

OGC Testbed-15
Machine Learning Engineering Report

Table of Contents

1. Subject	4
2. Executive Summary	5
2.1. Document contributor contact points	6
2.2. Foreword	6
3. References	7
4. Terms and definitions	8
4.1. Abbreviated terms	8
5. Overview	11
6. Background	12
6.1. Relationship to OGC API - Processes (WPS 3)	14
6.2. Machine Learning Techniques	15
6.2.1. Reinforcement Learning	16
6.2.2. Convolutional Neural Networks	16
6.2.3. Recurrent Neural Networks	16
7. Thread Architecture	17
7.1. Petawawa Super Site research forest change prediction ML model scenario	17
7.2. New Brunswick forest supply management decision maker ML model scenario	18
7.3. Quebec Lake river differentiation ML model scenario	19
7.4. Richelieu River hydro linked data harvest model scenario	21
7.5. Arctic Web Services Discovery ML model scenario	21
8. Petawawa cloud mosaicking ML model	23
8.1. Component Summary	24
8.1.1. WPS Server	24
8.1.2. Job / Queue Handler	25
8.1.3. Internal Storage	25
8.1.4. Orchestrator	25
8.2. Component Design	26
8.2.1. Cloud free mosaic generation	27
8.2.2. ML model training	29
8.3. Implementation Approach	30
8.3.1. Job / Queue Handler	30
8.3.2. WPS Server	30
8.3.3. Cloud free mosaic generation status query	33
8.3.4. Cloud free mosaic download	34
8.3.5. Orchestrator	36
8.4. Conclusions	49
9. Petawawa Land Classification Model	51
9.1. Pixel-wise Classification with Deep Learning	51

9.1.1. Dataset	51
9.1.2. Model	52
9.1.3. Results	54
9.2. Implementation of Web Processing Service (WPS) for Deep Learning Model	55
9.2.1. Introduction of WPS wrapper implementation	55
9.2.2. WPS Interface Description	56
9.2.3. WPS Request Example and Result Demonstration	58
10. New Brunswick forest supply management decision maker ML model	62
10.1. Component Summary	62
10.2. Component Design	63
10.2.1. Set neural network	64
10.2.2. Train an agent	65
10.2.3. Harvest agent	66
10.2.4. Transport agent	66
10.2.5. Planning agent	67
10.2.6. Run episodes	68
10.3. Architecture	68
10.4. Input data	69
10.5. Routing engine	69
10.6. Preprocessed forest model	70
10.7. Price forecasting model	71
10.7.1. Wood pricing forecasting	71
10.7.2. Fuel pricing forecasting	72
10.8. Other models - Business process parameters	75
10.8.1. Harvest teams	75
10.8.2. Transport teams	75
10.8.3. Planning team and team allocation criteria	75
10.8.4. Mills	75
10.8.5. Machine Learning Model	76
10.9. Component Implementation	77
10.9.1. Implementations	77
10.10. WPS Request / Response examples	77
10.10.1. Configuration	77
10.10.2. Training	79
10.10.3. Execution	80
10.10.4. Results	81
10.11. Conclusions	83
11. Quebec River-Lake Classification and Vectorization Model	85
11.1. Component Summary	85
11.2. Component Design	85
11.3. Implementation Approach	86

11.3.1. Application	86
11.3.2. Data	87
11.3.3. Machine Learning Model	90
11.3.4. Convolutional Neural Network architecture	90
11.3.5. Other experimental findings	93
11.4. Component Implementation	94
11.4.1. Process Description	95
12. Arctic Discovery Catalog	97
12.1. Overview	97
12.2. Architecture	97
12.3. Machine Learning Model Training	99
12.3.1. Data Preparation	99
12.4. ML Models	100
12.4.1. Multilayer Perceptron (MLP) Neural Network Implementation	100
12.4.2. Training Results	100
12.5. Model Accuracy	104
12.5.1. Results	104
12.5.2. Standards	105
12.5.3. Interoperability	105
12.6. Future Directions	105
12.6.1. Convolutional Neural Network Implementation	105
12.6.2. Recurrent Neural Network Implementation	105
12.6.3. Evergreen Harvester	105
12.6.4. Unsupervised Learning	106
13. Discussion	107
13.1. OGC API - Processes Operations	107
13.1.1. Suggestions for OGC API – Processes endpoints	107
13.2. Recommendations	108
13.2.1. D102 Recommendations	108
13.2.2. D104 Recommendations	111
13.2.3. D105 (OGC API - Features service) Recommendations	112
14. Conclusion	114
Appendix A: Configuration file for the ML application	115
Appendix B: JSON file for ML App Process Description	117
Appendix C: CWL file for the helper ML Application Package	120
Appendix D: Log output of the training process for D104	121
Appendix E: Revision History	122
Appendix F: Bibliography	123

Publication Date: 2019-12-20

Approval Date: 2019-11-22

Submission Date: 2019-09-30

Reference number of this document: OGC 19-027r2

Reference URL for this document: <http://www.opengis.net/doc/PER/t15-D002>

Category: OGC Public Engineering Report

Editor: Sam Meek

Title: OGC Testbed-15: Machine Learning Engineering Report

OGC Public Engineering Report

COPYRIGHT

Copyright © 2019 Open Geospatial Consortium. To obtain additional rights of use, visit <http://www.opengeospatial.org/>

WARNING

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Public Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to

indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Subject

The Machine Learning (ML) Engineering Report (ER) documents the results of the ML thread in OGC Testbed-15. This thread explores the ability of ML to interact with and use OGC web standards in the context of natural resources applications. The thread includes five scenarios utilizing seven ML models in a solution architecture that includes implementations of the OGC Web Processing Service (WPS), Web Feature Service (WFS) and Catalogue Service for the Web (CSW) standards. This ER includes thorough investigation and documentation of the experiences of the thread participants.

Chapter 2. Executive Summary

This OGC ER documents work completed in the OGC Testbed-15 ML thread. This includes documentation of experimental methods and results as well as addressing the integration of ML models and outputs into an OGC Web Services (OWS) architecture. The thread covered several scenarios that have commonalities, but do not interact directly. The purpose of the research in the ML thread was to demonstrate the use of OGC standards in the ML domain through scenario development. The scenarios used in the ML thread were:

- Petawawa Super Site Research forest change prediction model.
- New Brunswick forest supply management decision maker ML models.
- Quebec Lake river differentiation ML models.
- Richelieu River Hydro linked data harvest models.
- Arctic web services discovery ML model.

Each scenario has a set of supporting data coupled with cataloging and processing services to support the aim. An ML model is at the core of each scenario. The objective was to have the model make key decisions that a human in the system would typically make under *normal* circumstances. Each scenario and corresponding implementations were supported by at least one client to demonstrate the execution and parsing of outputs for visualization.

Publication of specific ML results in the draft Map Markup Language (MapML) specification focuses on the client supporting the Quebec Lake scenario as the data service. This was an implementation of the OGC Application Programming Interface (API) - Features standard. This implementation was required to produce the outputs of the model in MapML. (Note: The OGC API - Features standard was previously named WFS 3.0.) Likewise, the corresponding client was required to parse and visualize the results using the MapML outputs from the data service. This client was provided as a separate work item. The other scenarios were supported by clients provided by the model originators to demonstrate their work. A full exploration and documentation of the MapML work is documented in the [MapML ER](http://www.opengis.net/doc/PER/t15-D023) [http://www.opengis.net/doc/PER/t15-D023].

Each of the different work activities incorporated one or more ML techniques using different datasets and parameters. The overall findings and recommendations from the ML thread consisted of: Those regarding ML and those concerning the usage of OGC standards in ML use cases. Many of the ML recommendations included further exploration of the techniques required to produce suitable results. Recommendations of interest to the OGC are as follows:

- Define and discuss the candidate OGC API - Processing pattern for use in machine learning. This type of exercise has already been done in the [OGC Open Routing API Pilot](https://www.opengeospatial.org/projects/initiatives/routingpilot) [https://www.opengeospatial.org/projects/initiatives/routingpilot] in which two different patterns were created to explore the functionality. These were:
 - Use of a lightweight concept *routes* as the path base with little constraint on the API design pattern and use of conformance classes to configure clients automatically.
 - A formal structure, based on the OGC API – Processes draft specification, for paths that start with */processes/* and has many of the same API calls as WPS 2.0.
- Understand the utility of OGC standards for feeding dynamic data to ML models. As these

models require considerable data to train, the thread participants felt that the current suite of OGC standards for data dissemination is better suited for static or mostly static datasets. Extensions specific for data streaming might be useful for all big data problems, not just ML.

- Explore the use of OGC standards to compare scenarios in previously trained ML models. There are already a number of pre-trained models freely available as well as *general feature models* that attempt to identify trends, patterns or objects from a variety of domains. Re-use of existing models is likely to be important in the future of geospatial ML applications.
- • Use OGC standards to enable stress testing of ML models. The use of parameters within ML processes is key to their ability to successfully predict based upon an unknown sample. Currently this testing is carried out manually. However, stress testing via the OGC API - Processes draft specification and then recording the parameters in a CSW would be useful in the future for OGC standards to support. This approach strays into the realm of metadata profiling for ML models, which may be a useful output of future endeavors that have a discovery aspect.

Overall, the thread produced a multitude of results that can be taken forward in future OGC Testbeds and Pilots or more widely in the community.

2.1. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Contacts

Name	Organization	Role
Sam Meek	Helyx SIS	Editor
Tom Landry	CRIM	Contributor
Pierre-Luc Saint-Charles	CRIM	Contributor
Francis Charette-Migneault	CRIM	Contributor
Mario Beaulieu	CRIM	Contributor
Ignacio Correas	Skymantics	Contributor
William Cross	Skymantics	Contributor
Jerome St-Louis	Ecere	Contributor

2.2. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 3. References

The following normative documents are referenced in this document.

- OGC: OGC 14-065r2, OGC WPS 2.0.2 Interface Standard: Corrigendum 2 (2018) [<https://portal.opengeospatial.org/files/14-065r2>]
- OGC: OGC Web Feature Service 2.0.2 (2014) [<http://docs.opengeospatial.org/is/09-025r2/09-025r2.html>]
- OGC: OGC Catalog Service for the Web 2.0.2 (2007) [http://portal.opengeospatial.org/files/?artifact_id=20555]
- OGC: OGC 06-121r9, OGC® Web Services Common Standard (2010) [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2]

Chapter 4. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard (OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2]) shall apply. In addition, the following terms and definitions apply:

- **overfitting**

The production of an analysis that corresponds too closely or exactly to a particular set of data, and therefore fails to fit additional data or predict future observations reliably. Source: [Oxford English Dictionary](https://www.oed.com/view/Entry/258314?rskey=mJU00t&result=2&isAdvanced=false#eid) [https://www.oed.com/view/Entry/258314?rskey=mJU00t&result=2&isAdvanced=false#eid]

- **dropout**

The procedure of randomly dropping components of a neural network from a neural network layer. This results in a scenario where at each layer more neurons are forced to learn the multiple characteristics of the neural network. This can prevent overfitting. Source: medium.com [http://medium.com]

- **activation function**

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. Source: [Wikipedia](https://en.wikipedia.org/wiki/Activation_function) [https://en.wikipedia.org/wiki/Activation_function]

- **hyperparameter**

In Bayesian statistics, a hyperparameter is a parameter of a prior distribution. The term is used to distinguish them from parameters of the model for the underlying system under analysis. Source: [Wikipedia](https://en.wikipedia.org/wiki/Hyperparameter) [https://en.wikipedia.org/wiki/Hyperparameter]

4.1. Abbreviated terms

- ADES - Application Deployment and Execution System
- AI - Artificial Intelligence
- API - Application Programming Interface
- AUPRC - Area Under Precision Recall Curve
- CLI - Command Line Interface
- CNN - Convolutional Neural Networks
- CRIM - Computer Research Institute of Montréal
- CRS - Coordinate Reference System
- CSW - Catalogue Service for the Web
- CVM - Controlled Vocabulary Manager
- CWL - Common Workflow Language
- DL - Deep Learning

- DNN - Deep Neural Network
- DVC - Data Version Control
- ER - Engineering Report
- EMS - Execution Management System
- GRHQ - Géobase du réseau hydrographique du Québec
- Helyx SIS - Helyx Secure Information Systems Limited
- HLS - Harmonized Landsat and Sentinel-2
- HRDEM - High Resolution Digital Elevation Model
- HTTP - Hypertext Transfer Protocol
- JSON - JavaScript Object Notation
- KB - Knowledge Base
- LiDAR - Light Detection and Ranging
- mAP - Mean Average Precision
- MapML - Map Markup Language
- ML - Machine Learning
- MLP - Multilayer Perceptron
- NIR - Near-Infrared
- NLTK - Natural Language Toolkit
- OGC - Open Geospatial Consortium
- ONNX - Open Neural Network Exchange Format
- OpenMI - Open Modeling Interface
- OWS - OGC Web Services
- PaaS - Platform as a Service
- Pub/Sub - Publication/Subscription
- RAKE - Rapid Automatic Keyword Extraction
- RDF - Resource Description Framework
- REST - Representational State Transfer
- RGB - Red, Green, Blue
- RL - Reinforcement Learning
- RNN - Recurrent Neural Network
- SPF - Spruce, Pine, Fir
- SOS - Sensor Observation Service
- TC - Technical Committee
- TF/IDF - Term Frequency-Inverse Document Frequency
- TIE - Technology Integration Experiments

- URL - Uniform Resource Locator
- VCS - Version Control Systems
- WCS - Web Coverage Service
- WES - Web Enterprise Suite
- WFS - Web Feature Services
- WICS - Web Image Classification Service
- WMS - Web Map Service
- WPS - Web Processing Service
- WPS-T - Transactional Web Processing Service

Chapter 5. Overview

The rest of the ER is organized as follows:

Chapter 6 provides an overview of previous ML work in the OGC and an overview of the work items.

Chapter 7 describes the thread architecture.

Chapter 8 describes the component implementation that provides the Petawawa cloud mosaicking model capability.

Chapter 9 describes the component implementation that provides the Petawawa land classification model capability.

Chapter 10 describes the component implementation that provides the New Brunswick forest supply management decision maker ML model capability.

Chapter 11 describes the Quebec River-Lake Classification and Vectorization ML model capability.

Chapter 12 describes the Arctic Discovery catalog.

Chapter 13 provides the overall discussion and recommendations from the work.

Chapter 14 provides the concluding remarks.

Chapter 6. Background

This OGC Engineering Report (ER) reports on the work performed and completed as part of the *Machine Learning* (ML) thread in the OGC Testbed-15 initiative. ML has previously been explored in the OGC through the ML thread in Testbed-14. While the work reported in this ER is not a direct continuation from Testbed-14, the Testbed-14 Machine Learning ER provides many of the recommendations and design influences leading to the work described in this ER. A major driving factor behind this ER is a movement towards standardization of an interface designed for interacting with ML models and processes.

The documents reviewed are largely from the OGC, but academic and industrial references are included where relevant.

Previous OGC work that has influenced the Testbed-15 ML activity consists of the following documents:

- [18-038r2](http://docs.opengeospatial.org/per/18-038r2.html) [http://docs.opengeospatial.org/per/18-038r2.html] - OGC Testbed-14: Machine Learning Engineering Report
- [18-094r1](http://docs.opengeospatial.org/per/18-094r1.html) [http://docs.opengeospatial.org/per/18-094r1.html] - OGC Testbed-14: Characterization of RDF Application Profiles for Simple Linked Data Application and Complex Analytic Applications Engineering Report
- [18-097](https://docs.opengeospatial.org/per/18-097.html) [https://docs.opengeospatial.org/per/18-097.html] - OGC Environmental Linked Features Interoperability Experiment Engineering Report
- [18-022r1](https://docs.opengeospatial.org/per/18-022r1.html) [https://docs.opengeospatial.org/per/18-022r1.html] - OGC Testbed-14: SWIM Information Registry Engineering Report
- [18-090r1](https://portal.opengeospatial.org/files/?artifact_id=82623) [https://portal.opengeospatial.org/files/?artifact_id=82623] - OGC Testbed-14 Federated Clouds Engineering Report
- [16-059](http://docs.opengeospatial.org/per/16-059.html) [http://docs.opengeospatial.org/per/16-059.html] - Testbed-12 Semantic Portrayal, Registry and Mediation Engineering Report
- [15-054](https://portal.opengeospatial.org/files/?artifact_id=64405) [https://portal.opengeospatial.org/files/?artifact_id=64405] - Testbed-11 Implementing Linked Data and Semantically Enabling OGC Services Engineering Report
- 14-049 - Testbed 10 Cross Community Interoperability (CCI) Ontology Engineering Report
- [19-003](http://docs.opengeospatial.org/per/19-003.html) [http://docs.opengeospatial.org/per/19-003.html] - OGC Testbed: Earth System Grid Federation (ESGF) Compute Challenge
- [18-050r1](https://docs.opengeospatial.org/per/18-050r1.html) [https://docs.opengeospatial.org/per/18-050r1.html] - OGC Testbed-14: ADES & EMS Results and Best Practices Engineering Report
- [18-049r1](http://docs.opengeospatial.org/per/18-049r1.html) [http://docs.opengeospatial.org/per/18-049r1.html] - OGC Testbed-14: Application Package Engineering Report
- [17-035](http://docs.opengeospatial.org/per/17-035.html) [http://docs.opengeospatial.org/per/17-035.html] - OGC Testbed-13: Cloud ER

The earliest example of ML-type operations being exposed via an OGC interface is via Web Image Classification Service (WICS). This service includes several calls that are suitable for configuring and executing ML models behind an OGC interface. Specifically, these calls include: *GetClassification*, *TrainClassifier* and *DescribeClassifier*. Although suitable for use in a small set of

circumstances, the WICS only supports image-specific calls. It does this through OGC *web services* style applications that represent an older architecture model, prior to the recent move to a resources-based model through OpenAPI. ML in the Testbed-15 context has broadened to include different types of ML beyond image classification. The work in this Testbed moves towards a decision support tool that utilizes multiple data types to build models and predict results.

The OGC Testbed-14 ML ER describes work that extends beyond WICS. It identifies and implements several new calls that follow a similar pattern to WICS, but go beyond image classification. These calls are as follows:

- TrainML
- RetrainML
- ExecuteML

These three calls follow the *web services* pattern of OGC services and offer the ability to create, modify and execute ML models through a standardized interface. In addition to these calls are the following ML Knowledge Base (KB) interactions:

- GetModels
- GetImages
- GetFeatures
- GetMetadata

As well as opening up the interfaces to include ML specific calls, the ML space has undergone semantic enablement via a Controlled Vocabulary Manager (CVM). The interfaces used in Testbed-14 consisted mainly of WPS-T 2.0 with Representational State Transfer (REST) and JavaScript Object Notation (JSON) bindings. There is an ongoing initiative within the OGC Technical Committee (TC) to enhance Web Processing Service (WPS) version 2.0 to 3.0 by implementing the REST/JSON bindings as core functionality rather than as an extension. This is designed to bring WPS in line with the OGC API - Features standard, which is also based on OpenAPI. Additionally, Testbed-14 brought about experimental implementations of Web Map Service (WMS), Web Feature Service (WFS) and WPS standards specifically for transparency and usage of ML models.

There are several explicit recommendations taken from the Testbed-14 ML ER, garnered from experiences of ML in OGC Web Services (OWS). These include:

- Use of a Catalogue Service for the Web (CSW) as an interface to an ML KB.
- An International Organization for Standardization (ISO) application profile to record and distribute KB Information
- Use of the OGC API - Features standard as a design pattern to manage the interaction with ML capabilities.
- Consideration for use of the Open Modeling Interface (OpenMI) standard.

Moving on from interfaces and service standards, a further area enabling ML discussed in Testbed-14 is the concept of Federated Clouds, that is, disparate cloud computing services that are federated to share access credentials and therefore services, data and resources, are likely to play a role in the

ML space. Federation of cloud services is not a new concept and is simply managed when cloud services (or any services for that matter) sit within the same administrative domain. Recently there has been a shift in the computing world to assume resources-on-demand, including elastic computational storage and computing power, that can be surged or stood down as required. All of this is usually outsourced in a *Platform-as-a-Service* (PaaS) approach, where on-demand computing is provided by an organization with significant resources (server farms) that are allocated according to demand and provision. In short, federation enables participating organizations to selectively share information across administrative domains for purposes of their choosing.

The Testbed-14 Federated Clouds ER sought to research and test the implications of utilizing federated cloud architecture with a focus on cross administrative domain security for use case such as data sharing. In the case of ML and in particular the ML thread in this Testbed, components including ML models and clients are designed and maintained by different vendors (as in the real world) but all need to interoperate. Therefore, understanding and applying the lessons learned and recommendations from Testbed-14 Federated Cloud ER as needed is paramount to a successful set of interoperability experiments.

ML models, outputs and predictions are complex, therefore cataloging, presenting and disseminating data and metadata is of high importance. There are multiple languages, models and standards that can be used to document, discover and disseminate complex offerings utilizing OGC standards. In Testbed-14, dissemination of complex analytic applications and data was explored in the OGC Testbed-14 Characterization of RDF Application Profiles for Simple Linked Data Applications and Complex Analytic Applications ER. The ER covers several aspects of interest including Resource Description Framework (RDF) profiles, Web Ontology Language (OWL) and ontologies to describe certain aspects of complex analytical use cases. RDF is of particular consideration as Testbed-14 sought to define a metadata model to describe RDF application profiles. If operationalized, this was of tangible use within the ML thread as it provides a facility to discover application profiles based upon specific ontologies. The work documented in this ER seeks to utilize RDF where suitable to enable discovery of the complex analytical applications.

The architecture of the thread consists of a set of well-defined ML scenarios. The requirements across the thread deliverables are broad enough to cover typical ML usage such as analysis of imagery content through to the discovery of ML datasets, models and practices through the Arctic Discovery catalog. The latter, activity is concerned with ML process metadata rather than just the outputs. There are a set of *stretch goals* within the Call For Participation (CFP) that are also discussed and prioritized according to likely value gain for the sponsors.

In terms of interfaces, each component is fronted by the relevant OGC service. Each of the ML models is fronted by a WPS, either 1.0 or with the REST/JSON bindings and the catalog is an OGC Compliant CSW (there is currently further work going on in OGC to define an OGC API specification for catalogs). Additionally, data created by the ML models are exposed by the relevant data interface, OGC API - Features for features and Web Coverage Service (WCS) for coverages.

6.1. Relationship to OGC API - Processes (WPS 3)

WPS

NOTE

Prior to the OGC API - Processes naming convention, the draft specification was referred to as WPS 3.0. The official name of the draft specification is now OGC API - Processes. This ER therefore, at times, acceptably refers to implementations of OGC API - Processes as WPS.

As mentioned previously, there are several commonalities between each of the scenarios in terms of the requirements. The scenarios are separate as they aim to deliver completely different outputs, are focused on different areas and in some cases, are using different approaches to ML to achieve their goals. However, each of the server-side components are required to be fronted by a WPS (with REST bindings in some instances) and each has the option of utilizing Common Workflow Language (CWL). Note that since the WPS implementations described in this ER conform to the draft OGC API – Processes specification, they are referred to using both terms throughout this document.

At the time of writing, there is a debate within the OGC on how processing services should be exposed using OpenAPI fronted, resource-based architectures. Some of the viewpoints are captured in the OGC API Hackathon 2019 Engineering Report (OGC 19-062) which presents results from the OGC API Hackathon 2019 event. The debate is largely concerned with the role of legacy WPS calls in versions 1, 2, and transactional versions that include:

- From WPS 1.0
 - GetCapabilities - provide the capabilities document describing the processes available
 - DescribeProcess - describe a particular process
 - Execute - execute a process
- Introduced in WPS 2.0
 - GetStatus - provide the status of an asynchronous processes
 - GetResult - provide the result of an asynchronous process
- Introduced in WPS-T
 - DeployProcess - deploy a new process ready for Execution
 - UnDeployProcess - undeploy a deployed process so it is no longer available

These calls provide functionality in a web services architecture that performs specific actions in relation to processing. In the resource-based architecture approach, the calls are based upon the HTTP verbs GET, POST, HEAD, PUT and DELETE.

6.2. Machine Learning Techniques

The terms "artificial intelligence" and "machine learning" are often used interchangeably or at minimum in a hyphenated fashion. In truth, ML can be considered as a subset of Artificial Intelligence (AI) techniques. Additionally, ML as an array of techniques contains a multitude of different algorithms that are selected to produce the best result depending on the use case. Related to the generic concept of ML is Deep Learning (DL), which is a subset of ML that uses large, multi-layered, artificial neural networks for supervised or unsupervised ML problems.

This section contains a short overview of the techniques used in this thread. While there are several nuanced differences, the main one to consider is the automation of model feedback.

In addition to this functionality, there are several non-functional requirements including:

- Use of [TensorFlow](https://www.tensorflow.org) [https://www.tensorflow.org]
- Continuing to work on CWL best practices from previous Testbeds.
- The demonstrator should be compatible with the Boreal Cloud OpenStack cloud environment of Natural Resources Canada (NRCAN). Boreal Cloud is NRCAN's high performance cloud infrastructure based on OpenStack technology, located at the Pacific Forestry Centre in Victoria, BC.

Supervision of a classification application depends on how much human intervention is required to achieve a suitable model for prediction. *Supervised Learning* requires human intervention to different degrees depending on the use case. *Unsupervised Learning* does not require any human interaction while training the models as the ML model uses automated techniques to assess the likely performance of the model.

6.2.1. Reinforcement Learning

This type of learning is usually implemented in game play applications and in use cases that include autonomous vehicle navigation as there is no "correct" answer to a particular problem. Instead the ML model looks to make the best decision given the circumstances with a view to maximizing cumulative reward. The reinforcement aspect is the application of the reward within the system, if cumulative reward increases then the system has a notion of a *good decision* and will seek to perform similar actions to further increase reward.

6.2.2. Convolutional Neural Networks

A Convolutional Neural Network (CNN) uses convolutions to extract features from local regions of an input. CNNs have gained popularity particularly through their excellent performance on visual recognition tasks. CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage. They have applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing.

6.2.3. Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows the model to exhibit temporal dynamic behavior. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.

Chapter 7. Thread Architecture

The ML thread is comprised of a set of five scenarios with eight formal deliverables and several clients provided by vendors in kind. As there are five separate scenarios, the thread participants defined five separate architectures that were utilized to demonstrate the interoperability of components through the interoperability testing process, also known as Technology Integration Experiments (TIEs). This section describes the scenarios and supporting architectures in detail to provide the reader with an overview of the thread goals, architectures for each of the threads, a motivation for each of the threads, and any changes made during the course of the Testbed to mitigate issues experienced. The five scenarios are as follows:

- Petawawa Super Site Research Forest Change Prediction ML Model
- New Brunswick forest supply management decision maker ML model
- Quebec lake-river differentiation model
- Richelieu River linked data harvest model
- Arctic web services discovery ML models

There are a common set of technical requirements for each of the scenarios. As with all Testbeds, one of the goals is to utilize the latest versions of OGC standards. This policy was operationalized in this domain via usage of the OGC API - Processes draft specification. Likewise, data services are made available through OGC API - Features implementations using the new OpenAPI style resource-based approach and WMS/WCS for mapping and coverages. Although not an enforced requirement, it is expected that each of the ML models is built using open source software with a mention of TensorFlow. Each of these scenarios is discussed in turn in the following sections. Many of the ML participants opted to use WPS 2.0 because they had existing operational implementations.

7.1. Petawawa Super Site research forest change prediction ML model scenario

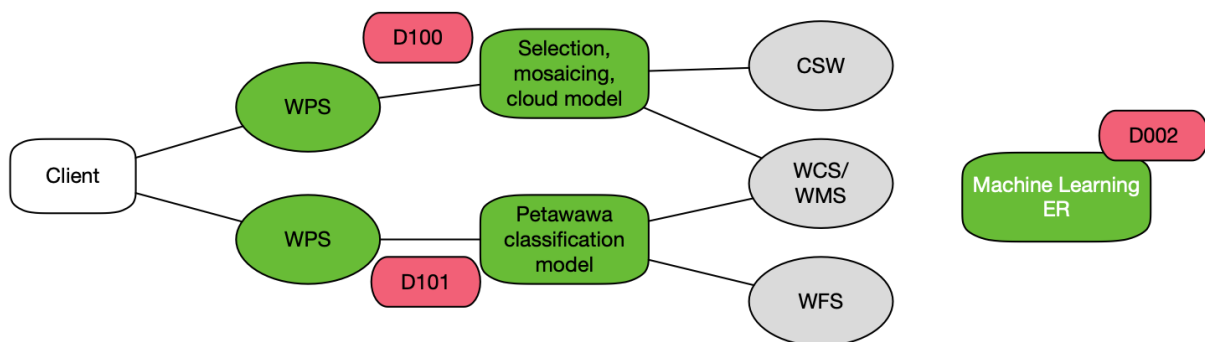


Figure 1. Petawawa Super Site forest change architecture

The aim of this component deliverable was to 1) produce an ML model for detecting and removing high altitude cloudlets (popcorn clouds) from Landsat 1 data in the Petawawa Super Site, and 2) produce a second model for classifying a cloudless, automatically generated image mosaic into land cover categories. The ML model performed the following functions:

1. Data discovery using an OGC CSW.
2. Discovery of usable imagery that has less than 70% cloud coverage.
3. Identification of parts of an image that are either cloud or cloud shadow.
4. Creation of a cloud free composite image using automated techniques.
5. Classify the resultant composite image into land cover using a second ML model.

In addition to this functionality, there are several non-functional requirements including:

- Use of TensorFlow
- Continuing to work on CWL best practices from previous Testbeds. CWL could potentially be used to automate some of the test workflows or to enable the discovery to dissemination aspect of the system. Although a non-functional requirement, the implementation aspect is optional.
- Ensuring demonstrator compatibility with the NRCan Boreal Cloud OpenStack environment.

This scenario is designed to exercise two ML models that are made available to a single client. These include:

- A cloud and cloud shadow (artifact) identification model.
- A land cover classification model.

These two ML models form the backbone of the ML thread; however, they are supported by the following services:

- Each ML model is fronted by a WPS 2.0 for simple execution of the services exposed by the models.
- A CSW facilitates discovery of time-series enabled satellite imagery from Landsat and Sentinel-2 products.
- Attached to the cloud artifact identification model that creates a mosaic using multiple images to build a cloud-free composite.
- Results are made available via the relevant interface (WFS 3.0, WMS, WCS).

Additionally, there is a requirement to continue the work done in Testbed-14 to utilize the CWL to potentially automate some test workflows or to enable the *discovery* → *ML1* → *ML2* → *dissemination* aspect of the system via pre-configuration. The CWL aspects of the thread are optional and implemented where specified.

7.2. New Brunswick forest supply management decision maker ML model scenario

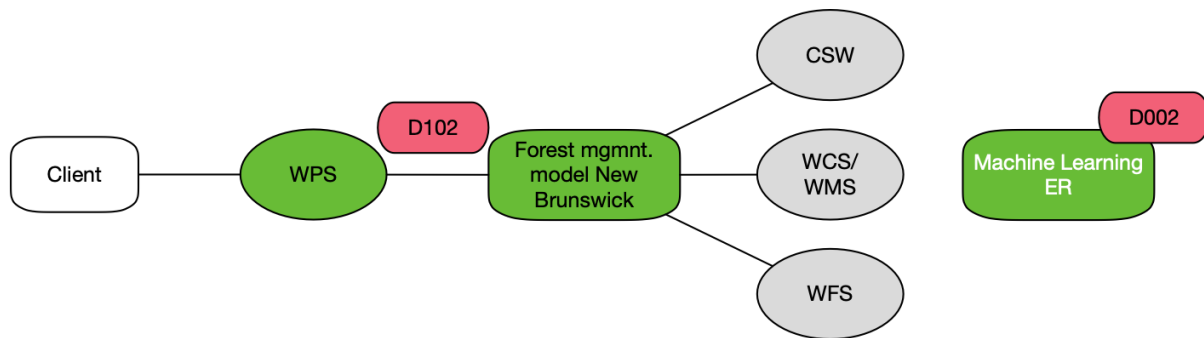


Figure 2. New Brunswick forest supply management scenario architecture

This ML model was concerned with the efficient routing of timber from a managed woodland area in New Brunswick. Road building and infrastructure management were also considered. This model was atypical in terms of its usage of ML practices. It performed the following functions while working towards similar non-functional requirements as described in the previous section:

1. Create a "wood flow model" to optimize routing for timber from source to market.
2. Recommend areas for new road construction to make the route more efficient.
3. Provide a list of recommended road closure locations and times to minimize disruption.
4. Consider data from different sources including: primary infrastructure, secondary infrastructure, and prices of lumber, fuel and energy.

As mentioned previously, the scenarios in this thread were distinct and therefore treated as their own work-item sets, rather than one large, interoperable thread. The New Brunswick scenario contains many of the same constraints and requirements as the other scenarios, such as using a WPS instance to front the model, a CSW for data discovery and cataloging, and WCS/WFS/WMS for data dissemination. The ML model in this work package was complex and consisted of a set of ML models to achieve the desired outcome. The ML model aspects of this work package were as follows:

- Creation of a *wood flow model*, that is, optimization of resource allocation considering optimized flow from forest to market.
- Recommendation of new infrastructure including roads and bridges to further optimize wood flow considering life-cycle analysis.
- Utilization of peripheral supporting information including market prices of lumber, secondary infrastructure, primary infrastructure and efficiency.
- Deployment of the capability on the NRCan Boreal Cloud OpenStack environment.

Execution of the workflow is somewhat simpler than the Petawawa scenario as the ML service can be configured and executed without reaching back to client at any point, except when providing the result.

7.3. Quebec Lake river differentiation ML model scenario

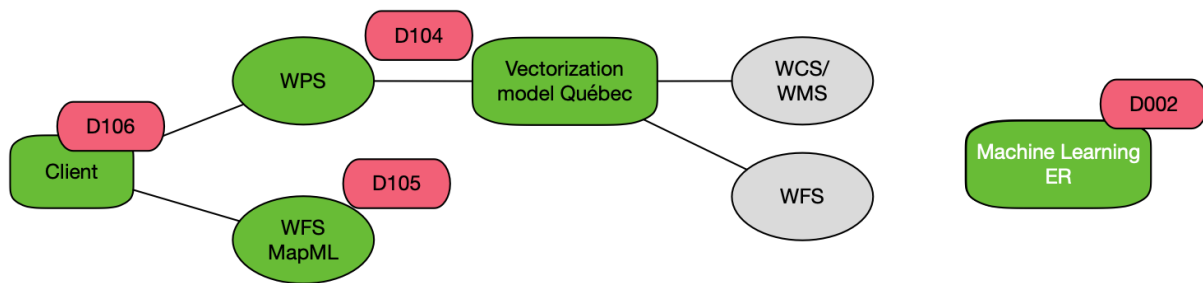


Figure 3. Quebec Lake river differentiation model architecture

The objective of this work package was to create and deploy an ML model to differentiate between rivers and lakes from otherwise unlabeled bodies of water in an image. The main focus of the work was to provide a service to determine whether a body of water should be split into a lake and a river. If so, then the lake and river portions of the split should be identified and labeled. If no split is required, then each identified body of water should be labeled as either lake or river. The procedure for applying the model is as follows:

1. Recommend whether a water body should be split into lake and river features.
2. Evaluate the confidence level of a recommendation.
3. Apply the recommendation to the dataset.
4. Test and correct the resultant dataset for topological and cartographical issues.
5. Present the data in a WFS 3.0 using MapML (described in another ER).

This scenario requires an ML model that is capable of differentiating between lakes and rivers from imagery and Light Detection and Ranging (LiDAR) data. Currently bodies of water from these datasets can be distinguished, but there is not a clear indication of where the line is drawn between when a water body changes from a lake to a river and vice versa. This is not just an ML problem but also an ontological problem. Therefore, any definition of the two concepts is built upon a somewhat arbitrary definition, although a consistent one if an ML service is to be successful. In addition to identifying rivers and lakes, the entire ML service needed to perform the following functions in a workflow:

- Identify a water body and recommend whether a split needs to be made and apply a confidence level to the recommendation.
- If a split is made then vectorize the bodies of water into *lakes* and *rivers*.
- Apply topological correction algorithms if required to remove errors including:
 - Overlaps
 - Slivers
 - Gaps
- Name each feature according to a suitable naming convention as not all rivers and lakes have accessible names.
- Serve the results via MapML using WFS 3.0.

Unlike the previous work packages, there is no data discovery requirement via a CSW. However, there is a requirement to serve the results via an implementation of OGC API – Features, using MapML.

7.4. Richelieu River hydro linked data harvest model scenario

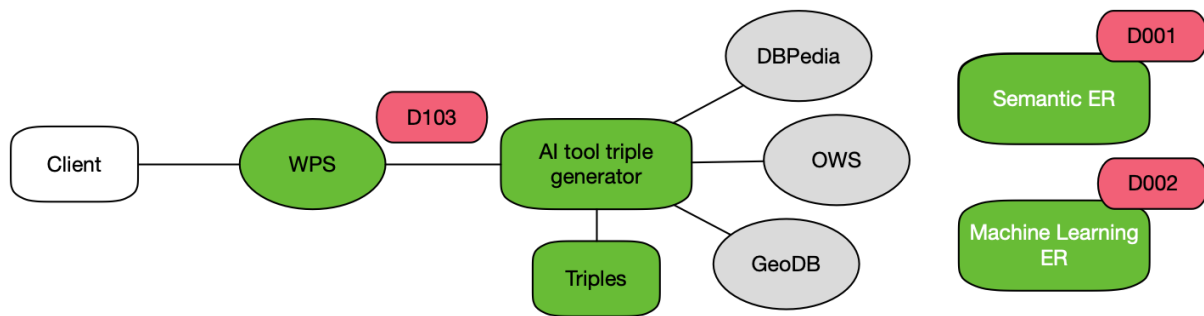


Figure 4. Richelieu River linked data harvesting scenario architecture

This work package differs from the others as it does not require imagery or ML in the traditional sense. Instead this scenario seeks to mine the semantic web for relevant relations between datasets and store the results as triples in the appropriate database. The model was based upon a set of provided ontologies for features and relations to be harvested by the ML model. This scenario was concerned with establishing links between datasets via the semantic web. The main work item in this work package was the AI tool triple generator, which sought to harvest data from specific datasets and gather relations between items of data. The details regarding the semantic aspects of this work package are described in the OGC Testbed-15: Semantic Web Link Builder and Triple Generator Engineering Report (OGC 19-021) and the ML aspects are described in the Components section of this ER.

7.5. Arctic Web Services Discovery ML model scenario

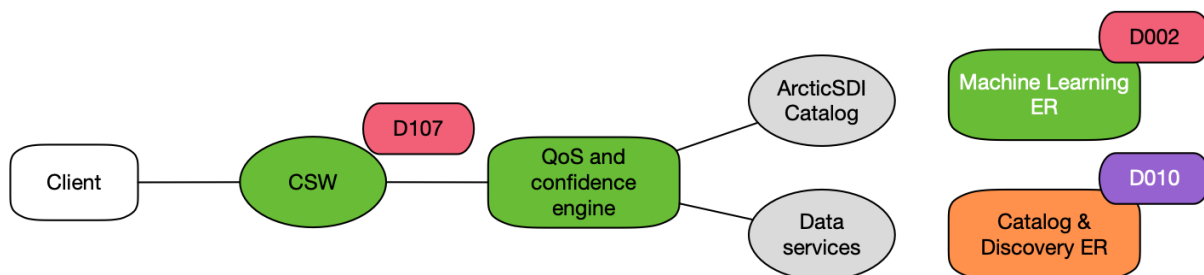


Figure 5. Arctic Web Service discovery model ML architecture

The goal for this work package was to understand the data holdings of a particular domain and its utility to the Arctic domain in terms of relevance to circumpolar science. The following structure was used for this approach:

- The model was focused on the .ca domain to understand the assets that are available within this domain and their relationship to other data assets.
- The ML model was trained to cycle through and categorize endpoints on the .ca domain and make a decision on whether each has any relevance to circumpolar science.
- The identified datasets were given a confidence score and then entered into a CSW for later discovery and use.

The concept of relevance can be determined in a variety of ways. For example, a geographical bounding box can be used as a geofence but the model may also rely on keyword search as well as

other parameterization options. Essentially the ML aspect of the service was trained on a set of attributes of a test ML service that was deemed to be relevant. It then crawled through all ESRI REST endpoints and OGC services within the domain and made an assessment of each of the services, providing information on their relevance.

Chapter 8. Petawawa cloud mosaicking ML model

In the context of the Petawawa Super Site research forest change prediction ML model, the Testbed-15 D100 component (i.e. cloud mosaicking ML model) aimed to create a cloud-free mosaic over the Petawawa Research Forest by assembling the best non-cloud and most recent segments over a given time frame. The cloud detection system was based on ML and CNN.

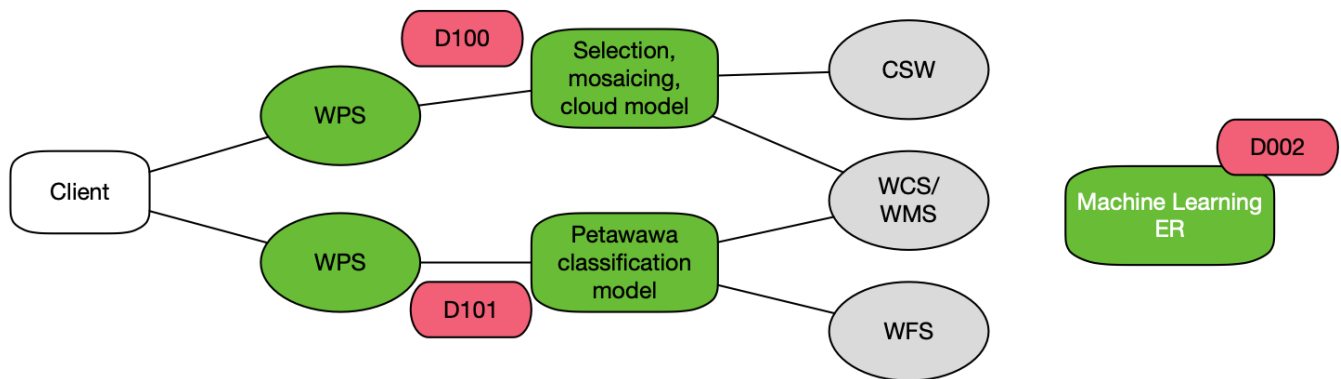


Figure 6. Petawawa Super Site research forest change prediction ML model

NOTE

Petawawa Research Forest

The 100 km² Petawawa Research Forest is situated in Ontario, approximately two-hours northwest of Canada’s capital city, Ottawa. Located in the mixedwood forests of the Great Lakes–St. Lawrence Forest region, common tree species include white pine (*Pinus strobus* L.), trembling aspen (*Populus tremuloides* Michx.), red oak (*Quercus rubra* L.), red pine (*P. resinosa* Ait.), white birch (*Betula papyrifera*), maple (*Acer* spp.), and white spruce (*Picea glauca*), among others (Wetzel et al. 2011). This forest region is considered a transition between the boreal forests to the north, which are dominated by coniferous species, and the deciduous-dominated forests to the south.

The whole system was developed and deployed to be compatible with NRCan’s Boreal Cloud (OpenStack cloud environment). The model can be accessed via a generic WPS client [here](https://borealweb.nfis.org/tb15d100wps/) [https://borealweb.nfis.org/tb15d100wps/]. The mosaic is generated starting from surface reflectance products available from NRCan’s [National Forest Information System](https://saforah2.nfis.org/index.html) [https://saforah2.nfis.org/index.html] for the following datasets:

Table 1. Datasets

Dataset	Description
Landsat	Archived Landsat Collection 1 data (1972–2018). Includes Landsat Multispectral Scanner (MSS), Thematic Mapper (TM), Enhanced Thematic Mapper Plus (ETM+), and the Operational Land Imager (OLI). With the exception of MSS, all data is corrected to surface reflectance. Search terms: “PRF” AND “Landsat”, “Landsat4”, “Landsat5”, “MSS”, “TM”, “ETM+”, “OLI”, etc.
Sentinel-2	Archived Sentinel-2 data (2016–2018), corrected to surface reflectance.

Dataset	Description
<i>Harmonized Landsat and Sentinel-2 (HLS)</i>	Harmonized Landsat and Sentinel-2 surface reflectance data generated by NASA/USGS (2013–2018)

WARNING

Landsat availability

With respect to the **Landsat** Dataset defined in [Table 1](#), for this component the search is based only on **Landsat 7 ETM+**. In any case, the **MSS** products cannot be used due to the missing **Blue** band.

8.1. Component Summary

The following figure summarizes the main software constituting the component.

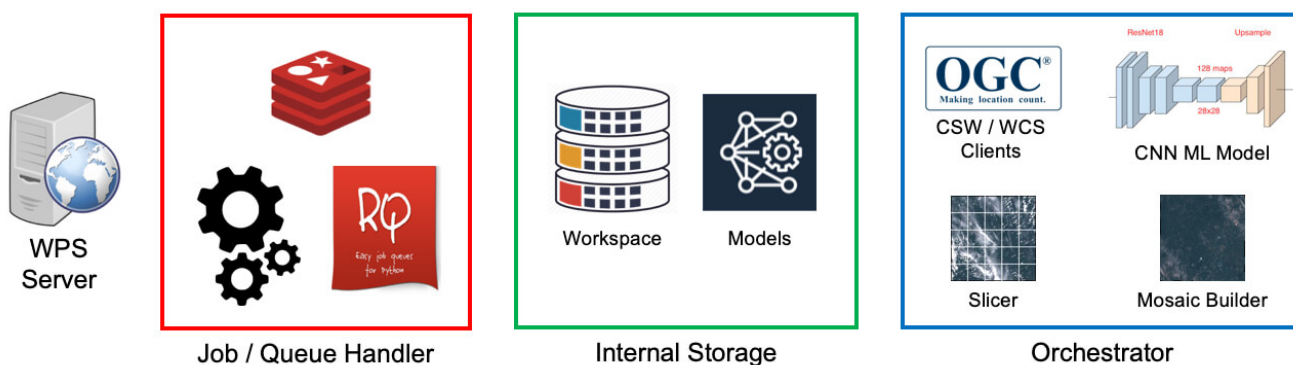


Figure 7. D100 High level architecture

The **D100** component design is based on four main elements:

- WPS Server
- Job / Queue Handler
- Internal Storage
- Orchestrator

WARNING

Architecture Deltas

At the time of publication of this ER, the D100 component is based on WPS version 1.0 and not WPS 3.0 as stated in the architecture. This was a stretch goal for this work package. Likewise, the implementation does not use CWL.

8.1.1. WPS Server

The D100 component exposes a dedicated WPS enabled server (implementing the WPS 1.0 standard) for the requests. The WPS server is running on Flask, a Python lightweight Web Server Gateway Interface (WSGI) web application framework using PyWPS. The endpoint handles the following four different requests:

- Network training;

- Cloud free mosaic generation;
- Cloud free mosaic generation status query;
- Cloud free mosaic download.

Considering that both network training and mosaic generation are demanding activities, these types of requests are queued in order to avoid blocking the WPS server. All the other requests are immediately served. For the cloud-free mosaic generation, requesting either one of the two defaults: ready network (one trained with 3 bands for Red-Green-Blue (RGB) and one with 4 bands for RGB + Near Infrared), or a new network trained (either 3 or 4 bands) with a dedicated WPS request is possible.

8.1.2. Job / Queue Handler

The queue mechanism relies on Remote Dictionary Server (Redis), an in-memory data structure and object persisting system supporting different kinds of abstract data structures. Every time the WPS server receives the training and mosaic generation requests, the new request is pushed on a Redis Queue (RQ), a Python library for queuing jobs and processing them in the background with so called "workers". The relevant job is not automatically run and its execution is remanded to RQ. A worker is another Python process running in the background as a work-horse to perform lengthy or blocking tasks instead of performing the task inside a web process. At least one worker is always up-and-running, but more than a single request instantiating more workers as needed according to application loading and hardware resources available (default configuration is 5 workers) can be served. Every time a worker is available, the job is retrieved from the RQ in a First In - First Out (FIFO) order and executed in a new dedicated process.

8.1.3. Internal Storage

The Internal Storage component contains two different kinds of data: Trained network models and, inside what is called Workspace, all downloaded bands, tiles, intermediate cloud masks generated by the ML model, and the final mosaic for each request. By default, two models are present (as stated before one trained with 3 bands for RGB and one with 4 bands for RGB + Near Infrared). Any custom training requested by dedicated WPS call is also stored to be called later. All the models (stored as PyTorch checkpoints) are persistent in time. The Workspace content, instead, is preserved for each job only for a specific retention time. When time expires the specific folder is deleted to save storage.

8.1.4. Orchestrator

When a WPS request is received for cloud mosaicking, the worker runs a dedicated job named Orchestrator. This was the core part of the D100 component and was composed of several different subcomponents as follows:

- OGC Clients
- Bands Slicer
- ML Model
- Mosaic Builder

8.1.4.1. OGC Clients

Access to the catalog is required to create the mosaic. In order to search and to retrieve products bands, the Orchestrator uses OWSLib Python library for both CSW and WCS requests. For each product discovered from the catalogue service, several links are returned, one for each available band. The number of bands downloaded depends on the model requested for the cloud detection (i.e. either 3 (RGB) or 4 (RGB + Near Infrared) bands).

Table 2. Bands Number Mapping

Dataset	Red	Green	Blue	NIR	Resolution (m)
Sentinel-2	4	3	2	8	10
Landsat-7	3	2	1	4	30
HLS	4	3	2	5	30

All required bands are downloaded to a dedicated folder inside the Workspace, one for each WPS mosaic generation request. For each product found in the search, the relevant bands are downloaded via WCS and stored in the Workspace of the Internal Storage. Only when all the bands have been downloaded and just after the Bands Slicer is the ML model called.

8.1.4.2. Bands Slicer

In order to provide the ML model with proper input, all the downloaded bands are sliced into tiles of 224 x 224 pixels and marked with proper geolocation / geographic information (needed later to rebuild a single cloud mask image). This tile size was chosen to balance speed and performance in the training phase.

8.1.4.3. ML Models

The generic ML model is based on a [ResNet 18](https://www.mathworks.com/help/deeplearning/ref/resnet18.html) [https://www.mathworks.com/help/deeplearning/ref/resnet18.html] architecture developed on the **PyTorch** framework. The model accepts one single tile (224 x 224 pixels) composed of several bands (three or four) and generates a black and white image representing the cloud mask of the inferred data with the same size. The pure white areas represent pixels containing clouds while the black areas represent pixels where clouds are not present. Two default models were made available: One trained with three bands (RGB) and one with four bands (RGB + NIR).

8.1.4.4. Mosaic Builder

When all the tiles are processed by the ML model, the Mosaic Builder merges them into a single cloud mask. The cloud mask is used as an alpha channel to be applied to the original product bands. This result is then combined with the other cloud-free mosaics in a reverse time order. This allows cloud pixels from earlier images to be substituted for non-cloud pixels from more recent images. The final mosaic is generated in GeoTIFF RGB format.

8.2. Component Design

This section describes the overall lifecycle of the D100 component considering two main use cases

covering all the functionalities:

- Cloud-free mosaic generation;
- ML model training / retraining.

8.2.1. Cloud free mosaic generation

The mosaic generation was triggered by a specific WPS request. The following figure shows the sequence diagram for a generic mosaic generation process.

D100 Petawawa cloud mosaicing ML model - Execution

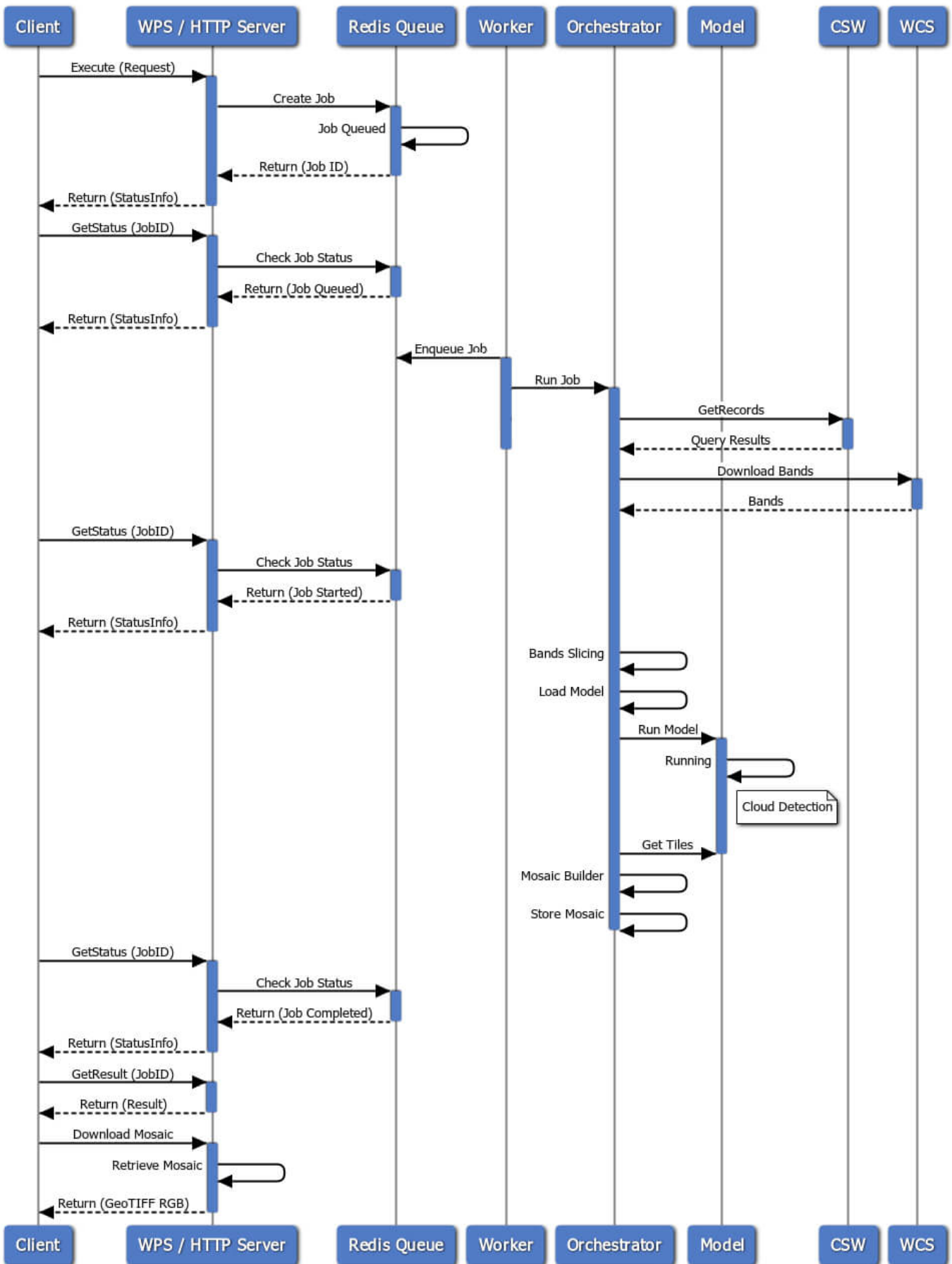


Figure 8. Cloud Free Mosaic Sequence Diagram

The D100 component receives a WPS execution request containing several input parameters (e.g. time

window, ML model to be used and so on). The request is queued and waits for the first available worker to run the job. The client is notified with a response message indicating that the request was received and a new job was created with a specific Job ID. This ID is later used by the client to query the status of the request's progress.

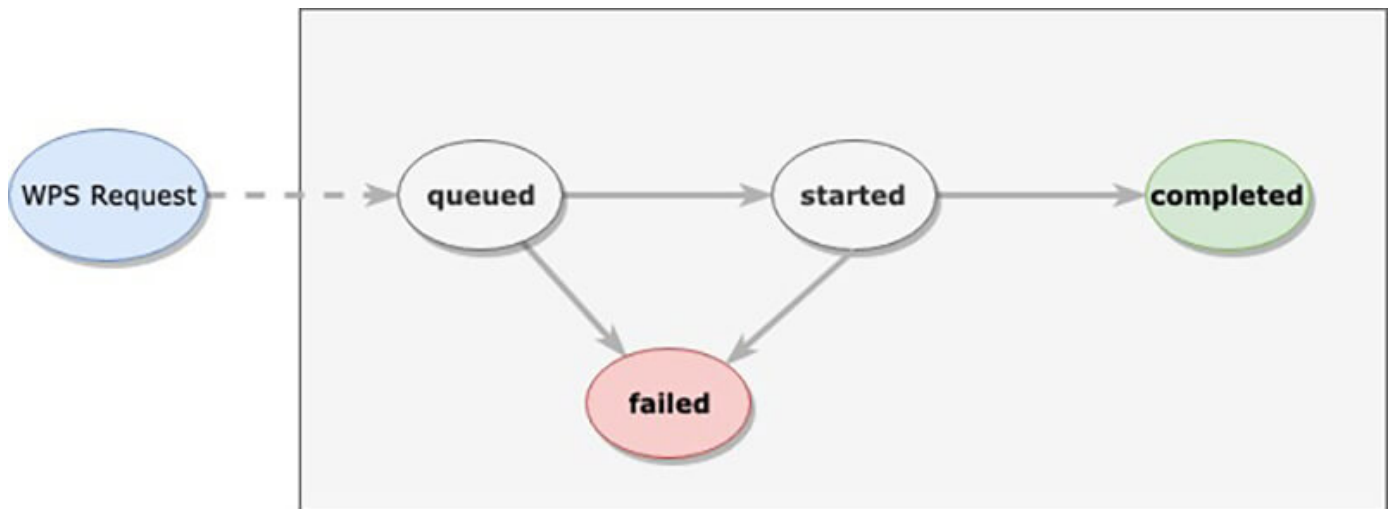


Figure 9. Job Status Diagram

Any subsequent request about Job status will return one of the following status:

- **queued:** The WPS request for a mosaic generation is received and queued but has not yet started.
- **started:** The Job is queued by a worker and is running.
- **failed:** The Job has encountered an unexpected error and is blocked.
- **completed:** The mosaic has been generated and is available for download.
- **NONE:** Either the Job ID is not valid or is no longer available (retention time expired).

As soon as a worker is available, it queues a Job and runs the Orchestrator. As a first step, the Orchestrator queries the WCS server to retrieve all products covering the requested time window. The result list is then sorted in descending percentage coverage order, and all the products having cloud coverage greater than 70% are discarded. For each product found, the relevant bands are downloaded, sliced into tiles, inferred in the ML Model, and reassembled to generate a single cloud mask for the whole product. The Mosaic Builder also takes the product bands plus the cloud mask and merges one product at a time, stacking the different results respecting the descending sorting order. This process, from downloading the bands to image stacking is performed iteratively (i.e. handling one product at a time) until either the area is entirely cloud free or no more products are available. Finally, the resultant GeoTIFF RGB file is downloaded by the client.

8.2.2. ML model training

The D100 implementation was delivered using two different networks having the same architectures but coping with tiles defined by either 3 (RGB) or 4 (RGB + NIR) bands. This approach is consistent with the two process profiles described in the [OGC Testbed-14 Machine Learning ER](http://docs.openeospatial.org/per/18-038r2.html) [http://docs.openeospatial.org/per/18-038r2.html] and the ML best practices work from Testbed-14. This means that training or retraining of a network can be triggered by a dedicated WPS call. Considering the nature and context of the cloud detection system, having a dynamic dataset (mainly to validate the quality and accuracy of the network) is quite complex. Instead what can be requested is training a new

instance of the network (choosing 3 or 4 bands architecture), asking for specific batch size and number of epochs. This new model is then stored in the Internal Storage and can be later recalled for the generation of a cloud free mosaic.

8.3. Implementation Approach

8.3.1. Job / Queue Handler

In order to handle all the **WPS execute** calls, at least one running **worker** shall be present. Run the following command line to start a new **worker**.

RQ Worker start-up command

```
prompt> rq worker -worker-ttl -1
```

In order to assure that each job queued is served, the parameter `-worker-ttl` is set to `-1` to disable expiration of the job.

Reviewing the status of the **workers** (i.e. how many are running and their queue status) can be achieved with the **rqinfo** command.

RQ Worker status command

```
prompt> rqinfo

default      | 0
1 queues, 0 jobs total

a24018a5cc594e9cb73779d5a8908afa (None None): ?
dde85a5880574f55853107e0899fa669 (None None): ?
db90b2fbb3e64b428d324766dff52ea0 (None None): ?
6833fe6f60fa4fe88426ca7aba88429a (None None): ?
3ee415e050a34403b4140b2d34b83f67 (None None): ?
5 workers, 1 queues
```

To stop the workers either kill the processes or close the prompt.

8.3.2. WPS Server

As described above, the WPS server is based on Flask and handles four different types of requests: Network training, cloud free mosaic generation, generation status query, and cloud free mosaic download. Beside these, the WPS instance exposes a generic interface such as the standard `GetCapabilities`.

8.3.2.1. GetCapabilities

The `GetCapabilities` operation requests details of the services offered by the D100 component, including service metadata and metadata describing the available processes. The response is an XML document

called the **capabilities document**, which contains a list of all available services. An example of a GetCapabilities request is:

```
https://borealweb.nfis.org/tb15d100wps?  
service=WPS&  
version=1.0.0&  
request=GetCapabilities
```

The response is a standard WPS GetCapabilities XML response. The following is a snippet of the services offered by the D100 component:

GetCapabilities XML sample response snippet

```
<!-- PyWPS 4.2.1 -->  
<wps:Capabilities service="WPS" version="1.0.0" xml:lang="en-CA" xsi:schemaLocation=  
"http://www.opengis.net/wps/1.0.0 ../wpsGetCapabilities_response.xsd" updateSequence="1">  
  ...  
  ...  
  <wps:ProcessOffering>  
    <wps:Process wps:processVersion="1.0.0">  
      <ows:Identifier>train_network</ows:Identifier>  
      <ows:Title>Train Network</ows:Title>  
      <ows:Abstract>Trigger a process to train the neural network</ows:Abstract>  
    </wps:Process>  
    <wps:Process wps:processVersion="1.0.0">  
      <ows:Identifier>compose_mosaic</ows:Identifier>  
      <ows:Title>Compose Mosaic</ows:Title>  
      <ows:Abstract>Trigger a process to compose a mosaic over a time  
range</ows:Abstract>  
    </wps:Process>  
    <wps:Process wps:processVersion="1.0.0">  
      <ows:Identifier>get_status</ows:Identifier>  
      <ows:Title>Get Status</ows:Title>  
      <ows:Abstract>Retrieve the Job Status</ows:Abstract>  
    </wps:Process>  
    <wps:Process wps:processVersion="1.0.0">  
      <ows:Identifier>get_result</ows:Identifier>  
      <ows:Title>Get Result</ows:Title>  
      <ows:Abstract>Retrieve the Job Result</ows:Abstract>  
    </wps:Process>  
  </wps:ProcessOfferings>  
  ...  
  ...  
</wps:Capabilities>
```

8.3.2.2. DescribeProcess

The DescribeProcess operation requests details of any services offered by the D100 component.

An example of a DescribeProcess request for a compose_mosaic service is:

```
https://borealweb.nfis.org/tb15d100wps?  
service=WPS&  
version=1.0.0&  
request=DescribeProcess  
identifier=compose_mosaic
```

All the available parameters, their nature and possible values (e.g. model type to train or model name to infer) if constrained are provided in a standard response package. In the following sections all the available services with relevant parameters are described.

8.3.2.3. Cloud free mosaic generation

In order to trigger the generation of a new cloud free mosaic, a specific WPS execute service is exposed with the following parameters:

Table 3. Cloud free mosaic generation request parameters

Keyword	Description	Sample Value
<i>identifier</i>	The name of action to be executed. Fixed compose_mosaic	compose_mosaic
<i>model</i>	The name of network to be used. Default values of pretrained network are always available: RGB and RGBNIR	RGBNIR
<i>start</i>	The start date of the time window in ISO Date format.	2017-06-01
<i>stop</i>	The stop date of the time window in ISO Date format.	2017-07-01

An example of this request is:

```
https://borealweb.nfis.org/tb15d100wps?  
service=WPS&  
version=1.0.0&  
request=Execute&  
identifier=compose_mosaic  
datainputs=model=RGBNIR;start=2017-06-01;end=2017-07-01
```

If the request is accepted and queued correctly, the client is provided with the Job ID (e.g. d7111976-3a9f-401c-a3d2-c1a5c30329ac) uniquely identifying the request. This Job Id is needed to perform a status query.

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:ExecuteResponse xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:ows=
"http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://www.opengis.net/wps/1.0.0 ../wpsExecute_response.xsd" service="WPS" version=
"1.0.0" xml:lang="en-US" serviceInstance=
"https://borealweb.nfis.org/tb15d100wps?request=GetCapabilities&service=WPS"
statusLocation="">
  <wps:Process wps:processVersion="1.0.0">
    <ows:Identifier>compose_mosaic</ows:Identifier>
    <ows:Title>Compose Mosaic</ows:Title>
    <ows:Abstract>Trigger a process to compose a mosaic over a time
range</ows:Abstract>
  </wps:Process>
  <wps:Status creationTime="2019-08-21T15:34:49Z">
    <wps:ProcessSucceeded>PyWPS Process Compose Mosaic
finished</wps:ProcessSucceeded>
  </wps:Status>
  <wps:ProcessOutputs>
    <wps:Output>
      <ows:Identifier>jobID</ows:Identifier>
      <ows:Title>Job Identifier</ows:Title>
      <ows:Abstract></ows:Abstract>
      <wps>Data>
        <wps:LiteralData uom="urn:ogc:def:uom:OGC:1.0:unity" dataType="string"
>d7111976-3a9f-401c-a3d2-c1a5c30329ac</wps:LiteralData>
      </wps>Data>
    </wps:Output>
  </wps:ProcessOutputs>
</wps:ExecuteResponse>
```

8.3.3. Cloud free mosaic generation status query

In order to query the system about the status of a Job, a specific WPS execute service is exposed with the following parameters.

Table 4. Cloud free mosaic generation status query request parameters

Keyword	Description	Sample Value
<i>identifier</i>	The name of action to be executed. Fixed get_status	get_status
<i>job_id</i>	The Job ID returned by the cloud free mosaic generation XML sample response.	d7111976-3a9f-401c-a3d2-c1a5c30329ac

An example of this request is:

```
https://borealweb.nfis.org/tb15d100wps?  
service=WPS&  
version=1.0.0&  
request=Execute&  
identifier=get_status  
datainputs=job_id=d7111976-3a9f-401c-a3d2-c1a5c30329ac
```

The status of the job follows the flow defined in [Figure 9](#).

Cloud free mosaic generation status query XML sample response

```
<?xml version="1.0" encoding="UTF-8"?>  
<wps:ExecuteResponse xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:ows=  
"http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi=  
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=  
"http://www.opengis.net/wps/1.0.0 ../wpsExecute_response.xsd" service="WPS" version=  
"1.0.0" xml:lang="en-US" serviceInstance=  
"https://borealweb.nfis.org/tb15d100wps?request=GetCapabilities&service=WPS"  
statusLocation="">  
  <wps:Process wps:processVersion="1.0.0">  
    <ows:Identifier>get_status</ows:Identifier>  
    <ows:Title>Get Status</ows:Title>  
    <ows:Abstract>Retrieve the Job Status</ows:Abstract>  
  </wps:Process>  
  <wps:Status creationTime="2019-08-21T15:46:22Z">  
    <wps:ProcessSucceeded>PyWPS Process Get Status finished</wps:ProcessSucceeded>  
  </wps:Status>  
  <wps:ProcessOutputs>  
    <wps:Output>  
      <ows:Identifier>status</ows:Identifier>  
      <ows:Title>Job Status</ows:Title>  
      <ows:Abstract></ows:Abstract>  
      <wps>Data>  
        <wps:LiteralData uom="urn:ogc:def:uom:OGC:1.0:unity" dataType="string"  
>finished</wps:LiteralData>  
      </wps>Data>  
    </wps:Output>  
  </wps:ProcessOutputs>  
</wps:ExecuteResponse>
```

8.3.4. Cloud free mosaic download

When the status query indicates processing has completed for the required Job ID, the URL for downloading of generated mosaic can be retrieved. A specific WPS execute service is exposed with the following parameters:

Table 5. Cloud free mosaic generation status query request parameters

Keyword	Description	Sample Value
<i>identifier</i>	The name of action to be executed. Fixed get_result	get_result
<i>job_id</i>	The Job ID returned by the cloud free mosaic generation XML sample response.	d7111976-3a9f-401c-a3d2-c1a5c30329ac

An example of this request is:

```
https://borealweb.nfis.org/tb15d100wps?
service=WPS&
version=1.0.0&
request=Execute&
identifier=get_result
datainputs=job_id=d7111976-3a9f-401c-a3d2-c1a5c30329ac
```

If the requested job is finished and the completion time is within the retention time period, the URL of the GeoTIFF RGB cloud free mosaic is returned. The URL is used to download the mosaic via standard HTTP protocol.

```

<?xml version="1.0" encoding="UTF-8"?>
<wps:ExecuteResponse xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:ows=
"http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://www.opengis.net/wps/1.0.0 ../wpsExecute_response.xsd" service="WPS" version=
"1.0.0" xml:lang="en-US" serviceInstance=
"https://borealweb.nfis.org/tb15d100wps?request=GetCapabilities&service=WPS"
statusLocation="">
  <wps:Process wps:processVersion="1.0.0">
    <ows:Identifier>get_result</ows:Identifier>
    <ows:Title>Get Result</ows:Title>
    <ows:Abstract>Retrieve the Job Result</ows:Abstract>
  </wps:Process>
  <wps:Status creationTime="2019-08-21T15:46:53Z">
    <wps:ProcessSucceeded>PyWPS Process Get Result finished</wps:ProcessSucceeded>
  </wps:Status>
  <wps:ProcessOutputs>
    <wps:Output>
      <ows:Identifier>status</ows:Identifier>
      <ows:Title>Job Status</ows:Title>
      <ows:Abstract></ows:Abstract>
      <wps:Data>
        <wps:LiteralData uom="urn:ogc:def:uom:OGC:1.0:unity" dataType="string"
>https://borealweb.nfis.org/tb15d100wps/d7111976-3a9f-401c-a3d2-
c1a5c30329ac</wps:LiteralData>
      </wps:Data>
    </wps:Output>
  </wps:ProcessOutputs>
</wps:ExecuteResponse>

```

8.3.5. Orchestrator

This software component is the core of the ML system and is in charge of searching and downloading product bands, loading and triggering the model, and creating the final cloud free mosaic in GeoTIFF RGB format. In order to optimize the execution performance of the Orchestrator (considering also that downloading of product bands is time consuming), the Orchestrator was designed with the following requirements:

- Only one product at the time is handled
- Product bands are retrieved **only** if the area covered by the current product still contains some clouds in the temporary mosaic; otherwise it skips to the next product
- The mosaic generation ends as soon as the **Petawawa** area is entirely cloud free, or imagery products are no longer available.

The **NRCan** forestry **CSW** service endpoint is located at <https://saforah2.nfis.org/geonetwork-main/srv/eng/csw> [<https://saforah2.nfis.org/geonetwork-main/srv/eng/csw>].

Filtering and metadata result assumption

NOTE

The GetRecords request is sent with **prf** as specific filter in order to retrieve only products covering the Petawawa Research Forest. The csw:GetRecordsResponse does not contain a dedicated field for cloud coverage but this information is available in the response (refer to the following snippet) as a "free text property". There is an assumption that this value is always present in order to skip products with a cloud coverage greater than 70%.

Following a GetRecords request, the WCS Server returns all matching products tagged with the prf string and that were acquired within the range of the start / stop parameters provided in the job request.

csw:GetRecordsResponse snippet for Cloud Coverage

```
<gmd:abstract xsi:type="gmd:PT_FreeText_PropertyType">
  <gco:CharacterString>Sentinel-2 surface reflectances images (L2A) in .SAFE format.
  The surface reflectance products were generated by applying the Sen2Cor algorithm to the
  Top of Atmosphere (L1C) Sentinel-2 images provided by the European Space Agency. For more
  information on the Sen2Cor algorithm please visit http://step.esa.int/main/third-party-
  plugins-2/sen2cor/.

  Sensor: MSI
  Platform: Sentinel2A
  Acquisition Date: 2017-07-18
  Provider: European Space Agency
  Cell Size (m): 20
  Cloud Cover (%): 7.9388
</gco:CharacterString>
...
...
...
</gmd:abstract>
```

The main configuration parameters for the Orchestrator are stored inside the config.yaml file:

```
# Logging level
logging_level: INFO

# Workspace and products download path
workspace_path: /data/ogctb15/workspace
download_path: downloads

#
# Mosaicing
#

# CSV endpoint
csw_endpoint: "https://saforah2.nfis.org/geonetwork-main/srv/eng/csw"

# Cloud coverage percentage threshold to accept image
cloud_threshold: 70

#
# Neural Network
#

# Check point paths
cnn_checkpoint_path: "/data/ogctb15/checkpoints"
cnn_checkpoint_RGB: "ModelV2-CloudDetectionNetV2_RGB_epoch-100.20190727.pth"
cnn_checkpoint_RGBNIR: "ModelV2-CloudDetectionNetV2_RGBNIR_epoch-100.20190727.pth"

# Tile size
tile_width: 224
tile_height: 224

# Petawawa shape file
petawawa_shp: '/data/ogctb15/shapefiles/prf/petawawa_research_forest.shp'

# Mosaic retention time (minutes)
retention_time: 30
```

First the Orchestrator checks if the area of the current product still contains clouds. If this is true, the bands are downloaded via a WCS endpoint (one WCS request for each band) in the dedicated job folder in the Workspace. The number of bands retrieved will be either 3 or 4 according to the required ML model.

WCS Coordinate Reference Systems

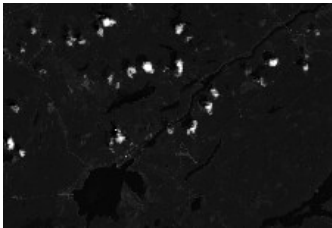
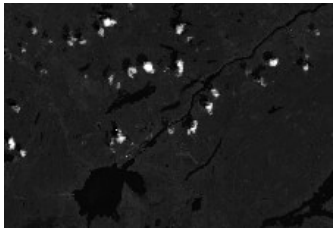
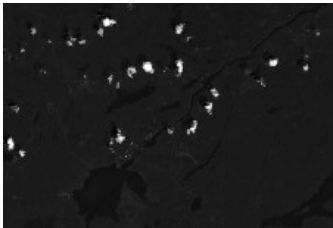

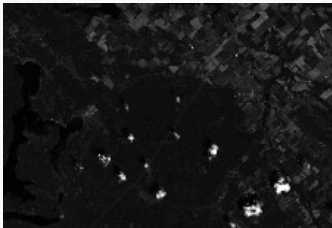
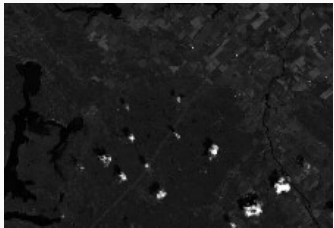
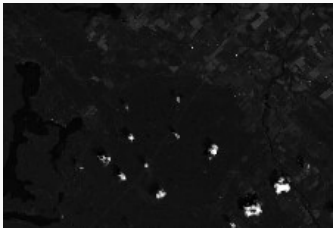

NOTE

For all **GetCoverage** requests, for the **BoundingBox** coordinates either **EPSG:26917** or **EPSG:26918** is used as **CRS**, according to the relevant product.

Once all bands are locally available, they are cut into 224 x 224 pixels tiles via the Bands Slicer to be used for training the ML model. The requested ML model is loaded and run tile by tile (each tile


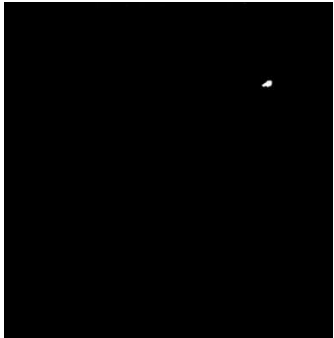

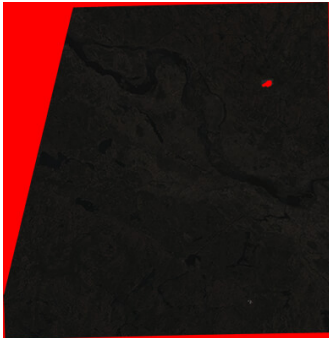
composed by the different bands). The output is an equivalently sized black & white image showing cloud presence (in white).

Table 6. Sample cloud masks generated for the RGB bands for two different tiles

Red	Green	Blue	Cloud Mask
			
			

When all tiles are processed and the relevant cloud masks created, the Mosaic Builder starts. The outputs from the ML model are merged to generate the overall cloud mask for the whole product. The cloud mask is then used as an alpha channel to maintain the areas being cloud-free and to have transparency (i.e. holes) for cloudy pixels. In the opposite way from the cloud mask, the alpha channel works by considering black as transparent (or 0% opacity) and white as solid (100% opacity). In order to use the cloud mask as an alpha channel the mask has to be inverted. When the alpha channel is applied, the resulting image becomes transparent in black portions of the alpha while in the pure white areas (the ones without clouds) the image is preserved. An example is shown in Table 7. Images in this table show transparent areas as red for viewing purposes only.

Table 7. Cloud masks generated for all bands of a single tile

Product	Cloud Mask	Alpha Channel	Final
			

The newly generated image is then stacked below previous ones (if any) in order to cover the transparencies (holes) of the previous round with the newest image. This stacking order is needed to maintain the original reverse sorting designed to show most recent imagery of the area for the requested time range at the top of the stack.

Every time the Mosaic Builder terminates, it checks whether the flattened stacked images overlapping the Petawawa Forest area still contain clouds. If clouds are present, the Orchestrator performs another

round handling the next product until either the area is either cloud free or no more products are available. When the job is terminated the final GeoTIFF RGB image is generated and stored in the Workspace and becomes available for download. From then on, any `get_status` request will return the job statistics as completed and the client can perform the `get_result` call to retrieve the download URL. The download is available until retention time expires. At that time the Workspace will be cleaned and the mosaic deleted.

8.3.5.1. Machine Learning Model

8.3.5.1.1. Model training

The D100 component is provided with two different pre-trained networks: One that works with RGB bands and the other having an additional NIR band. In either case, the training of this network with different custom parameters is triggered via a standard WPS execute call.

Table 8. Training network request parameters

Keyword	Description	Sample Value
<i>identifier</i>	The name of action to be executed. Fixed train_network	train_network
<i>modelType</i>	The type of network to be used. Either RGB or RGBNIR	RGBNIR
<i>model</i>	The name of network to be created	RGBNIR_EPOCH100_BATCH500
<i>epoch</i>	The number of epochs to be used for training	100
<i>batch</i>	The number of batch size to be used for training	500

An example of this request is:

```
https://borealweb.nfis.org/tb15d100wps?
service=WPS&
version=1.0.0&
request=Execute&
identifier=train_network
datainputs=modelType=RGBNIR;model=RGBNIR_EPOCH100_BATCH500;epoch=100;batch=500
```

If the request is accepted and queued correctly the client is returned with a Job ID (e.g. d7111976-3a9f-401c-a3d2-c1a5c30329ac) uniquely identifying the request. This Job Id is needed to perform a status query and to monitor completion of training.

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:ExecuteResponse xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:ows=
"http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://www.opengis.net/wps/1.0.0 ../wpsExecute_response.xsd" service="WPS" version=
"1.0.0" xml:lang="en-US" serviceInstance=
"https://borealweb.nfis.org/tb15d100wps?request=GetCapabilities&service=WPS"
statusLocation="">
  <wps:Process wps:processVersion="1.0.0">
    <ows:Identifier>train_network</ows:Identifier>
    <ows:Title>Training network</ows:Title>
    <ows:Abstract>Trigger a process to train a custom network</ows:Abstract>
  </wps:Process>
  <wps:Status creationTime="2019-08-21T15:34:49Z">
    <wps:ProcessSucceeded>PyWPS Process Training network
finished</wps:ProcessSucceeded>
  </wps:Status>
  <wps:ProcessOutputs>
    <wps:Output>
      <ows:Identifier>jobID</ows:Identifier>
      <ows:Title>Job Identifier</ows:Title>
      <ows:Abstract></ows:Abstract>
      <wps>Data>
        <wps:LiteralData uom="urn:ogc:def:uom:OGC:1.0:unity" dataType="string"
>d7111976-3a9f-401c-a3d2-c1a5c30329ac</wps:LiteralData>
      </wps>Data>
    </wps:Output>
  </wps:ProcessOutputs>
</wps:ExecuteResponse>
```

The job lifecycle follows the same behavior as a `compose_mosaic` execution. Therefore, with `get_status` it is possible to monitor the training status. Check if the network is available for the generation of a new mosaic or if it is still in training is important. If still in training a `get_result` execution will not provide any valuable results. As soon as the network is available, this is listed in the `DescribeProcess` response for `compose_mosaic` service.

8.3.5.1.2. Model architecture

The network was developed in Python using the PyTorch framework, whereas the architecture is based on a ResNet18 network followed by 3 deconvolution / upsample layers. Residual networks [1] were developed to solve the so called vanishing gradient problem. This problem occurs when the network is too deep. The gradients used to calculate the weights associated with the different layers quickly tend to zero, resulting in the weights never being updated, which prevents learning from taking place. In order to avoid this problem, ResNets use residual connections between layers:

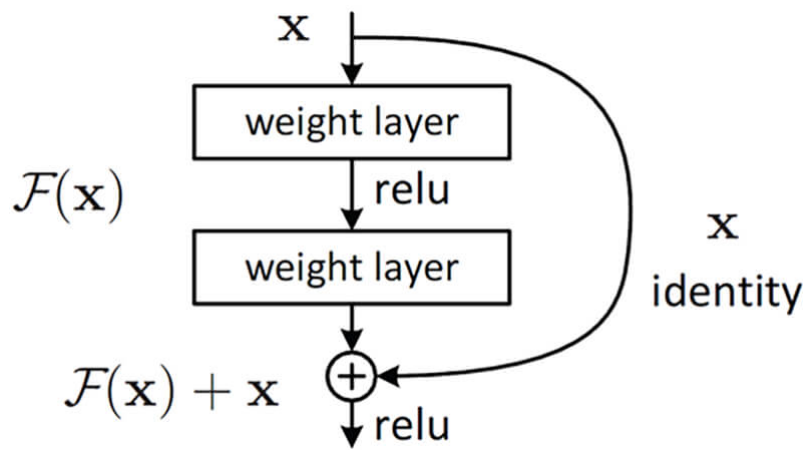


Figure 10. Residual connection

Residual connections ease the solution of the vanishing gradient problem allowing networks to grow deeper without the learning problems that would otherwise be found.

The particular flavor of ResNet that was adopted uses 18 layers (Figure 11) hence the name **ResNet18**.

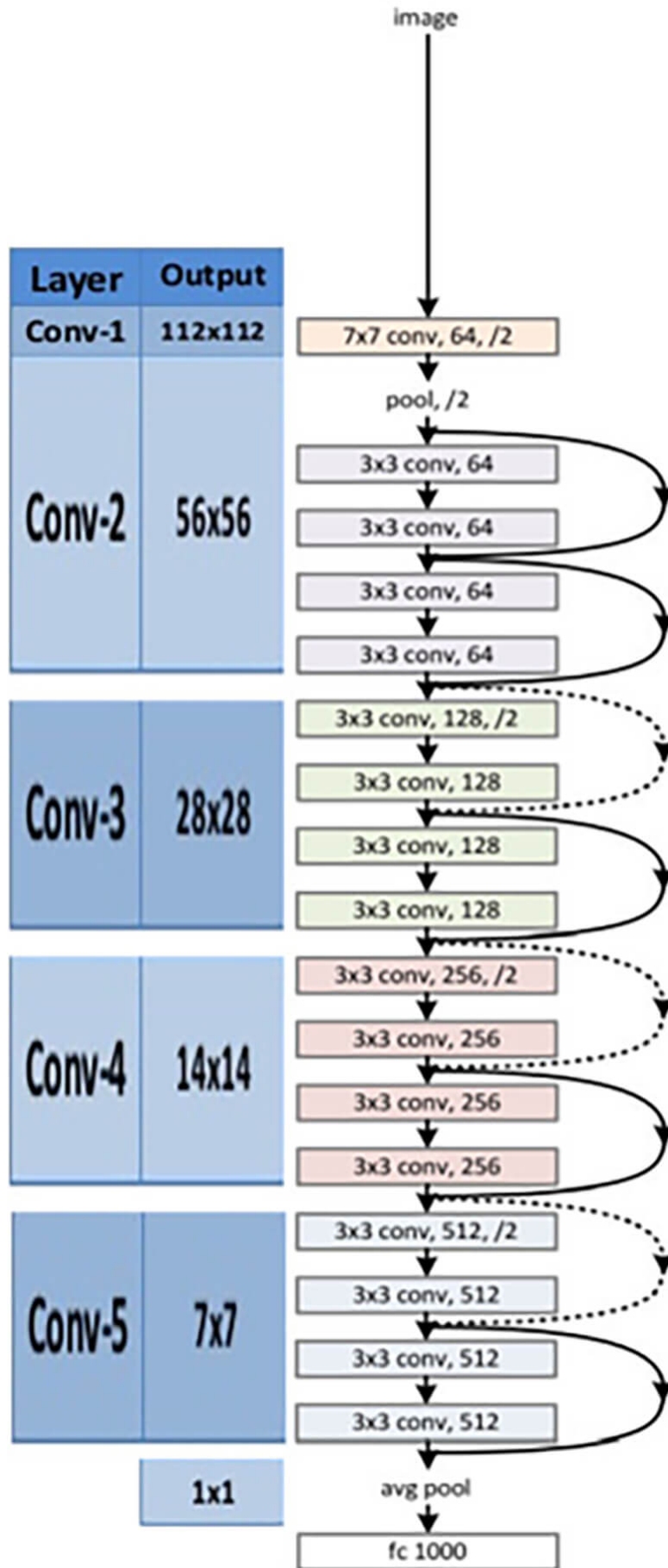


Figure 11. ResNet18

Standard **ResNets** are used to solve classification problems such as what type of clouds are present in a given image. However, the goal of the model was to produce an image with the same dimensions as the

input one, flagging every pixel occupied by a cloud. In order to achieve this, the ResNet was “truncated”, essentially feeding the output of the 6th layer to the input of the up-sample layers. This approach was successfully used in coloring networks taking input black and white images and producing colored versions as outputs (e.g. as per the Sen2Cor model). This has implications on the scale of details that can be captured by the network. A simplified diagram of the basic network used is shown in [Figure 12](#) and a summary of the full network in [Figure 13](#).

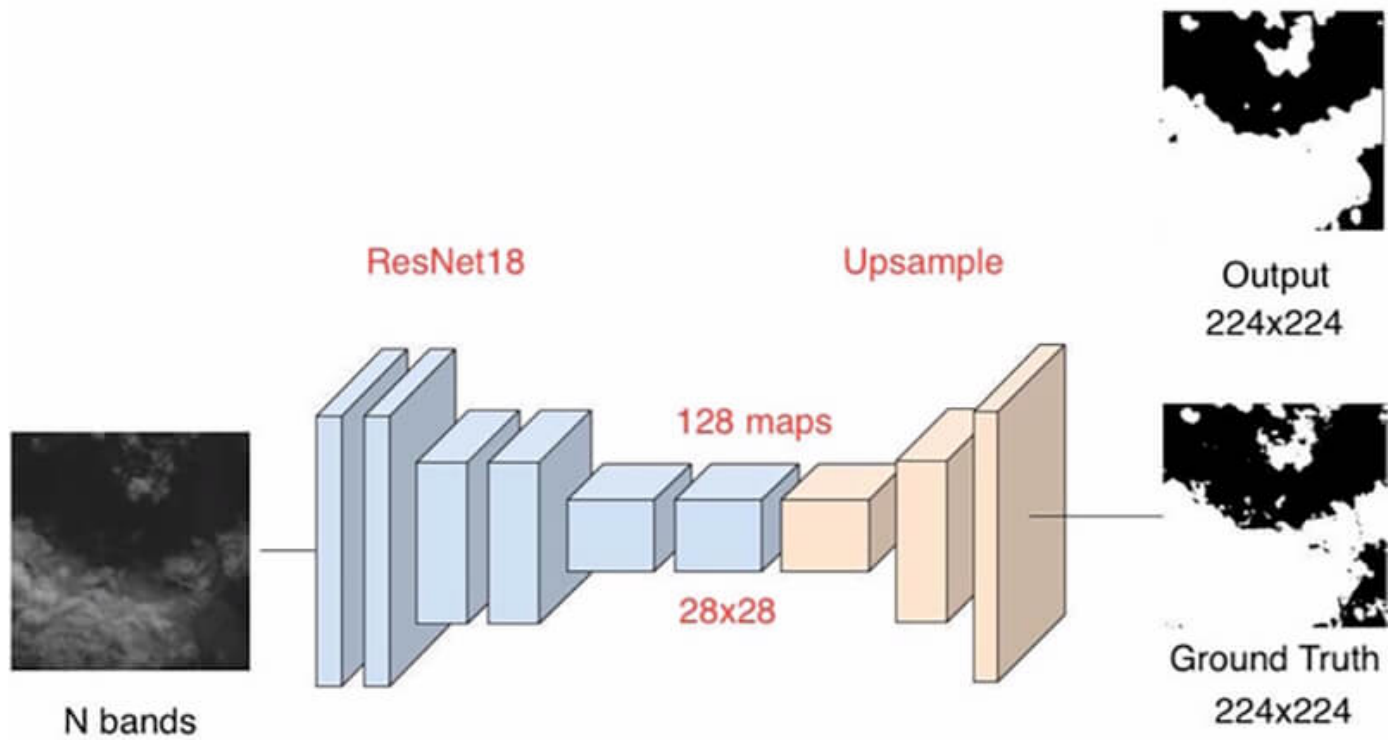


Figure 12. Network simplified diagram

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	12,544
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 128, 28, 28]	73,728
BatchNorm2d-20	[-1, 128, 28, 28]	256
ReLU-21	[-1, 128, 28, 28]	0
Conv2d-22	[-1, 128, 28, 28]	147,456
BatchNorm2d-23	[-1, 128, 28, 28]	256
Conv2d-24	[-1, 128, 28, 28]	8,192
BatchNorm2d-25	[-1, 128, 28, 28]	256
ReLU-26	[-1, 128, 28, 28]	0
BasicBlock-27	[-1, 128, 28, 28]	0
Conv2d-28	[-1, 128, 28, 28]	147,456
BatchNorm2d-29	[-1, 128, 28, 28]	256
ReLU-30	[-1, 128, 28, 28]	0
Conv2d-31	[-1, 128, 28, 28]	147,456
BatchNorm2d-32	[-1, 128, 28, 28]	256
ReLU-33	[-1, 128, 28, 28]	0
BasicBlock-34	[-1, 128, 28, 28]	0
Conv2d-35	[-1, 128, 28, 28]	147,584
BatchNorm2d-36	[-1, 128, 28, 28]	256
ReLU-37	[-1, 128, 28, 28]	0
Upsample-38	[-1, 128, 56, 56]	0
Conv2d-39	[-1, 64, 56, 56]	73,792
BatchNorm2d-40	[-1, 64, 56, 56]	128
ReLU-41	[-1, 64, 56, 56]	0
Conv2d-42	[-1, 64, 56, 56]	36,928
BatchNorm2d-43	[-1, 64, 56, 56]	128
ReLU-44	[-1, 64, 56, 56]	0
Upsample-45	[-1, 64, 112, 112]	0
Conv2d-46	[-1, 32, 112, 112]	18,464
BatchNorm2d-47	[-1, 32, 112, 112]	64
ReLU-48	[-1, 32, 112, 112]	0
Conv2d-49	[-1, 1, 112, 112]	289
Upsample-50	[-1, 1, 224, 224]	0

Total params: 963,841
 Trainable params: 963,841
 Non-trainable params: 0

Input size (MB): 0.77
 Forward/backward pass size (MB): 83.93
 Params size (MB): 3.68
 Estimated Total Size (MB): 88.37

Figure 13. Network full architecture

The input of the network consists of 3 (or 4) images (according to the bands either RGB or RGB + NIR) of size 224 x 224 pixels. For the training phase, an Adam Optimizer (**Diederik P. Kingma** and **Jimmy Lei Ba. Adam**) and a Minimum Squares Error loss function as follows was used:

$$MSE = \frac{\sum_{i=1}^n (y_i - y_i^p)^2}{n}$$

Figure 14. Minimum Squares Error loss function

Where y and y^p are the output of the network ground truth.

8.3.5.1.3. Dataset preparation

As usual with neural networks, the preparation steps for training data are very important. Publicly available datasets consisting of 18 Landsat images and four bands (RGB + NIR), each of them sliced into 384 x 384 pixels tiles, were used for training. For each of the bands, two random rotations and four mirroring operations were performed from the original 8400 images. Since the network was fed with 224 x 224 pixel tiles, two additional random crops for each input were performed. In summary, the dataset contains 8400 x 6 (rotation and mirroring) x 2 crops leading to a total of 100800 different images where 70% were used for training and 30% for testing. Note that Landsat imagery was selected as it provided the best results in cloud detection.

Dataset source

NOTE

Concerning the source of the dataset, it was decided to use Landsat imagery from a different source (i.e. <https://github.com/SorourMo/38-Cloud-A-Cloud-Segmentation-Dataset> [https://github.com/SorourMo/38-Cloud-A-Cloud-Segmentation-Dataset]), rather than the one originally provided by NRCAN. According to **S. Mohajerani** and **P. Saeedi**: [An end-to-end Cloud Detection Algorithm for Landsat 8 Imagery](https://arxiv.org/abs/1901.10077) [https://arxiv.org/abs/1901.10077], the selected dataset compared with **FMask** cloud detection resulted in a better training output (also as described later in [Table 10](#)).

8.3.5.1.4. Training

Training was performed for 100 epochs, splitting the training dataset into 150 batches. [Figure 15](#) shows the evolution of the loss with 4 bands. For every epoch the loss at the beginning of the epoch (blue squares) and the average for the epoch (green squares) is shown. At the end of training the loss is below 0.02, while with 3 bands the final loss was around 10% worse. The loss function was used to evaluate how well the model responds to the training, this includes its learning capability and how the predictions deviate from the actual result

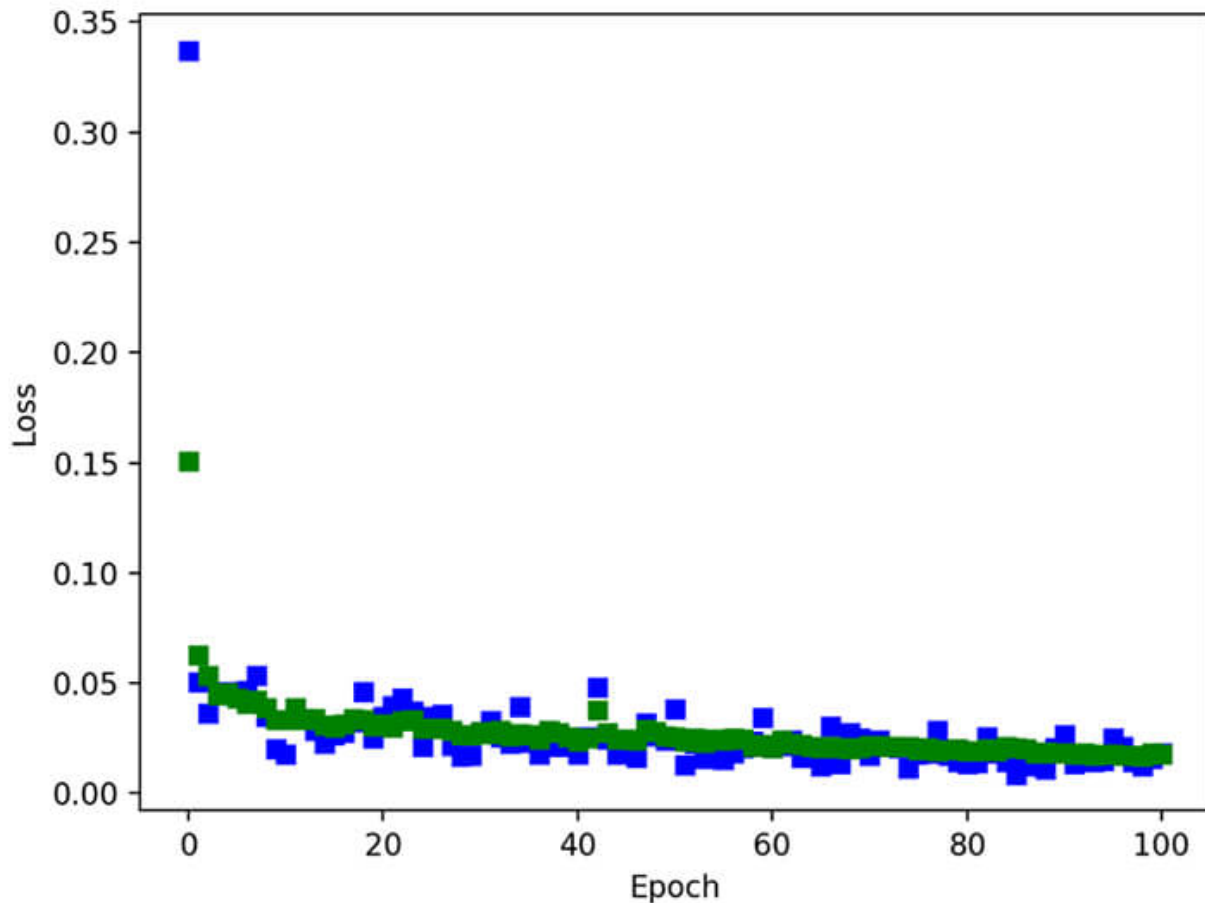


Figure 15. Evolution of loss with epoch with 4 bands

Figure 16 shows some examples of the cloud masks produced by the network during the last epoch of training for the 4-bands case.



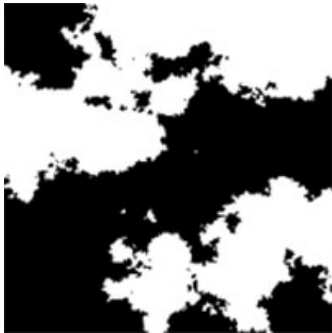

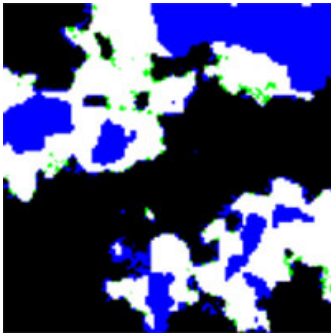
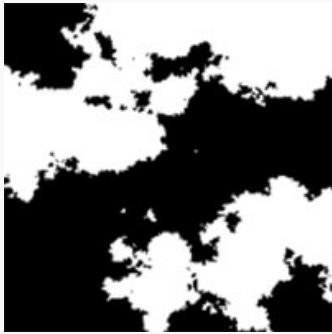

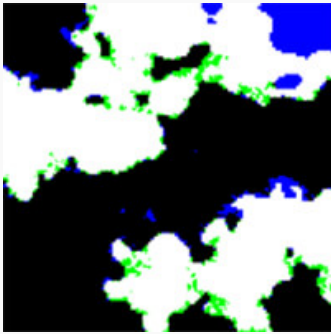
Figure 16. Four Bands training output

The left image displays the ground truth, the middle image shows the output, and the right image presents the comparison. Green shows false positives (non-cloudy area detected by the model as a cloud) and blue shows false negatives (cloudy area not recognized).

8.3.5.1.5. Testing and model accuracy

For testing the output, the ground truth images with the output of the network were compared as shown in [Figure 16](#) and looking at false positives / negatives.

Table 9. Cloud masks testing for 3 and 4 bands

Bands	Ground Truth	Output	Differences
RGB			
RGB + NIR			

From these examples, it can be noted that there is a tendency for the 3 bands model to overproduce false negatives and that the 4-band model is more accurate. This is also confirmed by Overall Accuracy in the performance metrics shown in [Table 10](#). The performance is measured by computing the [Jaccard Index](https://en.wikipedia.org/wiki/Jaccard_index) [https://en.wikipedia.org/wiki/Jaccard_index], Precision, Recall, Specificity and Overall Accuracy ([Figure 17](#)).

$$\text{Jaccard Index} = \frac{TP}{TP + FN + FP},$$

$$\text{Precision} = \frac{TP}{TP + FP},$$

$$\text{Recall} = \frac{TP}{TP + FN},$$

$$\text{Specificity} = \frac{TN}{TN + FP},$$

$$\text{Overall Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

Figure 17. Machine Learning Model performance measurements

Where **TP**, **TN**, **FP** and **FN** are respectively the total number of true positive, true negative, false positive and false negative pixels. The measured performances are reported in [Table 10](#):

Table 10. Model Accuracy

Method	Performance (%)	
	3 Bands	4 Bands
<i>Jaccard</i>	62.55	75.99
<i>Precision</i>	90.50	83.00
<i>Recall</i>	65.65	89.99
<i>Specificity</i>	95.15	92.19
<i>Overall Accuracy</i>	86.09	91.39

As can be seen, the Overall Accuracy for the 3-band case is around 5% lower than the 4-band case. Although some of the indicators seem to be better for the 3-band case, the numbers reflect the fact while the network is - in this case - good at picking pixels where there are no clouds, it is less able to identify pixels where there are clouds. This confirms the observations regarding the examples shown above.

These results for the 4-band case are very encouraging and are similar to results obtained with other networks with more complex topologies (e.g. **G. Morales, A. Ramírez and J. Telles**: [End-to-end Cloud Segmentation in High-Resolution Multispectral Satellite Imagery Using Deep Learning](https://arxiv.org/abs/1904.12743) [https://arxiv.org/abs/1904.12743]; **Z. Zhu, S. Wang and C. E. Woodcock**: [Improvement and expansion of the Fmask algorithm: cloud, cloud shadow, and snow detection for Landsats 4-7, 8, and Sentinel-2 Images](https://www.sciencedirect.com/science/article/pii/S0034425714005069) [https://www.sciencedirect.com/science/article/pii/S0034425714005069]). Since we used the same training dataset (albeit with a few variations) it is possible to make a direct comparison as the variations are not significant. The performance of implemented network is comparable to that of **FMask** and, for some of the quantities in [Table 10](#), with a slightly better percentage.

8.4. Conclusions

The results of the Petawawa cloud mosaicking ML model (D100) component implementation and testing prove that it is possible to use artificial neural networks to identify cloud coverage and to use implementations of OGC standards to create, with ease, a cloud free mosaic over the Petawawa Research Forest area. At the same time, using a reasonably simple network topology, the results obtained are very close to the ones coming from far more complex networks. Not surprisingly, the 4-band model produced better results with a higher accuracy. This is due to the fact that the network is trained with more information.

Using NRCan’s Boreal Cloud resources assisted the development of this work, reducing processing time for ML model training. This allowed the testbed participants to develop two differently trained networks. Even if a GUI were not available, the OpenStack cloud environment was easy to access and to use.

Concerning the pure development (with the exception of the network), plenty of Python libraries are available that implement OGC standards. This is another positive even if some of them do not use the latest version of the standard (e.g. the PyWPS library we used for the WPS server was limited to WPS version 1.0.0). In any case, it was quite simple to integrate the code with external CSW, WMS and WCS services.

Analyzing the ML model's overall results, in more detail, the outcome is summarized in the following three points:

1. The network is still not able to identify levels of detail at very high resolution. This is partly due to the up-sample layers of the network that start their work looking at somewhat degraded feature maps. A solution adopted in other work is to feed each level of the ResNet to an appropriate level of the up-sample part of the network. This would guarantee that the reconstructed maps also consider the higher resolution feature maps. On the other hand, using more complex methods for up-sampling can in part mitigate this problem.
2. Regarding training, a fixed training data set was used thereby allowing for comparison of networks trained with similar datasets without worrying about random effects introduced in testing different training datasets. Performing random rotations and crops on a fraction of the training data for each epoch would extend the training data set, making sure the network sees a much larger number of cases while at the same time also seeing a fixed number of common cases throughout training;
3. Looking at the improvement passing from 3 to 4 bands, very likely the accuracy of the network can be somewhat enhanced by performing training including additional bands such as, considering Landsat-8 case, the cirrus band (band 9) and bands 6 and 7 (short-wave infrared bands).

Chapter 9. Petawawa Land Classification Model

9.1. Pixel-wise Classification with Deep Learning

Applying deep learning techniques to support the land-cover classification of the Petawawa forest area is described in this section. Pixel-wise classification, also known as Semantic Segmentation, is used to assign a class label to every pixel of an image instead of just one label for the whole image.

9.1.1. Dataset

This classification component takes the output from component D100 as the input for the deep classifier. It is represented by Landsat-8 cloud-free images (in .tif format and with three bands).

The label data, representing a ground-truth land cover classification of the Petawawa Research Forest, can be downloaded from [this link](https://pfc.cfsnet.nfis.org/cgi-bin/getDownload.cgi?MAPSHEET_ID=031F,031K) [https://pfc.cfsnet.nfis.org/cgi-bin/getDownload.cgi?MAPSHEET_ID=031F,031K]. Grayscale pixel values are defined to represent specific land cover classes.

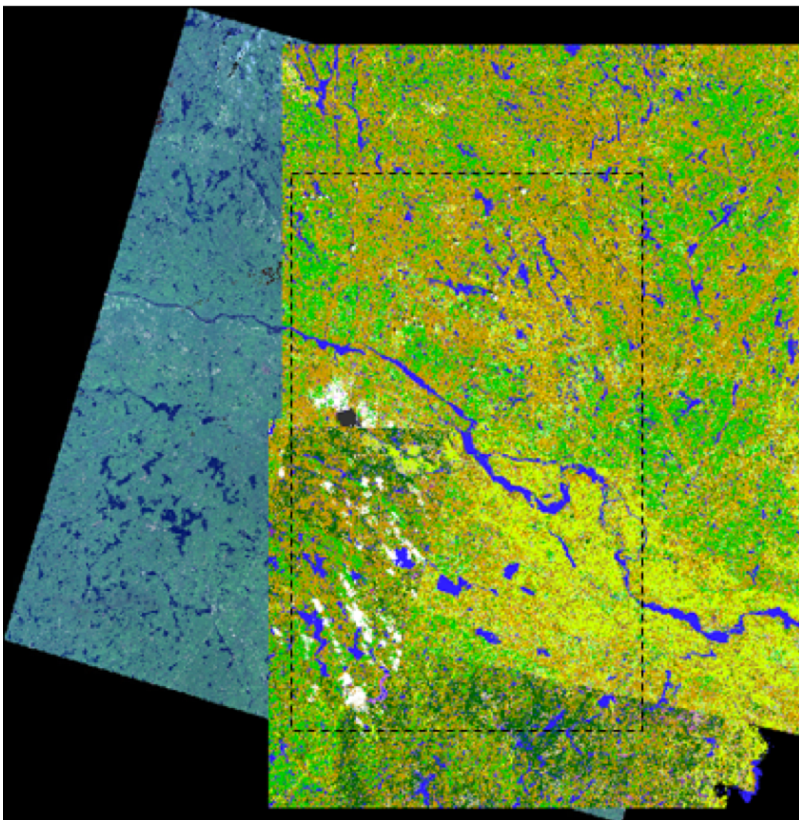


Figure 18. The remote sensing image (bottom) and the label image (top).

Only the overlapping part (the dashed box shown on Figure 18) of both the Landsat-8 imagery and the land cover classification dataset were used for this experiment. The data were eventually split into 375 pairs of remotely sensed and label image patches, with a size of 256*256 pixels. These paired patches were then randomly split into training set and validation set of 9:1. Specifically, the training set contains 337 image pairs, and the validation set contains 38 pairs.

For this dataset, there are a total of 21 land cover classes. The table below shows that the pixel

distributions of the classes are unbalanced.

Table 11. Class distribution of the TreeSpecies dataset

#	Class	label value	pixels in training set	pixels in validation set
0	Cloud	11	587160	45842
1	Shadow	12	84182	1793
2	Water	20	2101281	209702
3	Rock/Rubble	32	14021	183
4	Exposed land	33	453132	67174
5	Bryoids	40	1692	47
6	Shrub tall	51	1126655	106207
7	Shrub low	52	122065	12193
8	Wetland-tree	81	136808	16209
9	Wetland-shrub	82	101967	7259
10	Wetland-herb	83	109245	7094
11	Herb	100	2021569	207416
12	Coniferous Dense	211	1964217	161203
13	Coniferous Open	212	2744294	23303
14	Coniferous Sparse	213	5942	587
15	Broadleaf Dense	221	4146148	392693
16	Broadleaf Open	222	1317120	125172
17	Mixedwood Dense	223	10508786	947408
18	Mixedwood Open	232	1079412	100624
19	Mixedwood Sparse	233	456503	43564
20	No data	0	86902	4695

9.1.2. Model

9.1.2.1. Choosing a deep learning model for the pixel-wise classification

In deep learning, a [convolutional neural network \(CNN\)](https://en.wikipedia.org/wiki/Convolutional_neural_network) [https://en.wikipedia.org/wiki/Convolutional_neural_network] is probably one of the most commonly used models for analyzing visual images. The original CNN is more suitable for an image-level classification. However, CNN can still be used for pixel-wise classifications by taking a small area around each pixel as the input for training and prediction. However, the process would be very computationally intensive.

Fully Convolutional Networks (FCNs) are based on the original CNN, but the fully connected layers in CNN are replaced by convolutional layers. Specifically, an FCN contains an encoder and a decoder. The encoder gradually transforms an input into a series of representations with smaller spatial dimensions, and then the decoder restores the representation to an output by deconvolution. In this way, FCNs maintain a 2D structure of feature maps and is the first to implement the semantic segmentation of the

image, i.e. the pixel-wise classification. However, one of the problems with FCNs is that their segmentation results are often coarse. This is because some position information gets lost during the pooling operation in the decoder.

U-NETs are an evolution of FCNs. They follow the encoder-decoder structure of FCNs, but with the additional use of shortcut connections between the mirrored layers to easily pass the details of objects from the encoder to the decoder. As a result, the segmentation result is precise and fine-grained.

SegNet also follows the encoder-decoder pattern. It is very similar to U-Net, but the upsampling method is quite different. The upsampling in SegNet takes use of max-pooling indices which were stored during downsampling in the encoder. This makes the model smaller and requires less memory.

9.1.2.2. The SegNet architecture

SegNet [<http://mi.eng.cam.ac.uk/projects/segnet/>] is therefore used as the deep classifier. The architecture is shown below. The architecture consists of an encoder based on the 13 convolutional layers of the VGG-16 [42] and a decoder which is a reverse process of the encoder.

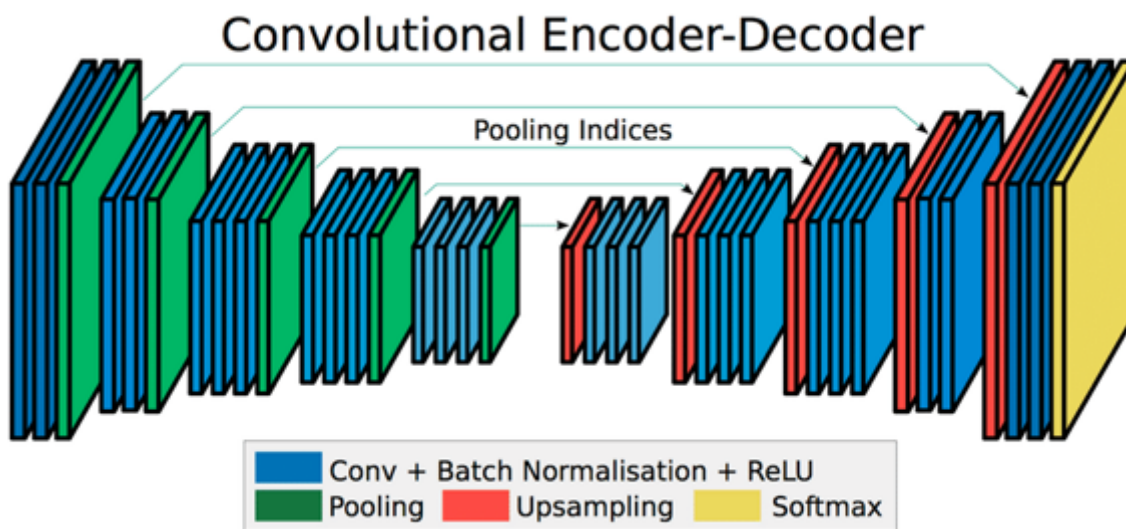


Figure 19. Architecture of SegNet.

Blocks	# conv/deconv layers	#Filters	Output size
conv_block_1	2	64	128×128
conv_block_2	2	128	64×64
conv_block_3	3	256	32×32
conv_block_4	3	512	16×16
conv_block_5	3	512	8×8
deconv_block_5	3	512	16×16
deconv_block_4	3	512	32×32
deconv_block_3	3	256	64×64
deconv_block_2	2	128	128×128
deconv_block_1	2	64	256×256

9.1.3. Results

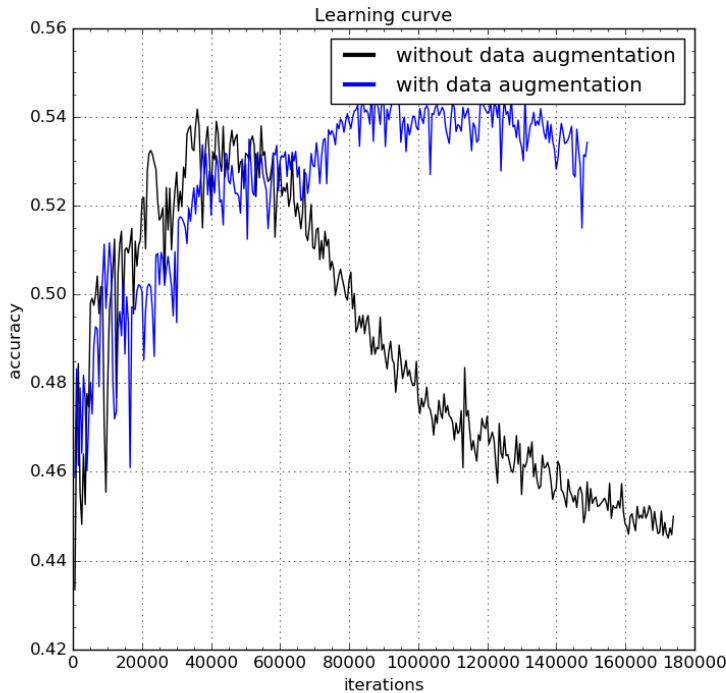


Figure 20. Accuracy on test data. (Black: without augmentation; Blue: with augmentation)

Figure 20. shows the learning curves of the model during training on the validation dataset with regards to accuracy. Accuracy indicates that, among all the pixels, how many of them are correctly classified by the model.

9.1.3.1. The learning behavior of the model during training

9.1.3.1.1. 1) Overfitting problem

The model (without augmentation) performs best when the number of iterations reaches about 40,000 and at that point the accuracy goes down. This is the overfitting phenomenon. Overfitting refers to a model that models the training data too well but cannot generalize well for new data. This happens when a model is trained too much or the training data set is too small.

9.1.3.1.2. 2) Solution

Accordingly, there are two simple ways to avoid overfitting. First, stop the training earlier, such as, before 40,000 iterations. Second, try to use a large dataset for training. Sometimes, an existing dataset is too limited and small, but can be enriched by data augmentation. Data augmentation is a technique to artificially create new data from the original data. For example, data augmentation was done by rotating, translation and flipping. The training dataset was expanded 7 times. As a result, as indicated by the blue line in the Figure 20, the model trained with augmented dataset is less likely to be over fitted.

9.1.3.2. Model performance

9.1.3.2.1. 1) Poor accuracy

As seen from Figure 20, the accuracy attained is low (0.545). But as there are 20 classes in the classification task, the results are acceptable for this Testbed.

9.1.3.2.2. 2) The reasons

- Deep learning has great advantages for image-level classification but has limited potential for pixel-wise classification. High-level semantic information needs to be extracted from the low-level features for high level image-level classification. The model architecture makes deep learning suitable for multi-level learning, which is useful for image-level classification. However, for pixel-wise classification, the task of predicting the label of pixels in the RS image is relatively shallow, so there is less opportunity for a deep learning model to be performant. In a review article, the authors also conclude that the achievement of DL techniques in pixel-based classification has not been groundbreaking.
- The test dataset presents a forest area with fine classes. So, there is a similarity between different classes (e.g. Coniferous, Herb, Broadleaf and Mixedwood). This makes the classification difficult. In contrast, many RS image classifications with deep learning techniques only focus on urban areas, where fewer classes (e.g. Buildings, Roads, Water, and Vegetation) that differ greatly.
- The class imbalance problem for this data set is severe. Therefore, the imbalance complicates the utility of the outputs. Even if the training set is large, the size of minority classes is still small due to their extremely low proportion, which will cause sparseness in minority classes. Second, standard classifiers with overall accuracy tend to be overwhelmed by majority classes. Therefore, minority classes are ignored. These issues compound to negatively influence performance.
- Additional experiments with exactly the same model but using a different dataset (i.e. street view image dataset [CamVid](http://mi.eng.cam.ac.uk/research/projects/VideoRec/) [http://mi.eng.cam.ac.uk/research/projects/VideoRec/]) has been conducted, and the best accuracy obtained is about 0.85. This indicates the Petawawa forest dataset could be a particularly challenging dataset to classify. This may explain the sub optimal results.

9.1.3.2.3. 3) Possible improvements

For the class imbalance problem, there might be several ways to overcome the issue, including:

- Changing the evaluation metric: Use other metrics like ROC (i.e., Receiver Operating Characteristic) rather than accuracy.
- Over-sampling majority classes and under-sampling minority classes.
- Class-weighting: Increase the weight of minority classes. When minority classes are misclassified, the loss value should be multiplied by the corresponding weight, so as to make the classifier pay more attention to such minority classes.

9.2. Implementation of Web Processing Service (WPS) for Deep Learning Model

9.2.1. Introduction of WPS wrapper implementation

To help Testbed participants and the general public obtain convenient access to the trained deep

learning model, the model was wrapped into a WPS and deployed using with Tomcat 8.5 and GeoServer 2.12 (official WPS extension included). If needed, the whole WPS can be easily deployed to any other desired server with some setup and configuration. Currently, the WPS can take any WCS or arbitrary web service that responds a valid GeoTIFF image as input, and provide an output image of land classification by performing preprocessing and deep learning model prediction. The figure below describes the general workflow of the WPS.

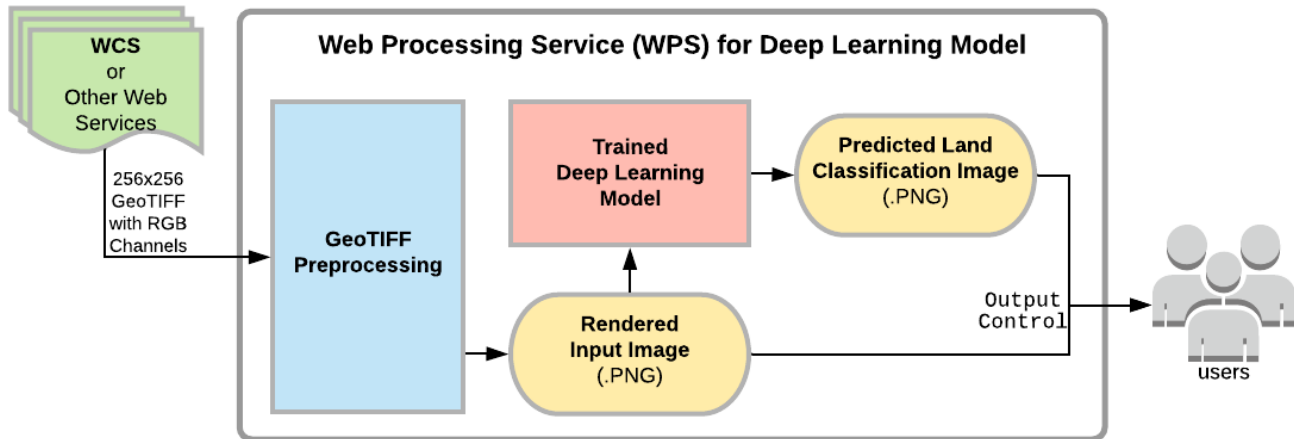


Figure 21. fig.

Once the user makes an Execute request, the WPS takes a 256 by 256 pixel GeoTIFF with RGB channels as input. This should be described in the request body. More commonly, the data would be provided by a WCS instance with a subsetting operation implemented. However, the WPS can be connected with any arbitrary web service as long as it returns GeoTIFFs that meet the implementation requirements. The WPS first downloads the input GeoTIFF from the given web service to the WPS server. The data will then be normalized and converted into a PNG image file. This operation not only generates the compatible input file for the trained deep learning model, but also provide users the convenience of comparing the input images and predictions. The normalization applied in each channel of the GeoTIFF linearly maps the values from an arbitrary range to 0-255. This PNG image file is cached in the server for future use and fed into a trained deep learning model. For each pixel in the input image, a corresponding land classification label is predicted by the model. With a proper colormap, the predicted land classification image is rendered. The WPS provides an optional control to decide whether to return the normalized/rendered input image or the prediction image (input image by default). Finally, the WPS prepares the response containing the specified image with some auxiliary information, such as response mimetype, and returns the result to the user.

9.2.2. WPS Interface Description

An interface description of the implemented WPS can be obtained using the WPS DescribeProcess request: <http://cici.lab.asu.edu/geoserver2.12.0/ows?service=wps&version=1.0.0&request=DescribeProcess&Identifier=gs:CNNProcessor>. The DescribeProcess document is:

```

<wps:ProcessDescriptions xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xml:lang="en" service="WPS" version="1.0.0" xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
  
```

```

http://schemas.opengis.net/wps/1.0.0/wpsAll.xsd">
<ProcessDescription wps:processVersion="1.0.0" statusSupported="true" storeSupported="true">
<ows:Identifier>gs:CNNProcessor</ows:Identifier>
<ows:Title>WPS4CNN</ows:Title>
<ows:Abstract>WPS for CNN model.</ows:Abstract>
<DataInputs>
<Input maxOccurs="1" minOccurs="1">
<ows:Identifier>coverage</ows:Identifier>
<ows:Title>coverage</ows:Title>
<ows:Abstract>The raster to be predicted</ows:Abstract>
<ComplexData>
<Default>
<Format>
<MimeType>image/tiff</MimeType>
</Format>
</Default>
<Supported>
<Format>
<MimeType>image/tiff</MimeType>
<Encoding>base64</Encoding>
</Format>
<Format>
<MimeType>application/arcgrid</MimeType>
</Format>
</Supported>
</ComplexData>
</Input>
<Input maxOccurs="1" minOccurs="1">
<ows:Identifier>output</ows:Identifier>
<ows:Title>output</ows:Title>
<ows:Abstract>
"prediction" for predicted image, otherwise original rendered image.
</ows:Abstract>
<LiteralData>
<ows:AnyValue/>
</LiteralData>
</Input>
</DataInputs>
<ProcessOutputs>
<Output>
<ows:Identifier>result</ows:Identifier>
<ows:Title>result</ows:Title>
<ComplexOutput>
<Default>
<Format>
<MimeType>image/png</MimeType>
</Format>
</Default>
<Supported>
<Format>

```

```
<MimeType>image/png</MimeType>  
</Format>  
</Supported>  
</ComplexOutput>  
</Output>  
</ProcessOutputs>  
</ProcessDescription>  
</wps:ProcessDescriptions>
```

There are two input parameters for this WPS instance. The first one is titled “coverage”, which should be a WCS request that delegates to the WPS server. This request should respond with a 256 by 256 pixels GeoTIFF with RGB channels. The second input is titled as “output”. It takes a string as input. If the string is exactly “prediction”, the WPS returns a predicted land classification image. Otherwise the WPS returns the normalized/rendered input image. The output of the WPS is titled as “result”. Currently only PNG as output format is supported.

9.2.3. WPS Request Example and Result Demonstration

The prediction result can be obtained using a WPS Execute request. Users should prepare a valid WPS Execute XML body that meets the WPS interface and post the XML to <http://cici.lab.asu.edu/geoserver2.12.0/wps>. An example post body is:

```

<?xml version="1.0" encoding="UTF-8"?>
<wps:Execute version="1.0.0" service="WPS" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns="http://www.opengis.net/wps/1.0.0" xmlns:wfs="http://www.opengis.net/wfs"
xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:gml="http://www.opengis.net/gml" xmlns:ogc="http://www.opengis.net/ogc" xmlns:wcs=
"http://www.opengis.net/wcs/1.1.1" xmlns:xlink="http://www.w3.org/1999/xlink"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
http://schemas.opengis.net/wps/1.0.0/wpsAll.xsd">
  <ows:Identifier>gs:CNNProcessor</ows:Identifier>
  <wps>DataInputs>
    <wps:Input>
      <ows:Identifier>coverage</ows:Identifier>
      <wps:Reference mimeType="image/tiff" xlink:href="http://geoserver/wcs"
method="POST">
        <wps:Body>
          <wcs:GetCoverage service="WCS" version="1.1.1">
            <ows:Identifier>sf:L8</ows:Identifier>
            <wcs:DomainSubset>
              <ows:BoundingBox crs=
"http://www.opengis.net/gml/srs/epsg.xml#32618">
                <ows:LowerCorner>300000.0 5020000.0</ows:LowerCorner>
                <ows:UpperCorner>307680.0 5027680.0</ows:UpperCorner>
              </ows:BoundingBox>
            </wcs:DomainSubset>
            <wcs:Output format="image/tiff"/>
          </wcs:GetCoverage>
        </wps:Body>
      </wps:Reference>
    </wps:Input>
    <wps:Input>
      <ows:Identifier>output</ows:Identifier>
      <wps>Data>
        <wps:LiteralData>prediction</wps:LiteralData>
      </wps>Data>
    </wps:Input>
  </wps>DataInputs>
  <wps:ResponseForm>
    <wps:RawDataOutput mimeType="application/octet-stream">
      <ows:Identifier>result</ows:Identifier>
    </wps:RawDataOutput>
  </wps:ResponseForm>
</wps:Execute>

```

In this request body, the input parameter “coverage” is described as a WCS post request with DomainSubset operation that subsets a 256 by 256 pixel area from the whole coverage. The post request delegates to the WPS server and sends a request to <http://geoserver/wcs>. This link is the short reference of the WCS service implemented by the same GeoServer as the WPS. The link is the same as <http://cici.lab.asu.edu/geoserver2.12.0/wcs>. Input parameter “output” is specified as “prediction”, so the response will contain the predicted land classification image. The response image and corresponding

legend in the right-hand side are demonstrated as follows:

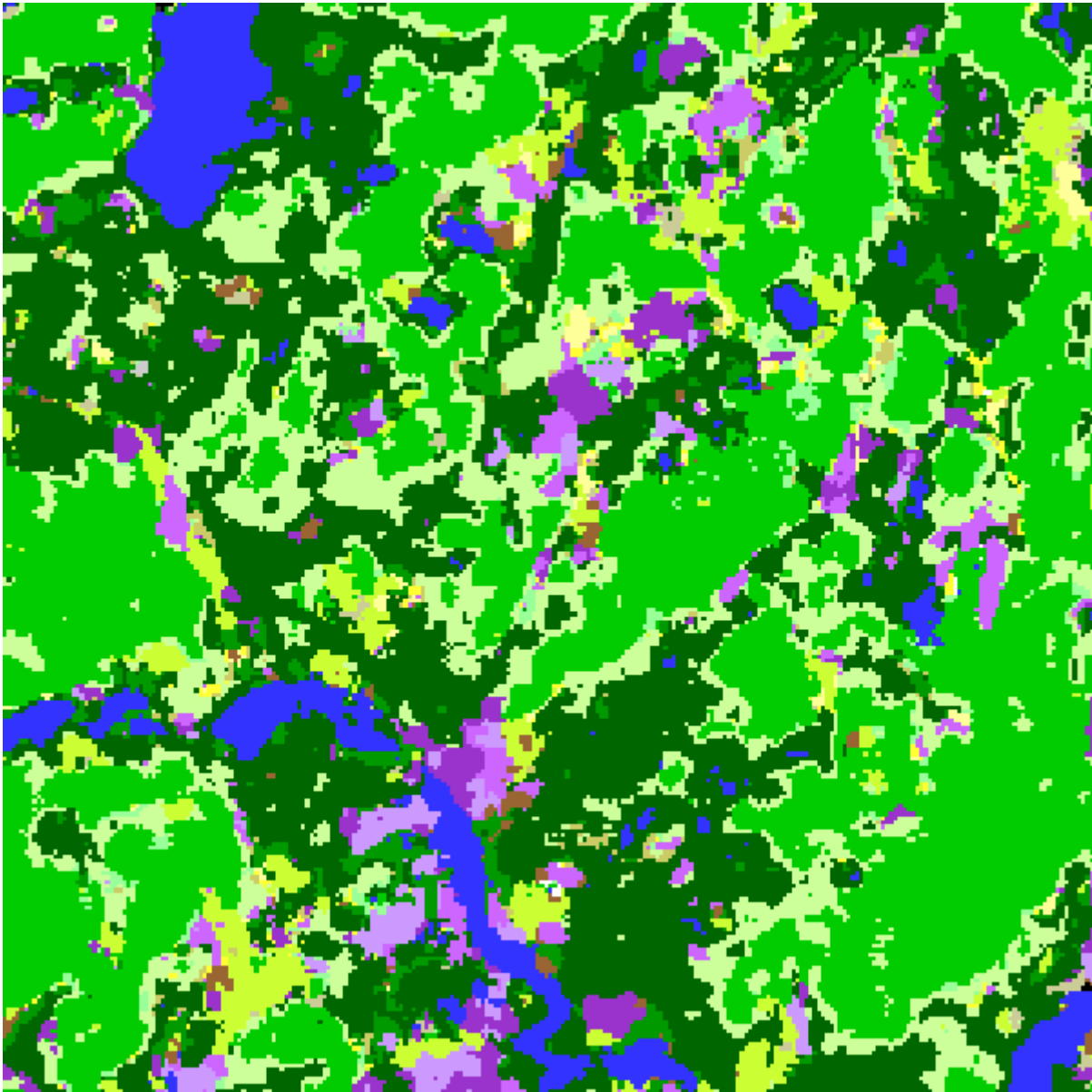
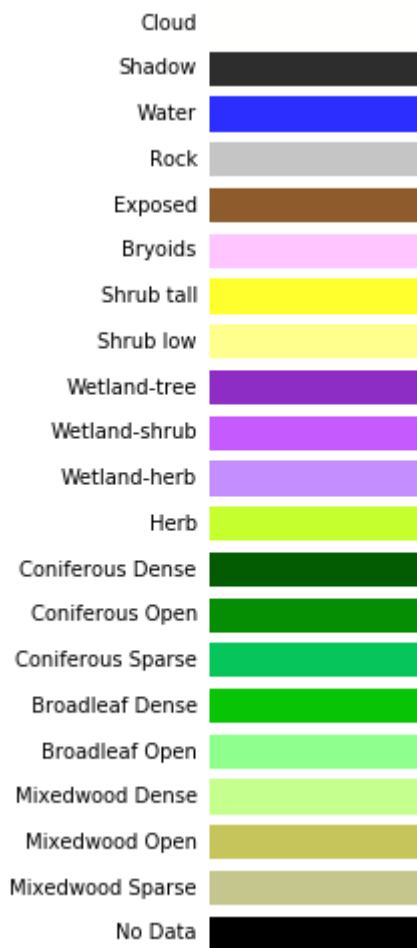


Figure 22. fig.



Change the input parameter “output” to an arbitrary word, we get the following normalized/rendered input image as the comparison:



Figure 23. fig.

Chapter 10. New Brunswick forest supply management decision maker ML model

The New Brunswick forest ML model (D102) delivered a forest supply management decision maker ML model for the province of New Brunswick. The capability was demonstrated through a live demonstration and is documented in this chapter. The component modeled the New Brunswick road network as a graph and estimated transportation costs from forest blocks to mills. Production volumes and costs of each forest block were calculated based on historical data. The component then calculated an optimal 5-year harvest plan based upon net revenues. As with the other sub-threads within the ML thread, the New Brunswick forest ML model is a standalone ML component and does not depend on other Testbed-15 ML deliverables.

10.1. Component Summary

Initial requirements were reformulated as the challenge proposed is a multidimensional optimization problem that is not easily addressable with a traditional ML approach. The goal was to produce an optimal or close-to-optimal multi-year harvesting plan for New Brunswick. However, the complexity of the problem was increased by an undefined number of harvesting teams (each acting autonomously and with its own characteristics). Additionally, there was an undefined number of transportation means to deliver raw materials of different tree species and qualities to a series of mills and factories to produce different products. A further complication is that the environment changes continuously. Wood prices vary, winter snow makes transportation difficult, trees grow increasing inventory, or decisions are made to modify the environment such as building a new road to a forest block or closing a road for maintenance.

To solve these challenges, Reinforcement Learning (RL) was used. RL is an area of ML well suited to finding the best decisions in a known environment. In a RL algorithm, an agent takes actions in an environment in order to maximize a reward. Once the algorithm was set up, it was run many times to train a neural network to learn what the best decisions to make in different situations are. This is the same methodology used by AlphaGo to learn how to play Go and beat the best human player in the world (more info at <https://arxiv.org/abs/1712.01815>). There are many resources in the Internet for a quick introduction to RL, for example <https://www.youtube.com/watch?v=JgvyzlkxFO>

The approach used in D102 was to divide the problem space into decision units or agent types within a limited environment and action-set, and then train them in parallel to find the optimal combination. Three different agent types were designed:

- Harvesting teams: responsible for the optimization of the operational decisions in an assigned forest block, from harvesting wood to building new roads.
 - Actions: continue harvesting current block, move to a different block, build a road to access the block.
 - Reward: value of wood volume harvested (independent of species) - costs (harvesting and moving wood to the nearest road).
- Transportation teams: responsible for selection of the optimal mill to take delivery of harvested wood from each forest stand.

- Actions: take wood to the closest mill, take wood to the emptiest mill, take wood to the mill with highest yield - the revenue realized from the harvested trees.
- Reward: value of wood volume transported (depends on species and mill yield) - costs of storage (if mill is full) - costs of transportation.
- Planning team: responsible for the selection of policies to allocate forest stands to harvesting and transportation teams.
 - Actions: increase priority for hardwood species, increase priority for Spruce, Pine, and Fir (SFP) species, increase priority for softwood species (affect forest allocation for harvesting teams).
 - Reward: accumulated rewards from transportation teams.

10.2. Component Design

The solution was designed using RL as the main ML methodology. In order to interact with an environment, RL is basically an agent that has a series of actions to choose from that return a state and a reward for each action. The bigger the reward, the more the agent will be inclined to take that same action in similar situations.

For the New Brunswick scenario, three different agent types were considered to divide the problem space into three different areas for decision-making: planning, harvesting and transport. Planning was responsible for deciding the species-based priorities, harvesting will take care of the operating decisions in the forest and transport to choose the best mill to take the wood to.

Three main use cases were defined to cover interaction with the user:

- *Set neural network* - the user tunes the parameters for the neural networks.
- *Train an agent* - the user individually trains the harvest, transport or planning agents.
- *Run episodes* - once all the agents are trained, it can run episodes to find the best solution for multi-year forest planning.

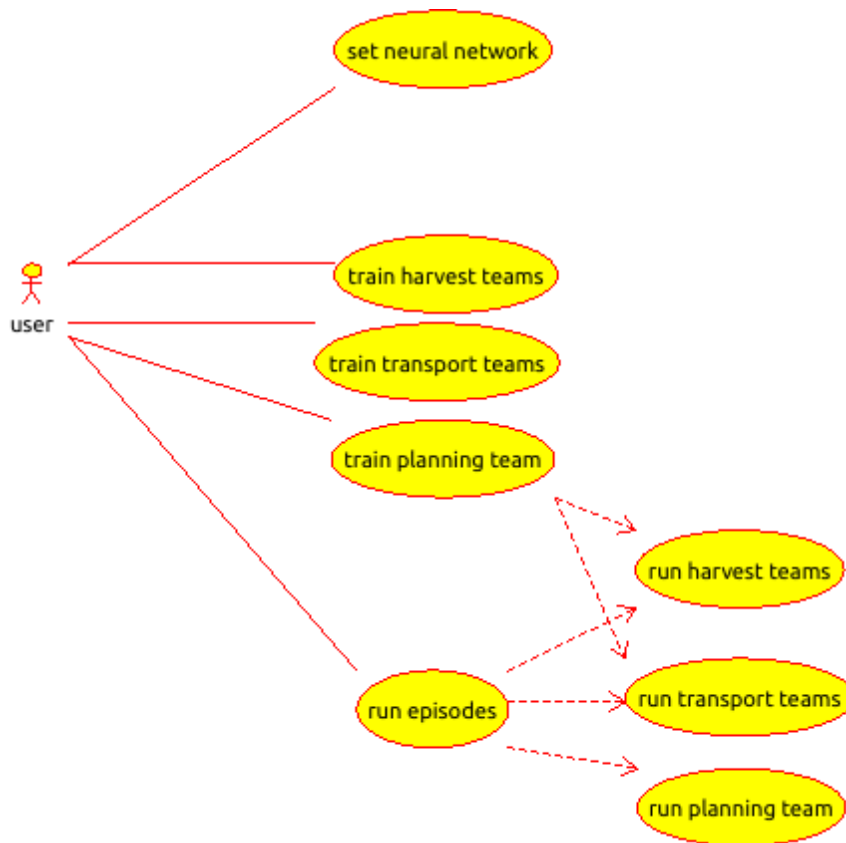


Figure 24. Use cases defined for D102

10.2.1. Set neural network

There were many parameters to tune optimally for RL. Each type of agent has a single neural network to train, even if there can be several agent instances running in parallel in New Brunswick. This is the case for the harvest and transport teams: Sharing a single brain that concentrates all experiences from all the different teams, but each acts independently in their own area.

The parameters that can be tuned for each agent type are:

- **Batch size:** the number of training samples (that is, the list of state, action, reward, new state) taken from memory to feed the neural network in each learning iteration. The higher the number, the faster the neural network learns and the longer training takes. Typical values used in this project range between 100 and 500.
- **Learning rate:** the speed at which newly acquired information overrides old information. The higher the number, the faster the neural network will learn, at the risk of becoming unstable. Typical values used in this project range between 0.01 and 0.0001.
- **Gamma:** also called the discount factor, determines how future expected rewards are considered, balancing the importance of future rewards versus immediate ones. For example, building a road to lower future harvesting costs, even if the immediate reward of this action is negative. For lower gamma values, the agent will tend to focus on immediate rewards only. Typical values used in this project range between 0.95 and 0.8.
- **Memory capacity:** also sometimes called the memory buffer, it defines the storage size of all the experiences (that is, the list of state, action, reward, new state). The bigger the value, the more variety of experiences it will store, taking a batch size of these in each learning iteration. For this project, memory capacity was set up as cyclic, making sure that new experiences (based on new,

learned behavior), override the old ones that may no longer be relevant. Typical values used in this project range between 1000 and 10000.

- **Epsilon start, end and decay:** an agent's neural network starts with an empty buffer of experiences and needs to explore the environment and experiment with the possible actions before it can figure out the rules of the game. In order to facilitate this stage, an agent will choose actions randomly with a probability calculated in the base of a threshold, calculated according to this formula: $\text{threshold} = \text{epsilon end} + (\text{epsilon start} - \text{epsilon end}) * \exp(-1. * \text{number of steps done} / \text{epsilon decay})$. Typical values used in this project for epsilon start range between 0.9 and 0.95, for epsilon end between 0.01 and 0.05, and for epsilon decay between 100 and 10000.

The following graph shows the evolution of the value of threshold; that is the probability that an agent chooses a random action vs. deciding it based on its experience. Actions will be chosen randomly as the memory is filled with experiences, and only then, when the neural network starts to learn, will the Epsilon parameters be used.

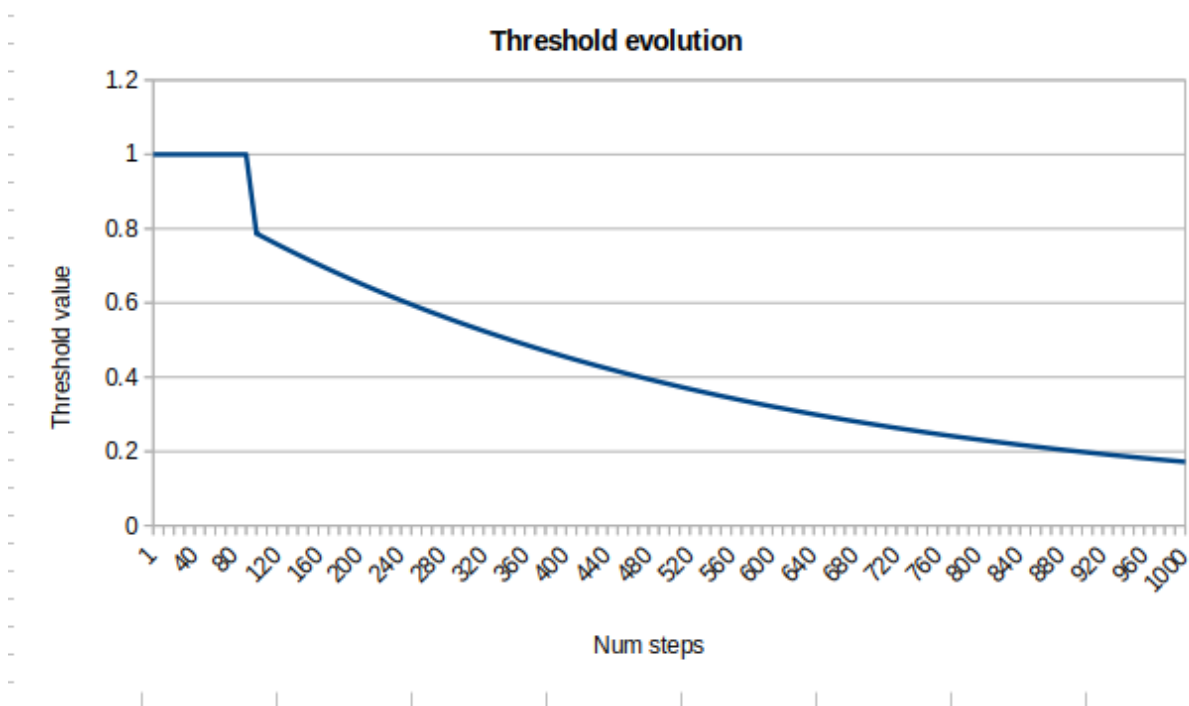


Figure 25. Threshold evolution for memory capacity = 100, Epsilon start = 0.95, Epsilon end = 0.05 and Epsilon decay = 1000

- **Number of layers and number of neurons per layer:** these parameters define the structure of each neural network. The complexity of agent decisions is low and the tests performed well with simple networks, using one or two layers with between 20 and 50 neurons each.
- **Number of episodes:** this parameter sets the number of episodes needed to complete training.

10.2.2. Train an agent

The process to train an agent is similar in all three cases. The environment provides the state to the agent, then the agent gets the best action from the neural network, modifying the state from the environment and receiving a reward. This process is stored and triggers a learning cycle, improving the neural network.

10.2.3. Harvest agent

In the case of a harvest agent that has as an assigned forest block and stand, the environment provides a state containing the following information:

- Area left to harvest in current stand
- Density of current stand
- Density of next stand in current block
- Distance to road in current stand

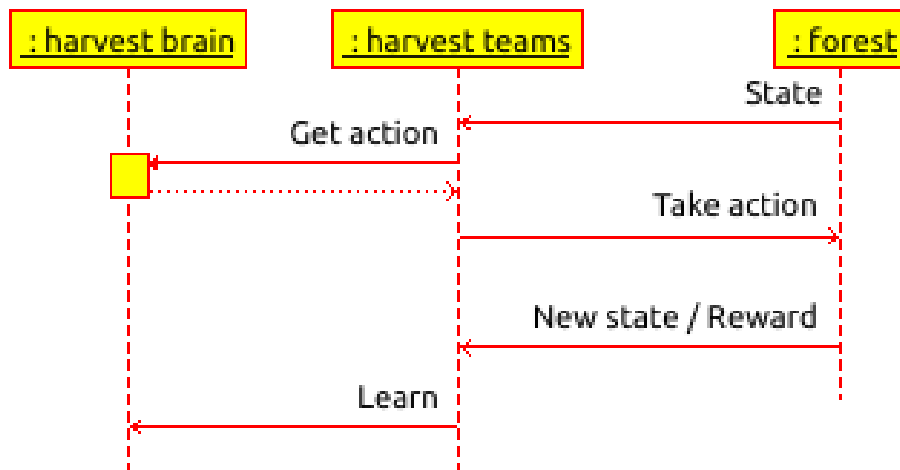


Figure 26. Training process for the harvest agent

With this information, the agent chooses one of the following actions:

- Continue harvesting current stand, which will be taken if the density is high enough. If the stand is completely harvested it will automatically move to the next stand in the block. The reward received will be a value of the chopped wood (independent of the species) minus harvesting costs.
- Move to a different block, which will be taken if the density of the current and next stands is not high enough. The decision on which block to move next is external to the agent, taking into consideration volume and density in the block, distance to the team's base and type of wood matching the priorities set by the planning team (if any). The reward received will be negative (i.e. the cost of moving the team).
- Build a road, which will be taken if the area and density left to harvest is high enough to compensate the costs that will depend on the distance to the road. The reward received will be negative (i.e. the cost to build the road)

10.2.4. Transport agent

In the case of the transport agent, which has a wood stock assigned, the environment provides a state containing the following information:

- Transport costs, stock and yield for the closest mill that fits the wood species to transport
- Transport costs, stock and yield for the emptiest mill that fits the wood species to transport
- Transport costs, stock and yield for the mill with the highest yield that fits the wood species to transport

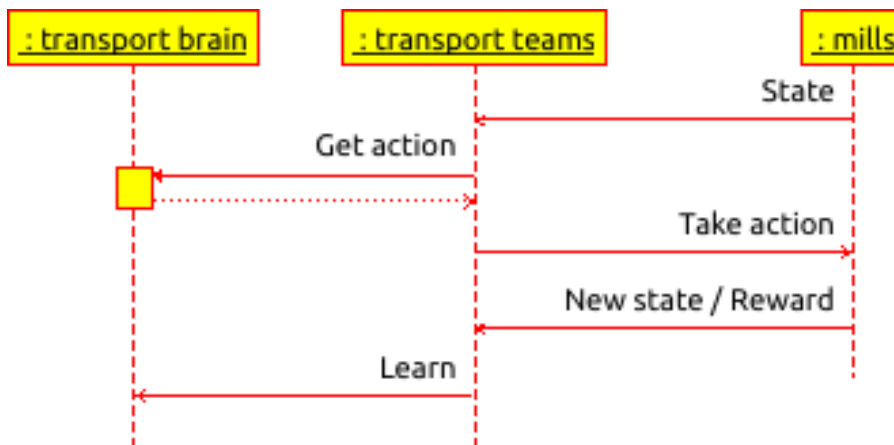


Figure 27. Training process for the transport agent

With this information, the agent chooses one of the following actions:

- Take the wood to the closest mill
- Take the wood to the emptiest mill
- Take the wood to the mill with the highest yield

The reward received will be the value of the transported wood (which will depend on the species, the type of mill, the stock of the mill and the wood price at that moment) minus the transportation cost.

10.2.5. Planning agent

The case of the planning agent is different, as this agent cannot be trained independently of the other two agents. The goal is to balance the harvesting of different species in order to optimize flow. This agent only takes into consideration the status of the stock in the different mills as well as current wood prices for each type (hardwood, SPF, softwood) and their trend.

With this information, the agent chooses one of the following actions:

- Increase priority for hardwood species
- Increase priority for SPF species
- Increase priority for softwood species

Each priority increase for one type of species makes a small priority decrease for the rest, keeping the numbers balanced. Before the agent takes the new state and the reward, in order to see the effect of its action it waits for several steps from the other teams.

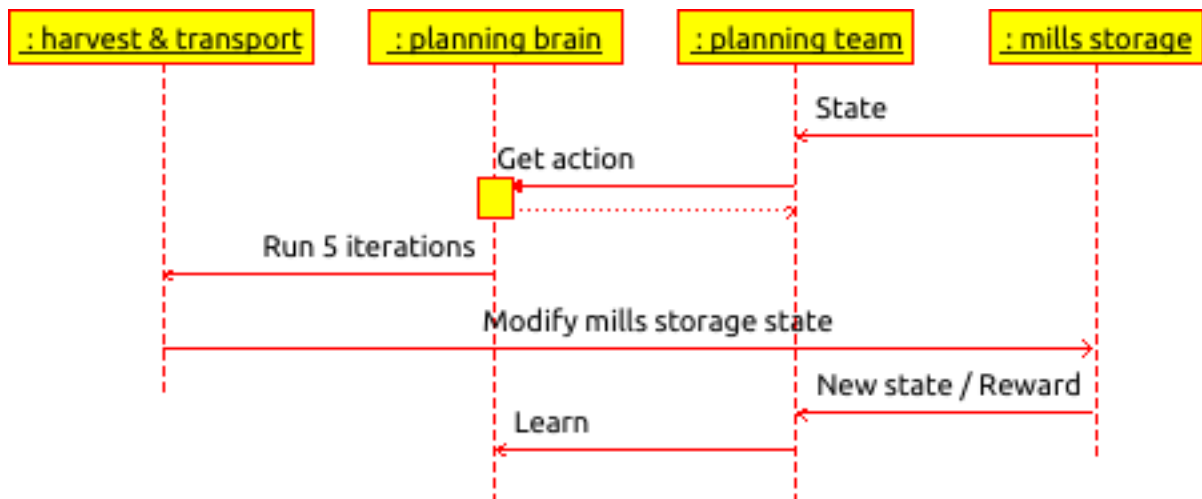


Figure 28. Training process for the planning agent

10.2.6. Run episodes

Once all agents are properly trained, they can be set to play the game harmoniously and reassess during the number of episodes set by the user. While running the episodes, the neural networks do not go through learning iterations and the threshold is set fixed to Epsilon end in order to keep a minimum randomness in the process.

The episode with the highest reward is then recorded for further manual analysis. Additionally, the top 25 percentile episodes generate aggregated statistics to find further insights that can help prepare a multi-year plan. These statistics include the forest blocks that were harvested in most of these top episodes, as well as the blocks where a road was built or the average stock in the mills.

10.3. Architecture

The architecture of the solution is described in the following diagram:

- Data is fed from NRCan's web services or from direct download (e.g. data on prices).
- These data help build several models: forest, price forecasting, and routing engine. Other models for which there is not sufficient data are made up based on educated guesses, such as agents, mills, and road building models.
- These models help train the three different agents.
- Once trained, episodes can be run in order to select the best results.

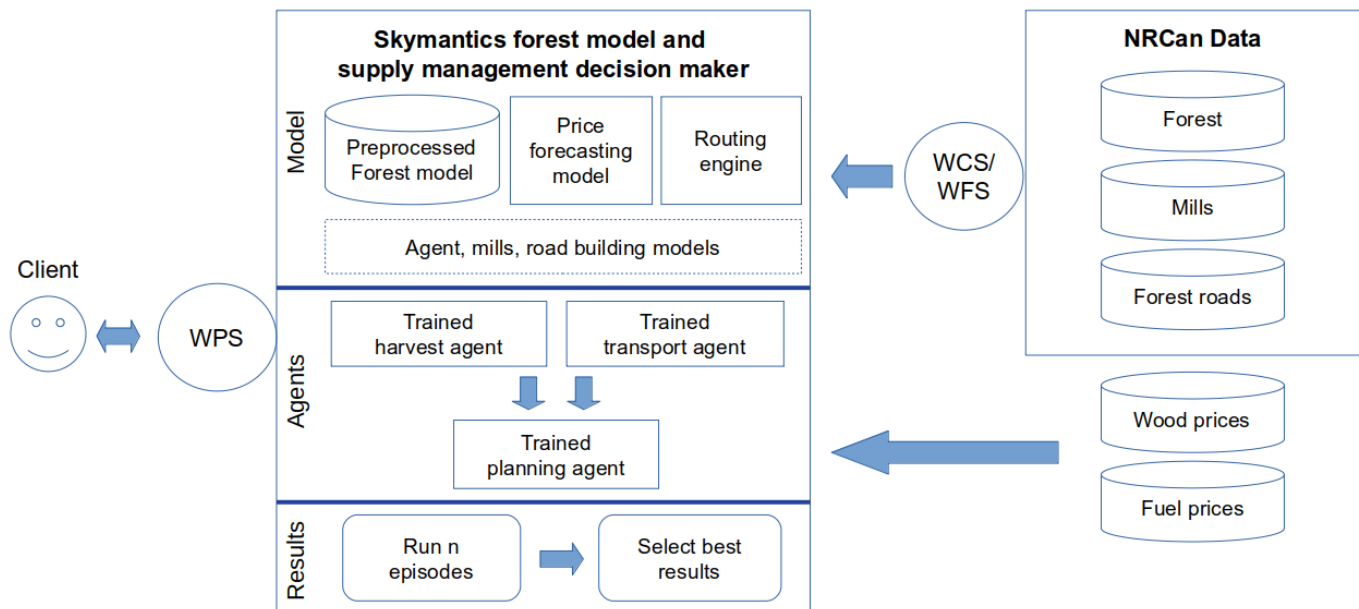


Figure 29. Architecture diagram

10.4. Input data

The Skymantics forest model and supply management decision maker has access via WCS/WFS to NRCan data, in particular layers providing information on forests, road networks, forest resource roads, and mills:

- Forest layer: with more than 800,000 elements, each with more than 80 properties, it provides very detailed information on all forest stands and blocks in New Brunswick, from stand taxonomy and tree species distribution to treatment history. It is easy to make a rough estimation of the wood volume per species in each stand based on this information, as well as its geographical location and geometrical distribution.
- New Brunswick Road Network layer and Transportation Forest Resource Roads layer: these layers provide all the information required to build a routing engine that can estimate the costs (both in time and fuel consumption) from any forest road to any mill location in New Brunswick. Moreover, it can help calculate the distance from any forest stand to the closest road.
- Mills layer: it provides the location, type, wood species consumed and product of all the mills in New Brunswick, as well as their operational status.

These data are essential to build the decision maker, but they cannot be consumed directly and need to be processed in order to build needed intermediate systems.

Apart from these, historical data on wood and fuel prices are downloaded in order to build a price forecasting model.

10.5. Routing engine

There are two possible sources of road data to build the routing engine:

- New Brunswick Road Network layer, which includes only the main road network of New Brunswick but with enough information to understand the taxonomy and properties of each road

- Transportation Forest Resource Roads layer, which includes the whole road network, including every forest road, but without much additional information

Ideally, both sources should be merged in order to have the most accurate information for each road. Due to time constraints, in this project only the Transportation Forest Resource Roads layer was used to build the routing engine.

The routing engine was implemented in Python3, making use of [ogr](https://gdal.org/) [https://gdal.org/] (to extract data from WCS/WFS) and [Vincenty's Formula](https://en.wikipedia.org/wiki/Vincenty%27s_formulae) [https://en.wikipedia.org/wiki/Vincenty%27s_formulae] (to calculate distances) libraries to speed up development. For the first stage, road data is extracted from WCS/WFS and saved in a local MySQL database to speed up data manipulation. Next, by treating road segments as edges of a graph and road junctions as nodes of the same graph, the graph is preprocessed in order to spot intersections and calculate shortcuts, as well as to infer speed limits and the road hierarchy for every edge.

Once preprocessing is complete, the graph is built and loaded into memory for faster computation. The routing algorithm implemented is A* with Contraction Hierarchy optimization, defining three hierarchy levels: 1 for forest roads, 2 for local roads and 3 for freeways and highways. With this graph and algorithm it is now possible to find optimal or close to optimal routes from any point A to any point B in New Brunswick in a fraction of a second, providing both the cost in time and the distance travelled, which can easily be used to infer the cost of transportation. For this Testbed the cost was simplified as a fixed amount per hour or fraction, as well as a fixed amount per kilometer. However, more advanced and accurate estimations are possible.

10.6. Preprocessed forest model

The forest layer provides very detailed information on all forest stands and blocks in New Brunswick. For this Testbed, the information extracted assisted in the location of each forest stand / block and drawing its limits, and helped estimate the amount of wood in the stand, grouped per type of wood (hardwood, SPF or softwood). The rest of the information was not used in this version of the forest model.

The preprocessed forest model was implemented in Python3, making use of ogr (to extract data from WCS/WFS) and Vincenty's formula (to calculate distances) libraries to speed up development. To begin, forest data is extracted from WCS/WFS and saved in a local MySQL database to speed up data manipulation. Then, several calculations are carried out and the results stored in the database, covering stand volume (total and per type of wood), density, distance to closest road and distance to all currently active mills (making use of the routing engine). As the agents need to have all data ready for consumption when performing training or running episodes, this preprocessing stage is of great importance. For example, consider the case of route calculation to mills for the transport team. For every model iteration the cost of transportation to the mills needed to be calculated several times. Taking into account that every episode is made of more than a thousand iterations, that there should be several transport teams working in parallel and that users would probably want to run several hundreds of episodes in a row, having these costs pre-calculated has a very big impact on overall system performance.

The Enhanced Forest Inventory data was not used to build the preprocessed forest model for two reasons: 1) lack of time to process the data and include the data in the model and 2) there were other

factors much more critical to build the decision maker, which were related to the business process parameters (see below) that were missing. For eventual future versions of the decision maker and once the business process parameters have been properly dimensioned, integrating the Enhanced Forest Inventory data will become important. This is in order to have more accurate numbers of wood availability in the forest stands.

10.7. Price forecasting model

There are two prices that have an important effect on the operations that are modelled in this deliverable: the wood price at the mills and fuel prices for wood transportation. Prices are acquired in the form of timeseries. Timeseries allows the price evolution according to a time interval (days, weeks or months) to be followed. Predicting future prices of either wood or fuel for the long term required for this model (5 years) is almost impossible. However, decomposing the timeseries into its three main components (trend, seasonal and residual) and analyzing them is possible. If the seasonal component can be accurately predicted and is important with respect to trend and residual components, the decomposition can serve as a good approximation.

10.7.1. Wood pricing forecasting

Wood prices are available at <https://www.nrcan.gc.ca/current-lumber-pulp-panel-prices/13309>

Being able to forecast wood prices should have a positive impact in fine-tuning the behavior of transport agents. This is because the reward they receive will be more accurately estimated and therefore they will be able to make better decisions on what wood to transport to which mill at each moment. This approach should also have a positive impact in improving the behavior of the planning agent. Instead of just monitoring the storage status of mills, the model is also aware of the price evolution of each type of wood; it therefore has a richer set of information to base its decisions.

The wood prices data presented two main challenges: 1) they are not directly related to wood species but to a mixture of wood type and mill products, and 2) they only provide 12-month data. The first challenge was solved assuming direct relations. "Softwood lumber" refers to prices for every wood from softwood species, "Panel" refers to prices for every wood from hardwood species and "Pulp" refers to prices for every wood from SPF species. This is a gross approximation and should be refined in further revisions.

The second challenge was solved by assuming that the 12-month data is actually the seasonal component of the timeseries. Again, this approximation is objectionable but necessary, as without multi-year data it is simply not possible to find the seasonal component of a timeseries.

With these assumptions, price variations might vary up to 40% in a year. This should have an important effect on the rewards and the decisions made.

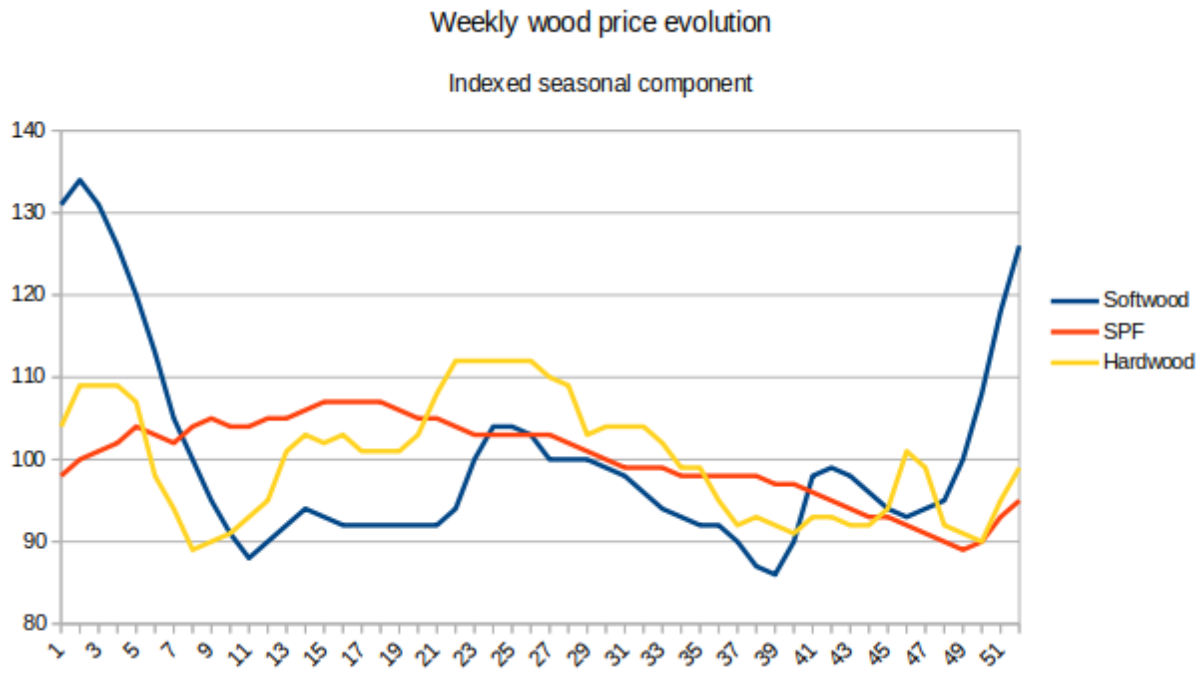


Figure 30. Yearly wood price evolution (normalized)

10.7.2. Fuel pricing forecasting

Fuel prices are available at <https://www.nrcan.gc.ca/current-lumber-pulp-panel-prices/13309>

As costs will be more accurately estimated, being able to forecast fuel prices should have a positive impact on fine-tuning the behavior of transport agents. Thus, transport teams should favor closer mills when fuel prices are high and high yield mills when fuel prices are low. The fuel prices data offer a very complete dataset of the monthly evolution of fuel prices (gasoline, diesel, others) since 1979 in different parts of Canada. For this model, the data included starts in year 2000, is from Saint John, New Brunswick, and represents diesel fuel at full service or self-service filling stations. In order to have consistent comparisons, prices are adjusted to their 2019-equivalent using historical [Canadian inflation rates](https://www.inflation.eu/) [https://www.inflation.eu/].

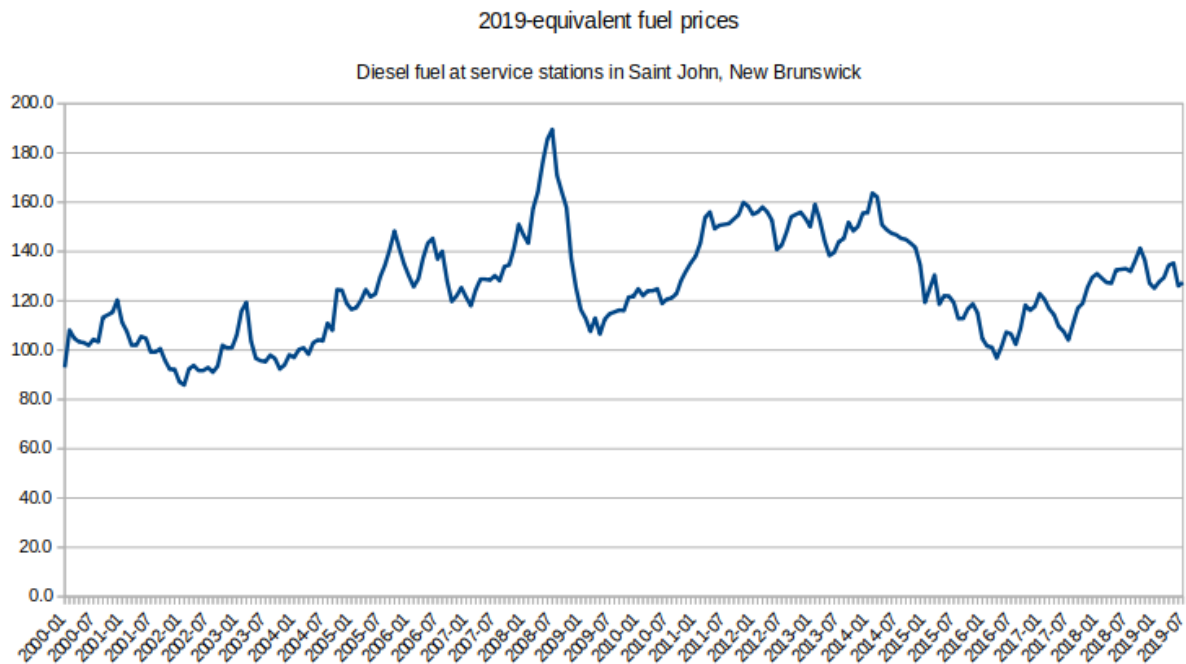


Figure 31. Fuel prices timeseries

Analyzing the data series of fuel prices with the [Dickey-Fuller test](https://en.wikipedia.org/wiki/Dickey%E2%80%93Fuller_test) [https://en.wikipedia.org/wiki/Dickey%E2%80%93Fuller_test] draws a p-value of 0.060783, meaning that the series is not stationary and can be decomposed.

Once the data are decomposed into the three main components, analyzing the residual component with the Dickey-Fuller test draws a p-value of 0.000009, meaning that that component is strongly stationary as would be expected. The different components are visible in the next image:

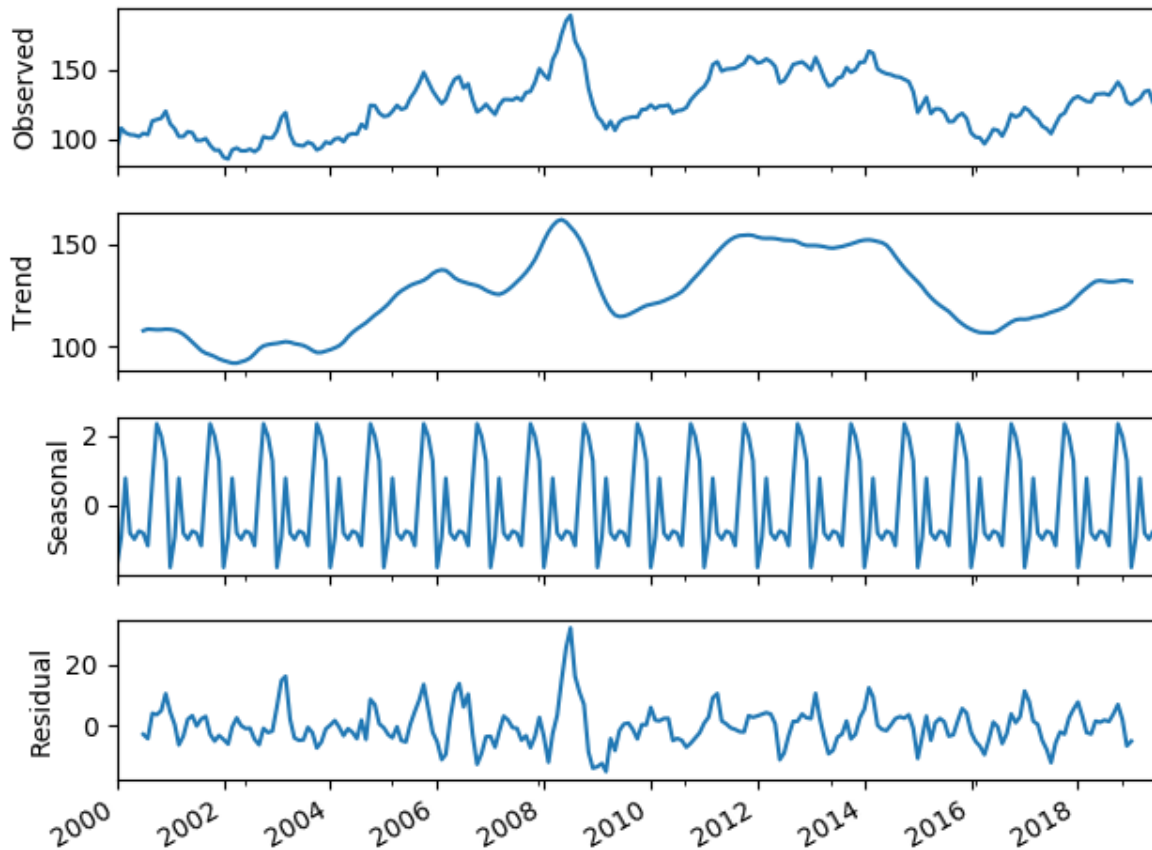


Figure 32. Fuel timeseries decomposition

Further analyzing the different components, three important conclusions are drawn:

- The trend component shows how the price of diesel is affected by economic and geopolitical factors. It is easy to spot increasing oil prices after the beginning of the 2008 crisis due to speculation, followed by an important fall due to economic slowdowns and a corresponding reduction in consumption. As well the effects of an OPEC production cut in 2009 and the outbreak of Libyan civil war in 2011 are visible in this component. It is not possible to forecast these events, which are responsible for the main part of price variation, up to \$70.
- The seasonal component shows how the price of diesel is affected by the time of the year. This is the part of pricing that is 100% predictable and that can be included in the model without any doubt. However, it accounts for just \$3 of price variation and thus, its effect in the model is very limited.
- The residual component shows how the price of diesel is affected by random aspects. This part is very difficult to foresee or interpret, and it accounts for as much as \$30 of price variation, about 10x the seasonal component.

Taking all this into consideration, the seasonal component of the diesel price will be included in the model, even though it is not going to have a significant impact nor is it going to help improve accuracy, as it is just a minimal part of the price forecast.

However, even if it does not seem possible to have an accurate long-term forecast of fuel prices, it might be interesting in the future to run the model with different scenarios depending on fuel (or wood) prices, such as high fuel prices - low wood prices, and evaluate how all the different agents would behave. Or, what would be the price level at which the rewards will be higher for teams to stay at home than going to work (that is, revenues are lower than costs). These applications are beyond the

scope of this project.

10.8. Other models - Business process parameters

Some elements of the solution, mostly models related to business processes, were built without enough data. In these cases, parameters were defined making educated guesses after some Internet research. These are the models that should be redone first in order to expect any accuracy in the results.

The parameters to properly design these models is described in the following subsections:

10.8.1. Harvest teams

- Number of harvest teams and the location of their bases
- Average area harvested per day
- Daily cost of harvesting
- Cost of transporting wood to the closest road
- Cost (time and \$) to move the harvesting team to a different forest block
- Maximum distance from base for proper operation
- Daily costs of operating far from base
- Costs of road building
- Are these parameters similar to all harvest teams or do they vary significantly?

10.8.2. Transport teams

- Number of transport teams and the location of their bases
- Volume capacity
- Time required to load and unload trucks
- Relation between transport time (and load/unload time) and costs. Effect of distance to base.
- Fuel consumption per mile, for slow and fast roads.
- Are these parameters similar to all transport teams or do they vary significantly?

10.8.3. Planning team and team allocation criteria

- What are the criteria to select a stand to harvest? Only economical (potential revenues minus costs) or are there other aspects to consider?

10.8.4. Mills

- Yield per product and type of wood consumed. Relation with public prices.
- Storage capacity per mill
- Relation between wood stock and yield
- Speed of wood consumption

- Are these parameters similar to all mill teams or do they vary significantly?

10.8.5. Machine Learning Model

PyTorch was used to create and train the neural networks. For each agent, the network starts with a layer that has the same number of artificial neurons as the number of state variables. Then, the agent creates as many hidden layers with as many artificial neurons as specified by the user. Finally, the last layer contains as many neurons as the number of actions the agent can take. Between artificial neuron layers, it adds a simple Relu activation function, that is, the positive part of its argument.

In order to train the model, an algorithm called Q-learning is used, which is able to find the optimal action taking into consideration not only the immediate reward received, but also the expected rewards generated by all successive steps applying the gamma parameter (or discount factor) described previously. Q is actually the name of the function that calculates the quality of an action used to provide the reinforcement.

However, when applying Q-learning in combination with a neural network (also called Deep Q-Learning), there is the risk of making the learning process unstable or divergent. This is because small updates to Q may substantially change the policy and data distribution, and thus change the correlations between Q and the target values. In order to overcome this loss of stability, two complementary techniques were used in this project: Experience Replay and Target Network.

- **Experience Replay:** this is where the batch size parameter comes to play, in combination with the memory capacity. At each learning iteration, instead of just feeding the ML model with the latest episodes, which would quickly lead to overfitting, the algorithm is fed with a random batch of past experiences to update the neural network. Apart from avoiding overfitting, this technique helps reduce correlation between experiences, it increases learning speed and it reuses past transitions to avoid forgetting.
- **Target Network:** the Q-value should be calculated based on the rewards received and an estimation of the future rewards produced by the subsequent new states. However, this would mean that the function (or neural network) would try to predict its own output, which would easily make the model diverge. In order to avoid that, a new neural network is created, called Target Network, which initially is a copy of the training network. It is updated every 100 steps.

With these techniques there will be a stable and convergent Deep Q Network. However, it is easy for the network by adjusting the parameters, for example:

- Set a small memory capacity, especially combined with a relatively large batch size. This way the effect of the Experience Replay technique will be diminished or canceled, and the agent will start to overfit.
- Set a small Epsilon decay, especially with a small memory capacity. The agent will not have enough time to explore the environment and will quickly start overfitting its decisions.
- Set a high learning rate, such as 0.1 or even 0.05. The model will probably become unstable.
- Set a very large number of episodes, especially with a small memory capacity and a small epsilon decay. The model will soon replace the exploratory experiences with the exploitation ones, learning only and repeatedly from its own decisions, leading to overfitting.

10.9. Component Implementation

10.9.1. Implementations

52North's WPS2.0 compliant JavaPS was provided to extend ML processing in a networked manner. Users could use the JavaPS to conduct three actions via the WPS interface: Configure the neural network, train a given agent type, and run a specific model. These actions respectively are reachable at:

```
http://52.224.191.252:8080/tb15/service?request=Execute&identifier=org.n52.javaps.service
.ConfigureNetwork&version=2.0.0
http://52.224.191.252:8080/tb15/service?request=Execute&identifier=org.n52.javaps.service
.TrainAgent&version=2.0.0
http://52.224.191.252:8080/tb15/service?request=Execute&identifier=org.n52.javaps.service
.ExecuteModel&version=2.0.0
```

10.10. WPS Request / Response examples

10.10.1. Configuration

The configuration endpoint allows users to both query current model configuration and set new configuration values. All parameters except team_type are optional.

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:Execute xmlns:wps="http://www.opengis.net/wps/2.0"
  xmlns:ows="http://www.opengis.net/ows/2.0" xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wps/2.0
http://schemas.opengis.net/wps/2.0/wps.xsd" service="WPS" mode="sync" response="document"
  version="2.0.0">
  <ows:Identifier>org.n52.javaps.service.TB15Actions</ows:Identifier>
  <wps:Input id="nn">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>20</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Input id="bs">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>200</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Input id="es">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>.95</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Input id="nl">
```

```

    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>2</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="ed">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>100</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="ee">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>.05</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="gm">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>.9</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="lr">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>.001</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="mc">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>1000</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="ne">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>100</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Input id="team_type">
    <wps:Data mimeType="text/xml">
      <wps:LiteralValue>harvest</wps:LiteralValue>
    </wps:Data>
  </wps:Input>
  <wps:Output id="configuration" mimeType="application/xml"
    transmission="value" />
</wps:Execute>

```

A successful POST request to the configuration endpoint returns back all current configuration settings for the specified agent, including those not changed by the user's configuration request:

```

<?xml version="1.0" encoding="UTF-8"?>
<wps:Result xmlns:wps="http://www.opengis.net/wps/2.0"
xmlns:ows="http://www.opengis.net/ows/2.0" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/2.0
http://schemas.opengis.net/wps/2.0/wps.xsd">
  <wps:JobID>3fbd0d14-ef79-46a5-9d12-37e468501bec</wps:JobID>
  <wps:ExpirationDate>2019-09-13T09:35:23.589033Z</wps:ExpirationDate>
  <wps:Output id="result">
    <wps>Data mimeType="application/xml" encoding="UTF-8">
      <wps:LiteralValue xmlns:wps="http://www.opengis.net/wps/2.0"
dataType="https://www.w3.org/2001/XMLSchema-datatypes#string">{"epsilon start": .95,
"epsilon end": .05, "number of layers": 2, "memory count": 1000, "agent": "harvest",
"epsilon decay": 100, "number of neurons": 20, "gamma": .9, "batch size": 200, "number of
episodes": 100, "learning rate": .001}</wps:LiteralValue>
    </wps>Data>
  </wps:Output>
</wps:Result>

```

10.10.2. Training

The various agents must be trained prior to execution. A user specifies which of the three agent types to train and the number of agents running in parallel. Harvest and transport agents must be trained before training the planning agent.

```

<?xml version="1.0" encoding="UTF-8"?>
<wps:Execute xmlns:wps="http://www.opengis.net/wps/2.0"
xmlns:ows="http://www.opengis.net/ows/2.0" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/2.0
http://schemas.opengis.net/wps/2.0/wps.xsd" service="WPS" mode="sync" response="document"
version="2.0.0">
  <ows:Identifier>org.n52.javaps.service.TB15Actions</ows:Identifier>
  <wps:Input id="team_type">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>harvest</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Input id="number_of_agents">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>7</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Output id="trainingResult" mimeType="application/xml"
transmission="value" />
</wps:Execute>

```

The WPS returns a link to a JSON collection describing the results of the training process.

```

<?xml version="1.0" encoding="UTF-8"?>
<wps:Result xmlns:wps="http://www.opengis.net/wps/2.0"
xmlns:ows="http://www.opengis.net/ows/2.0" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/2.0
http://schemas.opengis.net/wps/2.0/wps.xsd">
  <wps:JobID>5b841718-6ec9-4829-94c8-8bc605945b2b</wps:JobID>
  <wps:ExpirationDate>2019-09-13T09:35:23.589033Z</wps:ExpirationDate>
  <wps:Output id="trainingResult">
    <wps>Data mimeType="application/xml" encoding="UTF-8">
      <wps:LiteralValue xmlns:wps="http://www.opengis.net/wps/2.0"
dataType="https://www.w3.org/2001/XMLSchema-datatypes#string"> ... </wps:LiteralValue>
    </wps>Data>
  </wps:Output>
</wps:Result>

```

10.10.3. Execution

When launching the execution, users may optionally specify the number of episodes to run and the length of those episodes.

```

<?xml version="1.0" encoding="UTF-8"?>
<wps:Execute xmlns:wps="http://www.opengis.net/wps/2.0"
xmlns:ows="http://www.opengis.net/ows/2.0" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/2.0
http://schemas.opengis.net/wps/2.0/wps.xsd" service="WPS" mode="sync" response="document"
version="2.0.0">
  <ows:Identifier>org.n52.javaps.service.TB15Actions</ows:Identifier>
  <wps:Input id="episode_length">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>3</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Input id="number_of_episodes">
    <wps>Data mimeType="text/xml">
      <wps:LiteralValue>240</wps:LiteralValue>
    </wps>Data>
  </wps:Input>
  <wps:Output id="executionResult" mimeType="application/xml"
transmission="value" />
</wps:Execute>

```

Much like the training process, the WPS provides a response storing the best episode results and an aggregated summary of execution in JSON and GeoJSON. These results are best retrieved asynchronously, as long-running tasks may run into issues with connectivity being maintained across long periods of time.

Additionally, the WPS provides all compliant methods such as GetCapabilities and DescribeProcess.

10.10.4. Results

10.10.4.1. Training process

When training an agent, it is necessary to specify the agent to train and, for the case of "harvest" and "transport", it is optional to specify the number of agents used in the training. This is an important parameter, particularly for the transport agent; the more agents running in parallel, the fuller mills storage will be and, thus agents will need to deal with diminishing rewards.

During the training process, it is important to understand how agents' learn. The process is divided in two different phases: exploring and learning.

- **Exploring phase:** the agent will take actions randomly and will record the experience in the form of [state, action, reward, new state] until the memory capacity is full. These experiences will serve as the base for learning in the next state.
- **Learning phase:** at each step the agent will make a decision on which action to take. This decision will be either random or made by the neural network according to the epsilon parameters (start, end and decay). The experience will be recorded in memory, overwriting an old experience, and a batch (defined by the batch size parameter) of randomly selected experiences will be fed into the neural network for learning (at a rate defined by the learning rate parameter). As training progresses, the agent will be improving its decisions and the reward received at the end of each episode.

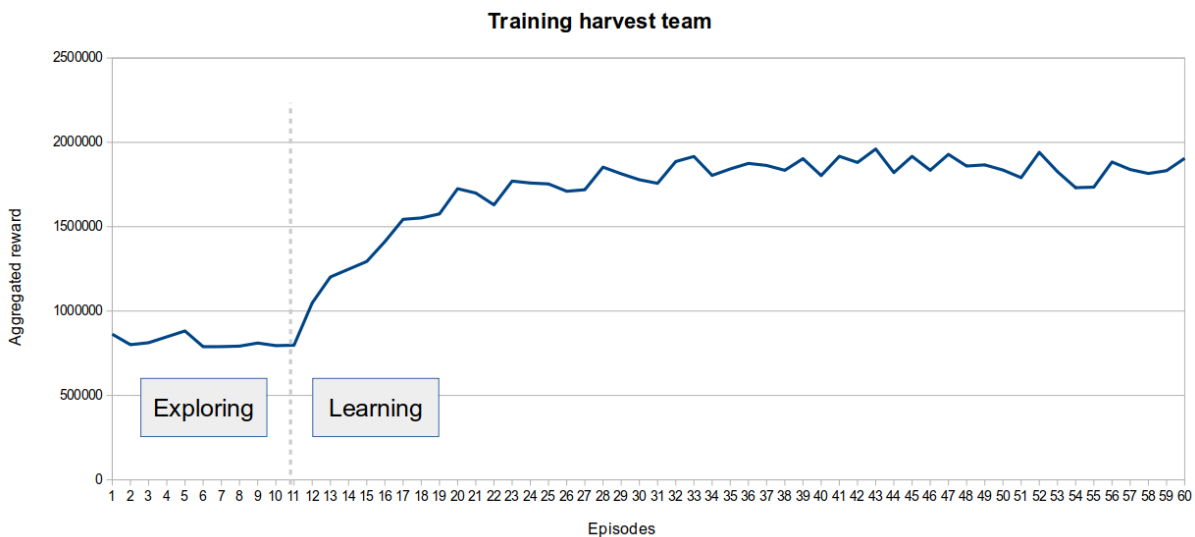


Figure 33. Evolution of reward during the training process

As would be expected, the frequency each different action is taken evolves as agent training progresses. Initially, when the agent training is in the exploring phase, all actions are taken with a similar frequency but once the learning phase kicks in, they start to diverge. For example, the harvest teams will usually receive a higher reward when chopping wood, and moving to a different forest block or building a road will be actions taken only when the right conditions are met, which is at a much lower frequency than chopping.

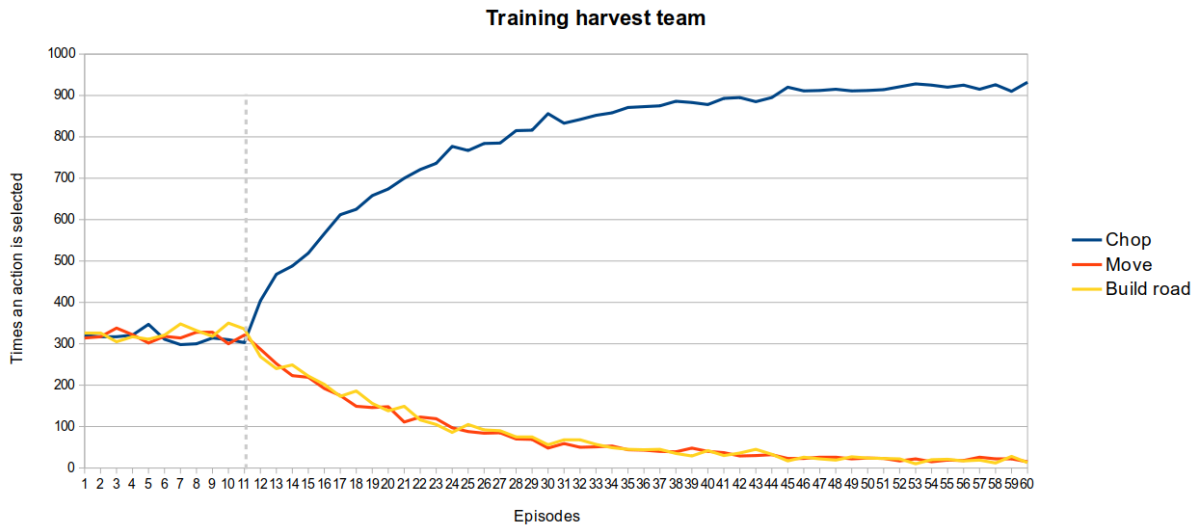


Figure 34. Evolution of the frequency of each action during the training process

10.10.4.2. Execution

When configuring the model, it is optional to specify the number of episodes to run and the length in years of each episode. Once completed, the response sent back will include links to the best episode and an aggregated summary of the best results.

Once the model is launched, episodes are run without any training being performed and recording all of the results. At the end, the episode with the best result serves as a base to generate a detailed list of the actions taken by all the agents, which can be used as a blueprint for the multiyear plan. However, in reality this might not be too useful for a variety of reasons, such as the model not taking into account every possible aspect or reluctance to follow a machine-generated plan. A second set of results comes from the aggregated outcome of the top 25 percentile of the episodes with the highest reward. The most commonly harvested forest stands, most commonly new roads built and busiest mills in these episodes are marked and returned in a GeoJSON that can be visualized using standard tools, in order to serve as a guide to elaborate the multiyear plan.

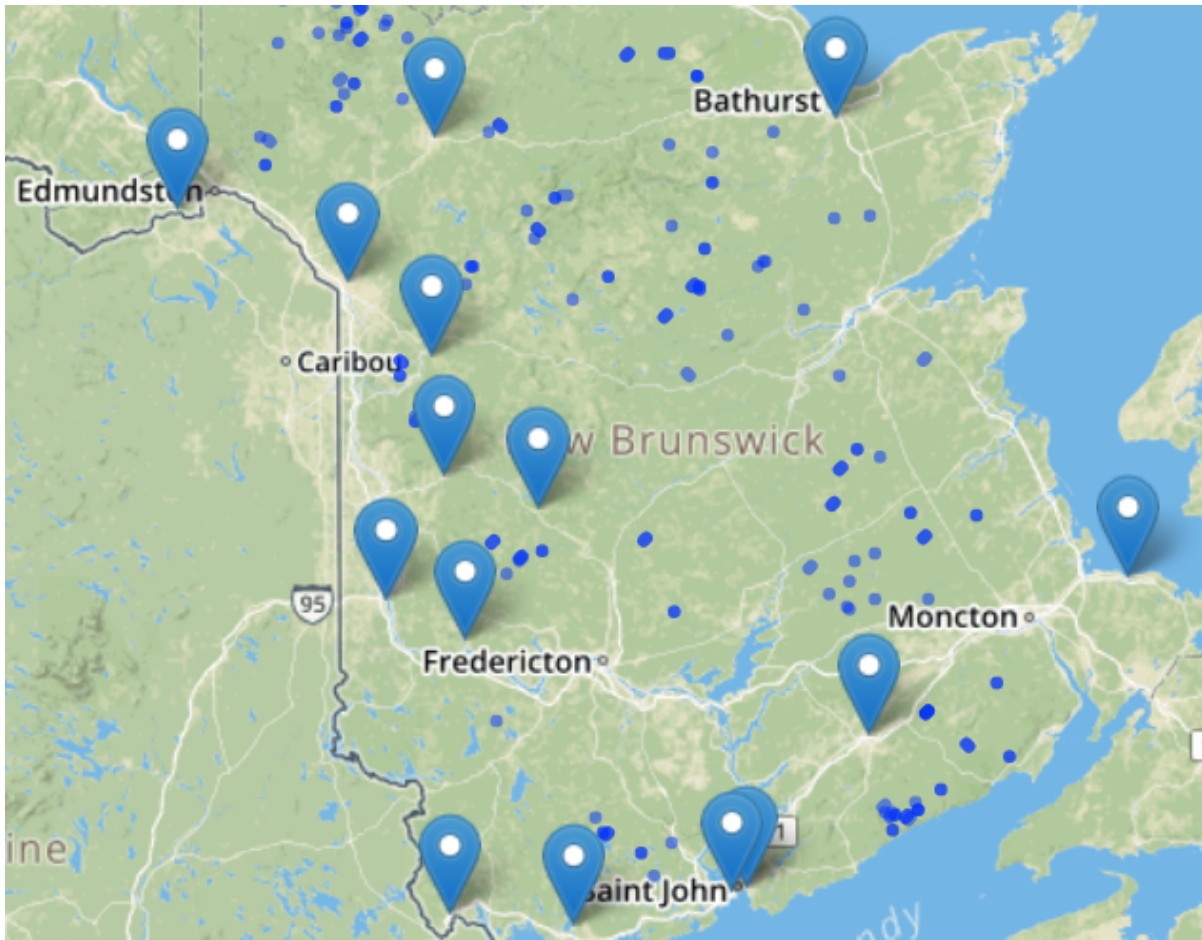


Figure 35. Screenshot of top harvested stands and busiest mills (overall view, using <http://geojsonlint.com/>)

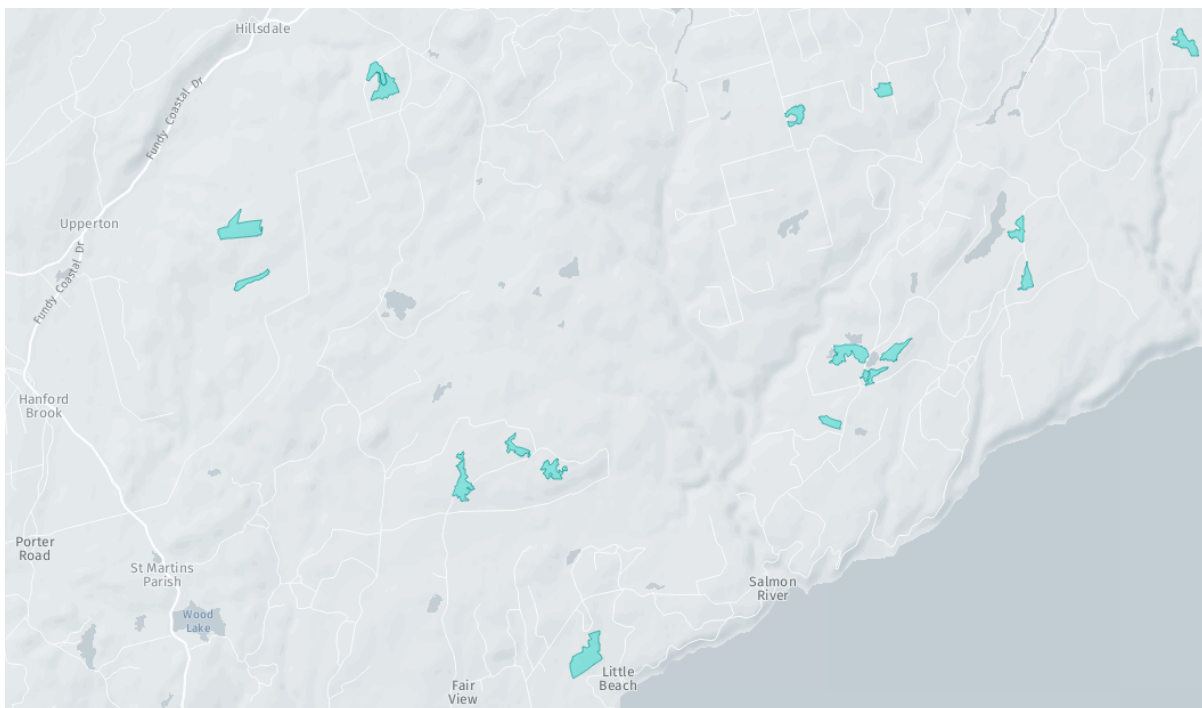


Figure 36. Screenshot of top harvested stands (zoom view, using <http://geojson.tools/>)

10.11. Conclusions

During the implementation of the D102 component, RL techniques were successfully applied to make use of data available through services conforming to OGC standards, and to simulate the operation of

wood harvest and transport at a province-wide scale. The model can be improved with a more advanced description and implementation of existing business processes, more accurate environmental data, as well as more data on how to define the different agents that interact in the simulation. The model can be used as an additional information source to develop multi-year plans, but with a little twist the model can also be used to help make business decisions and to find new ways of optimizing the value chain. The model can also be used to evaluate the strengths and weaknesses of the whole industry by performing stress tests that could help understand how future crises could trigger disruption, identify where weak links are and determine what contingency plans could be developed to soften a crisis. A series of recommendations have been drafted to help improve the use of OGC standards for their use with RL applications. These can be found in the [Discussion](#) section.

Chapter 11. Quebec River-Lake Classification and Vectorization Model

11.1. Component Summary

The trained model in component D104 (Quebec River-Lake Classification) was developed with an application package that includes all required dependencies for data transformation, deep learning model training, and model execution. The model distinguishes lakes from rivers by first detecting lakes and then separating detected lakes from the undifferentiated hydrographic layer. The application package is deployed as a process through an Execution Management System (EMS), and finally executed either locally or on a distant system via an Application Deployment and Execution System (ADES).

11.2. Component Design

The figure below depicts the conceptual model for the deliverable of the D104 component, showing the various processes, either remote or local. The processes are launched through a series of manual operations, function calls, or via OWS. The features datasets are downloaded then mounted on read-only volumes so that the training process can access them. Trained models and metrics are also exchanged through this shared volume, and constitute important inputs and outputs for the component.

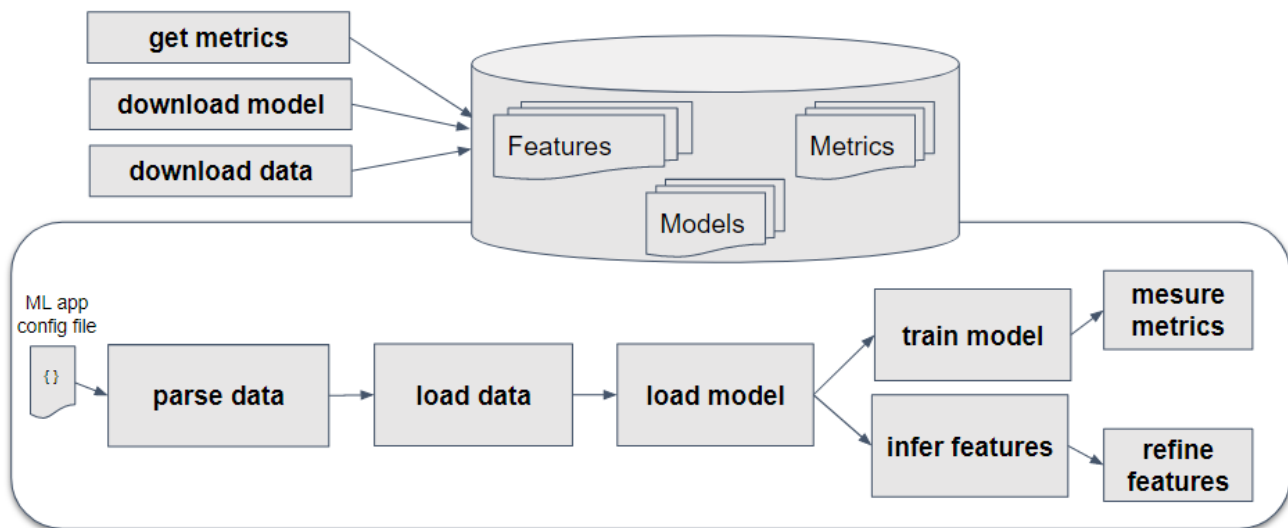


Figure 37. Conceptual model of the various processes involved

As a whole, the processes could be the basis of a basic ML API. As this is a conceptual model, the actual process names employed in the implementation or documentation may differ. Below a short description of the processes:

- *parse data*: opens the input data, selects the target features, balances the training dataset
- *load data*: prepares local chunks, splits the dataset into training/validation sets, transforms into tensor
- *load model*: retrieves and loads neural network architecture, updates model weights

- *train model*: launches training process, produces updated models until criteria are met
- *infer features*: uses a trained model to produce an inference from inputs, yielding new features
- *measure metrics*: updates the metadata of the training process and of trained models
- *refine features*: uses geospatial operators to transform the inferred features
- *get metrics*: access the metadata produced by the training process
- *download model*: makes a trained model accessible to the local filesystem of the application
- *download data*: makes the input data accessible to the local filesystem of the application

For consistency with the [Testbed-14 Machine Learning ER](http://docs.opengeospatial.org/per/18-038r2.html) [http://docs.opengeospatial.org/per/18-038r2.html], the addition of a get annotations process to this list would enable uses cases where an analyst conducts in-the-loop machine learning operations.

CRIM’s framework for training and testing machine learning models as part of inference services and workflows is called **thelper**. The code is hosted on GitHub as an open-source project at [crim-ca/thelper](https://github.com/crim-ca/thelper) [https://github.com/crim-ca/thelper]. This framework primarily allows users to build and evaluate Deep Neural Networks (DNNs) on predefined image analysis tasks (e.g. classification, object detection) using a Command-Line Interface (CLI). The CLI and its underlying API provide an abstraction layer between low-level ML libraries (e.g. scikit-learn, PyTorch) and applications seeking new ML capabilities. Pre-designed model recipes (provided as JSON configuration files) are used to specify the data preparation operations, the neural network architecture settings, and training behavior. This simplifies the model creation process when working with evolving datasets and annotations.

11.3. Implementation Approach

11.3.1. Application

The application’s workflow was fixed and configuration-driven. All dependencies were packaged in a base Docker image which was extended to include a trained model, thus becoming an application. The application then offered a single-entry point, either for training or inference. The figure below depicts the ML application and the embedded workflow.

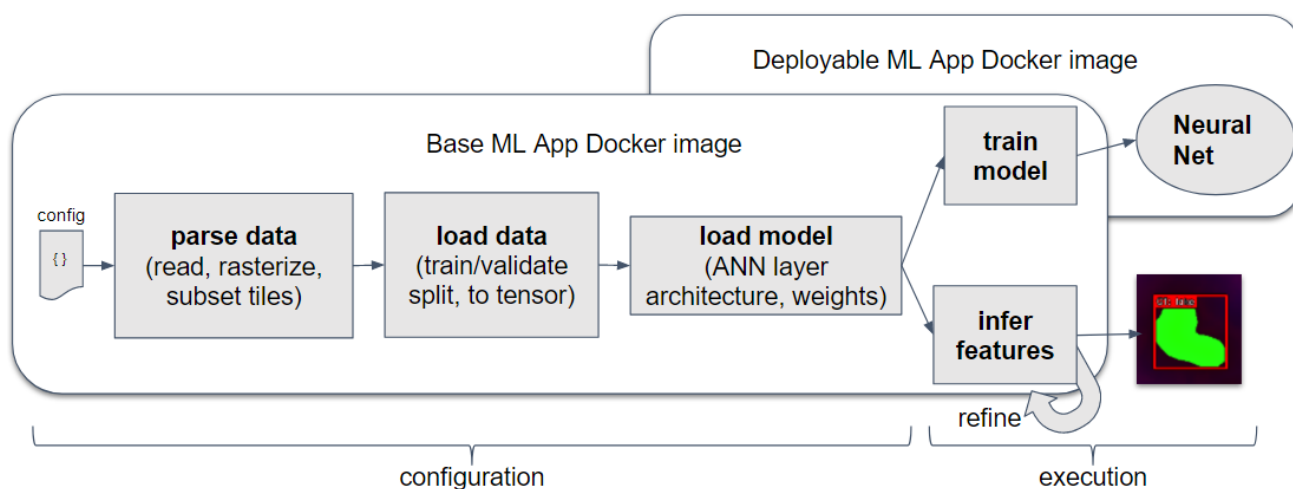


Figure 38. High-level Machine Learning workflow for D104

A configuration file for the application is presented in [Annex A](#), while some excerpts are provided in

subsections below.

11.3.2. Data

The two main sources of input data were a High-Resolution Digital Elevation Model (HRDEM) and a subset of the [Geobase du réseau hydrographique du Québec \(GRHQ\)](https://mern.gouv.qc.ca/repertoire-geographique/reseau-hydrographique-grhq/) [https://mern.gouv.qc.ca/repertoire-geographique/reseau-hydrographique-grhq/] hydrographic network. The addition of satellite imagery was considered, but discarded due to potential temporal inconsistencies. The region of interest was a patch of land of about 200 square kilometers, north-west of the cities of Gatineau and Ottawa. Both data sources were downloaded from NRCan's NFIS Geoserver at <https://opendata.nfis.org/>, through WCS and WFS 3.0 respectively. The figure below presents the region.

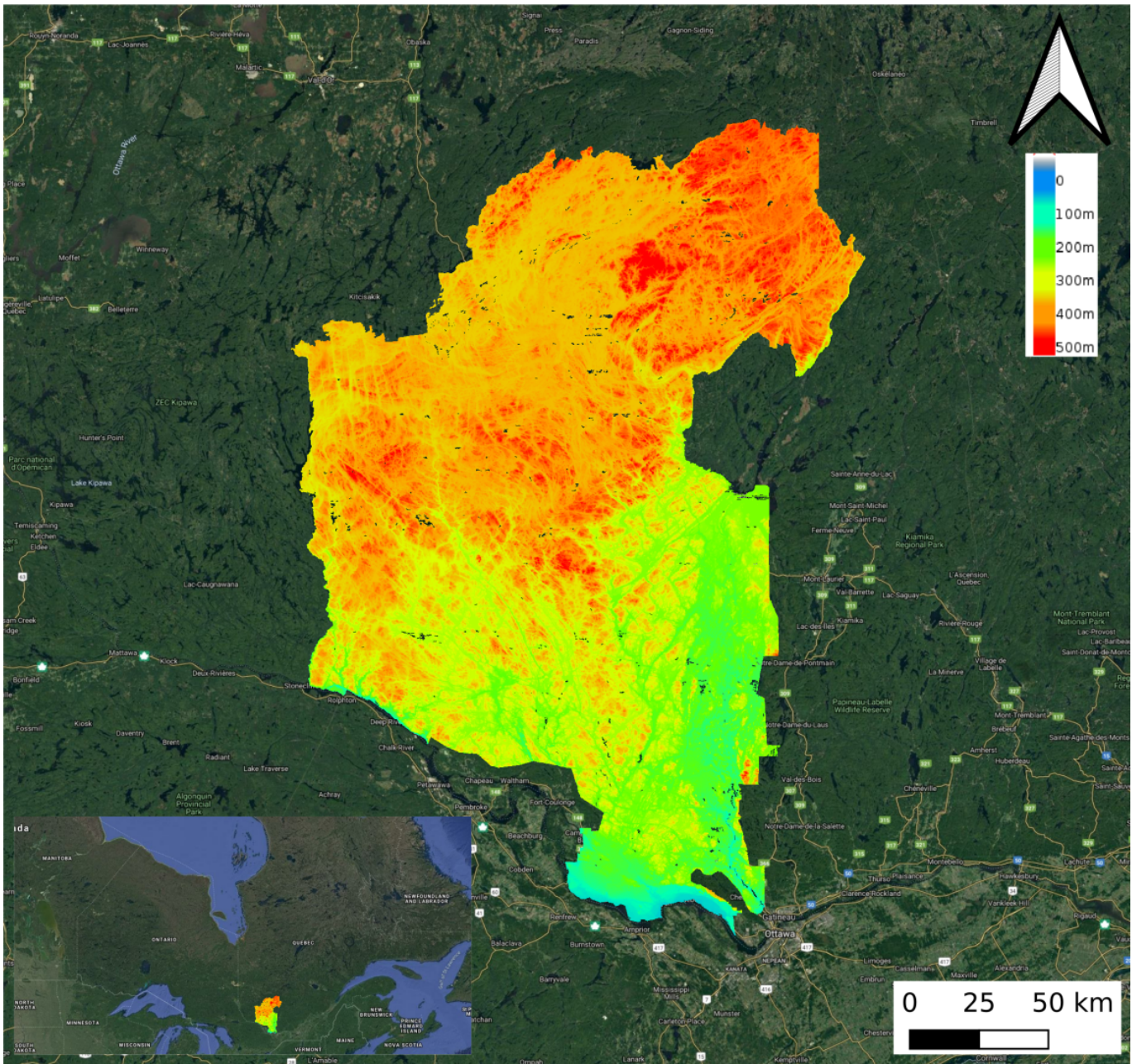


Figure 39. Region of interest used in the experiments. The colored region is the HRDEM (left).

For the considered region of interest, the HRDEM has a file size of 252 GB, for a coverage of 210 000 by 300 000 pixels and a spatial resolution of 1 meter. The maximum elevation value found in this region is 609.1 meters. The hydrographic network dataset contains 36,109 differentiated features, composed of

34 678 lakes and 1431 rivers. The projection used in both was NAD 83 / UTM zone 18N. The main attributes of the hydrographic network features are presented in the following table. The type of feature is represented by TYPECE, where a value of 10 corresponds to rivers and values of 21 and over represent lakes.

Attribute	Description
<i>fid</i>	Record number in the dataset
<i>ID_RHS</i>	Unique identifier of the object
<i>TYPECE</i>	Object type

11.3.2.1. Data preparation

As stated previously, the data was assumed to be readily available for the application. In an operational scenario, the ‘download data’ process can constitute an initial step in a workflow. In the general case, the process uses a URL to fetch the data locally. Ideally, the application is to be deployed at run-time on a distant infrastructure, near the data. In that scenario, the download data process would have the responsibility of providing the ML application with a local path to the files by mounting a drive to the data archive, instead of downloading the data from an OWS endpoint.

Once the data is locally available, the ML application parses the dataset. The following code excerpt presents the data parser. A custom Python class named TB15D0104Dataset takes as parameters the path of the vector and raster data, then reads the data from disk and applies the heuristics. The data parser also receives *area_min*, *area_max* and *max_dist* as parameters. These last three heuristics allow filtering of the input hydrographic features based on thresholds. This step allows the explicit selection of features of interest to the task, namely lakes that are adjacent to rivers. This step also balances the size distribution of the training set. Note that with relatively minor changes to the implementation, the ‘data parser’ process could have been executed as a WPS instead of a local procedure call.

Excerpt of the data parser configuration for the ML application

```
datasets:  
  testbed15:  
    type: "thelper.data.geo.ogc.TB15D104Dataset"  
    params:  
      raster_path: "data/testbed15/roi_hrdem.tif"  
      vector_path: "data/testbed15/hydro_original.geojson"  
      lake_area_min: 100  
      lake_area_max: 200000  
      lake_river_max_dist: 300
```

Once the data was parsed, all the vector information was rasterized. The resulting stack was transformed into a PyTorch tensor, as depicted in the following code excerpt. The tensor was then ready to be used by the train model process, which were local in the application space. There is a possibility to serialize the tensor obtained for use in a further re-training of a model.

Transformation of the input data into a tensor

```
loaders:  
  base_transforms:  
    operation: torchvision.transforms.ToTensor  
    target_key: input
```

The following figure illustrates the content of the tensor. The content is a coverage stack composed of the HRDEM and the rasterized hydro network. A third layer encodes the distance between any pixel and its nearest lake pixel. This handcrafted feature is used as an attention layer to the model trainer. The three images are then transformed to a tensor. Visualization of the tensor is done by mapping values to RGB channels.

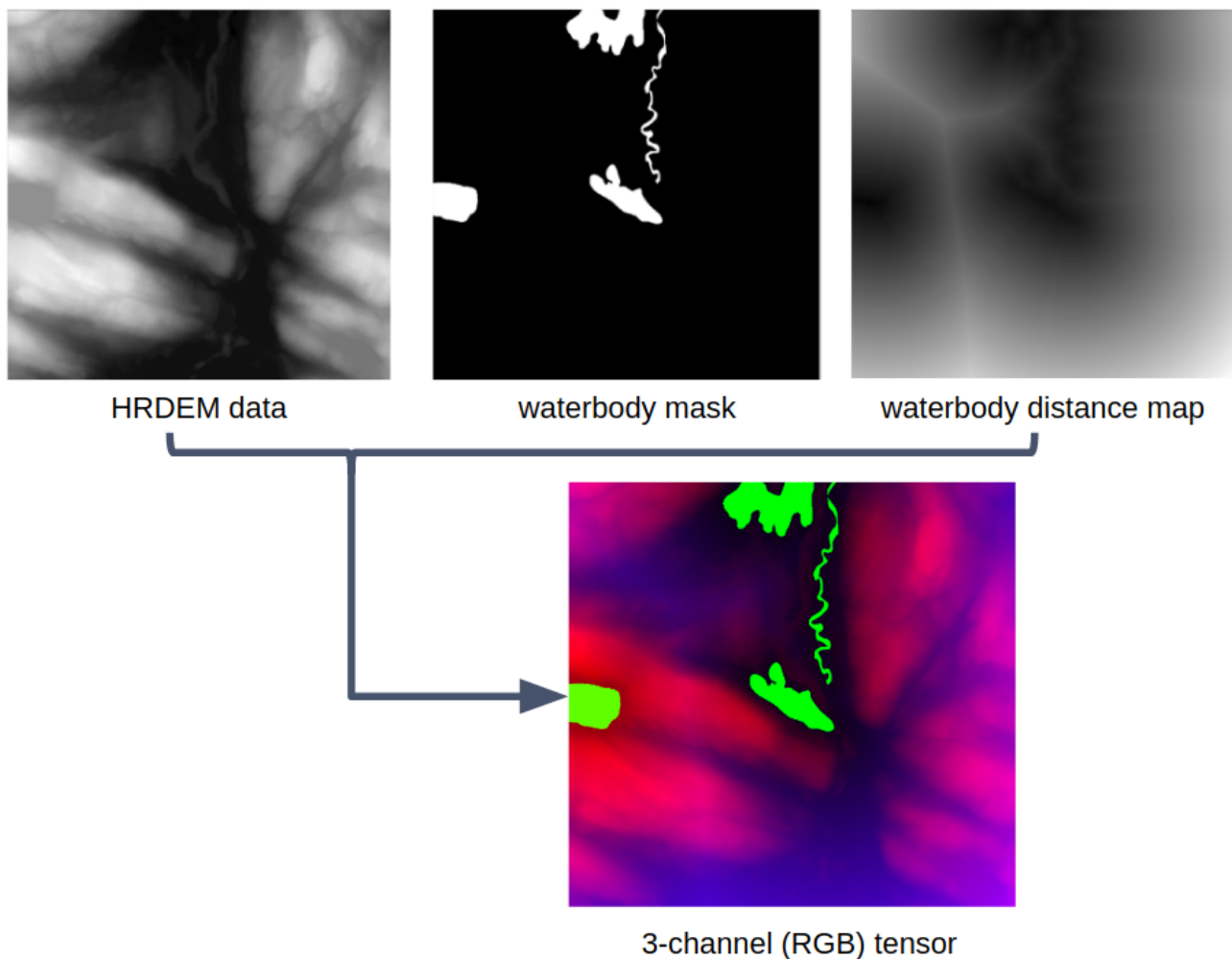


Figure 40. Formation of the input tensor used in training of a deep learning model

The evaluation of the lake detection model was accomplished by separating the available HRDEM and the differentiated hydro features into two geographical subsets: one for training and one for validation. These subsets are shown in the figure below: the purple region is used for training, and the yellow region for validation. After feature preprocessing and filtering, the training subset consists of roughly 1500 crops of one square kilometer each that contain at least one lake of interest. The validation subset counts roughly 600 crops of similar size.

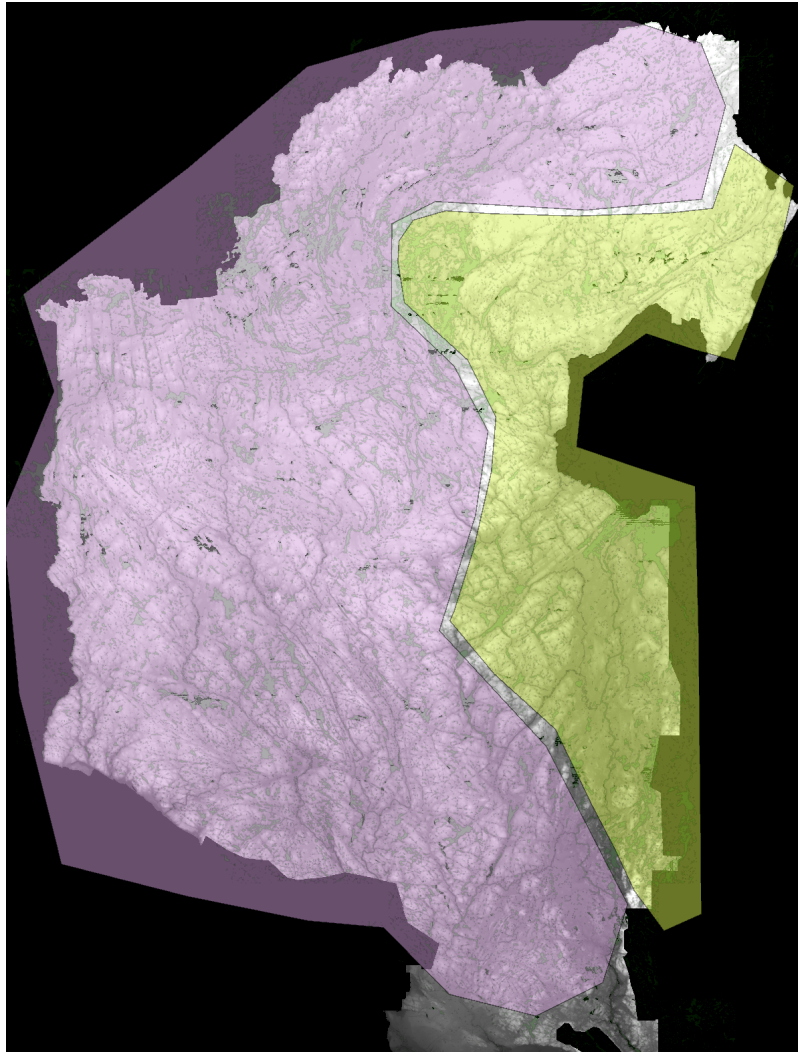


Figure 41. Validation and training datasets

11.3.3. Machine Learning Model

The goal of the model is, given a set of hydrological features as well as HRDEM raster data, to differentiate lakes from rivers. This goal can be reformulated as the detection of lakes since these are more easily defined as entities and easier to observe as a whole. As such, using a ML model trained to recognize and localize lakes from a set of mixed hydrological features, the goal of differentiating them from rivers was also accomplished. Therefore, the object detection in this model targets only lakes.

11.3.4. Convolutional Neural Network architecture

Object detection models in the context of machine learning now come in many different forms. To simplify development, the application uses a deep convolutional network combined with a region proposal network [1]. This combination of models coined Faster R-CNN can be trained for the regression of bounding boxes around objects of interest. While such axis-aligned bounding boxes are not ideal for the fine-grained fragmentation of hydrological features, it was determined that post-processing can be used to optimize or refine the boundaries between rivers and lakes. The use of an instance-based detection and segmentation approach as in [2] would be more appropriate, as the segmentation masks would remove the need for the post-processing of region boundaries.

The *train model* process then uses the tensor and starts the iterations. A log output of model training can be found in [Annex D](#). Below is the description of a default, pre-trained model as a starting point for

the process. It can be noted that while custom architectures can be specified, their metadata, implementation and dependencies need to be provided separately. The implementation of D104 used mostly default architectures supported by PyTorch.

Specification of a deep learning architecture

```
model:  
  type: torchvision.models.detection.fasterrcnn_resnet50_fpn  
  params:  
    pretrained: true
```

11.3.4.1. Model training and performance

The lake detection performance was evaluated based on bounding box localization accuracy based on mean Average Precision (mAP). The model reached its peak performance on the validation subset after only two training epochs (mAP=0.949) before starting to overfit. Average Precision is often used to evaluate object detection models that produce bounding boxes. It was first defined by in the context of the [PASCAL VOC challenge](http://host.robots.ox.ac.uk/pascal/VOC/) [http://host.robots.ox.ac.uk/pascal/VOC/]. This metric reflects the precision and sensitivity of a detection model by computing the area under the Precision-Recall curve formed while evaluating bounding box predictions on an annotated dataset. Therefore, the metric can be used to assess the overall performance of a model on a specific detection task. Along with this metric, assessing the performance of the model can track the value of the model's loss function during training as well as evaluation. This makes it possible to further understand the performance of our module.

Below is an excerpt from the ML app configuration file that sets the metric measurement to Average Precision.

Specification of a deep learning architecture

```
trainer:  
  metrics:  
    mAP:  
      type: thelper.optim.metrics.AveragePrecision
```

11.3.4.2. Inference outputs results

The following figure presents results of the *infer features* process, part of the ML workflow local to the application.

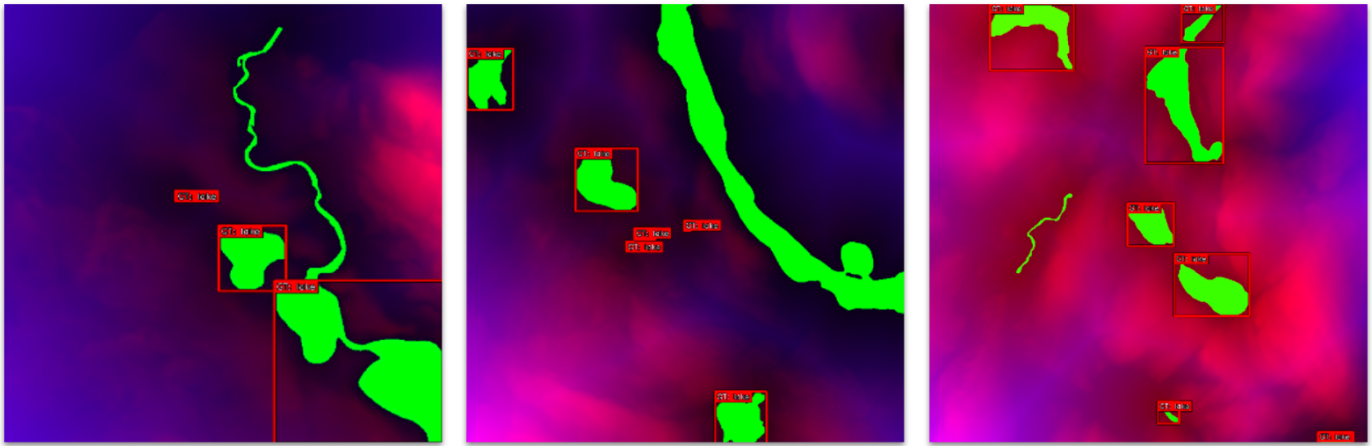


Figure 42. Results of the trained lake detection model

To generate a new set of features that differentiate lakes and rivers, the predicted lake bounding boxes are assembled into a geometry collection that is used to slice the original hydro features. A visualization of the collection of bounding boxes that correspond to lake predictions is shown below. In the following figure, large lakes are covered by several individual detections.

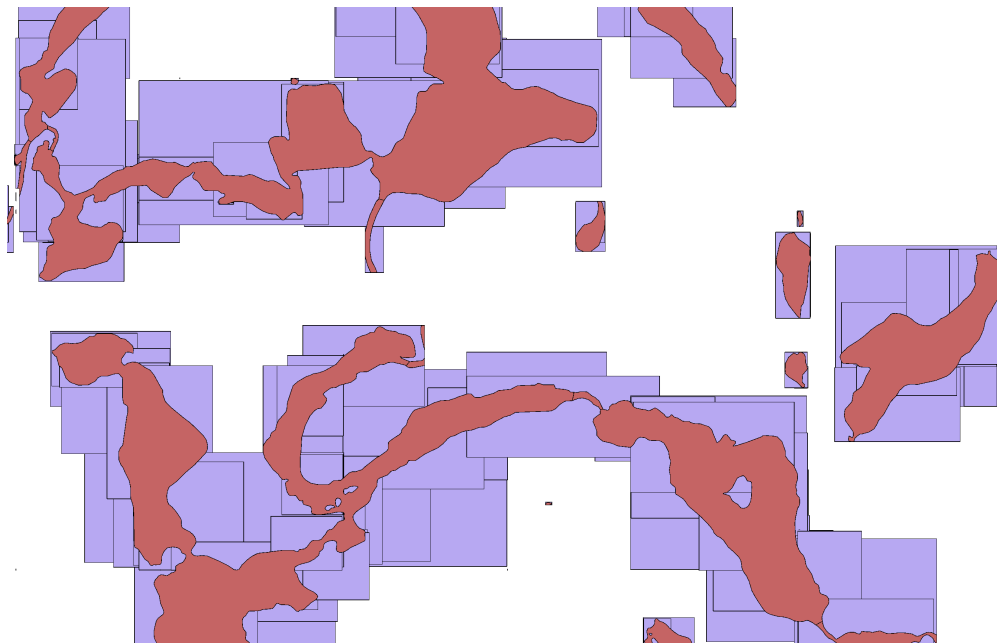


Figure 43. Undifferentiated hydro network over detected bounding boxes of lake regions

The following figure presents the lake and river separations. Artefacts are still clearly visible, as use of post-processing morphological and geometric operators was kept to a minimum. As described previously, an instance-based detection and segmentation approach as in [2] would greatly reduce the need for such post-processing features operations. Another advantage of a segmentation-first model can be easily computed pixel-wise over the undifferentiated network as an operator on two coverages.

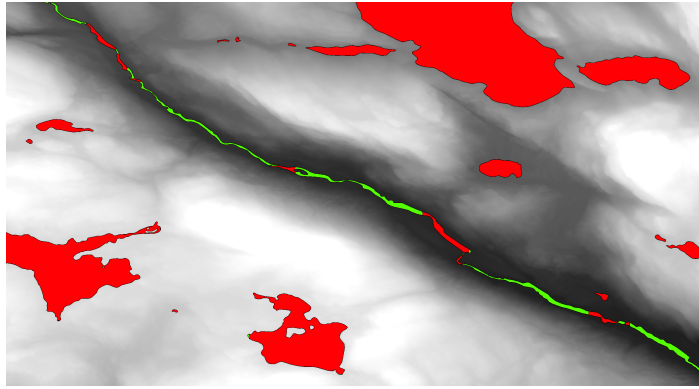


Figure 44. Detected lakes in red, rivers in green. Grey shaded area is the original HRDEM.

11.3.5. Other experimental findings

11.3.5.1. Model exportation and transfer learning

Models trained with this ML application can be exported in different formats (including the [Open Neural Network Exchange Format \(ONNX\)](https://onnx.ai/) [https://onnx.ai/], an open ecosystem for interchangeable AI models) for further refinements or to be reused in other systems. Once a model is trained, it should ideally be exported in a format that would allow inference (evaluation) on generic platforms and retraining. Retraining can either refer to "fine-tuning" for specific sub-tasks or more generally to "transfer learning" for new tasks. Recent initiatives such as ONNX have introduced exchange formats to simplify the inference process for neural networks using various back-ends. However, numerous deep learning frameworks (including PyTorch and TensorFlow) only allow models to be exported in the ONNX format, but are not easily imported from ONNX for retraining.

11.3.5.2. Use of WPS for interactive sessions

Training a model often requires monitoring its performance in real-time to evaluate when to stop, without actually stopping the process execution to avoid reloading data. Obtaining results from an ongoing WPS process execution is not well supported. There is often a need to go through a history of model checkpoints and logs to analyze interesting cases. This approach is neither standardized nor easily retrievable from WPS servers. While the application implements and supports 'train model' functions, experiments assumed an offline training phase resulting in a trained model. The infer features process that follows is very well adapted to be offered through a WPS.

To circumvent these apparent limitations of WPS in interactive settings, specific triggers can be set in advance, such as the divergence or convergence of metrics. Through a Publication/Subscription (Pub/Sub) mechanism, a user can be alerted of the completion of the job, or that specific models or data had been changed. The user could also decide to cancel or stop the train model process due to notifications of divergence or potential overfitting. Another scenario is to allow a 'parse logs' process to get metrics in real-time. This approach would be akin to dashboards such as TensorBoard, where widgets could stop, cancel or restart the train model WPS.

11.3.5.3. Change tracking for models, data and annotations

Even if the source code used to implement predictive models is kept static, the behavior of the models can change due to the varying availability and constant evolution of their training data. This affects the reliability of models and reproduction of experiments, which is a cornerstone of scientific research.

Keeping track of changes in data is not an easy task, as Version Control Systems (VCS) are not typically made to track large binary files. There exist solutions for small projects (e.g. [Data Version Control \(DVC\)](https://dvc.org/) [https://dvc.org/]), but these are usually limited to locally hosted datasets that are not updated often.

In a large-scale distributed system, keeping track of data changes happening on remote servers is difficult. Most changes can only be observed in downstream processes when new requests for the data are made. The responses to most standard requests formats also tend to not contain metadata indicating data source or the latest modification time. Some efforts have been made to develop data tracking solutions specifically for the geospatial domain, but have been limited to vector data (see e.g. [GeoGig](http://geogig.org/) [http://geogig.org/]). Keeping proper logs of all experiments including resource metadata and access times has proven fairly useful for manual tracking of data updates. Furthermore, the "forensic" analysis of these logs can lead to lowered model performance. Adding rigorous metadata fields related to data sources and modification times to standardized web service requests would greatly improve the robustness of ML training and evaluation services.

11.4. Component Implementation

The following figure presents the various modes of operation of the D104 application. As previously described, the helper library allows interactive command-line invocations of ML pipelines. The Application Package is completed with creation of a CWL file describing the execution unit. This CWL file can in turn be used in more elaborate workflows, subsequently packaged as applications. Once the application is deployed, the EMS automatically exposes the process description as WPS 2.0 REST/JSON.

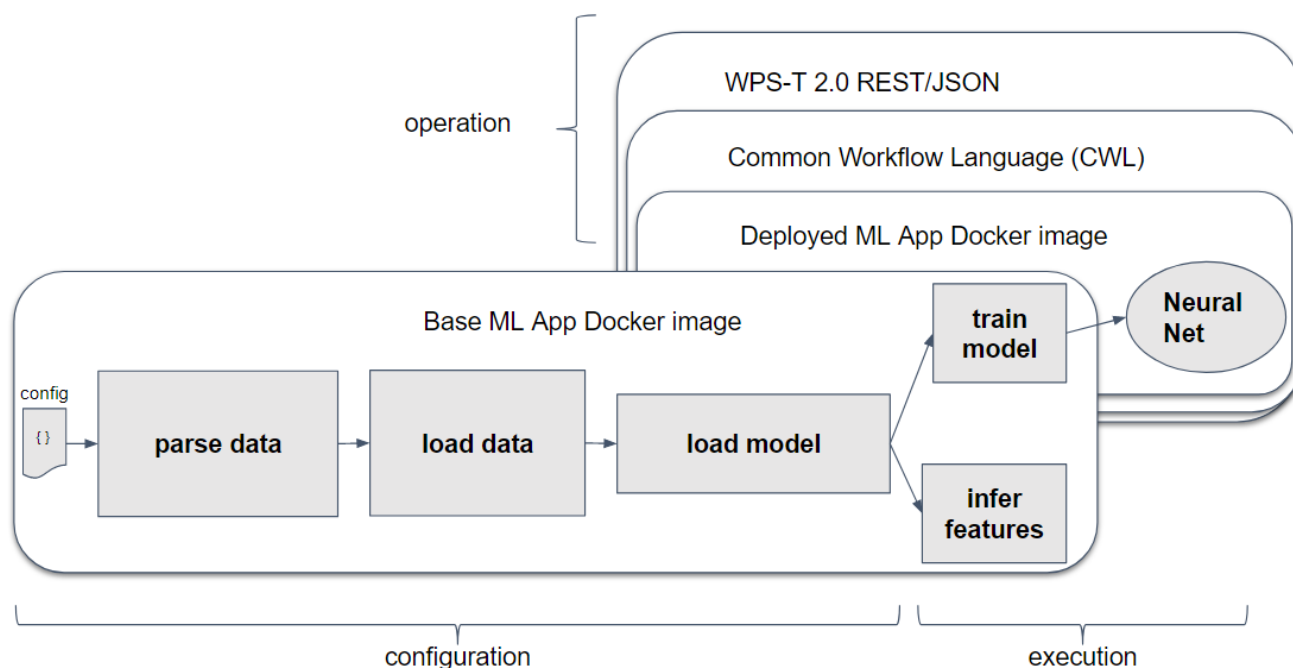


Figure 45. Command line, CWL and WPS-T 2.0 REST mode of operation

Below are the endpoints to test the two main operations of the ADES/EMS.

- *Deployment:* POST <https://ogc-ems.crim.ca/weaver/processes>
- *Execution:* POST <https://ogc-ems.crim.ca/weaver/processes/toy-example/jobs>

Below is the detailed architecture of the ADES/EMS component called Weaver. Aside from deployment

and execution of Application Packages, and as documented in the [OGC Testbed-14 Earth System Grid Federation Compute Challenge ER](http://docs.openeospatial.org/per/19-003.html) [http://docs.openeospatial.org/per/19-003.html], the component also allows for the deployment of existing PS 1.0 endpoints. Registration of such a pre-deployed WPS instance allows for CWL and WPS 2.0 REST/JSON support. This approach was proposed in a series of Technology Integration Experiments (TIE) that other deployed machine learning WPS instances for the ML thread used in that fashion.

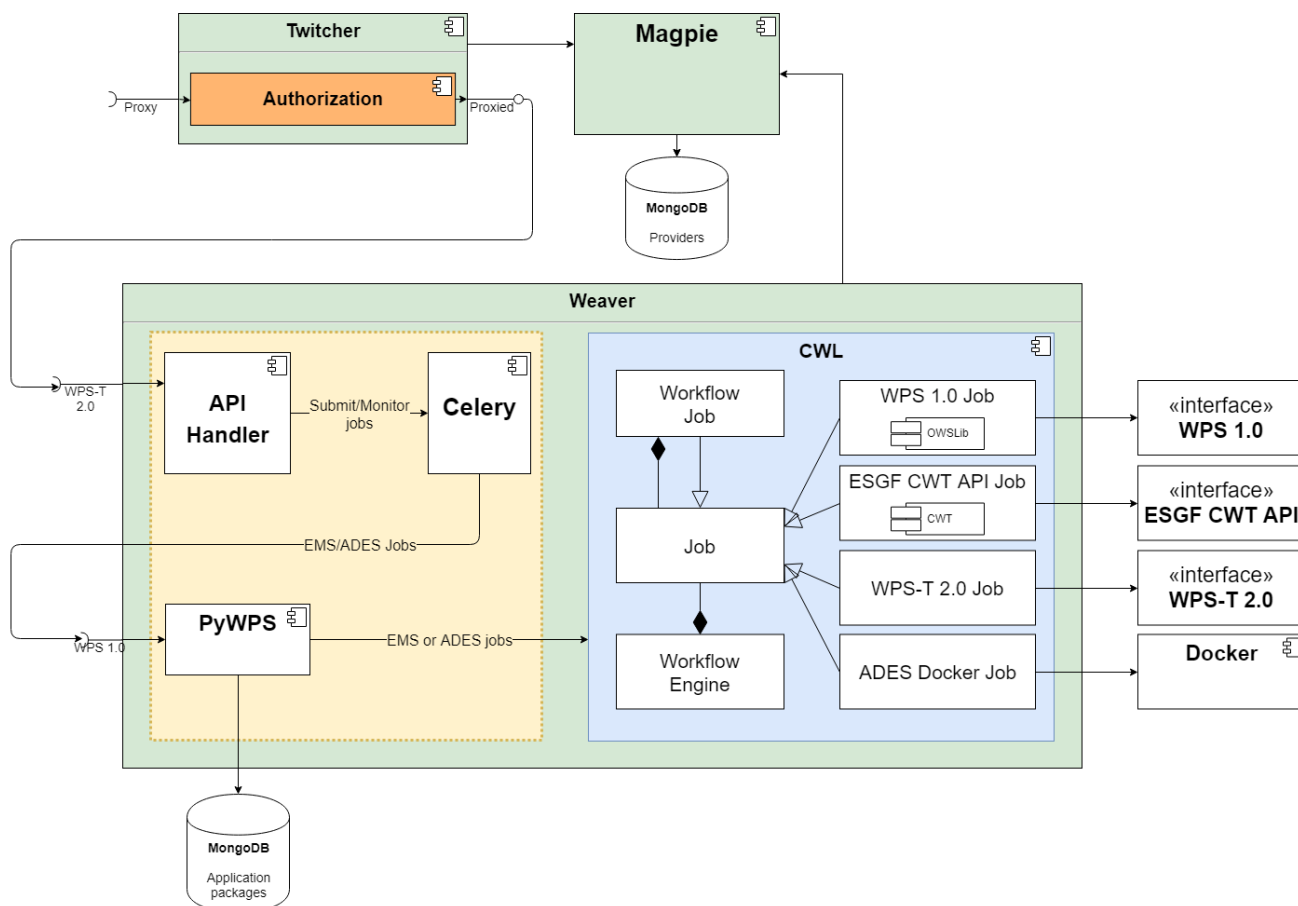


Figure 46. Overall component design of the EMS/ADES solution

11.4.1. Process Description

As documented in *ADES and EMS best practices*, a CWL descriptor is first produced for the application. Below is an excerpt of [Appendix A](#), presenting the process description that encapsulates the ML app. The automatically generated CWL file, to be passed in the `content:unit` field of the `owsContext`, is presented in [Appendix B](#).

.Excerpt of process Description for the ML application

```
"executionUnit":
{
  "unit": {
    "cwlVersion": "v1.0",
    "class": "CommandLineTool",
    "requirements": {
      "DockerRequirement": {
        "dockerPull": "docker-registry.crim.ca/ogc-public/ogc-thelper-
tb15:0.1.2"
      }
    },
    "baseCommand": "ogc_thelper_tb15",
    "arguments": ["-o", "${runtime.outdir}"],
    "inputs": {
      "raster_file": { <...> }
    },
    "vector_file": { <...> }
  },
  "outputs": {
    "output": {
      "outputBinding": {
        "glob": "${runtime.outdir}/output.json"
      },
      "type": "File"
    }
  }
}
}
```

Chapter 12. Arctic Discovery Catalog

12.1. Overview



Figure 47. The Arctic

The Arctic Discovery Catalog component (D107) built an evergreen catalog of relevant Arctic circumpolar web services from OGC and ESRI REST services that have some relevance to circumpolar science. The component evaluated the confidence level of the service conformance to selection criteria, and made the results available via an OGC CSW implementation. The component made use of open source libraries (TensorFlow, PyTorch) and utilized various training datasets such as the Arctic-SDI catalog, Arctic keywords, and an Arctic boundary file to train the model.

The Arctic Discovery Catalog was a standalone ML component in Testbed-15 that was not dependent on any other Testbed-15 ML deliverables.

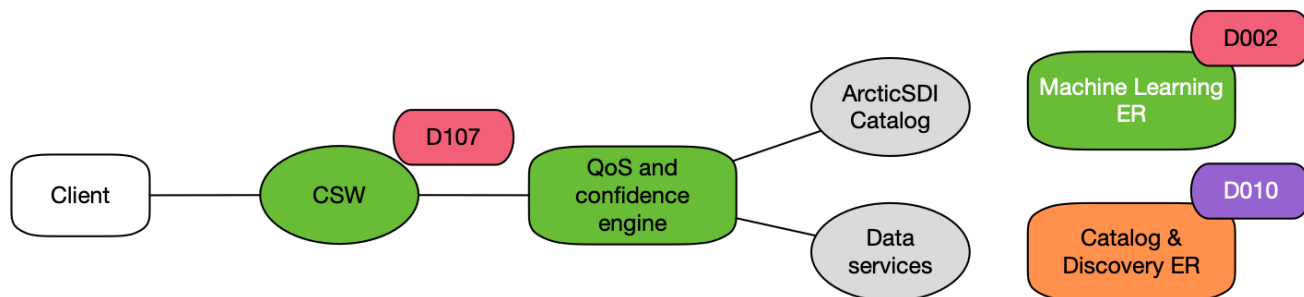
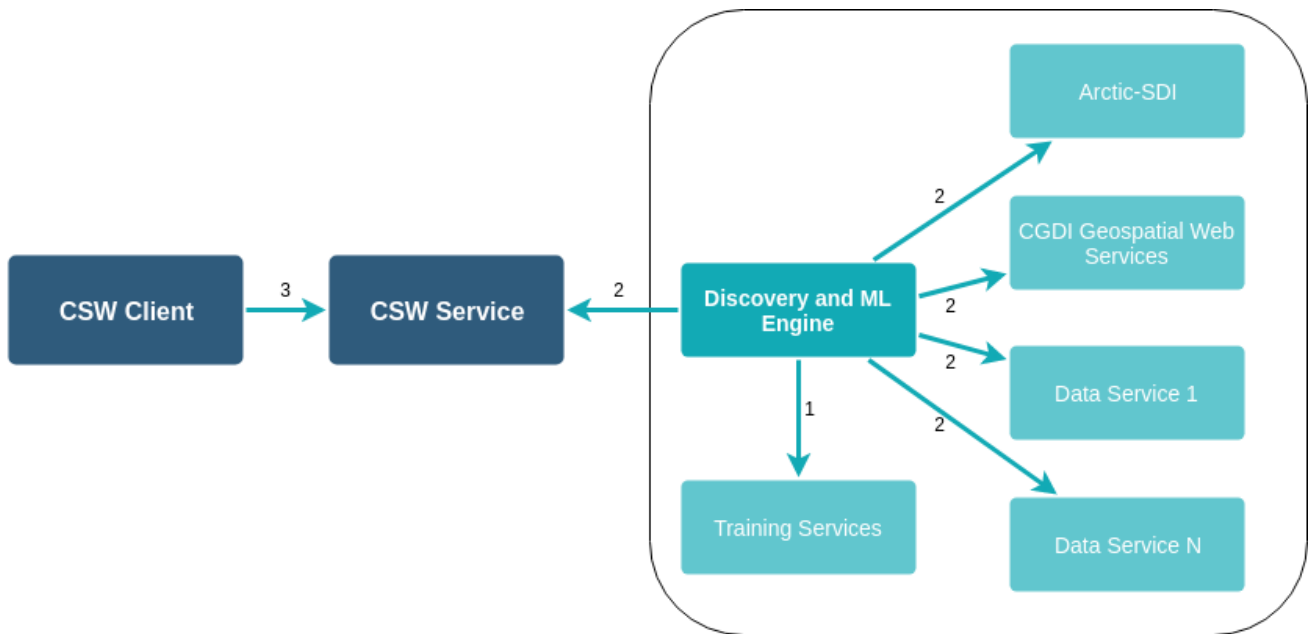


Figure 48. D107 Overview

12.2. Architecture

The architecture of the Arctic Discovery Catalog was as follows:



1. The Discovery and ML Engine trains the ML model using keywords, geographic areas, and positive/negative training datasets
2. The Discovery and ML Engine runs in the background, harvests and processes each configured endpoint and populates the WES CSW ebRIM data model with items that match the model
3. Users use the CSW Client to query the Arctic relevant content of the CSW

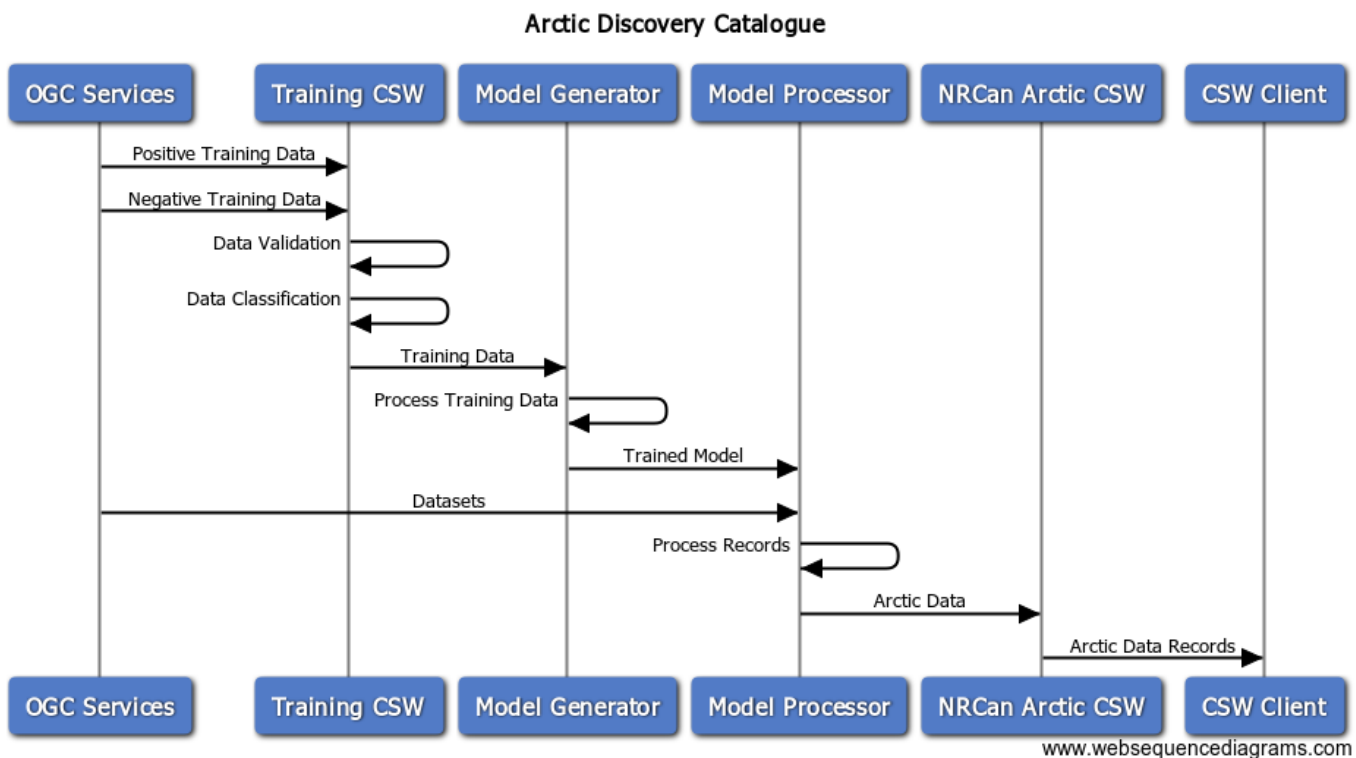


Figure 49. Sequence Diagram

1. Both positive and negative training data is harvested from various OGC web services into the Training CSW
2. The training data is validated
3. The training data is classified

4. The Model Generator queries the training data from the Training CSW
5. The Model Generator processes the training data and builds the ML model
6. The Model Processor retrieves records from various online OGC services and applies the trained model to filter the results
7. The filtered results are populated into an NRCan Arctic CSW
8. Clients can access the NRCan Arctic CSW to retrieve the Arctic-only records

12.3. Machine Learning Model Training

Training the ML model is one of the most important early stage steps of the entire process. The model was trained using a variety of datasets as outlined below. These datasets included Arctic related keywords, Arctic location names, Arctic regions, Arctic projections and existing Arctic and non-Arctic catalogs.

Table 12. Training Datasets

Dataset	Description
Keywords - General	ArcticSDI, Arctic, Albedo, SeaIceExtent
Keywords - Geographical Names	svalbard, greenland, baffin, victoria island, ellesmere island, resolute, beaufort, north sea, barents sea, laptev, kara sea, east siberian sea, chukchi, severny, arkhangelsk, archangel, bulunsky
Projections	Polar Projections North (EPSG:3571-3576)
Catalogs	<p>Positive:</p> <p>Arctic SDI Portal [https://geoportal.arctic-sdi.org], CGDI Geospatial Web Services [https://www.nrcan.gc.ca/earth-sciences/geomatics/canadas-spatial-data-infrastructure/19359], Atlas of the Cryosphere: Northern Hemisphere [https://nsidc.org/cgi-bin/atlas_north?service=WMS&request=GetCapabilities&version=1.1.1], Harvested Conservation of Arctic Flora and Fauna (CAFF) abds [http://geo.abds.is/geoserver/base/wms?Request=GetCapabilities&version=1.3.0], Harvested Arctic SDI_US ArcGIS service [https://services.nationalmap.gov/arcgis/rest/services/ArcticSDI_US/MapServer], Harvested Canadian Cryospheric Information Network [https://ccin.ca], Polar Data Catalog [https://polardata.ca]</p> <p>Negative:</p> <p>Geoscience Australia Product Catalog [https://ecat.ga.gov.au/geonetwork/srv/eng/csw?request=GetCapabilities&service=CSW], Afghanistan Disaster Risk WebGIS [http://disasterrisk.af/catalog/csw?service=CSW&version=2.0.2&request=GetCapabilities], Tuna atlas [http://tunaatlas.d4science.org/geonetwork/srv/eng/csw?service=CSW&request=GetCapabilities]</p>

12.3.1. Data Preparation

CSW records were read from the Training CSW and 5 data fields were used:

1. Title
2. Abstract
3. Keywords / Subject
4. Projection
5. Bounding Box

The first three fields above are text-based fields which needed to be converted to numeric representations for use in the model. The Natural Language Toolkit (NLTK) library was used to normalize and extract stop word and training word tokens in the Title, and Abstract fields. Once the word tokens were extracted, the Testbed participants investigated using two algorithms for computing numeric keyword scores:

1. Word Tokens → TF/IDF → Numeric Score
2. Word Tokens → RAKE → Numeric Score

TIP *TF/IDF = Term Frequency-Inverse Document Frequency*
 A numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. [\[dm\]](#)

TIP *RAKE = Rapid Automatic Keyword Extraction*
 An algorithm for extracting and ranking the keywords/phrases out of a document without any other context except for the document itself. Used to create a keyword vocabulary and search inputs for keywords

The records in the Training CSW were then used to train the various ML models.

12.4. ML Models

12.4.1. Multilayer Perceptron (MLP) Neural Network Implementation

Compusult has implemented a Multilayer Perceptron (MLP) neural network using Python and Tensorflow. The MLP contains three layers as:

1. Total Length of Input Layer
2. 1024
3. Output Layer

12.4.2. Training Results

The results that were generated during the MLP model training are outlined in [Table 13](#) below. [Table 12](#) describes the columns that make up that table.

Table 13. Training/ Testing Results Table Description

<i>Iteration</i>	Iteration #
------------------	-------------

<i>Time/Sample</i>	the time used to process each training sample
<i>Training Loss</i>	$-(y \log(p) + (1-y) \log(1-p))$ where y - binary indicator (0 or 1) if class label c is the correct classification for observation o $y = 0$ or 1 if class label is the correct classification for an observation. p - predicted probability an observation is a class
<i>Training Accuracy</i>	the correct prediction/total # of records
<i>Test Loss</i>	Loss value for test data set
<i>Test Accuracy</i>	Accuracy value for test data set

Table 14. Sample of Training / Testing Results

Iteration		Time / Sample	Training Loss	Training Accuracy %	Test Loss	Test Accuracy %
<i>Epoch 1/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 24.6065	acc: 0.5156		
4402/4402	[===== ===== ===== =====]	0s 79us/sample	loss: 25.5396	acc: 0.5091	val_loss : 36.2915	val_acc: 0.1579
<i>Epoch 2/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 11.3121	acc: 0.4944		
4402/4402	[===== ===== ===== =====]	0s 39us/sample	loss: 10.3177	acc: 0.4982	val_loss : 13.5758	val_acc: 0.1579
<i>Epoch 3/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 4.4615	acc: 0.5031		
4402/4402	[===== ===== ===== =====]	0s 37us/sample	loss: 4.6499	acc: 0.4968	val_loss : 0.3788	val_acc: 0.8463
<i>Epoch 4/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 1.6558	acc: 0.5759		

Iteration		Time / Sample	Training Loss	Training Accuracy %	Test Loss	Test Accuracy %
4402/4402	[=====	0s 37us/sample	loss: 1.5971	acc: 0.5566	val_loss : 2.8048	val_acc: 0.1579
<i>Epoch 5/50</i>						
3584/4402	[=====	ETA: 0s	loss: 0.9894	acc: 0.4894		
4402/4402	[=====	0s 37us/sample	loss: 0.9313	acc: 0.4952	val_loss : 0.3311	val_acc: 0.8463
<i>Epoch 6/50</i>						
3584/4402	[=====	ETA: 0s	loss: 0.5921	acc: 0.6992		
4402/4402	[=====	0s 37us/sample	loss: 0.5758	acc: 0.7008	val_loss : 0.3570	val_acc: 0.8937
<i>Epoch 7/50</i>						
4096/4402	[=====	ETA: 0s	loss: 0.4718	acc: 0.8193		
4402/4402	[=====	0s 37us/sample	loss: 0.4685	acc: 0.8310	val_loss : 0.3903	val_acc: 0.9663
<i>Epoch 8/50</i>						
3584/4402	[=====	ETA: 0s	loss: 0.4191	acc: 0.9807		
4402/4402	[=====	0s 37us/sample	loss: 0.4134	acc: 0.9832	val_loss : 0.3574	val_acc: 0.9663
<i>Epoch 9/50</i>						
4096/4402	[=====	ETA: 0s	loss: 0.3932	acc: 0.9683		
4402/4402	[=====	0s 37us/sample	loss: 0.3909	acc: 0.9702	val_loss : 0.3024	val_acc: 0.9695
<i>Epoch 10/50</i>						

Iteration		Time / Sample	Training Loss	Training Accuracy %	Test Loss	Test Accuracy %
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 0.3789	acc: 0.9467		
4402/4402	[===== ===== ===== =====]	0s 40us/sample	loss: 0.3704	acc: 0.9541	val_loss : 0.4982	val_acc: 0.9516
<i>Epoch 11/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 0.3242	acc: 0.9891		
4402/4402	[===== ===== ===== =====]	0s 39us/sample	loss: 0.3241	acc: 0.9891	val_loss : 0.2345	val_acc: 0.9747
<i>Epoch 12/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 0.3086	acc: 0.9858		
4402/4402	[===== ===== ===== =====]	0s 37us/sample	loss: 0.3033	acc: 0.9873	val_loss : 0.3659	val_acc: 0.9516
...						
<i>Epoch 48/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 0.0349	acc: 0.9992		
4402/4402	[===== ===== ===== =====]	0s 38us/sample	loss: 0.0348	acc: 0.9993	val_loss : 0.0757	val_acc: 0.9789
<i>Epoch 49/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 0.0331	acc: 0.9997		
4402/4402	[===== ===== ===== =====]	0s 40us/sample	loss: 0.0330	acc: 0.9995	val_loss : 0.0668	val_acc: 0.9789
<i>Epoch 50/50</i>						
3584/4402	[===== ===== ===== =>.....]	ETA: 0s	loss: 0.0309	acc: 0.9994		

Iteration		Time / Sample	Training Loss	Training Accuracy %	Test Loss	Test Accuracy %
4402/4402	[===== ===== ===== =====]	0s 37us/sample	loss: 0.0305	acc: 0.9995	val_loss : 0.0561	val_acc: 0.9789

After the model was generated the basic structure of the MLP model is as described in [Table 14](#).

Table 15. MLP structure

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 1799)	0
dense (Dense)	(None, 256)	460800
dense_1 (Dense)	(None, 128)	32896
dense_2 (Dense)	(None, 64)	8256
dense_3 (Dense)	(None, 1)	65

Total params: 502,017

Trainable params: 502,017

Non-trainable params: 0

12.5. Model Accuracy

The Precision and Recall Ratio metric is used to measure the accuracy of the model. The Area Under Precision Recall Curve (AUPRC) method was used.

Compusult found a success rate of over 98% accuracy when using an MLP to categorize records as "Arctic vs non-Arctic". This high success rate is somewhat alarming, and the model was analyzed to try to determine the cause of the high success rate.

Due to the simplification of training data labeling, negative training data was chosen from three CSW sources. This negative training data may share common keywords or features, which increased the "bias" of the negative training data and decrease the "variant". This may be the reason why the accuracy of the result is very high. Additionally, it was found that Arctic related records from the positive training datasets all contained common keywords which leads to a very binary "Arctic vs non-Arctic" classification.

12.5.1. Results

During this pilot the following results were achieved:

1. An instance of Compusult's Web Enterprise Suite (WES) was deployed. The WES Catalog product (based on CSW v2.0.2) used as an input catalog and populated with Arctic and non-Arctic records.
2. The catalog was configured to periodically harvest new records from various geospatial web services.
3. A ML model was developed using TensorFlow and PyTorch and trained to categorize input records

as Arctic or non-Arctic.

4. A process was developed to periodically read new records from the input catalog and apply the ML model as a filter to evaluate each record as Arctic or non-Arctic related.
5. All Arctic related records are written to another instance of WES, known as the Arctic Discovery Catalog (based on CSW v2.0.2).

This Arctic Discovery Catalog is available online to the sponsors at link: <http://ogc.compusult.com>

12.5.2. Standards

The output Arctic Discovery Catalog is based on the OGC CSW v2.0.2. It is available online via a CSW interface. Users can also log into WES and browse the catalog at <http://ogc.compusult.com> [<http://ogc.compusult.com>] (user account required). The catalog is documented in the Testbed-15 D010 Catalog ER.

12.5.3. Interoperability

Since the Arctic Discovery Catalog is a standards-based catalog, any user/system that has implemented the OGC CSW standard can inter-operate with the Arctic Discovery Catalog using standard interfaces.

12.6. Future Directions

12.6.1. Convolutional Neural Network Implementation

Compusult investigated implementing a multilayer CNN using Python and TensorFlow but due to the high accuracy rate of the MLP it was determined unnecessary due to the additional effort. The proposed CNN would have 7 layers as follows:

1. Input Layer
2. Five Hidden Layers
3. Output Layer

The same dataset that was used to train the MLP would be used to train the CNN.

12.6.2. Recurrent Neural Network Implementation

Compusult also investigated the use of an RNN to further enhance information extraction and processing of the Title and Abstract fields. An RNN would provide additional context to the complete sentence structure but this was deemed unnecessary for a simple categorization problem such as Arctic vs non-Arctic. This might be a candidate for a future Testbed.

12.6.3. Evergreen Harvester

An evergreen-catalog harvester that uses ML to help search engines associate documents with similar thumbnails.

The harvester discovers new candidate web pages for possible inclusion in the catalog. If that page has a

thumbnail in its metadata, the harvester could feed it to an ML model (almost like an image classification model) to help determine whether the candidate should be added to the catalog or not. The thumbnail similarity would be just one bit of evidence among many. However, if a minimally viable use case is being considered, the result of the thumbnail classification would determine whether to add the new page or not.

12.6.4. Unsupervised Learning

Explore unsupervised learning - The idea would be that there are some web assets (documents, datasets, etc.) that an ML model might consider worthy of including in an evergreen catalog. But we might not (yet) be able to describe the reason why in words. It could just be that the asset has the right mix of feature values to make it sufficiently similar to the assets used in model training. This is somewhat similar to the "anti-D103" approach, as it could be perceived as working against the goal of building a verbal semantic network.

Chapter 13. Discussion

As described in the [Testbed-14 Machine Learning ER](http://docs.openeospatial.org/per/18-038r2.html) [http://docs.openeospatial.org/per/18-038r2.html], there are in general a defined set of ML operations that are common to all ML operations. In the web services world, including WPS 2.0, these operations are as follows:

- TrainML
- RetrainML
- ExecuteML

The work carried out in this Testbed-15 thread followed the same basic pattern. Therefore there is some vindication and confirmation of work carried out in previous Testbeds. The next section describes how the work carried out in Testbed-14 and Testbed-15 in the ML space might translate to future profiles of the emerging OGC API – Processes specification.

13.1. OGC API - Processes Operations

As mentioned previously, the OGC API – Processes draft specification is described using the OpenAPI specification as a Swagger document. A final version of the Processes specification ready for OGC Member and community review has yet to be agreed to and documented. However, other OGC Pilot projects can be considered to get a feel for where the Processes specification might solidify. The OGC [Open Routing API Pilot](https://www.openeospatial.org/projects/initiatives/routingpilot) [https://www.openeospatial.org/projects/initiatives/routingpilot] seeks to make use of the OGC API – Processes draft specification for routing. The OGC [API Hackathon](https://www.openeospatial.org/projects/initiatives/oapihackathon19) [https://www.openeospatial.org/projects/initiatives/oapihackathon19] that took place in London in June 2019 was also influential in defining OGC APIs, although their relationship with WPS has yet to be rationalized.

13.1.1. Suggestions for OGC API – Processes endpoints

There are currently two schools of thought on API structure for OGC API – Processes. These are broadly as follows:

- Type 1: /<baseUrl>/<operation>
- Type 2: /processes/<baseUrl>/jobs/<job>

The Type 1 pattern is the most simplistic in terms of structure as it does not mandate prefixes or operations. In the Routing API Pilot, an example operation to get a route is as follows:

- GET /routes/<routeId>

The thinking behind this pattern is that the API architecture for the OGC is resource-based. Therefore all operations should be treated as resources. The purpose of the Routing Pilot work was to get a route via the interface. The method that the route is generated by is largely opaque to the user (or machine) that is calling the service.

The alternative (Type 2) describes a more typical OGC pattern that seeks to preserve some of the capabilities/patterns from WPS 2.0. This approach enables implementers to simply wrap their service according to the OGC API – Processes specification and have the calls pass through to the old service. This

approach does additional work in that it preserves the concept of WPS throughout the OGC and applies some rigidity for implementers to design clients against. Which method is eventually adopted remains to be seen. Some example, potential patterns for ML using Type 1 are as follows:

- POST /machineLearning/petawawa GET /machineLearning/newbrunswick/features?[id]

The equivalent patterns for the Type 2 pattern could be represented as:

- POST /processes/machinelearning/jobs

With the process ID put in the POST body, this returns a job-id GET /processes/machinelearning/jobs/<job-id>/result. Type 1 is clearly more flexible, whereas Type 2 offers structure. The actual decisions on an API pattern for the OGC could be defined in a Pilot program or future Testbed.

13.2. Recommendations

The work completed in the ML thread produced several recommendations that are documented in this section. The primary objectives of the Testbed-15 ML thread were to:

1. Understand and define how the latest set of OGC services should be configured to work with ML models.
2. Produce a Best Practices document specifically for ML and its use within the OGC. There is potentially nothing *geo-specific* regarding ML models. ML technology usage within the geospatial domain is what requires interaction and standardization.

13.2.1. D102 Recommendations

During the implementation of the D102 component, the input data were analyzed from an interoperability point of view, and three different types of data were found:

1. Model data
2. Parameters
3. Time series

13.2.1.1. Model data

These are the data needed to build the model - that is, train or retrain the algorithms. Any substantial change in the model data requires retraining the algorithms in order to keep their validity. For example, if the woods types classification changes, the algorithms need to be retrained in order to consider the new wood taxonomy. Feeding the model data to the algorithms iteratively from a WCS/WFS is not viable for two reasons:

1. Model data usually requires preprocessing before being properly curated for a training feed. Raw data is rarely useful for direct input into training. For example, the raw data provided by NRCAN consists, among others layers, of forest stand coordinates and the forest road network. However, in order to train the harvest agents the minimum distance to a road for each forest stand needs to be calculated. Calculating the distance on-the.
2. Even if model data is offered fully curated and ready to feed for training, fetching each element

remotely is extremely inefficient compared to fetching them locally. Taking into account that in each training cycle each element can be required thousands of times, it is just not feasible to fetch the element remotely every time. In summary, WCS/WFS endpoints are expected to provide generic data in a standard way to build value-added products or services. However, they are not expected to provide specific, pre-processed data as efficiently as a local database. This opens an interesting topic: Updates. If an ML model needs to download remote, generic data from a WCS/WFS and preprocess it in order to train its algorithms, the model will need two things:

3. Ensure that trained algorithms are valid after model data updates (within a reasonable range). For example, if harvest agents are now using more powerful machinery and they can chop a bigger area per day, the algorithms should be able to cope with the change. This requires the model to be trained using a variety of values in those parameters that are likely to change (see below).
4. Ensure that predictions and simulations are based on the latest available data. For example, if a forest block has been harvested in real life, the model should be able to update its database and not to suggest further harvesting in that same block during simulations.

Recommendation: Explore the specific case of ML models feeding from dynamically updated data from WCS/WFS instances. Alternatively, sensor related services such as [SensorThings API](https://www.opengeospatial.org/standards/sensorthings) [https://www.opengeospatial.org/standards/sensorthings] or [Sensor Observation Service \(SOS\)](https://www.opengeospatial.org/standards/sos) [https://www.opengeospatial.org/standards/sos] could also be applied. Such an exploration would help to draw a series of best practices.

13.2.1.2. Parameters

Each agent is defined by a series of parameters that can vary between different scenarios or even between agents. For example, transport agents can have different load capacities, different fuel consumption or travel costs, or even different costs for loading or unloading the wood cargo. Parameters are needed, either directly or indirectly, as input data for training. However, once this is complete, the model can be run with different parameter values in order to compare results. This requires that the parameter values used for training are carefully selected, allowing the algorithms to learn the effect of each parameter in the overall result.

If the algorithms have been properly trained and small changes in agent parameters do not require retraining, this approach can be a very powerful tool to compare scenarios to help make business decisions. For example, if we are considering adding a new truck to our fleet, would the new truck increase overall revenue or would it saturate the mills and generate costs? If a provider has a fleet of large and expensive trucks, if a customer chooses a provider with small and more mobile units, would it help the customer be more dynamic and optimize better the price variations or would it operate less efficiently?

The interest of parameters is not limited to active agents. The process can easily be applied to mills. For example, if a new mill is opened in a specific location, would it help boost overall production or would it just cannibalize the business of other similar mills in the province? If there is a mill with a particularly high demand, would it make sense to enlarge its capacity? What would be the estimated return on investment?

The use of standards can become crucial in this case if a ML model is carefully built, trained and offered publicly for land owners, forest companies, private mills or logistic operators as a powerful tool to help them make wiser decisions. Using standards, they could plug their data (such as the size and

characteristics of trucks, in the case of a logistic operator) and explore different scenarios to compare results (such as opening a new logistic base in Moncton in order to serve the Western part of the province more efficiently, or renew the fleet with more fuel-efficient trucks).

Recommendation: Explore the use of OGC standards to compare scenarios via a previously trained ML model.

13.2.1.3. Time series

The potential to predict prices accurately over long periods of time is limited, but time series can be exploited as a way to evaluate extreme scenarios and evaluate the ability of the entire industry to withstand stress. Different scenarios can be explored with extreme conditions to identify breaking points and prepare contingency plans to assess and soften incoming crises. For example, a scenario could be run with mounting fuel prices in order to find the point at which transport teams consider it more rewarding to stay at home than to move wood cargo to the mills. Or a stress test in which wood prices are decreasing and fuel prices increasing would show which mills would be the first to suffer from low wood stock levels.

Timeseries are a particular type of parameter data, and as such they are needed as input for training, with a carefully selected set of values that allow the algorithms to learn the effect of the state of each time series in the overall result. Similarly, as with parameters, the use of standards can become crucial if a ML model is offered publicly to different players in the industry. Continuing with the example of a logistic operator, this could allow them to analyze the scenario of mounting fuel prices and prepare contingency plans.

Recommendation: explore the use of OGC standards to perform stress tests via a previously trained ML model.

13.2.1.4. Recommendations for OGC Standards

At the outset of D102 New Brunswick development, it was assumed a "normal" processing workflow would suffice for WPS requests. However, this assumption turned out to be false. Training an agent or running a full simulation might take hours or days, depending on the configuration and the parameters. Conventional HTTP requests were too long-running to keep open sockets. Consequently, considerations into HTTP callbacks must be considered for future ML-WPS combinations. As processes on a ML server may take hours to days, it is much more efficient to split the request-response into two distinct sessions- one for job creation and ML processes execution, and another session for data callback via HTTP POST on the WPS.

Additional functionality could be envisioned using "OGC API" standards. For example, NRCan's data repository could model all mills, harvest and transport teams together, published through an OGC API - Features implementation. Time estimations for fuel and wood prices could be offered through an OGC API - Processes implementation using this data and fed back into the Features API.

Future work in these areas meshes well with proposed OGC API paradigms. In the previous example we discussed transport agents. The manipulation of parameters would be axiomatic in an OGC API - Processes world. For example, various Transport teams and their resources may be easily represented in a RESTful manner- i.e.: `NRCan/NewBrunswick/Scenario325/Teams/Transport/Transport2/Trucks/ToyoyaHatchback` where more expressive parameters may be outlined (i.e. POSTing of new transport truck resources, number of trucks, fuel prices, etc.) to facilitate finer-grained results. In this fashion, multiple teams may be either pulled from an OGC API - Features implementation or created as resources on an OGC API -

Processes implementation and used to run multiple, multivariate estimation processes. This would allow the generation of RL models similar to the one built by Skymantics. New scenarios could be explored by tuning the data via an OGC Process API.

An additional advantage of OGC APIs may be seen in the possible modularity of cloud-based processing while maintaining logical union. For example, the logical delineation of transport and harvest team endpoints or even different scenario endpoints may have distinct, differing physical storage areas in a cloud environment. Thus, a federated system of databases may be abstractly represented as a single Features API and offered via one single WFS endpoint. Consequently, multiple cloud-based processing services may be employed in ML processing of scenarios in an OGC API - Process fashion, with multiple scenarios represented as scaled-out services. Such an approach provides modularity and scalability benefits for both OGC WPS and WFS deployments, Big Data OGC working groups, and client-sponsored Pilots.

13.2.2. D104 Recommendations

Based on work presented in D104 Component section of this ER, the following recommendations for future work are suggested:

13.2.2.1. OGC API - Machine Learning

- Adopt a common API that captures typical geospatial ML workflows and processes.
- Enable OGC API - Processing in ML dashboards, such as Tensorboard.
- Add use cases considering in-the-loop data annotation and validation by an analyst.

13.2.2.2. Learned models

- Experiment with instance-based detection and segmentation approaches, instead of bounding box detection.
- Provide expert-tuned morphologic and geometric transformations of hydrographic networks.
- Develop services to export train models in various formats.

13.2.2.3. OGC API - Processes

- Add event-driven architectures such as *Pub/Sub* for machine learning processes, triggered by:
 - A threshold crossed on predetermined metrics of the *trained model* while in process, delivering alerts to the user.
 - Updated source data or annotated dataset, requiring retraining.
 - A feature with associated low confidence score, requiring user intervention.

13.2.2.4. Open Architecture

- Install on both an *ADES* and an *EMS* on Pacific Boreal Cloud.
- Conduct additional TIEs for:
 - *Application Packaging* of ML frameworks and workflows.
 - Execution of ML applications through *ADES and EMS*.

- Use of pre-deployed WPS endpoints in *CWL* workflows.
- Creation of data parsing (pre-processing) and refinement (post-processing) processes in *CWL*.

13.2.3. D105 (OGC API - Features service) Recommendations

Based on discussions with participants and experiences from this initiative, an approach for future work is discussed in this section.

13.2.3.1. Client-driven real-time distributed execution of Machine Learning model

Implementing a client-driven workflow executing the trained Machine Learning in a distributed manner in the cloud, with the ability to do so in real-time on any portion of a very large dataset, would likely best support the operational needs for which these experiments were devised. Given more time, this might have been the desired outcome for this activity, but this work was deemed out of scope.

The architecture defined in the CFP did not support achieving the suggested workflow approach. The intermediary data delivery component could be removed thereby allowing the clients to request the data directly from the server. This approach would avoid unnecessary back and forth communication and remove architectural complexity. Including the delivery service produces the following logical set of interactions:

1. The client makes a request to either the delivery service or the processing service.
2. The processing service uploads the results to the delivery service (with the mechanism as yet undefined)
3. The delivery service then informs the client the data is ready.
4. The client requests the data from the delivery service.
5. The delivery service then returns the data.

These steps show that the architectural choices made in the CFP have led to five interactions where only two were required in the simplified proposal (request & response). Another solution was processing the data for different geographical areas and at different scales. As such, tiles offer an attractive solution to the problem. Such a system could benefit from many core concepts introduced by the OGC API family of capabilities, such as its modular aspect; OGC API - Features, Tiles and Processes could be combined for the client to make processing requests for the specific area and scale of interest.

As discussed in the OGC Open Routing Pilot Engineering Reports, a hybrid approach to OGC API - Processes (*/processes* **and** */routes*) should be possible. Using this approach processes can still be described, but arbitrary APIs can be defined to initiate processing (with options for synchronous and asynchronous processing, and more options for polling, callbacks or alternative approach). This would enable a combination of the Processing and Tiles API so that issuing a tile request can initiate a process, or to define the Maps API for server-side rendering as a process.

If the required processing can be done on a tile-by-tile basis, the service can then execute the ML model (or any other type of processing allowing this) on each tile. This could be done in a parallel manner therefore enabling the great scalability of distributed computing, and leveraging large numbers of CPU and/or GPU cores. Caching can be introduced and calibrated to balance excessive repetitive processing with memory and disk storage capacity.

As recommended by CRIM as an outcome of the experiments, the Mask R-CNN method of detection would indeed lend itself very well to the approach discussed in this section and the workflow could be implemented extremely efficiently, and be returned almost immediately to the client.

On the far end of the processing side, the input to the processes can also be accessed on a tile-by-tile basis, minimizing the data load, and the requirement for transferring data across the network if it is stored remotely (e.g. the GRHQ and HRDEM data input into the Machine Learning model). Complex workflows could also be defined strictly in terms of chaining OGC APIs, offering much more flexibility to the user on the client-side than requiring the definition of the workflow beforehand to be deployed on a single processing server. For example, existing algorithms for hydrography analysis of DEM (such as these available in QGIS) could be chained as a preprocessing step before being interpreted by the Neural Network, likely resulting in much improved detection results.

This methodology should also enable integrating within the workflow any data, or processes, offered via any available OGC API. Also, this creates the possibility of uploading multiple algorithms to multiple datasets in a completely distributed manner, all the while still providing real-time input to the user.

This immediate feedback allows the user to adjust parameters for the processing without having to wait for long processes to complete before validating results, and wasting valuable computing resource.

The client could browse anywhere within an arbitrarily large or detailed dataset and experience the same instant responsiveness. At the same time also saving processing and storage power on the server side for only processing data requested by clients (but still caching as desired). Potentially this also enables the use of the latest data updated as often as necessary, even in real-time (e.g. satellite imagery being continuously received). This seems particularly well suited e.g. for monitoring natural disasters.

This is an approach that Ecere is actively researching and promoting, while also advocating to ensure that the new OGC API standards enable these types of capabilities. See these [OGC API - Common issue](https://github.com/opengeospatial/oapi_common/issues/17) [https://github.com/opengeospatial/oapi_common/issues/17] and [OGC API - Processes issue](https://github.com/opengeospatial/wps-rest-binding/issues/47) [https://github.com/opengeospatial/wps-rest-binding/issues/47] for additional details on these ideas.

Further exploration in future OGC Innovation Program activities is recommended.

Chapter 14. Conclusion

The work performed in the OGC Testbed-15 ML thread exercised OGC standards in the context of Machine Learning using five different scenarios. The scenarios incorporated a set of use cases that included traditional ML techniques for image recognition, understanding the linkages between different terms to identify a dataset, and vectorization of identified water bodies using satellite imagery. Overall, this Testbed thread provided opportunities to explore ML methods and application across different use cases while contributing to the discourse in terms of OGC standards requirements. The existing OGC standards utilized in the thread included:

- WPS
- WFS
- CSW

However, these are web service-based standards that will soon be complemented by OGC API specifications that are based on OpenAPI descriptions and RESTful principals. For example, the OGC API – Features standard is tightly controlled as conceptually, it is tightly coupled to geospatial data and operations. Contrarily, the OGC API - Processes draft specification is still undergoing activities to provide a definition. One option in this exercise is to have a very loose definition of an API and allow implementers to essentially follow the OpenAPI description of the OGC API.

Appendix A: Configuration file for the ML application

```
name: "testbed15-predict"
datasets:
  testbed15:
    type: "helper.data.geo.ogc.TB15D104Dataset"
    params:
      raster_path: "data/testbed15/roi_hrdem.tif"
      vector_path: "data/testbed15/hydro_original.geojson"
      px_size: 3
      lake_area_min: 100
      lake_area_max: 200000
      lake_river_max_dist: 300
      roi_buffer: 1000
      srs_target: "2959"
      reproj_rasters: false
      display_debug: true
      parallel: 0
loaders:
  workers: 0
  batch_size: 1
  base_transforms:
    - operation: torchvision.transforms.ToTensor
      target_key: input
  collate_fn:
    type: helper.data.loaders.default_collate
    params:
      force_tensor: false
  train_split:
    testbed15: 0.9
  valid_split:
    testbed15: 0.1
# with this section, we define the metrics that will be used by the test 'trainer'
# this is basically the prediction outputs that the model will produce during inference
trainer:
  metrics:
    mAP:
      type: helper.optim.metrics.AveragePrecision
    output:
      type: helper.train.utils.DetectLogger
      params:
        format: json
# details about the model
model:
  type: torchvision.models.detection.fasterrcnn_resnet50_fpn
  params:
    pretrained: true
```


Appendix B: JSON file for ML App Process Description

JSON file providing the Process Description for the ML App Package, where unit element of offering refers to CWL file in Annex

```
{
  "processDescription": {
    "processVersion": "0.1.2",
    "process": {
      "id": "ogc-tb15-lake-river-detector",
      "title": "OGC Testbed-15 Lake/River detector",
      "abstract": "Lake/River vector differentiation using deep learning model with bounding box detections. Built by researchers and developers from the Centre de Recherche Informatique de Montréal / Computer Research Institute of Montreal (CRIM).",
      "keywords": [
        "machine learning",
        "deep learning",
        "neural network",
        "detection",
        "lake",
        "river"
      ],
      "inputs": [
        {
          "id": "raster_file",
          "title": "Raster file",
          "abstract": "Input HRDEM TIF file (High-Resolution Digital Elevation Model).",
          "formats": [
            {
              "mimeType": "image/tif",
              "default": true
            }
          ],
          "minOccurs": 1,
          "maxOccurs": 1
        },
        {
          "id": "vector_file",
          "title": "Vector file",
          "abstract": "GeoJSON feature collection file representing Lake/River undifferentiated vectors.",
          "formats": [
            {
              "mimeType": "application/json",
              "default": true
            }
          ]
        }
      ]
    }
  }
}
```



```
    }  
  },  
  "outputs": {  
    "output": {  
      "outputBinding": {  
        "glob": "${runtime.outdir}/output.json"  
      },  
      "type": "File"  
    }  
  }  
},  
],  
"deploymentProfileName": "http://www.opengis.net/profiles/eoc/dockerizedApplication"  
}
```

Appendix C: CWL file for the helper ML Application Package

CWL file wrapping the Docker image of the helper as a ML Application Package

```
## following definition should match the 'executionUnit.unit' section of 'process-
deploy.cwl'
cwlVersion: v1.0
class: CommandLineTool
requirements:
  DockerRequirement:
    dockerPull: docker-registry.crim.ca/ogc-public/ogc-thehelper-tb15:0.1.1
baseCommand: ogc_thehelper_tb15
arguments:
- "-vv" # will log additional debug messages (not used for real process deployment)
- "-o"
- "$(runtime.outdir)"
inputs:
  raster_file:
    type: File
    inputBinding:
      position: 1
  vector_file:
    type: File
    inputBinding:
      position: 2
outputs:
  output:
    outputBinding:
      glob: "$(runtime.outdir)/output.json"
    type: File
```

Appendix D: Log output of the training process for D104

```
[2019-08-22 13:22:13,300 - 12196] INFO : created training log for session 'testbed15-train' [2019-08-22
13:22:13,598 - 12196] DEBUG : logstamp = GTW-LABOVISI1656-20190822-132213 [2019-08-22 13:22:13,598 -
12196] DEBUG : version = 0.3.9:52f42bae663aa1f3d635f0cf91213a915cc294ea [2019-08-22 13:22:13,598 -
12196] DEBUG : loading available devices [2019-08-22 13:22:21,184 - 12196] INFO : parsed metric 'mAP':
thelper.optim.metrics.AveragePrecision(target_class=None, max_win_size=None) [2019-08-22 13:22:21,184 -
12196] DEBUG : uploading model to '[2]'... [2019-08-22 13:22:21,294 - 12196] DEBUG : loss: None [2019-08-22
13:22:21,294 - 12196] DEBUG : optimizer: SGD ( Parameter Group 0 dampening: 0 lr: 0.005 momentum: 0.9
nesterov: False weight_decay: 0.0005 ) [2019-08-22 13:22:21,296 - 12196] INFO : at epoch#0 for 'testbed15-
train' (dev=[2]) [2019-08-22 13:22:21,296 - 12196] DEBUG : learning rate at 0.00500000 [2019-08-22
13:22:21,296 - 12196] DEBUG : fetching data loader samples... [2019-08-22 13:22:24,922 - 12196] INFO : train
epoch#0 (iter#0) batch: 1/2031 (0%) loss: 2.464282 [2019-08-22 13:22:26,703 - 12196] INFO : train epoch#0
(iter#1) batch: 2/2031 (0%) loss: 1.687276 [2019-08-22 13:22:28,438 - 12196] INFO : train epoch#0 (iter#2)
batch: 3/2031 (0%) loss: 1.156375 [2019-08-22 13:22:29,130 - 12196] INFO : train epoch#0 (iter#3) batch:
4/2031 (0%) loss: 1.014334 [2019-08-22 13:22:30,571 - 12196] INFO : train epoch#0 (iter#4) batch: 5/2031 (0%)
loss: 0.474855 [2019-08-22 13:22:32,073 - 12196] INFO : train epoch#0 (iter#5) batch: 6/2031 (0%) loss:
1.555177 [2019-08-22 13:22:33,637 - 12196] INFO : train epoch#0 (iter#6) batch: 7/2031 (0%) loss: 1.676842
[2019-08-22 13:22:35,412 - 12196] INFO : train epoch#0 (iter#7) batch: 8/2031 (0%) loss: 1.170705 [2019-08-22
13:22:36,895 - 12196] INFO : train epoch#0 (iter#8) batch: 9/2031 (0%) loss: 1.559234 [2019-08-22
13:22:38,043 - 12196] INFO : train epoch#0 (iter#9) batch: 10/2031 (0%) loss: 0.454997
```

Appendix E: Revision History

NOTE

Example History (Delete this note).
replace below entries as needed

Table 16. Revision History

Date	Editor	Release	Primary clauses modified	Descriptions
June 15, 2016	I. Simonis	.1	all	initial version
July 22, 2016	I. Simonis	.9	all	comments integrate
September 7, 2016	S. Simmons	1.0	various	preparation for publication
March 23, 2017	I. Simonis	2.0	all	template simplified
January 18, 2018	S. Serich	2.1	all	additional guidance to Editors; clean up headings in appendices

Appendix F: Bibliography

1. Ren, S., He, K., Girshick, R.B., Sun, J.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. CoRR. abs/1506.01497, (2015).
2. He, K., Gkioxari, G., Dollár, P., Girshick, R.B.: Mask R-CNN. CoRR. abs/1703.06870, (2017).
3. Rajaraman, A.; Ullman, J.D. (2011). "Data Mining" (PDF). Mining of Massive Datasets. pp. 1–17.
4. X. Zhang, S. Ren and J. Sun: Deep Residual Learning for Image Recognition, in CVPR, 2016.
5. Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. 2014. arXiv:1412.6980v9.
6. S. Mohajerani and P. Saeedi: An end-to-end Cloud Detection Algorithm for Landsat 8 Imagery - arXiv preprint arXiv:1901.10077.
7. G. Morales, A. Ramírez and J. Telles: End-to-end Cloud Segmentation in High-Resolution Multispectral Satellite Imagery Using Deep Learning arXiv:1904.12743v1.
8. Z. Zhu, S. Wang and C. E. Woodcock: Improvement and expansion of the fmask algorithm: cloud, cloud shadow, and snow detection for landsats 47, 8, and sentinel 2 images," Remote Sens. of Env., vol. 159, pp. 269 – 277, 2015. Last updated 2019-09-23 13:25:53 +0100.