows the main components of the Hopsworks Feature Store.

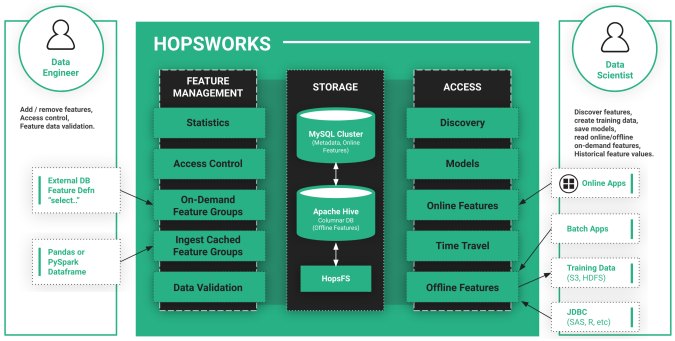


Fig. 7: Hopsworks Feature Store Architecture.

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 needs to be taken into account 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[7]. This validation framework is built on Apache Spark and Deequ[8].

[7] https://docs.hopsworks.ai/latest/generated/feature_validation/

D. Model Analysis and Training

Building ML models is an iterative process. This is because the models need to be built based on either new training data or feedback from model validation. Subsequently, distributing training on large datasets to develop and deploy ML models is non-trivial. 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*[9], a simple interface for expanding the understanding of a black-box regression or classification ML model.

The What-If Tool is shipped with Hopsworks as part of the default Python environments that Hopsworks projects come with. 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 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.

```
# @title 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.

E. Model Serving and Monitoring

Exporting, serving for inference, and monitoring the models developed in the previous stage are three folds of the last stages of the DL workflow.

The last stage of the DL workflow is threefold: *export*, *serve* for inference, and *monitor* the model(s) developed in the previous stage. Once a model is developed and exported from Hopsworks using the stages in the ML/DL pipeline, it needs to be served so that external applications (e.g., iceberg detection or water availability detection in food crops) can use it for inference. Users can submit inference requests using the Hopsworks built-in elastic model serving infrastructure for TensorFlow and scikit-learn. After the model is deployed, its performance also needs to be monitored in real-time so that users can decide when it would be the best time to trigger the training stage.

[8] https://github.com/awslabs/deequ
[9] https://github.com/pair-code/what-if-tool

Hopsworks has been extended to provide support for TensorFlow Serving and Flask model servers to deploy models trained with TensorFlow or any other Python library such as scikit-learn. Additionally, KFServing has been installed in the Kubernetes cluster to deploy more complex inference pipelines composed of the model server, an inference logger container, and a transformer component. KFServing provides additional features such as pre and post-processing of inputs and outputs during model inference, multi-model serving, scale-to-zero support, multi-armed bandits, and A/B testing. Users have the option to choose whether to use KFServing or the docker containers provided by Hopsworks. Moreover, users can select the minimum number of instances for a model serving in runtime, therefore Hopsworks provides users with the important property of elasticity. Hopsworks also provides infrastructure for model monitoring, which means continuously monitoring the requests being submitted to the model and its responses. The users can then apply their own business logic on which actions to take depending on how the monitoring metrics output changes over time.

In the context of *ExtremeEarth*, inference requests are proxied through the Hopsworks REST API to provide secure multi-tenant access to Hopsworks where role-based access control is done based on the abstraction of projects. Project members can only submit requests to the models being served from their projects. Inference requests are logged in Apache Kafka [8] which is provided as a multi-tenant service in Hopsworks. The overall architecture of model serving and monitoring in Hopsworks is depicted in Figure 8. The snippets of code presented in Listings 2-4 show end-to-end examples of model serving on Hopsworks using TensorFlow. The helper library for Hops that facilitates development by hiding the complexity of running applications and interacting with services is *hops-util-py*.

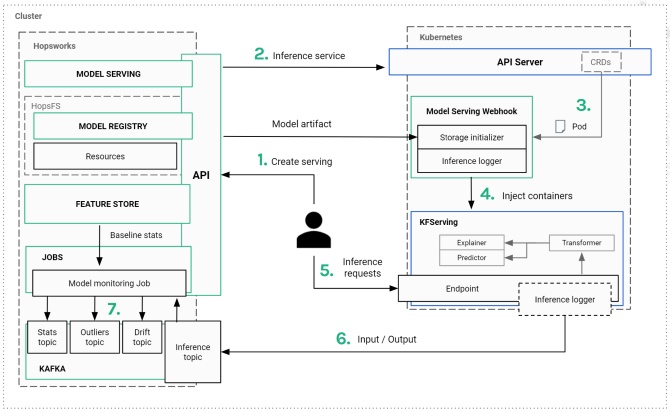


Fig. 8: The architecture for model serving and monitoring in Hopsworks [9].

```
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. Iif 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 of the exported model.

```
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)
```

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

III. 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 9 shows the challenges in scaling out ML and DL. Hopsworks makes use of PySpark as an orchestration layer that is transparent to the users in order 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 for the model we are going

to train. We typically run many experiments as we need to search for good values for 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 [10]. 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 [11], HyperOpt [12], and ASHA [13]) is provided. In the following subsections, we describe the Experiment API, distributed training techniques available in Hopsworks, Hyperparameter tuning with MAGGY, and ablation studies with AUTOABLATION.

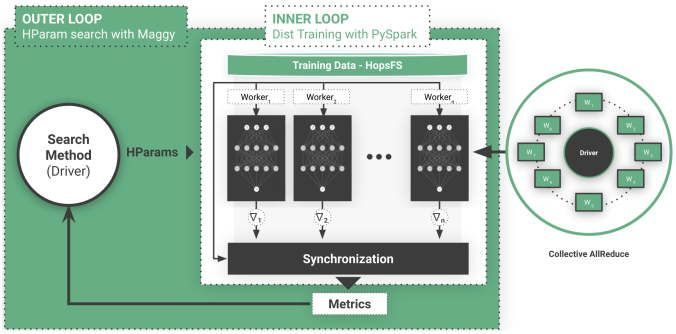


Fig. 9: Scaling out Distributed Training of ML and DL.

A. The Experiment API

An experiment can be defined as the process of using training samples, finding the best hyperparameters, 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 ML libraries and frameworks such as TensorFlow, TensorBoard, PyTorch, Keras, 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[10].

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 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 [4], 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 [4] 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.

B. 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 are able to request from one to potentially all the GPUs of the cluster, and Hopsworks will take care of allocating the resource to applications. Certain hyperparameter tuning algorithms such as random search and grid search are parallelizable by nature, which means that different *Executors* will run different hyperparameter combinations. 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[11]. Hopsworks provides parallelized versions of the grid search, random search, or *state-of-the-art* evolutionary optimization algorithms which will automatically search for hyperparameters to iteratively improve evaluation metrics for our models such as model accuracy.

[10] Documentationavailableat:https://hopsworks.readthedocs.io/en/latest/hopsml/hopsML.html#experimentation

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

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.

```
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 10 - HopsFS [4] 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 List 8 demonstrates how the Experiment API is used for distributed training.

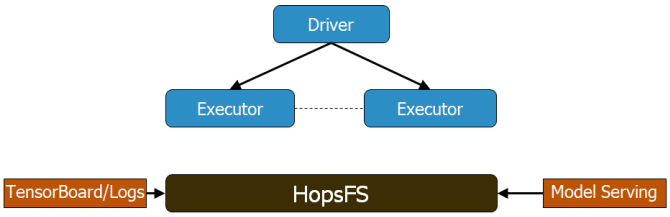


Fig. 10: Distributed File System.

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.

C. Hyperparameter Tuning with MAGGY

MAGGY[12] is a framework for asynchronous parallel execution of ML experiments that supports early stopping of under-performing trials by exploiting global knowledge, guided by an optimizer [10]. MAGGY's programming model is based on distribution oblivious training functions [14], in which the users have to factor out dataset creation and model creation code into their separate functions and pass them as parameters to the training function. This allows for the resulting parameterized code to be used and launched for different experiment types (e.g., hyperparameter tuning, ablation studies, and distributed training) and distribution settings (e.g., single-host or on several workers) without requiring further code changes.

For hyperparameter tuning, MAGGY currently ships with existing implementations of Random Search, Bayesian Optimization (with Tree Parzen Estimators [15] and Gaussian Processes [16]), as well as HyperBand [17] and ASHA [18] as optimizers, and a median early stopping rule for early stopping of under-performing trials [19]. Nevertheless, MAGGY also provides a developer API with base classes for both the

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

```
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.

optimizers and the early stopping rules so that users and developers can implement and use their own optimizers or early stopping rules. To run a hyperparameter tuning experiment, the users have to 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 9).

In MAGGY, as shown in Figure 11, the Spark *Driver* and *Executors* communicate with each other via Remote Procedure Calls (RPCs). The optimizer that guides hyperparameter search is located on the *Driver*, and it assigns trials to *Executors*. The 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, *Executors* are always kept busy running trials (see Figure 12), and global information needed for efficient early stopping is aggregated in the optimizer. This results in increased resource utilization and speedup for running the experiments.

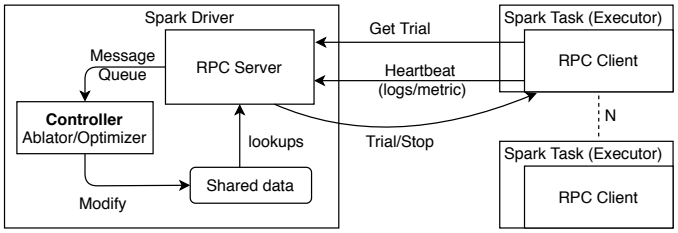


Fig. 11: MAGGY is setup as an RPC framework within the Spark Driver and Executors. The flow of information for the communication protocol and runtime behaviour is shown in this Figure.

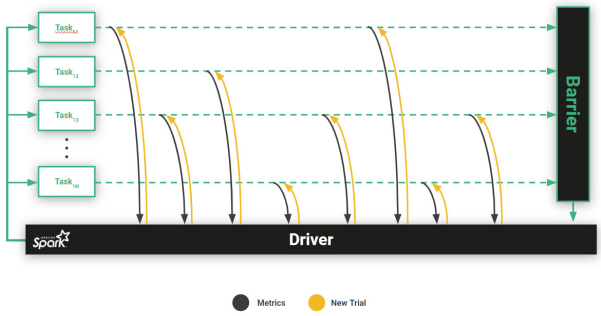


Fig. 12: Directed Asynchronous Search using MAGGY.

D. Ablation Studies

Ablation studies provide insights into the relative contribution of different architectural and regularization components to ML models' performance. Dataset features and model layers are notable examples of these components, but any design choice or module of the system may be included in an ablation study. 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, Hopsworks has been extended with AUTOABLATION, a framework for automated parallel ablation studies [20]. AUTOABLATION runs on top of MAGGY and includes a Leave-One-Component-Out (LOCO) ablator, which removes one component out of the training process at a time (i.e., in each trial). A component can be one or a group of dataset features, one or a group of network architecture layers, as well as "modules" of network architecture, such as Inception-v3 inception modules [21]. Similar to the concept of search space in hyperparameter tuning experiments, an ablation

study 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 specified components should be removed for each trial. Once the study is defined, AUTOABLATION would then automatically create the corresponding trials for the ablation study and run them in parallel.

An ablation study can be thought of as an experiment consisting of several trials. For example, each model ablation trial involves training a model with one or more of its components (e.g., a layer) removed (as shown in Figure13. Similarly, a feature ablation trial involves training a model using different dataset features and observing the outcomes. In Section IV we provide code snippets as examples for defining and running typical ablation study experiments.

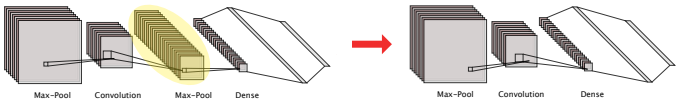


Fig. 13: Example of a model (layer) ablation trial. The layer highlighted in yellow is removed from the base model and the subsequent model is trained on the same training data to find out about the relative contribution of the ablated (removed) layer to the performance of the model.

IV. EXPERIMENTAL EVALUATION

In this section, we will present an experimental evaluation of the frameworks discussed in Section III, and in particular MAGGY and AUTOABLATION. We first report 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. Then we showcase Hopsworks' support for automated ablation studies for DL training workloads using AUTOABLATION.

A. Hyperparameter Tuning

For the task of hyperparameter tuning, we train a three-layer Convolutional Neural Network (CNN) with a fully connected layer on the Fashion-MNIST dataset [22]. We compare the performance of random search [23] using MAGGY (asynchronous parallel execution of trials over Spark) 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 9.

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

Listing 9: Hyperparameter search space for Fashion-MNIST.

As shown in Figure 14 and 15 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 [19] reduces the wall-clock time by roughly half compared to using Spark (Experiment API), without any loss in the final accuracy of the best trials. We can also see that adding more workers linearly reduces 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 I, as well as Figure 14. We can see that both MAGGY and Spark converge to similar accuracy.

TABLE I: Final accuracy after 100 trials.

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 II: 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.

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

The effect of early stopping of under-performing trials can be further emphasized by looking at Table II, 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 see that with MAGGY, using the median early stopping rule, on average, half of the trials are stopped and this results in the reduction of total wall-clock time by approximately half.

B. Ablation Studies

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

EXP1: Feature Ablation of the Titanic Dataset. Here we perform a feature ablation experiment on a customized version of the Titanic dataset[13]. The training dataset contains six features apart from the label, so we will have six trials where in each we exclude (leave out) one training feature, and one base trial that contains all the features (thus seven trials in total). We use a simple Keras Sequential model that has two hidden `Dense` layers. We use 80% of the dataset as the training set and the rest 20% as the test set and train the model for 10

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

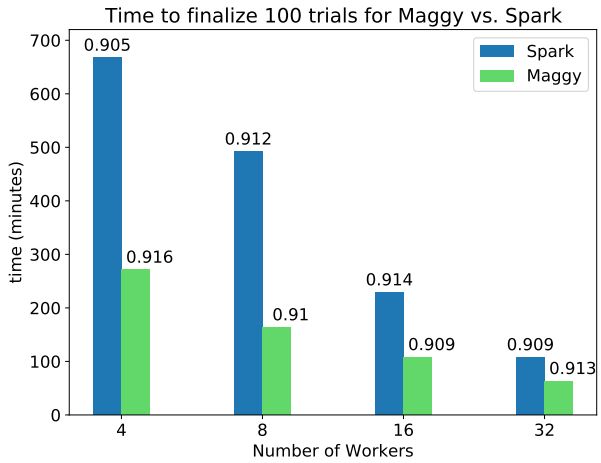


Fig. 14: Asynchronous execution of trials and the median stopping rule in MAGGY enables 100 hyperparameter trials to be executed in lower wall-clock time (lower is better) compared to Spark (the Experiment API) without any loss in accuracy (denoted on top of the bars). Linear scalability (linear reduction of wall-clock time of the experiment) by adding more workers can also be seen for both MAGGY and Spark.

epochs. Defining this experiment in AUTOABLATION requires only a few lines of code, as can be seen in Listing 10.

```
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 10: Code for defining the feature ablation experiment.

We repeat this experiment five times and then rank the training features with regards to their average effect on the test accuracy, as shown in Table III. Based on the results, we can see that training the model using all the features of the dataset(`None` ablated) results in the worst test accuracy, while the model trained on the dataset resulting from excluding the `fare` feature achieves the best test accuracy.

TABLE III: Average accuracy on the test set after excluding each feature from the training set.

Excluded Feature	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

EXP2: Model Ablation of a Keras Sequential CNN Model. Here, we perform a model (layer) ablation study on a CNN classifier for the MNIST dataset [24]. The network has 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. We are 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 11 shows the AUTOABLATION code for defining the study. After repeating the experiment for five times and then rank the selected layers with regards to their average effect on the test accuracy, as shown in Table IV. The results show that for this network-dataset pair, removing the second `Conv2D` layer has the worst effect on the test accuracy, while the model derived from removing the `Dropout` layers performs better compared to 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 11: Code for defining the CNN model ablation experiment.

TABLE IV: Average accuracy on the test set resulting from excluding layers of interest from the base model.

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

EXP3: Model Ablation of Inception-v3. Here we show the near-linear scalability achieved by parallel execution of ablation trials with AUTOABLATION through this experiment. We perform an ablation study on seven modules of the Inception-v3 network [25] in a transfer learning task on a subset of the TenGeoPSAR dataset [26] which is split into train (3200 images), validation (800 images), and test (1000 images) sets. Each image is labeled with one out of 10 classes that correspond to different geophysical phenomena.

We use an Inception-v3 network pre-trained on ImageNet [27] and then change the output layer to suit our 10-class classification task. This network consists of 11 blocks, also known as "inception modules", and we perform a module ablation study on the first seven modules of this network. This is a predefined network (i.e., we do not explicitly define how the layers and modules are structured), so we first compile it to find out about the names of different layers, and to identify the entrance and endpoint of each module. In Keras, this can be done either by plotting the architecture or observing the `model.summary()` output. Once we identify the layers, we define the ablation study using the code shown in Listing 12.

Each ablation trial consists of training (fine-tuning) the network on the TenGeoPSAR dataset for 40 epochs. To demonstrate scalability, we perform this experiment in

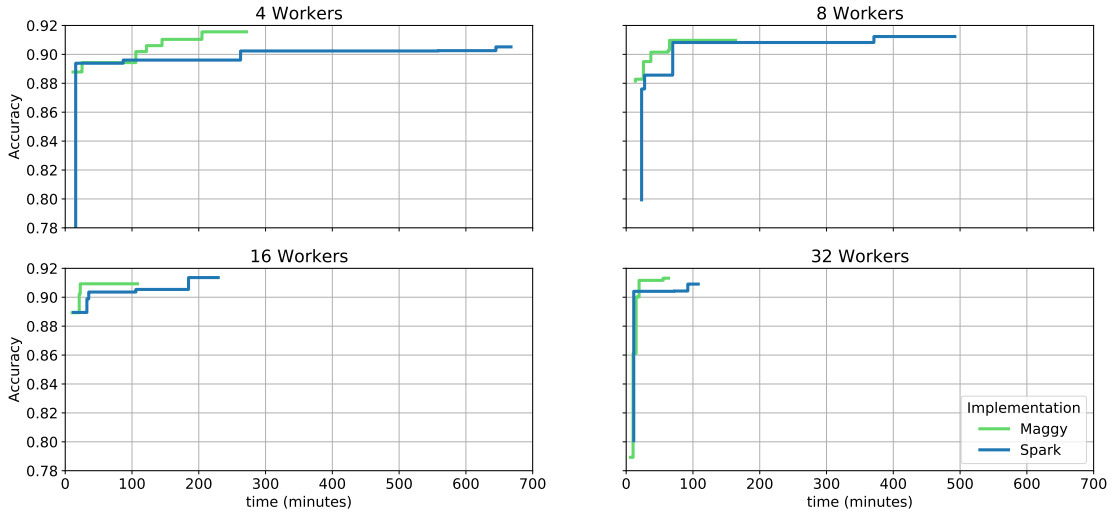


Fig. 15: MAGGY, through asynchronous parallel execution and early stopping of under-performing trials, finds better configurations faster. Due to shorter trials, MAGGY concludes experiments with the same number of trials in shorter wall-clock time. In Spark (Experiment API), each trial is executed to completion, i.e., no early stopping, yielding similar accuracy as expected, but resulting in higher wall-clock time to execute 100 trials compared to MAGGY.

```
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 12: Defining the Inception-v3 module ablation experiment.

three settings: (i) a single executor (sequential, i.e., no parallelization), (ii) two executors, and (iii) four executors. The wall-clock time for each settings can be seen in Figure 16. 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. We can see from Figure 16 that AUTOABLATION provides near-linear scalability through asynchronous, parallel execution of ablation trials.

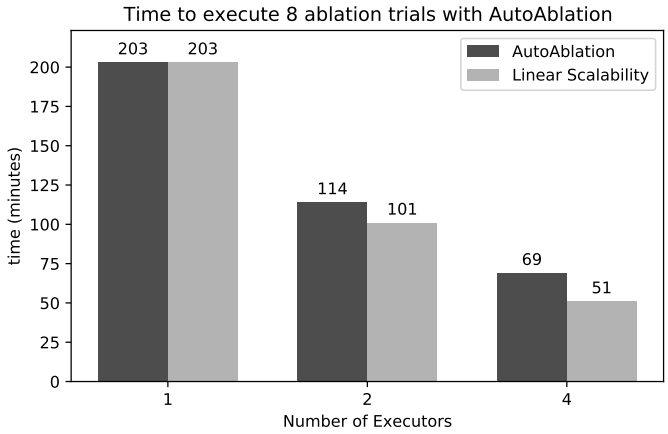


Fig. 16: AUTOABLATION provides near-linear scalability through asynchronous, parallel execution of ablation trials.

V. CONCLUSION

In this paper, we introduce the extended version of Hopsworks with EO data support for AI by adding enhanced and new features, such as the Feature Store and MAGGY framework, to enable and improve support for data parallel distributed DL and to enhance the scalability of the Hopsworks platform. To this end, this paper serves as a demonstrator and walk-through of the stages of building a production-level EO ML/DL pipeline using 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.

We also explained the different components and features available in Hopsworks that the EO community can use. In

addition to this, through experimental evaluation, we showed 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. We 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 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.

As future work, we will keep developing Hopsworks to make it even more fit to the tools and processes used by researchers across the entire EO community. We will also continue the development of our use cases with more sophisticated models using even more advanced distributed DL training techniques. In this paper, we focus the work of distributed learning on data parallelism. 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 be used to further improve training speed and resource utilization.

ACKNOWLEDGMENT

This work is supported by the ExtremeEarth project[14] funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258.

REFERENCES

[1] D. H. Hagos, T. Kakantousis, V. Vlassov, S. Sheikholeslami, T. Wang, J. Dowling, A. Fleming, A. Cziferszky, M. Muerth, F. Appel *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, 2021.

[2] D. H. Hagos, T. Kakantousis, V. Vlassov, S. Sheikholeslami, T. Wang, J. Dowling, C. Paris, D. Marinelli, G. Weikmann, L. Bruzzone *et al.*, "ExtremeEarth Meets Satellite Data From Space," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 14, pp. 9038–9063, 2021.

[3] M. Ismail, E. Gebremeskel, T. Kakantousis, G. Berthou, and J. Dowling, "Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2525–2528.

[4] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 89–104.

[5] R. Andersson, "Gpu integration for deep learning on yarn," 2017.

[6] T. Kakantousis, A. Kouzoupis, F. Buso, G. Berthou, J. Dowling, and S. Haridi, "Horizontally Scalable ML Pipelines with a Feature Store," in *Proc. 2nd SysML Conf., Palo Alto, USA*, 2019.

[7] G. Robbie, C. Owen, and L. Yevgeni, "Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning," https://eng.uber.com/petastorm/, 2018. [Online]. Available: https://eng.uber.com/petastorm/

[8] N. Garg, *Apache kafka*. Packt Publishing Ltd, 2013.

[9] J. de la Rúa Martínez, "Scalable Architecture for Automating Machine Learning Model Monitoring," 2020.

[10] M. Meister, S. Sheikholeslami, A. H. Payberah, V. Vlassov, and J. Dowling, "Maggy: Scalable Asynchronous Parallel Hyperparameter Search," in *Proceedings of the 1st Workshop on Distributed Machine Learning*, 2020, pp. 28–33.

[11] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on Bayesian optimization," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 26–40, 2019.

[12] J. Bergstra, D. Yamins, D. D. Cox *et al.*, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in science conference*, vol. 13. Citeseer, 2013, p. 20.

[13] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," *arXiv preprint arXiv:1810.05934*, 2018.

[14] M. Meister, S. Sheikholeslami, R. Andersson, A. A. Ormenisan, and J. Dowling, "Towards Distribution Transparency for Supervised ML With Oblivious Training Functions," in *Workshop on MLOps Systems*, 2020.

[15] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*. PMLR, 2013, pp. 115–123.

[16] D. Ginsbourger, J. Janusevskis, and R. Le Riche, "Dealing with asynchronicity in parallel Gaussian process based global optimization," Ph.D. dissertation, Mines Saint-Etienne, 2011.

[17] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[18] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar, "Massively parallel hyperparameter tuning," 2018.

[19] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[20] S. Sheikholeslami, M. Meister, T. Wang, A. H. Payberah, V. Vlassov, and J. Dowling, "Autoablation: Automated parallel ablation studies for deep learning," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021, pp. 55–61.

[21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[22] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[23] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[24] Y. LeCun, "The mnist database of handwritten digits," 1998, http://yann.lecun.com/exdb/mnist/.

[25] C. Szegedy et al., "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[26] C. Wang et al., "A labelled ocean sar imagery dataset of ten geophysical phenomena from sentinel-1 wave mode,"

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

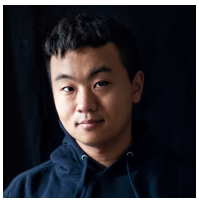
Geoscience Data Journal, vol. 6, no. 2, pp. 105–115, 2019.

[27] J. Deng et al., "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 248–255.

Desta Haileselassie Hagos (Member, IEEE) received a Ph.D. degree in Computer Science from the University of Oslo, Faculty of Mathematics and Natural Sciences in April 2020. Currently, he is a Postdoctoral Research Fellow at the Division of Software and Computer Systems (SCS), Department of Computer Science, School of Electrical Engineering and Computer Science (EECS), KTH Royal Institute of Technology, Stockholm, Sweden, working on the H2020-EU project, ExtremeEarth: From Copernicus Big Data to Extreme Earth Analytics. He received his B.Sc. degree in Computer Science from Mekelle University, Department of Computer Science, Tigray, in 2008. He obtained his M.Sc. degree in Computer Science and Engineering specializing in Mobile Systems from Luleå University of Technology, Department of Computer Science Electrical and Space Engineering, Sweden in June 2012. His current research interests are in the areas of machine learning, deep learning, and artificial intelligence.

Theofilos Kakantousis is a co-founder and VP of Product at Logical Clocks, the main developers of Hopsworks, an open-source Data and Artificial Intelligence (AI) platform. He holds an M.Sc. degree in Distributed Systems from KTH Royal Institute of Technology and has previously worked as a Middleware consultant at Oracle, Greece, and as a research engineer at SAP, Zurich, and at RISE SICS, Stockholm. He has published research papers on big data and ML platforms. He frequently gives talks on Hopsworks, and has presented Hopsworks at venues such as Strata San Jose/New York, Big Data Tech Warsaw, and BigData Moscow.

Sina Sheikholeslami (Student Member, IEEE) received the M.Sc. degree in data science from the KTH Royal Institute of Technology, Stockholm, Sweden, in 2019, and the Eindhoven University of Technology, Eindhoven, The Netherlands. He is currently working toward the Ph.D. degree with the Distributed Computing Group, KTH Royal Institute of Technology. His main research interest is design of systems for distributed deep learning.



Tianze Wang is a Ph.D. student in the Distributed Computing group at the Department of Computer Science of KTH Royal Institue of Technology. He received his B.Sc. degree in Software Engineering from Nankai University. He obtained his M.Sc. degree in Data Science from KTH Royal Institue of Technology and Eindhoven University of Technology. His current research interests are in distributed deep learning.

Vladimir Vlassov (Senior Member, IEEE) is a professor in Computer Systems at the Department of Computer Science, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology in Stockholm, Sweden. He was a visiting scientist with the Massachusetts Institute of Technology (1998) and at the University of Massachusetts Amherst (2004), USA. He has participated in a number of research projects funded by the European Commission, by Swedish funding agencies, and by the National Science Foundation (NSF) USA. He was one of the coordinators of the EMJD-DC Erasmus Mundus Joint Doctorate in Distributed Computing. Currently, he is a principal investigator from KTH in the H2020-EU project "ExtremeEarth: From Copernicus Big Data to Extreme Earth Analytics" (2018-2020). His research covers several areas in computer science, including data-intensive computing, stream processing, scalable distributed deep learning, distributed systems, autonomic computing, cloud, and edge computing.

Amir H. Payberah is an Assistant professor (Docent) of Computer Science at the Division of Software and Computer Systems (SCS) of KTH Royal Institute of Technology in Sweden. He is also a member of the Distributed Computing at KTH (DC@KTH) and the Center on Advanced Software Technology Research (CASTOR). Prior to that, Amir was a machine learning scientist at the University of Oxford (2017-2018), and a senior researcher at the Swedish Institute of Computer Science (2013-2017). He received his Ph.D. from KTH Royal Institute of Technology in June 2013.

Moritz Meister is a lead software engineer at Logical Clocks, the main developers of Hopsworks, an open-source Data and AI platform. Moritz has a background in Econometrics and received M.Sc. degrees in Computer Science from Politecnico di Milano and Universidad Politecnica de Madrid in 2019. Previously, he worked on various projects for large enterprises such as Deutsche Telekom, Deutsche Lufthansa, and Austrian Airlines, helping them to productionize machine learning models to improve customer relationship management.

Robin Andersson is a software engineer at Logical Clocks, the main developers of Hopsworks, an open-source Data and AI platform. Robin received the M.Sc. degree in Software Engineering of Distributed Systems from KTH Royal Institute of Technology in 2017. His main contributions to the Hopsworks platform include implementing support for GPU scheduling and isolation in Hopsworks's YARN component and building APIs for tracking machine learning experiments and managing models.

Jim Dowling is CEO of Logical Clocks and an Associate Professor at KTH Royal Institute of Technology. He is the lead architect of the open-source Hopsworks platform, a horizontally scalable data platform for machine learning that includes the industry's first Feature Store. He received his Ph.D. in Distributed Systems from Trinity College Dublin and has worked at MySQL AB. He is a distributed systems researcher and his research interests are in the area of large-scale distributed systems and machine learning.