July 3, 2019

Software Requirements Document

Version 1.0.0

OpenML

Members
Andrei Danila | 1034559
Bogdan Enache | 1035066
Gergana Goncheva | 1037010
Loïc Alexander Hijl | 1002745
Adrian-Stefan Mares | 0993873
Veselin Minev | 1014541
Thanh-Dat Nguyen | 1036672
Antoine Labasse Lutou Nijhuis | 1016657
Claudiu-Teodor Nohai | 1038442
Dragos Mihai Serban | 1033859
Tsvetan Zahariev | 1035269
Sonya Zarkova | 1034611

Project managers
Yuxuan Zhang
Stefan Tanja

Supervisor
Erik Luit
Ion Barosan

Customer
Joaquin Vanschoren

Abstract

This document contains the software requirements for the Deep Learning (DL) Extension Library for OpenML, which is developed by the team OpenML Support Squad. The requirements in this Software Requirements Document (SRD) satisfy the requirements in the User Requirements Document (URD) [1] of the DL Extension Library. This document complies with the Software Engineering Standard, defined by the European Space Agency (ESA) [2].

Contents

1 Introduction **6**
1.1 Purpose 6
1.2 Scope 6
1.3 Definitions and Abbreviations 8
1.3.1 Definitions 8
1.3.2 Abbreviations 11
1.4 List of references 11
1.5 Overview 12

2 General Description **14**
2.1 Relation to Current Projects 14
2.2 Relation to Predecessor and Successor Projects 14
2.3 Function and Purpose 14
2.4 Environment 15
2.5 Relation to Other Systems 16
2.5.1 OpenML Python API 16
2.5.2 Deep learning frameworks 17
2.5.3 Deep learning specifications 17
2.6 General Constraints 17
2.6.1 Visualization 17
2.6.2 Data Reliability 17
2.6.3 Environment constraints 18
2.6.4 Minimizing dependencies and storage space 18
2.7 Model Description 18
2.7.1 Abstract class diagrams 18
2.7.2 Abstract Keras model diagrams 19
2.7.3 Abstract PyTorch model diagrams 20
2.7.4 Abstract ONNX specification diagrams 22
2.7.5 Abstract MXNet model diagrams 23
2.7.6 Class diagrams 24
2.7.7 High-level sequence diagrams 40
2.7.8 Low level activity diagrams 46

3 Specific Requirements **61**
3.1 Extension «interface» 62
3.2 KerasExtension 67
3.3 PyTorchExtension 72
3.4 ONNXExtension 83
3.5 MXNetExtension 86
3.6 Visualization 93

4 Requirements Traceability Matrix 96
4.1 Software Requirements to User Requirements 96
4.2 User Requirements to Software Requirements 99

A User Interface Mockups 101
A.1 Visualizing a run 101
A.2 Visualizing a flow 105

B Transitions 108

Document Status Sheet

Document Title: Software Requirements Document
Identification: SRD/1.0.0
Version: 1.0.0
Authors: A.L.L.Nijhuis, A.Danila, B.Enache, G.Goncheva, T.Nguyen, T.Zahariev, S.Zarkova, L.Hijl

Document History

Version	Date	Author(s)	Summary
0.0.1	06-05-2019	T.Zahariev, L.A.Hijl	Created initial document
0.0.2	07-05-2019	B.Enache, S.Zarkova, G.Goncheva, T.Nguyen, T.Zahariev, A.Danila	Added first draft of abstract, sections 1.1, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6
0.0.3	11-05-2019	All authors	Added first draft for section 2.7 only for Keras
0.0.5	13-05-2019	All authors	Implement first internal feedback on all section until section 3
0.0.4	15-05-2019	All authors	Added first draft for sections 3.1.1, 3.1.2
0.0.5	15-05-2019	All authors	Implemented first internal feedback for sections 3.1.1, 3.1.2
0.1.0	20-05-2019	All authors	Implemented first feedback from supervisor in the functional requirements and description of sequence diagrams
0.1.1	10-06-2019	All authors	Implemented second internal feedback on everything
0.2.0	10-06-2019	All authors	Implemented second feedback from supervisor
0.3.0	21-06-2019	All authors	Implemented third feedback from supervisor
1.0.0	30-06-2019	All authors	Final edit

1 | Introduction

1.1 Purpose

This document contains the software requirements for the DL Extension Library, which represents an extension for the OpenML platform. These requirements are a translation of the user requirements found in Section 3 of the User Requirements Document (URD) [1].

Whereas the URD specifies the required functionalities, the Software Requirements Document (SRD) defines the manner according to which these are to be implemented.

The two documents are written by the OpenML Support Squad in accordance to Joaquin Vanschoren's preferences and guidance.

The purpose of project "Sharing deep learning models" is to create an extension for the already existent OpenML Python Application Programming Interface (API) and provide a visualization module. The extension, called DL Extension Library, represents the biggest part of the product that OpenML Support Squad has to deliver. DL Extension Library will enable sharing deep learning models to the OpenML platform. The visualization module allows visualization of a model's structure and performance metrics in Dash.

1.2 Scope

OpenML is an open-source online machine learning platform for sharing and organizing data, machine learning algorithms and experiments. It allows people all over the world to collaborate and build on each other's latest findings, data or results. OpenML is designed to create a seamless network that can be integrated into existing environments, regardless of the tools and infrastructure used. It allows users to upload datasets, models (called OpenML flows) and tasks for OpenML flows. It further provides support for uploading runs that combine a model with a task for it, to allow users to compare results.

The DL Extension Library is part of the OpenML Python API that will be imported by the users in their Python project. The OpenML Python API handles all interactions with the OpenML platform. Whenever necessary, the OpenML Python API calls the DL Extension Library in order to convert the deep learning model to an OpenML flow, the format supported by the OpenML platform. The OpenML Support Squad is not responsible for the development and maintenance of the OpenML Python API. The OpenML Support Squad will not modify the OpenML Python API since it is outside of the scope of the project "Sharing deep learning models".

The contribution of OpenML Support Squad consists of two parts: creating the DL Extension Library and the visualization module.

The DL Extension Library is built to provide support for the following deep learning libraries and specifications on the OpenML platform: Keras, PyTorch, Apache MXNet (MXNet) and Open Neural Network eXchange (ONNX). These libraries have already been created by their respective developers and, as such, are not part of this project and are not under the responsibility of OpenML Support Squad.

In the implementation of these extension packages, the OpenML Support Squad needs to take into consideration the already existent OpenML extension interface that the OpenML Python API provides. This extension interface can be seen in figure 2.7. The OpenML Support Squad is not responsible for the structure and design of this interface.

The visualization module is built to serve as a prototype which can be integrated into the website. It will allow users to visualize their models and model results.

An overview of the environment for this project can be seen in figure 1.1.

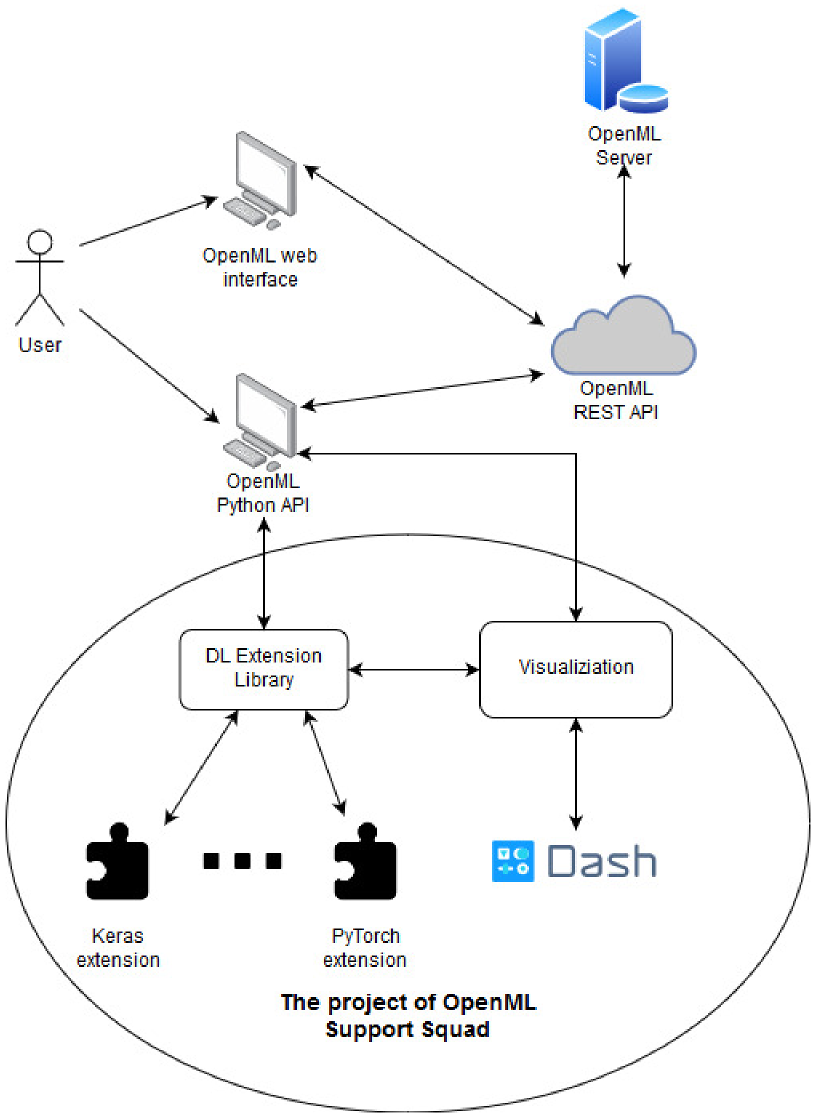


Figure 1.1: Overview diagram of the environment

1.3 Definitions and Abbreviations

1.3.1 Definitions

DL Extension Library	The product to be created in order to convert the deep learning models and the model transfer specifications (i.e. ONNX and MLflow) into a format supported by the OpenML platform.
Keras extension class	This implements OpenML extension interface and is part of the DL Extension Library.
MXNet extension class	This implements OpenML extension interface and is part of the DL Extension Library.
ONNX extension class	This implements OpenML extension interface and is part of the DL Extension Library.
OpenML Python API	"A connector to the collaborative machine learning platform OpenML.org. The OpenML Python package allows to use datasets and tasks from OpenML together with scikit-learn and share the results online" [3].
OpenML flow	The representation of untrained machine learning models in the OpenML platform.
PyTorch extension class	This implements OpenML extension interface and is part of the DL Extension Library.
Apache MXNet	"Apache MXNet is an open-source deep learning software framework, used to train and deploy deep neural networks. It is scalable, allowing for fast model training, and supports multiple programming languages (C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl, and Wolfram Language)" [4].
Caffe2	"CAFFE2 (Convolutional Architecture for Fast Feature Embedding) is a open-source deep learning framework" [5]. "Caffe2 comes with Python and C++ APIs" [6].
Classification task	"Classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known" [7].
Cross-validation	"Cross-validation or 'k-fold cross-validation' is when the dataset is randomly split up into 'k' groups (folds). One of the groups is used as the test set and the rest are used as the training set. The model is trained on the training set and scored on the test set. Then the process is repeated until each unique group as been used as the test set" [8].
Dash	"Dash is a user interface library for creating analytical web applications" [9].

Dataset	"Datasets are pretty straight-forward. They simply consist of a number of rows, also called instances, usually in tabular form" [3].
Deep learning model	A neural network with more than three layers. Deep learning is a subset of machine learning.
Direct conversion	A conversion from a deep learning library to the OpenML Flow format, which does not require an in-between conversion to a model transfer specification (such as ONNX).
Fold	A subset of the training data.
Hold-out	"Hold-out is when a dataset is split into a 'train' and 'test' set. The training set is what the model is trained on, and the test set is used to see how well that model performs on unseen data. A common split when using the hold-out method is using 80 % of data for training and the remaining 20 % of the data for testing" [8].
Hyperparameter	A hyperparameter is a parameter of a machine learning model that is set before training the model. Such parameters can be layers, seeds, etc.
Joaquin Vanschoren	Client of this project. Founder of OpenML.
JSON	"JSON (JavaScript Object Notation) is a lightweight data-interchange format" [10].
Keras	"Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano" [11].
Keras**	A placeholder for all implemented libraries. Can represent frameworks and specifications. Can be replaced with specific instances of Keras, PyTorch, ONNX and MXNet. Used for abstraction.
Machine learning model	A model that performs a "task without explicit instructions, relying on patterns and inference" [12] through learning from data.
Method Descriptor	A method descriptor is a reference, used by Python, to a function written in another programming language, e.g. *C*.
MLFlow specification	"An MLflow Model is a standard format for packaging machine learning models that can be used in a variety of downstream tools — for example, real-time serving through a REST API or batch inference on Apache Spark" [13].
Model	"A mathematical representation of a real-world process" [14].
ModuleDict container	A class from PyTorch - torch.nn.ModuleList(modules=None). "Holds submodules in a list. ModuleList can be indexed like a regular Python list, but modules it contains are properly registered, and will be visible by all Module methods" [8].

ModuleList container	A class from PyTorch - torch.nn.ModuleDict(modules=None). "Holds submodules in a dictionary. ModuleDict can be indexed like a regular Python dictionary, but modules it contains are properly registered, and will be visible by all Module methods" [8].
Neural Network	"Computing systems vaguely inspired by the biological neural networks that constitute animal brains." It is "a framework for many different machine learning algorithms" [15].
Non-redundant dependencies	The dependencies should be library-specific (based on the type of model used), only necessary dependencies being required for the proper operation of a certain type of a deep learning model.
Numpy	"NumPy is the fundamental package for scientific computing with Python" [16].
OpenML	An online machine learning platform for sharing and organizing data, machine learning algorithms and data experiments.
OpenML extension interface	This is the interface that the Keras extension class, PyTorch extension class and ONNX extension class concrete classes implement.
OpenML Support Squad	Name of the development team.
Project "Sharing deep learning models"	The project on which the OpenML Support Squad is working.
Protobuf	"Protocol Buffers (Protobuf) is a method of serializing structured data. It is useful in developing programs to communicate with each other over a wire or for storing data. The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data" [17].
PyTorch	"An open source deep learning platform that provides a seamless path from research prototyping to product deployment" [18].
Regression task	A statistical process that "attempts to determine the strength of the relationship between one dependent variable (usually denoted by Y) and a series of other changing variables (known as independent variables)" [19].
Run	"A run is a particular flow, that is algorithm, with a particular parameter setting, applied to a particular task" [3].
Sequential container	A class from PyTorch - torch.nn.Sequential(*args). "A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of modules can also be passed in" [8].

Task	"A task consists of a dataset, together with a machine learning task to perform, such as classification or clustering and an evaluation method. For supervised tasks, this also specifies the target column in the data" [3].
Tensor	"A tensor (torch.Tensor) is a multi-dimensional matrix containing elements of a single data type" [20].

1.3.2 Abbreviations

API	Application Programming Interface.
CV	Cross-validation.
DL	Deep Learning.
ESA	European Space Agency.
MXNet	Apache MXNet.
ONNX	Open Neural Network eXchange.
SRD	Software Requirements Document.
SRF	Software Requirements Functional.
UML	Unified Modeling Language.
URD	User Requirements Document.

1.4 List of references

[1] OpenML Support Squad. *DL Extension Library, URD User Requirement Document version 1.0.0.*

[2] Software Standardisation and Control. *ESA software engineering standards. 1991.*

[3] OpenML. *OpenML Documentation.* URL: `https://docs.openml.org/` (visited on 04/26/2019).

[4] Wikipedia. *Apache MXNet.* URL: `https://en.wikipedia.org/wiki/Apache_MXNet` (visited on 04/26/2019).

[5] Wikipedia. *Caffe (software.* URL: `https://en.wikipedia.org/wiki/Caffe_(software)` (visited on 04/26/2019).

[6] Caffe2. *Caffe2 homepage.* URL: `https://caffe2.ai/` (visited on 04/26/2019).

[7] Wikipedia. *Statistical classification.* URL: `https://en.wikipedia.org/wiki/Statistical_classification` (visited on 05/01/2019).

[8] Eijaz Allibhai. *Hold-out vs. Cross-validation in Machine Learning.* URL: `https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f` (visited on 05/17/2019).

[9] Kyle Kelley. *Introducing Dash.* URL: `https://medium.com/@plotlygraphs/introducing-dash-5ecf7191b503` (visited on 06/05/2019).

[10] JSON. *JSON homepage.* URL: `https://www.json.org/` (visited on 05/28/2019).

[11] Keras. *Keras homepage*. URL: `https://keras.io/` (visited on 04/26/2019).

[12] Wikipedia. *Machine Learning*. URL: `https://en.wikipedia.org/wiki/Machine_learning` (visited on 04/26/2019).

[13] MLflow. *MLflow Models*. URL: `https://mlflow.org/docs/latest/models.html` (visited on 04/26/2019).

[14] Joydeep Bhattacharjee. *Some Key Machine Learning Definitions*. URL: `https://medium.com/technology-nineleaps/some-key-machine-learning-definitions-b524eb6cb48` (visited on 05/01/2019).

[15] Wikipedia. *Artificial Neural Network*. URL: `https://en.wikipedia.org/wiki/Artificial_neural_network` (visited on 04/30/2019).

[16] Numpy Developers. *Numpy Homepage*. URL: `https://www.numpy.org/` (visited on 05/13/2019).

[17] *Protocol Buffers*. URL: `https://en.wikipedia.org/wiki/Protocol_Buffers` (visited on 05/29/2019).

[18] PyTorch. *PyTorch homepage*. URL: `https://pytorch.org/` (visited on 04/26/2019).

[19] Brian Beers. *Regression Definition*. URL: `https://www.investopedia.com/terms/r/regression.asp` (visited on 05/01/2019).

[20] PyTorch. *A Gentle Introduction to k-fold Cross-Validation*. URL: `https://machinelearningmastery.com/k-fold-cross-validation/` (visited on 05/24/2019).

[21] Agile Business Consortium. *MoSCoW Prioritisation*. URL: `https://www.agilebusiness.org/content/moscow-prioritisation-0` (visited on 04/26/2019).

[22] Wikipedia. *MoSCoW method*. URL: `https://en.wikipedia.org/wiki/MoSCoW_method` (visited on 04/26/2019).

1.5 Overview

The Software Requirements Document contains four chapters plus two appendices.

Chapter 1 is "Introduction". In Section 1.1, the purpose of this project is described. Section 1.2 presents the scope of our project and the context in which it is situated. In Section 1.3, a list of all definitions and abbreviations used in this document. Section 1.4 is a list of all references that are used in this document. Section 1.5 is the current section and describes the overview of this document.

Chapter 2 is "General Description". In Section 2.1, the context in relation to current projects is described. In Section 2.2, the correspondence to past/future project is depicted . In Section 2.3, a general overview of the purpose and function of the product can be found. Section 2.4 follows with environment description of the target and development system. Section 2.5 and 2.6 correspondingly outline the relation to other systems and the general constraints. Section 2.7 contains the logical description of the model.

Chapter 3 is "Specific Requirements". Section 3.1 depicts the list of all functional requirements related to the Extension interface. Section 3.2 depicts the list of all functional requirements related to the Keras extension. Section 3.3 depicts the list of all functional requirements related to the PyTorch extension. Section 3.4 depicts the list of all functional requirements related to the ONNX extension. Section 3.5 depicts the list of all functional requirements related to the MXNet extension. Section 3.6 depicts the list of all functional requirements related to the Visualization.

Chapter 4 is "Requirements traceability matrix". Section 4.1 contains a table with the translation from software requirements to user requirements. Section 4.2 contains a table with the translation from user requirements to software requirements.

There are the two appendices User Interface Mockups, and Transition Diagrams.

2 | General Description

2.1 Relation to Current Projects

The DL Extension Library serves as an extension of the OpenML Python API. At the beginning of this project, more precisely April 2019, OpenML did not support sharing of deep learning models on the platform. Hence, the aim of the project is implementing this functionality by adding new frameworks to the OpenML extension interface with the intention of sharing models and runs.

2.2 Relation to Predecessor and Successor Projects

This project has no predecessors. The successor of this project can extend the DL Extension Library's functionality. It is also possible that other libraries are created in the future which become more popular. These can then also be added to the DL Extension Library. This can be done by just converting from one format to another or by creating a new extension which holds the capabilities for conversion . Additional tasks for the successor could include compatibility issues that will arise, such as adapting to newer versions of Python.

2.3 Function and Purpose

The software product that the OpenML Support Squad will create, represents an extension library for the OpenML Python API, called DL Extension Library. The main reason for creating the DL Extension Library is to further increase the functionality of the OpenML platform. This is due to the lack of support that OpenML had for DL models at the start of the project. The main objective of the DL Extension Library is to convert Keras and PyTorch deep learning models into OpenML flow. After this task is completed, MXNet will be the next framework in line for implementation. Once such a conversion into OpenML flow is applied to deep learning models all current operations that the platform supports on non-deep learning models, such as downloading models, uploading models, running models on tasks and comparing the results of different models on the same task, will work on deep learning models as well. This is possible because these functions are supported by default in the OpenML flow format.

Another main task of the DL Extension Library is to convert an ONNX format to an OpenML flow. This way, the number of supported deep learning frameworks is increased. Furthermore, the majority of these frameworks can be transformed into an ONNX format by using some other external libraries, outside the scope of OpenML and the DL Extension Library. Thus, a framework can be converted into ONNX and afterwards into an OpenML flow in order to be used on the OpenML platform. If time allows, additional deep learning model frameworks such as Caffe2 and other formats such as MLFlow specification will be supported by the DL Extension Library.

2.4 Environment

Conversion of a deep learning model to an OpenML flow and vice versa, as well as conversion using ONNX specification require at least Python 3.5 to be installed on the user machine. Furthermore, a Python IDE and the OpenML Python API are also needed. Uploading or downloading OpenML flows requires interaction with the OpenML Python API or with the website of OpenML through a web browser. Running OpenML flows on tasks and publishing results is done via the OpenML Python API. The dependencies vary, depending on the deep learning framework used for creating the models and the task performed. These are depicted on figure 2.1 and are defined below as follows:

- Conversion of Keras deep learning models requires installing Keras
- Conversion of PyTorch deep learning models requires installing PyTorch
- Conversion of MXNet deep learning models requires installing MXNet
- Using the full functionality of ONNX extension requires the installation of the MXNet library, used for running the specification, beside the ONNX library
- Visualization of a OpenML flow or run requires installing Dash

The types of users are distinguished in terms of their preferred deep learning framework. These include the following categories:

- Keras users - they install and import only the Keras extension from the DL Extension Library
- PyTorch users - they install and import only the PyTorch extension from the DL Extension Library
- Apache MXNet users - they install and import only the Apache MXNet extension from the DL Extension Library
- Users of ONNX - they install and import the ONNX extension of the DL Extension Library

Users can be further split into categories regarding the tasks that they want to perform with the given models:

- Users that want to run an OpenML flow of a deep learning model on a specific task
- Users that want to upload a run of a deep learning model
- Users that want to download an OpenML flow of deep learning model
- Users that want to visualize a given OpenML flow
- Users that want to visualize a given run

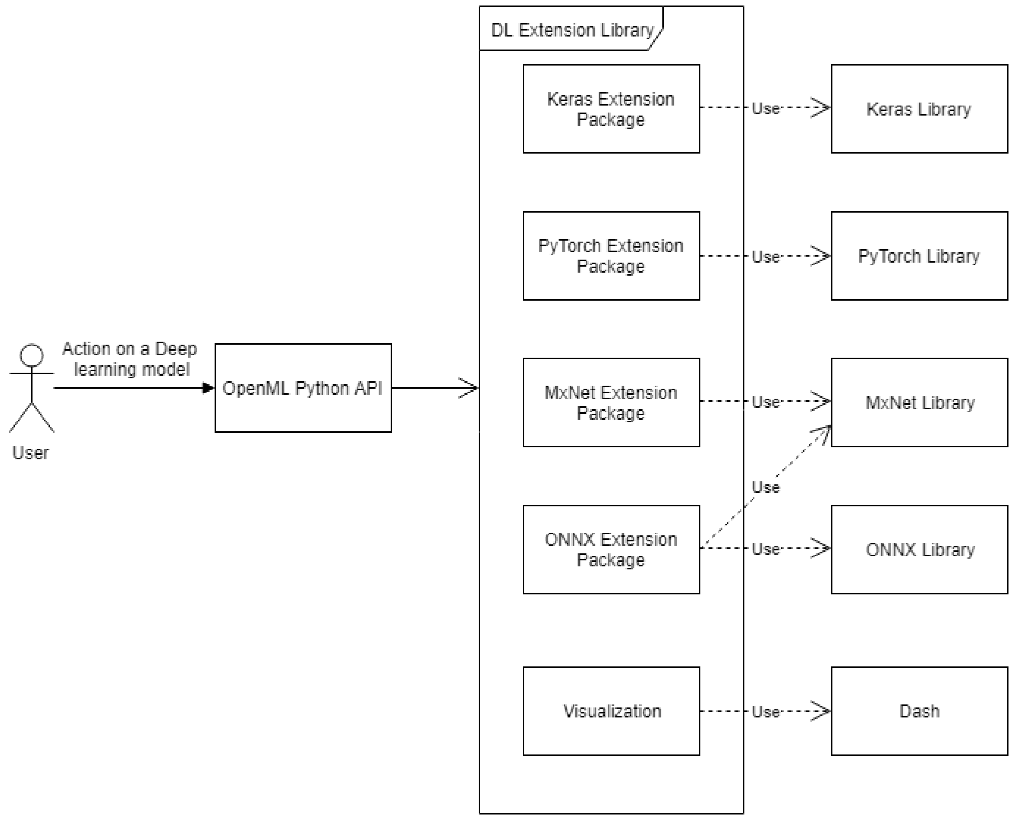


Figure 2.1: Diagram of the environment and the dependencies when working with deep learning models

2.5 Relation to Other Systems

Due to the nature of the DL Extension Library, to serve as a bridge between OpenML and deep learning models, this project has several related systems. First of which and most importantly is the OpenML Python API. It is an integral and mandatory part in order to use the DL Extension Library since all current operations on OpenML flow are done through the OpenML Python API . Other libraries are required depending on user preferences. For example, trying to convert a Keras deep learning model requires the Keras library. Following this rationale, other systems could be needed, such as: Keras, PyTorch, Apache MXNet and ONNX.

2.5.1 OpenML Python API

OpenML Python API is the Python API that has already been created and provided to our team. It allows users to download OpenML flows and datasets from the platform in order to edit or run them. It also allows the user to upload their OpenML flow and results of the run. Furthermore, this API enables the DL Extension Library to use additional operations such as running models on tasks and comparing the results of different models on the same task for the deep learning models.

2.5.2 Deep learning frameworks

The list of all deep learning libraries that DL Extension Library supports includes: Keras, PyTorch and MXNet. For the purpose of this project, these libraries all serve the same function, providing formats that DL Extension Library needs to convert into OpenML flow. While the conversion process can vary significantly from one format to another, for the purposes of this subsection, we will generalize all of them as deep learning libraries. It is important to emphasize that the user only needs to import in the code the packages (extensions) from DL Extension Library for the specific libraries they are using, as explained in 2.4.

2.5.3 Deep learning specifications

The specification that DL Extension Library implements is ONNX. This is the result of its ubiquity and the client's preferences. At the same time, it facilitates a possible intermediary step for transformations from one format to another.

2.6 General Constraints

The DL Extension Library that extends the functionality of OpenML needs to respect the following constraints which are discussed in the subsequent subsections.

2.6.1 Visualization

In order to visualize the structure of the deep learning models, the OpenML Python API has to utilize the Dash user interface library, as requested by the client. Hence, all layers (i.e. the hyperparameters) of the deep learning models will have to be present in the OpenML flow objects with which OpenML interacts. In this way, the user will be able to visualize OpenML flows created by the DL Extension Library since all the needed information will already be found in the OpenML flow. By using Dash, a local web server will be created on the local machine, which can be accessed by the user to see the structure of the layers of the deep learning models. Furthermore, the run will be visualized in terms of its loss and accuracy, if it was a classification task. If it was the run of a regression task, the loss, mean square error, mean absolute error and root mean square error graphs will be available for viewing. This is only available if the run has available training data to plot the graphs.

2.6.2 Data Reliability

Since the DL Extension Library converts a deep learning model into an OpenML flow and vice versa, the conversion should be done correctly i.e. lossless transformation. To achieve this goal, all the relevant data of a deep learning model should be stored in the OpenML flow as (hyper)parameters. The relevant data of a deep learning model consists of the layers (and other parameters which were used to create it). All this data does not need to be changed when converting, hence the user can always convert back from an OpenML flow to the initial deep learning model.

2.6.3 Environment constraints

The DL Extension Library extends OpenML Python API to support deep learning models. In this way, operations such as uploading, deleting and running flows on tasks, are supported on deep learning models as well, by using the already existing OpenML Python API. The OpenML platform supports only the OpenML flow format for machine learning models, so all interactions with the platform will need to be done taking this format into consideration. Since the deep learning models are created using deep learning libraries, such as Keras and PyTorch, the DL Extension Library has to support them.

2.6.4 Minimizing dependencies and storage space

To provide the users with the desired functionality, without having to download all supported deep learning libraries (i.e. Keras, PyTorch etc.), each library is implemented in a separate package in the DL Extension Library. Consequently, users are required to install and import only the package that provides support for their desired deep learning library. For example, if the user is interested in working with a Keras model, they will need to install and import only the package that provides conversion between Keras models and OpenML flows.

As deep learning models that the DL Extension Library supports can easily become very complex, it is important that there are constraints on the maximum size of these models. In the implementation for OpenML, there are existing constrains on the size of the models which can be published on the platform. Therefore, the DL Extension Library is limited to deep learning models of size at most 1GB and complexity of at most 30 layers.

2.7 Model Description

The following subsections depict the abstract structure of the DL Extension Library and of the deep learning libraries used. This is done by using Unified Modeling Language (UML) class diagrams. Furthermore, the interaction between the components is illustrated using UML abstract sequence diagrams and low-level activity diagrams.

2.7.1 Abstract class diagrams

The abstracted view of the DL Extension Library is depicted in figure 2.2. The DL Extension Library is composed of the Converter class, the Serializer and Deserializer classes as well as the Run class. This diagrams depicts the logical view of the project.

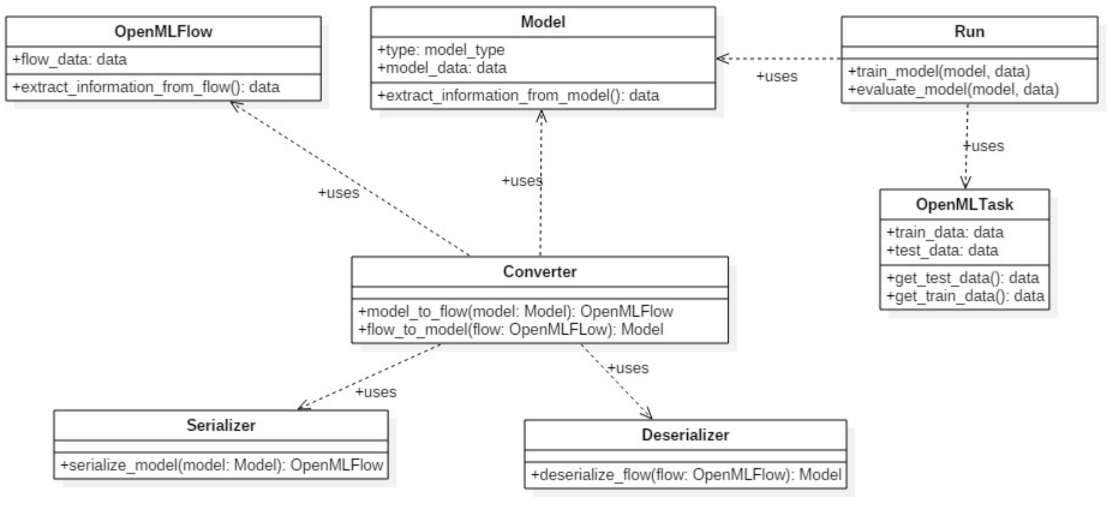

OpenMLFlow
+flow_data: data
+extract_information_from_flow(): data
Model
+type: model_type
+model_data: data
+extract_information_from_model(): data
Run
+train_model(model, data)
+evaluate_model(model, data)
OpenMLTask
+train_data: data
+test_data: data
+get_test_data(): data
+get_train_data(): data
Converter
+model_to_flow(model: Model): OpenMLFlow
+flow_to_model(flow: OpenMLFLow): Model
Serializer
+serialize_model(model: Model): OpenMLFlow
Deserializer
+deserialize_flow(flow: OpenMLFlow): Model
+uses

Figure 2.2: Abstract Class Diagram of the DL Extension Library

- *OpenML Flow:* Contains the information of an OpenML flow. The data can be extracted using the method *extract_information_from_flow*.
- *Model:* Contains the information of a Deep learning model. The data can be extracted using the method *extract_information_from_model*. The library used by the model is saved in the variable attribute *type*.
- *OpenML Task:* Contains the data necessary to run a model. The data consists of a training data and the test data stored respectively in the variables *train_data* and *test_data*. The data can be fetched using the methods *get_test_data* and *get_train_data*.
- *Run:* Provides functionality to run a given model on a given OpenML Task. This functionality can be invoked by calling the method *run_model* with taking as parameters the model and the OpenML task.
- *Converter:* Allows conversion between Models and OpenML flows. the conversion is done using the methods *model_to_flow* and *flow_to_model*.
- *Serializer:* Serializes a *Model* object. The conversion is done using the *serialize_model* function with the *Model* object passed in as a parameter. The function outputs an *OpenMLFlow* object.
- *Deserializer*: Deserializes an *OpenMLFlow* object. The conversion is done using the *deserialize_flow* function with the *OpenMLFlow* object passed in as a parameter. The function outputs a *Model* object.

2.7.2 Abstract Keras model diagrams

This subsection illustrates what components (i.e. variables, layers) can be found inside of a Keras model, and which of them we extract and store in the OpenML flow. Figure 2.3 depicts all the needed information by abstracting from the implementation details. This diagram only displays two layer for convenience purposes, as they are tens more.

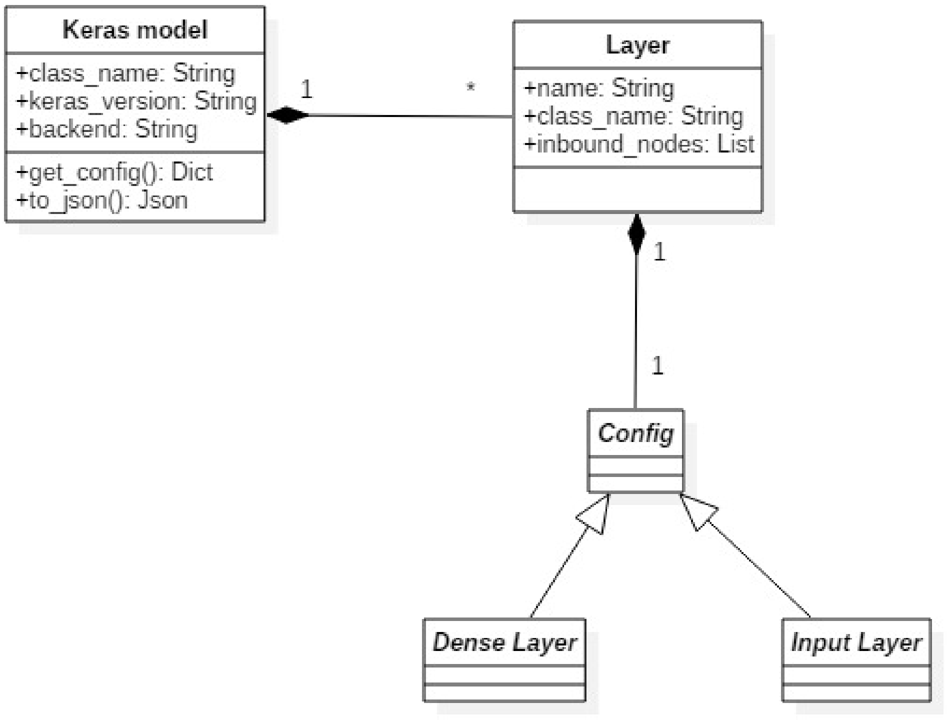


Figure 2.3: Abstract Keras Model

- *Keras model:* the deep learning model containing the needed variables and layers to be serialized/deserialized.
- *Layer:* the layers that the Keras model contains which need to be serialized/deserialized. Each layer contains a name, a class name, a list of incoming nodes and a configuration.
- *Config:* a dictionary which is specific for different types of layers, such as Dense layer, Input layer, etc, which are already available in the Keras library. Inside the dictionary, there are specific hyperparameters stored such as `name`, `trainable`, `momentum` etc.

Converting from a Keras model into an OpenML flow implies copying all the data that can be found inside the model and storing it into the variables of the OpenML flow. The data needed is class_name, keras_version, backend as well as the content of each layer (name, class_name, inbound_nodes and the config dictionary sepcific for each type of layer). The weights of the model are not stored in the OpenML flow in order to save storage space.

Converting from an OpenML flow into a Keras model implies copying all the data that can be found inside the OpenML flow and storing it into the variables of the Keras model.

2.7.3 Abstract PyTorch model diagrams

This subsection illustrates what components (i.e. variables, layers) can be found inside of a PyTorch model, and which of them we extract and store in the OpenML flow. Figure 2.4 depicts all the needed information by abstracting from the implementation details. This diagram only displays two layer for convenience purposes, as they are tens more.

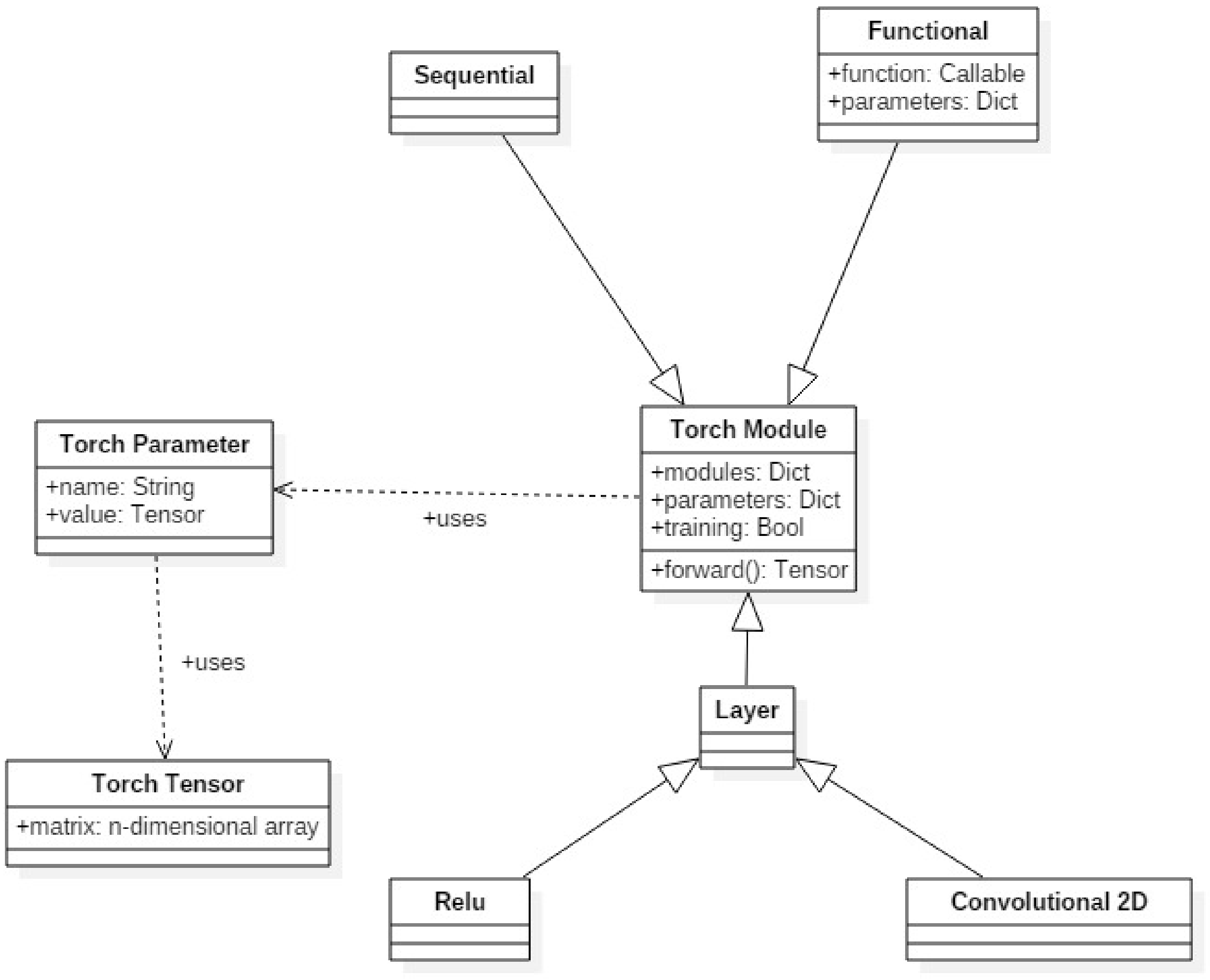


Figure 2.4: Abstract PyTorch Model

- *Torch module:* the general structure of a module in the PyTorch environment. It contains the needed variables, parameters and layers to be serialized/deserialized.
- *Torch parameter:* The parameters that a PyTorch model uses. Each parameter contains a Tensor component.
- *Torch tensor:* an n-dimensional array (i.e. a matrix) representing a geometric object that contains multiple vectors and scalars.
- *Sequential:* is a container that uses the forward function which it inherits from *Torch Module* to go layer by layer. In essence, it contains all the layers of the model.

- *Functional:* a layer that applies a given function (i.e. reshape) to the input Tensor with the provided parameters list. It is used to allow Tensor operations that generally would require a custom, non-serializable, layer.
- *Layer:* a component which contains different types of layers depending on the purpose of the PyTorch model. It includes types such as Relu, Convolutional 2D, etc, that are already available in the PyTorch library.

Converting from a PyTorch model into an OpenML flow implies copying all the data that can be found inside the model and storing it into the variables of the OpenML flow. All the data is the one represented in the diagram 2.4. The weights of the model are not stored in the OpenML flow in order to save storage space.

Converting from an OpenML flow into a PyTorch model implies copying all the data that can be found inside the OpenML flow and storing it into the variables of the PyTorch model.

2.7.4 Abstract ONNX specification diagrams

This subsection illustrates what components (i.e. parameters, layers) can be found inside of an ONNX specification, and which of them we extract and store in the OpenML flow. Figure 2.5 depicts all the needed information by abstracting from the implementation details.

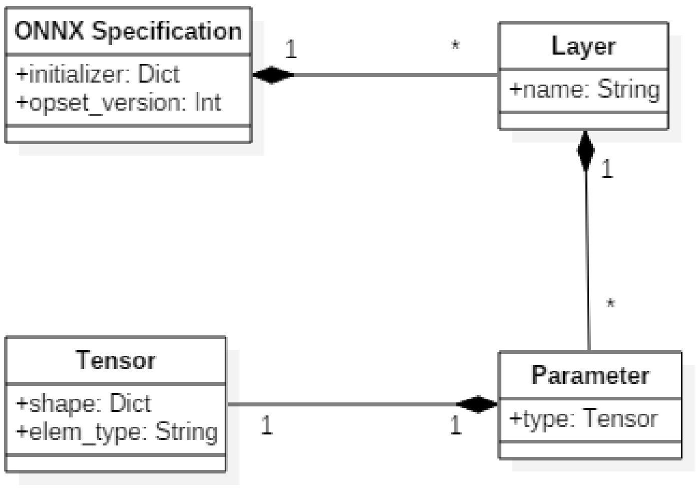


Figure 2.5: Abstract ONNX Model

- *ONNX Specification:* the specification of a deep learning model, containing the initializer and the version of ONNX Specification used.
- *Layer:* contains the type of layer used.
- *Parameter:* contains the type of tensor used.
- *Tensor:* an n-dimensional array, expressed in the shape dictionary with elements of type elem_type.

Converting from an ONNX specification into an OpenML flow implies copying all the data that can be found inside the specification and storing it into the variables of the OpenML flow.

Converting from a OpenML flow into an ONNX specification implies copying all the data that can be found inside the OpenML flow and storing it into the variables of the ONNX specification.

2.7.5 Abstract MXNet model diagrams

This subsection illustrates what components (i.e. parameters, layers) can be found inside of a MXNet model, and which of them we extract and store in the OpenML flow. Figure 2.6 depicts all the needed information by abstracting from the implementation details. This diagram only displays two layer for convenience purposes, as they are tens more.

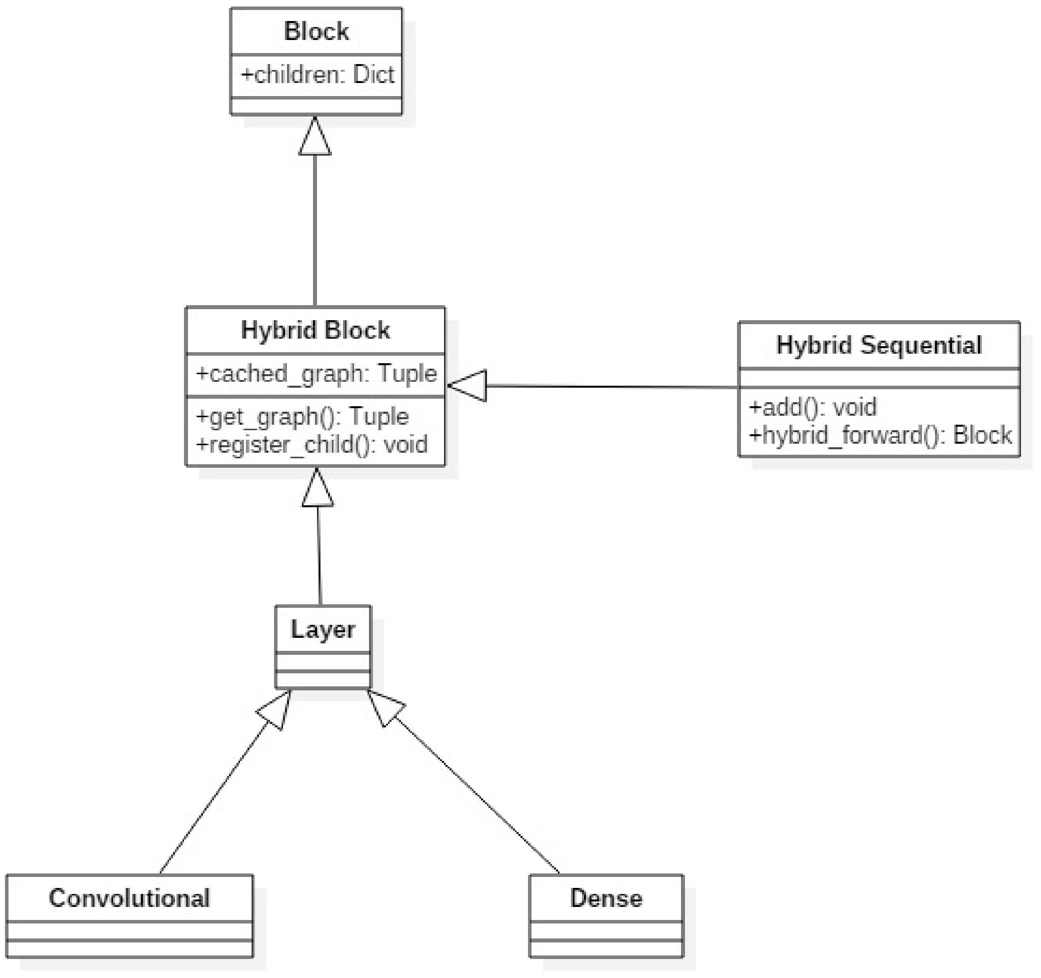


Figure 2.6: Abstract MXNet Model

- *Block:* is the most simple structure of data in an MXNet. It holds a dictionary containing all the subsequent blocks in the model.

- *Hybrid Block:* is a block that has the ability to store and retrieve a graph and add new elements to the children dictionary of the block.
- *Hybrid Sequential:* is the structure which contains a list of layers through which it goes one by one.
- *Layer:* a component which contains different types of layers depending on the purpose of the MXNet model. It includes types such as Convolutional, Dense etc, that are already available in the MXNet library.

Converting from an MXNet model into an OpenML flow implies copying all the data that can be found inside the specification and storing it into the variables of the OpenML flow.

Converting from a OpenML flow into an MXNet model implies copying all the data that can be found inside the OpenML flow and storing it into the variables of the MXNet model.

2.7.6 Class diagrams

The logical view of the DL Extension Library is depicted in figure 2.7. It depicts the interface of OpenML Python API that we need to implement.

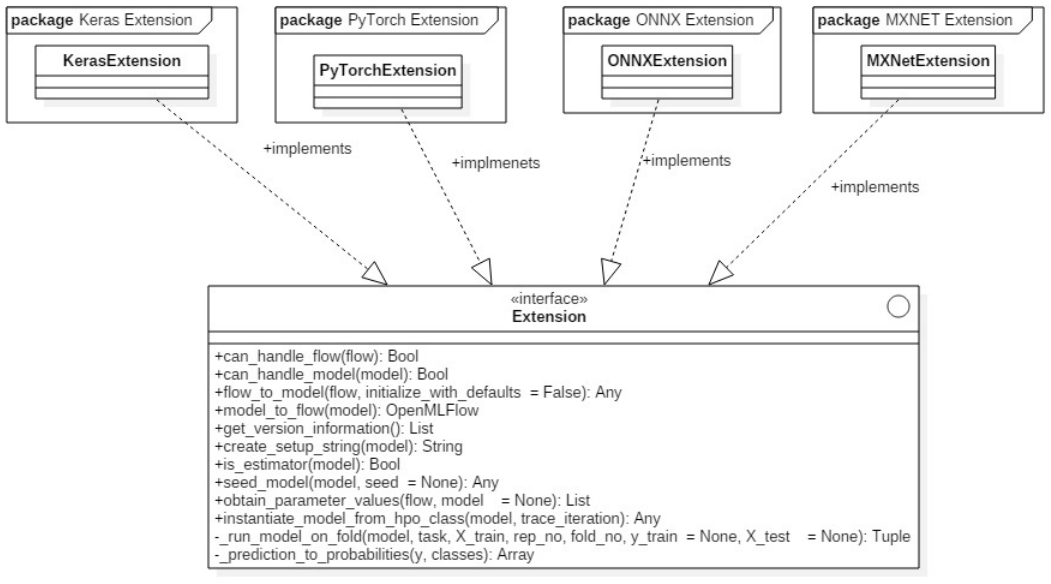

package Keras Extension
KerasExtension
package PyTorch Extension
PyTorchExtension
package ONNX Extension
ONNXExtension
package MXNET Extension
MXNetExtension
+implements
+implmenets
+implements
+implements
«interface»
Extension
+can_handle_flow(flow): Bool
+can_handle_model(model): Bool
+flow_to_model(flow, initialize_with_defaults = False): Any
+model_to_flow(model): OpenMLFlow
+get_version_information(): List
+create_setup_string(model): String
+is_estimator(model): Bool
+seed_model(model, seed = None): Any
+obtain_parameter_values(flow, model = None): List
+instantiate_model_from_hpo_class(model, trace_iteration): Any
-_run_model_on_fold(model, task, X_train, rep_no, fold_no, y_train = None, X_test = None): Tuple
-_prediction_to_probabilities(y, classes): Array

Figure 2.7: Top level class diagram

All the components in the class diagrams, together with their functions and corresponding attributes are described in the following subsections.

OpenML extension interface

The OpenML extension interface is an interface that allows to connect machine learning libraries to the OpenML Python API. The OpenML Support Squad has to use this extension interface when implementing all the other extension packages (Keras, PyTorch, ONNX, MXNet) since it is the only one supported by the OpenML Python API. This extension interface was not created by the OpenML Support Squad but was provided already by the OpenML Python API before the start of the project "Sharing deep learning models". The OpenML Support Squad is not responsible for the structure and design of this interface.

- *can_handle_flow*(*flow*: `OpenMLFlow`): `Boolean`
  Checks whether a given OpenML flow describes a neural network.
- *can_handle_model*(*model*: `Any`): `Boolean`
  Checks whether the OpenML extension interface can handle the framework of the given model.
- *flow_to_model*(*flow*: `OpenMLFlow`, *initialize_with_defaults* = `False`: `Boolean`): `Any`
  Initializes a model based on an OpenML flow.
- *model_to_flow*(*model*: `Any`): `OpenMLFlow`
  Transforms a model to an OpenML flow which can then be uploaded to OpenML.
- *get_version_information*(): `List`
  Lists versions of libraries required by the OpenML flow.
- *create_setup_string*(*model*: `Any`): `String`
  Creates a string from the model which can be used to re-instantiate the given model.
- *is_estimator*(*model*: `Any`): `Boolean`
  Checks whether the given model is a framework-specific neural network.
- *seed_model*(*model*: `Any`, *seed* = `None`: `int`): `Any`
  Set the seed of all the unseeded components of a model and return the seeded model. (Not applicable for Keras since Keras has no notion of RandomState)
- *obtain_parameter_values*(*flow*: `OpenMLFlow`, *model* = `None`: `Any`): `List[Dict[String, Any]]`
  Extracts all parameter settings required for the OpenML flow from the model.
- *instantiate_model_from_hpo_class*(*model*: `Any`, *trace_iteration*: `OpenMLTraceIteration`): `Any`
  Instantiates a base model which can be searched over by the hyperparameter optimization model.
- *_run_model_on_fold*(*model*: `Any`, *task*: `OpenMLTask`, *X_train*: `Union[np.ndarray, scipy.sparse.spmatrix, pd.DataFrame]`, *rep_no*: `int`, *fold_no*: `int`, *y_train* = `None`: `Array`, *X_test* = `None`: `Union[np.ndarray, scipy.sparse.spmatrix, pd.DataFrame]`): `Tuple[np.ndarray, np.ndarray, OrderedDict[String, float], Optional[OpenMLTrace]]`
  Runs a model on a task and return prediction information. Returns the data that is necessary to construct the OpenML run object.

Keras extension class

The Keras extension class extends the OpenML extension interface as depicted in figure 2.8, hence it inherits all functionality from the interface and extends it further with the functions described below. The Keras extension class provides the necessary functionality to convert Keras models to an OpenML flow and vice versa.

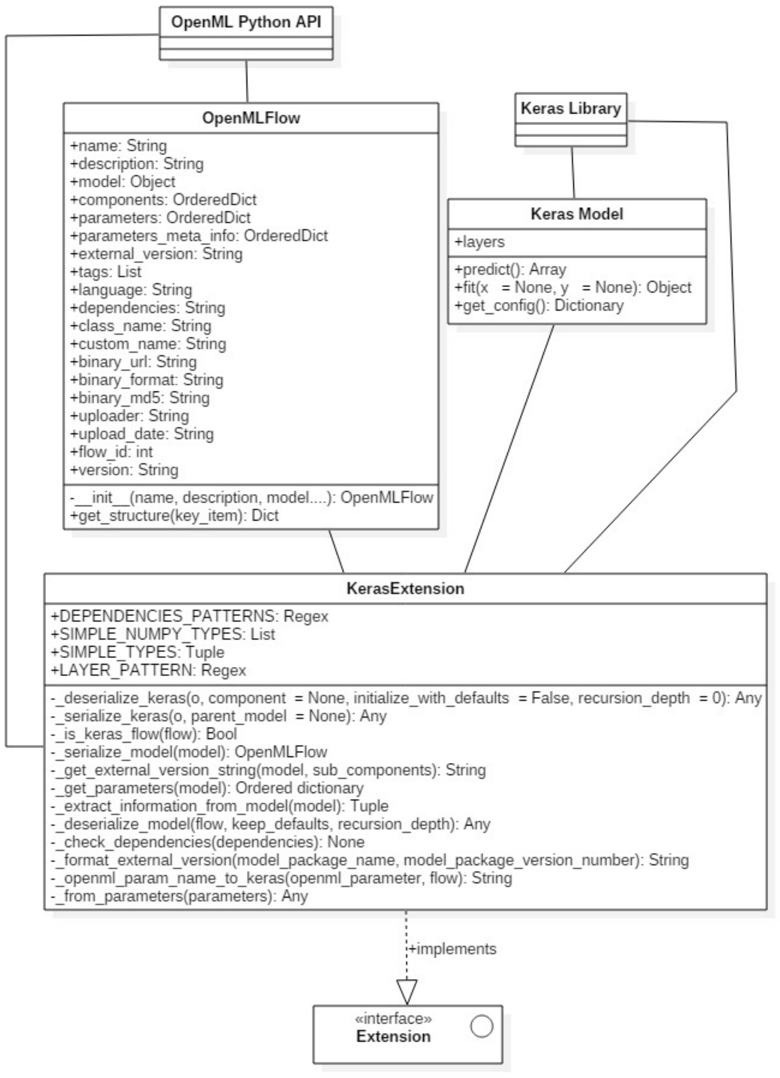

OpenML Python API
OpenMLFlow
+name: String
+description: String
+model: Object
+components: OrderedDict
+parameters: OrderedDict
+parameters_meta_info: OrderedDict
+external_version: String
+tags: List
+language: String
+dependencies: String
+class_name: String
+custom_name: String
+binary_url: String
+binary_format: String
+binary_md5: String
+uploader: String
+upload_date: String
+flow_id: int
+version: String
-__init__(name, description, model....): OpenMLFlow
+get_structure(key_item): Dict
Keras Library
Keras Model
+layers
+predict(): Array
+fit(x = None, y = None): Object
+get_config(): Dictionary
KerasExtension
+DEPENDENCIES_PATTERNS: Regex
+SIMPLE_NUMPY_TYPES: List
+SIMPLE_TYPES: Tuple
+LAYER_PATTERN: Regex
-_deserialize_keras(o, component = None, initialize_with_defaults = False, recursion_depth = 0): Any
-_serialize_keras(o, parent_model = None): Any
-_is_keras_flow(flow): Bool
-_serialize_model(model): OpenMLFlow
-_get_external_version_string(model, sub_components): String
-_get_parameters(model): Ordered dictionary
-_extract_information_from_model(model): Tuple
-_deserialize_model(flow, keep_defaults, recursion_depth): Any
-_check_dependencies(dependencies): None
-_format_external_version(model_package_name, model_package_version_number): String
-_openml_param_name_to_keras(openml_parameter, flow): String
-_from_parameters(parameters): Any
+implements
«interface»
Extension

Figure 2.8: Class KerasExtension diagram

Some of the methods in the class diagram will not be present in the sequence diagrams. The reason being that some of these functions are not related to the requirements specified in the URD [1]. They are only present in the Keras extension class since they must be implemented, because the Keras extension class implements the OpenML extension interface.

- *DEPENDENCIES_PATTERNS*: `Regex`
  Regex pattern of the required dependencies.

- *SIMPLE_NUMPY_TYPES*: `List[Type]`
  List that holds numpy types except types in the category "others".

- *SIMPLE_TYPES*: `List[Type]`
  List that holds *SIMPLE_NUMPY_TYPES* and simple types.

- *LAYER_PATTERNS*: `Regex`
  Regex pattern that describes how a Keras layer is formatted.
- *_deserialize_keras*(*o*: `Any`, *components* = `None`: `Dict`, *initialize_with_defaults* = `False`: `Boolean`, *recursion_depth* = `0`: `int`): `Any`
  Recursive function that serializes a Keras model into an OpenML flow.
- *_serialize_keras*(*o*: `Any`, *parent_model* = `None`: `Any`): `Any`
  Recursive function that deserializes an OpenML flow into a Keras model.
- *_is_keras_flow*(*flow*: `OpenMLFlow`): `Boolean`
  Checks if the OpenML flow is a Keras flow.
- *_serialize_model*(*model*: `Any`): `OpenMLFlow`
  Serializes Keras model to OpenML flow
- *_get_external_version_string*(*model*: `Any`, *sub_components*: `Dict`): `String`
  Gets the dependencies and their external versions required to produce the OpenML flow from a Keras model.
- *_get_parameters*(*model*: `Any`): `OrderedDict`
  Constructs a dictionary from the parameters of the Keras model.
- *_extract_information_from_model*(*model*: `Any`): `Tuple`
  Extracts parameters and sub-components from the Keras model.
- *_deserialize_model*(*flow*: `OpenMLFlow`, *keep_defaults*: `Boolean`, *recursion_depth*: `int`): `Any`
  Deserializes an OpenML flow into a Keras model.
- *_check_dependencies*(*dependencies*: `String`): `None`
  Checks whether the dependencies required for the deserialization of an OpenML flow are met.
- *_format_external_version*(*model_package_name*: `String`, *model_package_version_number*: `String`): `String`
  Returns a formatted string representing the required dependencies for an OpenML flow.
- *_openml_param_name_to_keras*(*openml_paramter*: `OpenMLParamter`, *flow*: `OpenMLFlow`): `String`
  Converts the name of an OpenMLParameter into the Keras name, given an OpenML flow.
- *_from_parameters*(*parameters*: `OrderedDict`): `Any`
  Get a Keras model from OpenML flow parameters.

PyTorch extension class

The PyTorch extension class extends OpenML extension interface as depicted in figure 2.9. It inherits all its functionality from the interface and extends it further with the functions described below. The PyTorch extension class provides the necessary functionality to convert PyTorch models to OpenML flow and vice versa. Furthermore, the PyTorch extension class relies on the Configuration package for PyTorch. The purpose of the Configuration package is to create global variables for later use. It also sets their default values or default functions depending on the type of the specific variable. The user still has the possibility to change these default values and function if they desire. This will be presented in more details below in the variable and function descriptions.

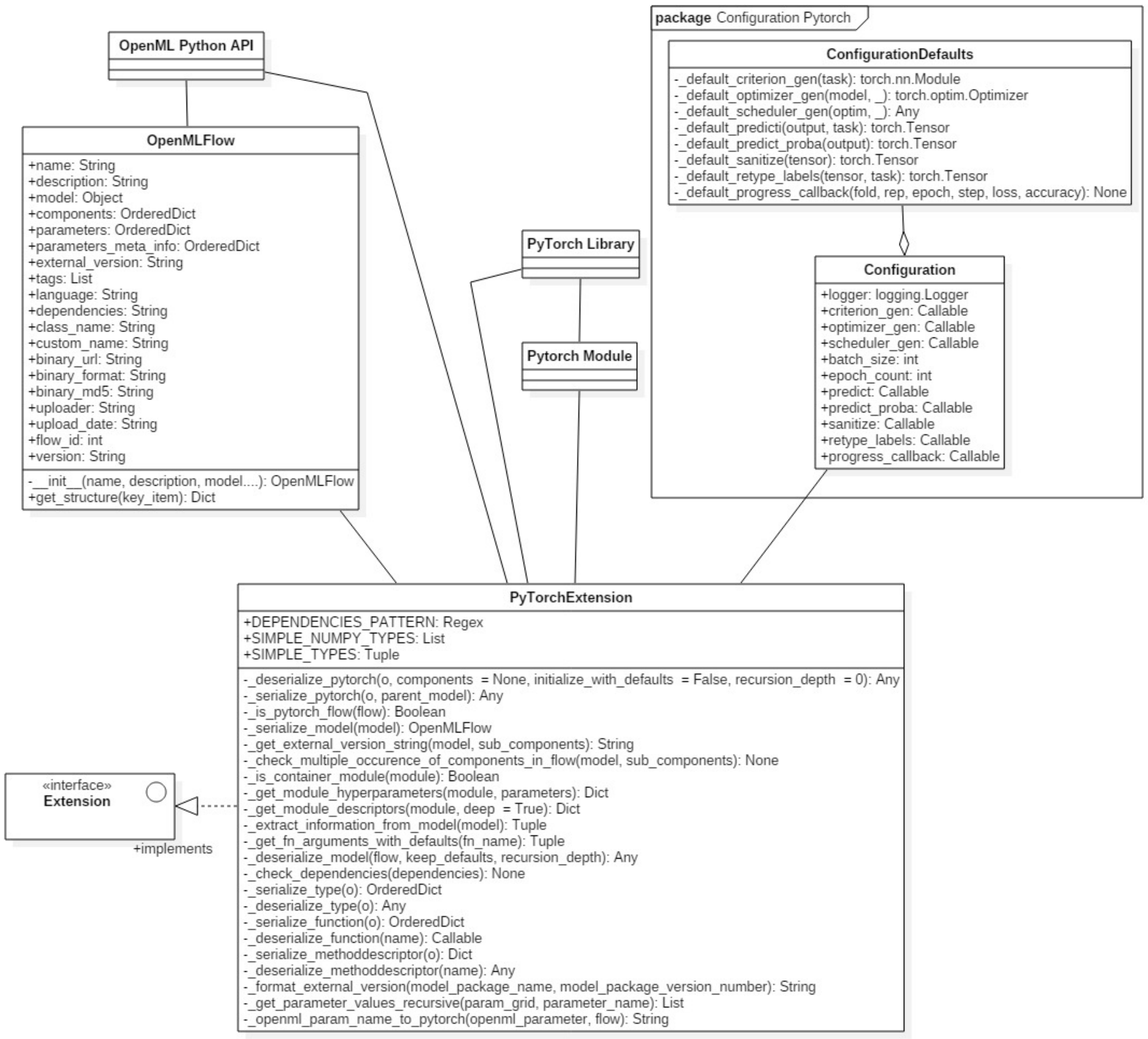

package Configuration Pytorch
ConfigurationDefaults
-_default_criterion_gen(task): torch.nn.Module
-_default_optimizer_gen(model, _): torch.optim.Optimizer
-_default_scheduler_gen(optim, _): Any
-_default_predicti(output, task): torch.Tensor
-_default_predict_proba(output): torch.Tensor
-_default_sanitize(tensor): torch.Tensor
-_default_retype_labels(tensor, task): torch.Tensor
-_default_progress_callback(fold, rep, epoch, step, loss, accuracy): None
Configuration
+logger: logging.Logger
+criterion_gen: Callable
+optimizer_gen: Callable
+scheduler_gen: Callable
+batch_size: int
+epoch_count: int
+predict: Callable
+predict_proba: Callable
+sanitize: Callable
+retype_labels: Callable
+progress_callback: Callable
OpenML Python API
OpenMLFlow
+name: String
+description: String
+model: Object
+components: OrderedDict
+parameters: OrderedDict
+parameters_meta_info: OrderedDict
+external_version: String
+tags: List
+language: String
+dependencies: String
+class_name: String
+custom_name: String
+binary_url: String
+binary_format: String
+binary_md5: String
+uploader: String
+upload_date: String
+flow_id: int
+version: String
-__init__(name, description, model....): OpenMLFlow
+get_structure(key_item): Dict
PyTorch Library
Pytorch Module
PyTorchExtension
+DEPENDENCIES_PATTERN: Regex
+SIMPLE_NUMPY_TYPES: List
+SIMPLE_TYPES: Tuple
-_deserialize_pytorch(o, components = None, initialize_with_defaults = False, recursion_depth = 0): Any
-_serialize_pytorch(o, parent_model): Any
-_is_pytorch_flow(flow): Boolean
-_serialize_model(model): OpenMLFlow
-_get_external_version_string(model, sub_components): String
-_check_multiple_occurence_of_components_in_flow(model, sub_components): None
-_is_container_module(module): Boolean
-_get_module_hyperparameters(module, parameters): Dict
-_get_module_descriptors(module, deep = True): Dict
-_extract_information_from_model(model): Tuple
-_get_fn_arguments_with_defaults(fn_name): Tuple
-_deserialize_model(flow, keep_defaults, recursion_depth): Any
-_check_dependencies(dependencies): None
-_serialize_type(o): OrderedDict
-_deserialize_type(o): Any
-_serialize_function(o): OrderedDict
-_deserialize_function(name): Callable
-_serialize_methoddescriptor(o): Dict
-_deserialize_methoddescriptor(name): Any
-_format_external_version(model_package_name, model_package_version_number): String
-_get_parameter_values_recursive(param_grid, parameter_name): List
-_openml_param_name_to_pytorch(openml_parameter, flow): String
«interface»
Extension
+implements

Figure 2.9: Class PytorchExtension diagram

Some of the methods in the class diagram will not be present in the sequence diagrams. The reason being that some of these functions are not related to the requirements specified in [1]. They are present in the PyTorch extension class since they must be implemented, due the fact that PyTorch extension class implements the OpenML extension interface.

- *_deserialize_pytorch*(*o*: `Any`, *components* = `None`: `Dict`, *initialize_with_defaults* = `False`: `Boolean`, *recursion_depth* = `0`: `int`): `Any`

Recursive function to deserialize an OpenML flow into a PyTorch model.

- *_serialize_pytorch*(*o*: `Any`, *parent_model* = `None`: `Any`): `Any`
  Recursive function to serialize a PyTorch model into a OpenML flow.

- *_is_pytorch_flow*(*flow*: `OpenMLFlow`): `Boolean`
  Checks whether the *flow* has been a PyTorch model originally

- *_serialize_model*(*model*: `Any`): `OpenMLFlow`
  Create an OpenMLFlow object from a PyTorch model.

- *_get_external_version_string*(*model*: `Any`, *sub_components*: `Dict[String, OpenMLFlow]`): `String`
  Gets the dependencies and their external versions required to produce the OpenML flow from a PyTorch model.

- *_check_multiple_occurence_of_components_in_flow*(*model*: `Any`, *sub_components*: `Dict[String, OpenMLFlow]`): `None`
  Checks if there is more than one occurrence of a component in a flow.

- *_is_container_module*(*module*: `torch.nn.Module`): `Boolean`
  Checks if a *module* is a Sequential container, ModuleDict container or a ModuleDict container.

- *_get_module_hyperparameters*(*module*: `torch.nn.Module`, *parameters*: `Dict[String, torch.nn.Paramete`
  `Dict[String, Any]`
  Finds the hyperparameters of a module.

- *_get_module_descriptors*(*module*: `torch.nn.Module`, *deep* = `True`: `Boolean`): `Dict[String, Any]`
  Finds the descriptors of a module.

- *_extract_information_from_model*(*model*: `Any`): `Tuple[OrderedDict[String, Optional[String]], OrderedDict[String, Optional[Dict], OrderedDict[String, OpenMLFlow]]`
  Extracts all necessary information (parameters and sub-components) for the OpenMLFlow object from the model.

- *_get_fn_arguments_with_defaults*(*fn_name*: `Callable`): `Tuple[Dict, Set]`
  Obtains all the parameter names of a function together with their default values.

- *_deserialize_model*(*flow*: `OpenMLFlow`, *keep_defaults*: `Boolean`, *recursion_depth*: `int`): `Any`
  Deserializes an OpenML flow into a PyTorch model.

- *_check_dependencies*(*dependencies*: `String`): `None`
  Checks whether the dependencies required for the deserialization of an OpenMLFlow object are met.

- *_serialize_type*(*o*: `Any`): `OrderedDict[String, String]`
  Converts a given type into a string representation of it.

- *_deserialize_type*(*o*: `String`): `Any`
  Converts a string representation of a type into an instance of that type.

- *_serialize_function*(*o*: `Callable`): `OrderedDict[String, String]`
  Converts a given function *o* into a string representation of that function.
- *_deserialize_function*(*name*: `String`): `Callable`
  Converts a given string *name* of a function into the function.
- *_serialize_methoddescriptor*(*o*: `Any`): `OrderedDict[String, String]`
  Creates a string representation of a method descriptor.
- *_deserialize_methoddescriptor*(*name*: `String`): `OrderedDict[String, String]`
  Turns a string representation of a method descriptor to a valid method descriptor.
- *_format_external_version*(*model_package_name*: `String`, *model_package_version_number*: `String`): `String`
  Returns a formatted string representing the necessary dependencies for a flow.
- *_get_parameter_values_recursive*(*param_grid*: `Union[Dict, List[dict]]`, *parameter_name*: `String`): `List[Any]`
  Creates a list of values for a given hyperparameter, encountered recursively throughout the flow. (e.g., n_jobs can be defined for various flows).
- *_openml_param_name_to_pytorch*(*openml_parameter*: `openml.setups.OpenMLParameter`, *flow*: `OpenMLFlow`): `String`
  Converts the name of an OpenMLParameter into the pytorch name, given a flow.

Below are the descriptions of the variables and functions in the Configuration PyTorch package from the diagram represented in Figure 2.9. The main use of the package is to export the variables represented in the Configuration class as global ones and give them default values. The functions represented in the ConfigurationDefault class are the default values for all variables in the package of type `Callable`. The rest of the values are simple types. All default values for each variable are represented below.

- *logger*: `logging.Logger`
  Default logger for the PyTorch extension
- *_default_criterion_gen*(*task*: `OpenMLTask`): `torch.nn.Module`
  Default implementation, returns the criterion based on the task type.
- *criterion_gen*: `Callable`
  Stores the result from *_default_criterion_gen*. Can be overridden by the user if they wish, but the new function should have an `OpenMLTask` parameter and return a `torch.nn.Module`.
- *_default_optimizer_gen*(*model*: `torch.nn.Module`, *_*: `OpenMLTask`): `torch.optim.Optimizer`
  Default implementation, returns an optimizer for the given model. The "_" variable is there in order to respect the function signature, but is ignored post that in the default implementation. The reasoning for which it was added is in case some user wants to create their own optimizer generator, they can take into account the type of task.
- *optimizer_gen*: `Callable`
  Stores the result from *_default_optimizer_gen*. Can be overridden by the user if they wish, but the new function should have a `torch.nn.Module` and a `OpenMLTask` parameter and return a `torch.optim.Optimizer`.

- *_default_scheduler_gen*(*model*: `torch.nn.Module`, *_*: `OpenMLTask`): `Any`
  Default implementation, returns a scheduler for the given optimizer. The "_" variable is there in order to respect the function signature, but is ignored in the default implementation. The reasoning for which it was added is in case some user wants to create their own scheduler generator, they can take into account the type of task.

- *scheduler_gen*: `Callable`
  Stores the result from *_default_scheduler_gen*. Can be overridden by the user if they wish, but the new function should have a `torch.nn.Module` and a `OpenMLTask` parameter and return a `Any`.

- *batch_size*: `int`
  Represents the processing batch size for training. Default value is 64.

- *epoch_count*: `int`
  Represents he number of epochs the model should be trained for. Default value is 32.

- *_default_predict*(*output*: `torch.Tensor`, *task*: `OpenMLTask`): `torch.Tensor`
  Default implementation, returns predictions given the outputs of the model.

- *predict*: `Callable`
  Stores the result from *_default_predict*. Can be overridden by the user if they wish, but the new function should have a `torch.Tensor` and a `OpenMLTask` as parameters and return a `torch.nn.Tensor`.

- *_default_predict_proba*(*output*: `torch.Tensor`): `torch.Tensor`
  Default implementation, returns predictions given the outputs of the model into probabilities for each class.

- *predict_proba*: `Callable`
  Stores the result from *_default_predict_proba*. Can be overridden by the user if they wish, but the new function should have a `torch.Tensor` parameter and return a `torch.Tensor`.

- *_default_sanitize*(*tensor*: `torch.Tensor`): `torch.Tensor`
  Default implementation, sanitizes the input data in order to ensure that models can be trained safely.

- *sanitize*: `Callable`
  Stores the result from *_default_sanitize*. Can be overridden by the user if they wish, but the new function should have a `torch.Tensor` parameter and return a `torch.Tensor`.

- *_default_retype_labels*(*tensor*: `torch.Tensor`, *task*: `OpenMLTask`): `torch.Tensor`
  Default implementation, changes the types of the labels in order to ensure type compatibility.

- *retype_labels*: `Callable`
  Stores the result from *_default_retype_labels*. Can be overridden by the user if they wish, but the new function should have a `torch.Tensor` and a `OpenMLTask` as parameters and return a `torch.Tensor`.

- *_default_progress_callback*(*fold*: `int`, *rep*: `int`, *epoch*: `int`, *step*: `int`, *loss*: `float`, *accuracy*: `float`): `None`
  Default implementation, reports the current progress when a training step is finished.
- *progress_callback*: `Callable`
  Stores the result from *_default_progress_callback*. Can be overridden by the user if they wish, but the new function should have three `int` parameters followed by two `float` parameters and return a `None`.

ONNX extension class

The ONNX extension class extends the OpenML extension interface as depicted in figure 2.10. It inherits all functionality from the interface and extends it further with the functions described below. The ONNX extension class provides the necessary functionality to convert ONNX specification to an OpenML flow and vice versa. Furthermore, the ONNX extension class relies on the Configuration package for ONNX. The purpose of the Configuration package is to create global variables for later use. It also sets their default values or default functions depending on the type of the specific variable. The user still has the possibility to change these default values and function if they desire. This will be presented in more details below in the variable and function descriptions

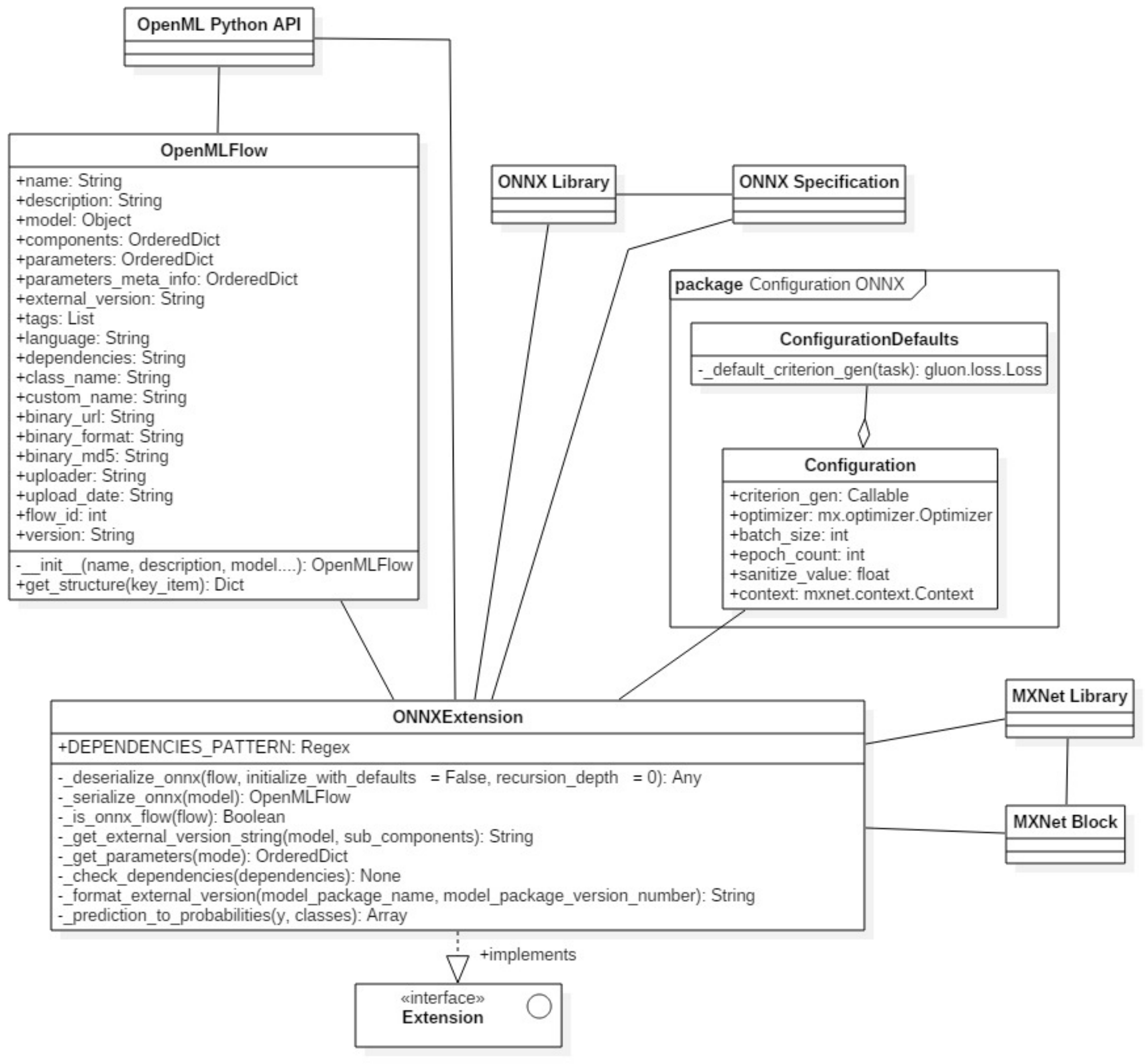

OpenML Python API
OpenMLFlow
+name: String
+description: String
+model: Object
+components: OrderedDict
+parameters: OrderedDict
+parameters_meta_info: OrderedDict
+external_version: String
+tags: List
+language: String
+dependencies: String
+class_name: String
+custom_name: String
+binary_url: String
+binary_format: String
+binary_md5: String
+uploader: String
+upload_date: String
+flow_id: int
+version: String
-__init__(name, description, model....): OpenMLFlow
+get_structure(key_item): Dict
ONNX Library
ONNX Specification
package Configuration ONNX
ConfigurationDefaults
-_default_criterion_gen(task): gluon.loss.Loss
Configuration
+criterion_gen: Callable
+optimizer: mx.optimizer.Optimizer
+batch_size: int
+epoch_count: int
+sanitize_value: float
+context: mxnet.context.Context
MXNet Library
MXNet Block
ONNXExtension
+DEPENDENCIES_PATTERN: Regex
-_deserialize_onnx(flow, initialize_with_defaults = False, recursion_depth = 0): Any
-_serialize_onnx(model): OpenMLFlow
-_is_onnx_flow(flow): Boolean
-_get_external_version_string(model, sub_components): String
-_get_parameters(mode): OrderedDict
-_check_dependencies(dependencies): None
-_format_external_version(model_package_name, model_package_version_number): String
-_prediction_to_probabilities(y, classes): Array
+implements
«interface»
Extension

Figure 2.10: Class ONNXExtension diagram

Some of the methods in the class diagram will not be present in the sequence diagrams. The reason being that some of these functions are not related to the requirements specified in the URD [1]. They are present in the ONNX extension class since they must be implemented, due the fact that ONNX extension class implements the OpenML extension interface.

- *DEPENDENCIES_PATTERNS*: `Regex`
  Regex pattern of the required dependencies.

- *ONNX_FILE_PATH*: `String`
  Contains the path to the ONNX labrary.

- *ONNX_ATTR_TYPES*: `Dictionary`
  A dictionary containing the ONNX attribute tasks.

- *_deserialize_onnx*(*flow*: `OpenMLFlow`, *initialize_with_defaults* = `False`: `Boolean`, *recursion_depth* = `0`: `int`): `Any`
  Deserializes an OpenMLFlow object into an ONNX model.
- *_serialize_onnx*(*model*: `Any`): `OpenMLFlow`
  Serializes the ONNX model to a an OpenML flow.
- *_is_onnx_flow*(*flow*: `OpenMLFlow`): `Boolean`
  Checks if the OpenML flow is a ONNX flow.
- *_get_external_version_string*(*model*: `Any`, *sub_components*: `Dict`): `String`
  Gets the dependencies and their external versions required to produce the OpenML flow from an ONNX model.
- *_get_parameters*(*model*: `Any`): `OrderedDict`
  Constructs a dictionary from the parameters of the ONNX model.
- *_check_dependencies*(*dependencies*: `String`): `None`
  Checks whether the dependencies required for the deserialization of an OpenML flow are met.
- *_format_external_version*(*model_package_name*: `String`, *model_package_version_number*: `String`): `String`
  Returns a formatted string representing the required dependencies for an OpenML flow.
- *_to_ordered*(*o*: `Any`): `Any`
  Recursively orders all the dictionaries (included nested dictionaries) in *o* by keys.

Below are the descriptions of the variables and functions in the Configuration ONNX package from the diagram represented in Figure 2.10. The main use of the package is to export the variables represented in the Configuration class as global ones and give them default values. The function represented in the ConfigurationDefault class is the default value the variable in the package of type `Callable`. The rest of the values are simple types or specific MXNet classes. All default values for each variable are represented below.

- *_default_criterion_gen*(*task*: `OpenMLTask`): `gluon.loss.Loss`
  Default implementation, returns the criterion based on the task type.
- *criterion_gen*: `Callable`
  Stores the result from *_default_criterion_gen*. Can be overridden by the user if they wish, but the new function should have an `OpenMLTask` parameter and return a `gluon.loss.Loss`.
- *optimizer*: `mx.optimizer.Optimizer`
  Is an instance of MXNet's Optimizer class. represents the optimizer to be used during training of the model
- *batch_size*: `int`
  Represents the processing batch size for training. Default value is 64.
- *epoch_count*: `int`
  Represents he number of epochs the model should be trained for. Default value is 32

- *sanitize_value*: `Float`
  Represents the number that will be used to replace NaNs in train and test data.

- *context*: `mx.cpu`
  Is an instance of MXNet's Context class (`mx.context.Context`). Represents the context of the MXNet model - by default it will use the CPU.

MXNet extension class

The MXNet extension class extends OpenML extension interface as depicted in figure 2.11. It inherits all its functionality from the interface and extends it further with the functions described below. The MXNet extension class provides the necessary functionality to convert MXNet models to OpenML flow and vice versa. Furthermore, the MXNet extension class relies on the Configuration package for MXNet. The purpose of the Configuration package is to create global variables for later use. It also sets their default values or default functions depending on the type of the specific variable. The user still has the possibility to change these default values and function if they desire. This will be presented in more details below in the variable and function descriptions.

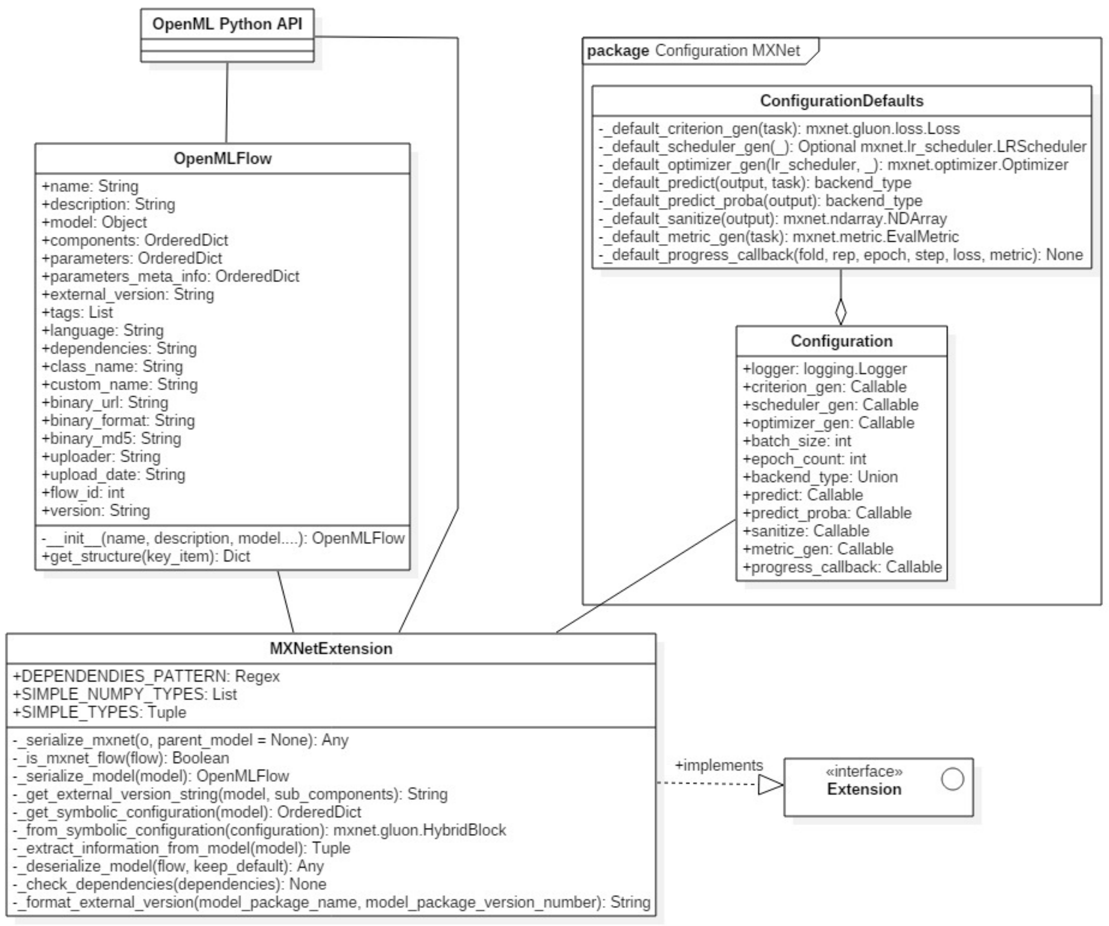

OpenML Python API
OpenMLFlow
+name: String
+description: String
+model: Object
+components: OrderedDict
+parameters: OrderedDict
+parameters_meta_info: OrderedDict
+external_version: String
+tags: List
+language: String
+dependencies: String
+class_name: String
+custom_name: String
+binary_url: String
+binary_format: String
+binary_md5: String
+uploader: String
+upload_date: String
+flow_id: int
+version: String
-__init__(name, description, model....): OpenMLFlow
+get_structure(key_item): Dict
package Configuration MXNet
ConfigurationDefaults
-_default_criterion_gen(task): mxnet.gluon.loss.Loss
-_default_scheduler_gen(_): Optional mxnet.lr_scheduler.LRScheduler
-_default_optimizer_gen(lr_scheduler, _): mxnet.optimizer.Optimizer
-_default_predict(output, task): backend_type
-_default_predict_proba(output): backend_type
-_default_sanitize(output): mxnet.ndarray.NDArray
-_default_metric_gen(task): mxnet.metric.EvalMetric
-_default_progress_callback(fold, rep, epoch, step, loss, metric): None
Configuration
+logger: logging.Logger
+criterion_gen: Callable
+scheduler_gen: Callable
+optimizer_gen: Callable
+batch_size: int
+epoch_count: int
+backend_type: Union
+predict: Callable
+predict_proba: Callable
+sanitize: Callable
+metric_gen: Callable
+progress_callback: Callable
MXNetExtension
+DEPENDENDIES_PATTERN: Regex
+SIMPLE_NUMPY_TYPES: List
+SIMPLE_TYPES: Tuple
-_serialize_mxnet(o, parent_model = None): Any
-_is_mxnet_flow(flow): Boolean
-_serialize_model(model): OpenMLFlow
-_get_external_version_string(model, sub_components): String
-_get_symbolic_configuration(model): OrderedDict
-_from_symbolic_configuration(configuration): mxnet.gluon.HybridBlock
-_extract_information_from_model(model): Tuple
-_deserialize_model(flow, keep_default): Any
-_check_dependencies(dependencies): None
-_format_external_version(model_package_name, model_package_version_number): String
+implements
«interface»
Extension

Figure 2.11: Class MXNetExtension diagram

Some of the methods in the class diagram will not be present in the sequence diagrams. The reason being that some of these functions are not related to the requirements specified in [1]. They are present in the PyTorch extension class since they must be implemented, due the fact that PyTorch extension class implements the OpenML extension interface.

- *DEPENDENCIES_PATTERNS*: `Regex`
  Regex pattern of the required dependencies.
- *SIMPLE_NUMPY_TYPES*: `List[Type]`
  List that holds numpy types except types in the category "others".
- *SIMPLE_TYPES*: `List[Type]`
  List that holds *SIMPLE_NUMPY_TYPES* and simple types.
- *_serialize_mxnet*(*o*: `Any`, *parent_model* = `None`: `Any`): `Any`
  Recursive function to serialize a MXNet model object.
- *_is_mxnet_flow*(*flow*: `OpenMLFlow`): `Boolean`
  Checks whether the *flow* has been a MXNet model originally
- *_serialize_model*(*model*: `Any`): `OpenMLFlow`
  Create an OpenMLFlow object from a MXNet model.
- *_get_external_version_string*(*model*: `Any`, *sub_components*: `Dict[String, OpenMLFlow]`): `String`
  Gets the dependencies and their external versions required to produce the OpenML flow from a MXNet model.
- *_get_symbolic_configuration*(*model*: `mxnet.gluon.HybridBlock`): `OrderedDicti[String, Any]`
  Extracts the nodes of an MXNet model and their order and creates a configuration of the model in the form of an ordered dictionary.
- *_from_symbolic_configuration*(*configuration*: `Dict[String, Any]`): `mxnet.gluon.HybridBlock`
  Retrieves the MXNet model from its the configuration.
- *_extract_information_from_model*(*model*: `Any`): `Tuple[OrderedDict[String, Optional[String]], OrderedDict[String, Optional[Dict], OrderedDict[String, OpenMLFlow]]`
  Extracts all necessary information (parameters and sub-components) for the OpenMLFlow object from the model.
- *_deserialize_model*(*flow*: `OpenMLFlow`, *keep_defaults*: `Boolean`): `Any`
  Deserializes an OpenML flow into a MXNet model.
- *_check_dependencies*(*dependencies*: `String`): `None`
  Checks whether the dependencies required for the deserialization of an OpenMLFlow object are met.
- *_format_external_version*(*model_package_name*: `String`, *model_package_version_number*: `String`): `String`
  Returns a formatted string representing the necessary dependencies for a flow.

Below are the descriptions of the variables and functions in the Configuration PyTorch package from the diagram represented in Figure 2.9. The main use of the package is to export the variables represented in the Configuration class as global ones and give them default values. The functions represented in the ConfigurationDefault class are the default values for all variables in the package of type `Callable`. The rest of the values are simple types. All default values for each variable are represented below.

- *logger*: `logging.Logger`
  Default logger for the MXNet extension
- *_default_criterion_gen*(*task*: `OpenMLTask`): `mxnet.gluon.loss.Loss`
  Default implementation, returns a loss criterion based on the task type.
- *criterion_gen*: `Callable`
  Stores the result from *_default_criterion_gen*. Can be overridden by the user if they wish, but the new function should have an `OpenMLTask` parameter and return a `mxnet.gluon.loss.Loss`.
- *_default_scheduler_gen*(*_*: `OpenMLTask`): `Optional[mxnet.lr_scheduler.LRScheduler]`
  Default implementation, returns the *None* scheduler for a given task. The "_" variable is there in order to respect the function signature, but is ignored in the default implementation. The reasoning for which it was added is in case some user wants to create their own scheduler generator, they can take into account the type of task.
- *scheduler_gen*: `Callable`
  Stores the result from *_default_scheduler_gen*. Can be overridden by the user if they wish, but the new function should have a `torch.nn.Module` and a `OpenMLTask` parameter and return a `Any`.
- *_default_optimizer_gen*(*lr_scheduler*: `mxnet.lr_scheduler.LRScheduler`, *_*: `OpenMLTask`): `mxnet.optimizer.Optimizer`
  Default implementation,returns the optimizer for the given task. The "_" variable is there in order to respect the function signature, but is ignored post that in the default implementation. The reasoning for which it was added is in case some user wants to create their own optimizer generator, they can take into account the type of task.
- *optimizer_gen*: `Callable`
  Stores the result from *_default_optimizer_gen*. Can be overridden by the user if they wish, but the new function should have a `mxnet.lr_scheduler.LRScheduler` and a `OpenMLTask` parameter and return a `mxnet.optimizer.Optimizer`.
- *batch_size*: `int`
  Represents the processing batch size for training. Default value is 64.
- *epoch_count*: `int`
  Represents he number of epochs the model should be trained for. Default value is 32.
- *backend_type*: `Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol]`
  Represents the possible MXNet backends. This is used in the default implementation below, in order to simplify writing: using `backend_type` instead of `Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol]`.

- *_default_predict*(*output*: `backend_type`, *task*: `OpenMLTask`): `backend_type`
  Default implementation, turns the outputs into predictions by returning the argmax of the output.
- *predict*: `Callable`
  Stores the result from *_default_predict*. Can be overridden by the user if they wish, but the new function should have a `backend_type` and a `OpenMLTask` as parameters and return a `backend_type`.
- *_default_predict_proba*(*output*: `backend_type`): `backend_type`
  Default implementation, turns the outputs into probabilities using softmax.
- *predict_proba*: `Callable`
  Stores the result from *_default_predict_proba*. Can be overridden by the user if they wish, but the new function should have a `backend_type` parameter and return a `backend_type`.
- *_default_sanitize*(*output*: `mxnet.ndarray.NDArray`): `mxnet.ndarray.NDArray`
  Default implementation, sanitizes the input data in order to ensure that models can be trained safely.
- *sanitize*: `Callable`
  Stores the result from *_default_sanitize*. Can be overridden by the user if they wish, but the new function should have a `mxnet.ndarray.NDArray` parameter and return a `mxnet.ndarray.NDArray`.
- *_default_metric_gen*(*task*: `OpenMLTask`): `mxnet.metric.EvalMetric`
  Default implementation, returns a composite metric composed of accuracy for classification tasks, and composed of mean squared error, mean absolute error, and root mean squared error for regression tasks.
- *metric_gen*: `Callable`
  Stores the result from *_default_metric_gen*. Can be overridden by the user if they wish, but the new function should have an `OpenMLTask` as parameter and return a `mxnet.metric.EvalMetric`.
- *_default_progress_callback*(*fold*: `int`, *rep*: `int`, *epoch*: `int`, *step*: `int`, *loss*: `mxnet.ndarray.NDArray`, *metric*: `mxnet.metric.EvalMetric`): `None`
  Default implementation, reports the current fold, rep, epoch, step, loss and metric for every training iteration to the default logger.
- *progress_callback*: `Callable`
  Stores the result from *_default_progress_callback*. Can be overridden by the user if they wish, but the new function should have three `int` parameters followed by a `mxnet.ndarray.NDArray` and a `mmxnet.metric.EvalMetric` parameters and return `None`.

Visualization

The visualization class diagram is depicted in figure 2.12. The Visualizer and the View depend on Dash. The View also contains a *visualizer* so that it can update. The Visualizer uses the OpenML run which is associated with an OpenML task. Alternatively, the Visualizer uses an

OpenML flow. The OpenML flow, OpenML run and OpenML task depend on OpenML Python API. Furthermore, the Visualizer uses the KerasExtension, PytorchExtension, MXNetExtension, ONNXExtension, and depends on the ONNX Library.

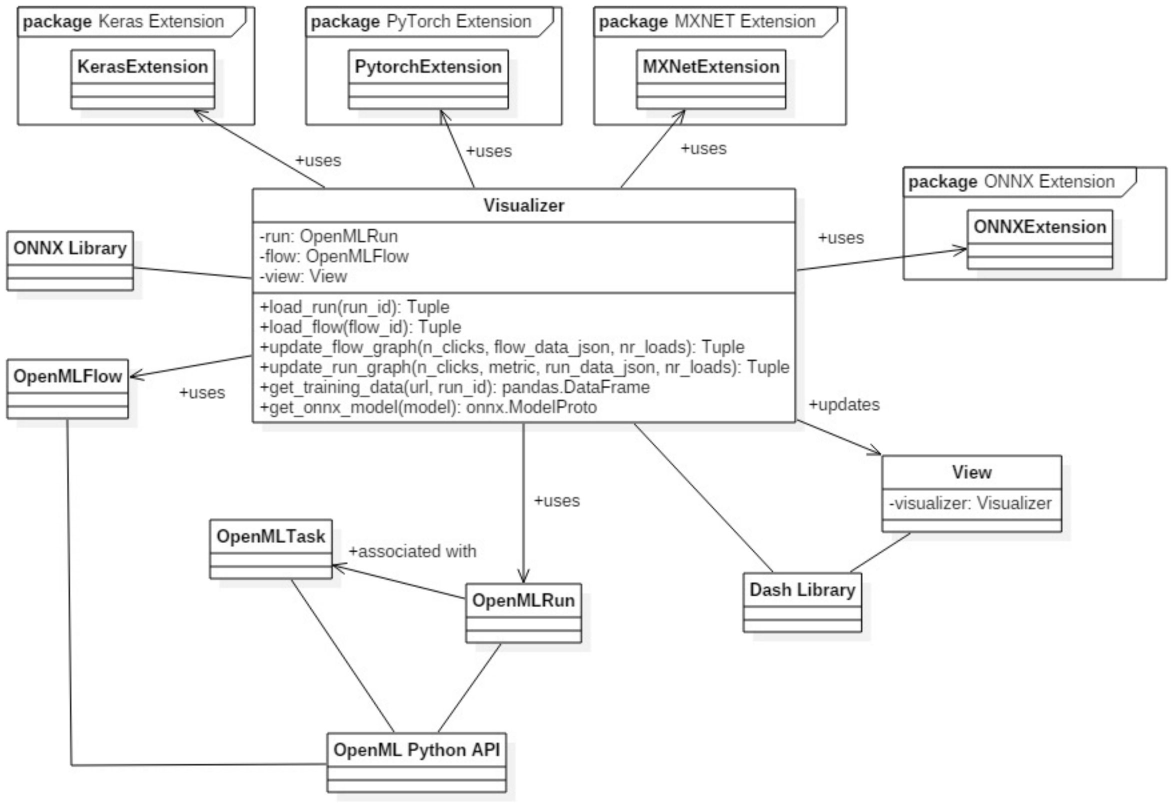


Figure 2.12: Visualization Class diagram

- *run*: `OpenMLRun`
  Stores the *run* that is to be visualized.
- *flow*: `OpenMLFlow`
  Stores the *flow* that is to be visualized.
- *view*: `View`
  The corresponding HTML rendered page that is shown for the visualization of the flow and run.
- *load_run*(*run_id*): `Tuple`
  This function gets the run from the OpenML website given the *run_id*. Then it extracts and stores all the necessary data that is needed to visualize the run.
- *load_flow*(*flow_id*): `Tuple`
  This function gets the flow from the OpenML website given the *flow_id*. Then it extracts and stores all the necessary data that is needed to visualize the flow.

- *update_flow_graph*(*n_clicks*, *flow_data_json*, *nr_loads*): `Tuple`
  Displays the flow in the window for visualization.
- *update_run_graph*(*n_clicks*, *metric*, *run_data_json*, *nr_loads*): `Tuple`
  Displays a graph for the selected metric in the window for visualization.
- *get_training_data*(*url*, *run_id*): `pandas.DataFrame`
  Downloads and loads the training data associated with the run with run id = *run_id*.
- *get_onnx_model*(*model*): `onnx.ModelProto`
  Extracts the ONNX specification from the *model*.

2.7.7 High-level sequence diagrams

The following subsections will offer some implementation details about all the use cases that have been described in the URD. The purpose of the high-level sequence diagrams is to illustrate the main idea behind the code implementation. They describe how problems, such as conversions, have been solved from a logical point of view. The solution is described in an abstract way, using natural language. This is done to make it more understandable, without going into technical and implementation details. Such low-level details are shown in 2.7.8. The activity diagrams from 2.7.8 are more representative for the way the code is implemented, but at the same time, harder to understand and follow.

2.7.7.1 Convert from the Keras** model to an OpenML flow

Use cases fulfilled: UC1, UC3

Figure 2.13 depicts a diagram with the abstract, logical idea behind converting a Keras** model to an OpenML flow. In order to do a conversion with non-redundant dependencies, the correct extension first needs to be identified. Hence, the very first call is `Decide on the extension` from `OpenML Python API` to `Extensions`. Once `Extensions` has identified and returned that the given model is a Keras** model, the actual conversion from a model to a flow begins with the call `Convert Keras** model into OpenML flow format`. Consequently the initialization parameters and layers from the model are extracted from the given Keras** model and are returned. If it is possible, this is done directly from the model, otherwise it is done manually in the Keras** Extension (step 4 & 5). Afterwards, it stores in OpenML hyperparameters the Keras** model's parameters and layers. In the end the `Keras** extension` returns the OpenML flow object. More low level details can be found in figure 2.18 for Keras, 2.19 for PyTorch and 2.20 for ONNX.

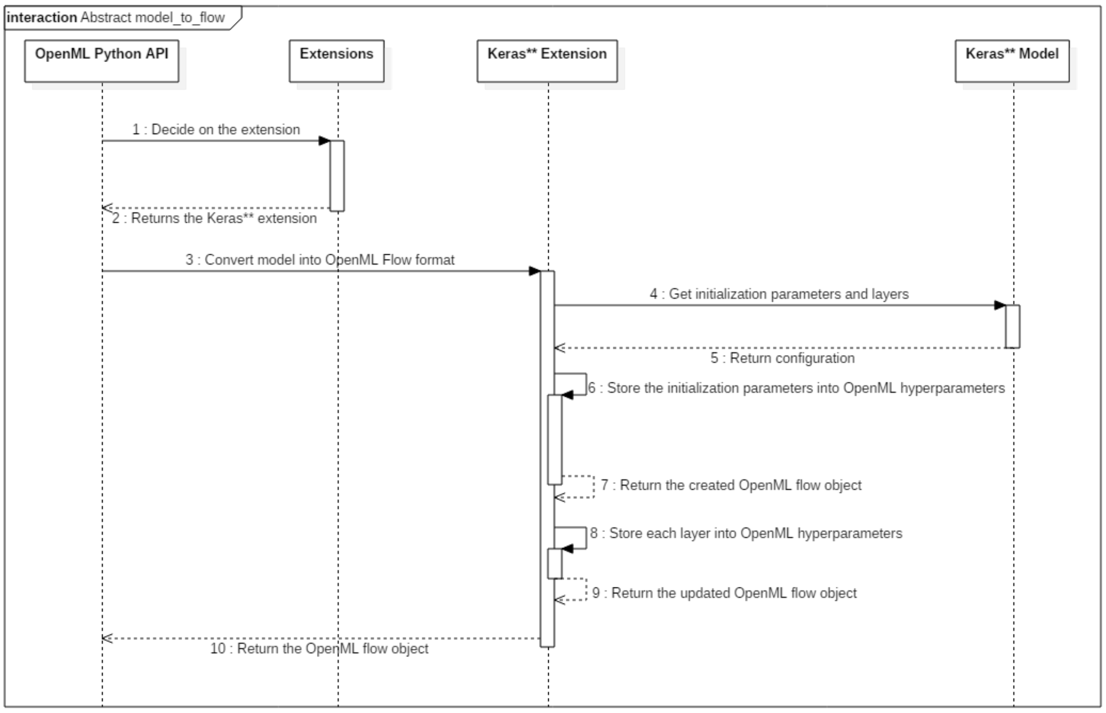

interaction Abstract model_to_flow
OpenML Python API
Extensions
Keras** Extension
Keras** Model
1 : Decide on the extension
2 : Returns the Keras** extension
3 : Convert model into OpenML Flow format
4 : Get initialization parameters and layers
5 : Return configuration
6 : Store the initialization parameters into OpenML hyperparameters
7 : Return the created OpenML flow object
8 : Store each layer into OpenML hyperparameters
9 : Return the updated OpenML flow object
10 : Return the OpenML flow object

Figure 2.13: Abstracted Sequence Diagram model_to_flow (Keras**)

2.7.7.2 Convert from an OpenML flow to a Keras** model

Use cases fulfilled: UC2, UC3

Figure 2.14 shows a diagram with the conversion process from an OpenML flow to a Keras** model is initiated by the OpenML Python API. It interacts with the `Extensions` package, which determines the library (Keras, PyTorch, etc.) needed to be used for the conversion. For diagram 2.14, it is known that the Keras** extension was chosen. The process now gets the parameters and the layers from the flow. A dictionary is created with those parameters and layers. Consequently, this dictionary is passed to the Keras** library in order to create a model based on it. The newly created model is now returned to the initial call of the process. More low level details can be found in figure 2.22 for Keras, figure 2.23 for PyTorch, and figure 2.24 for ONNX.

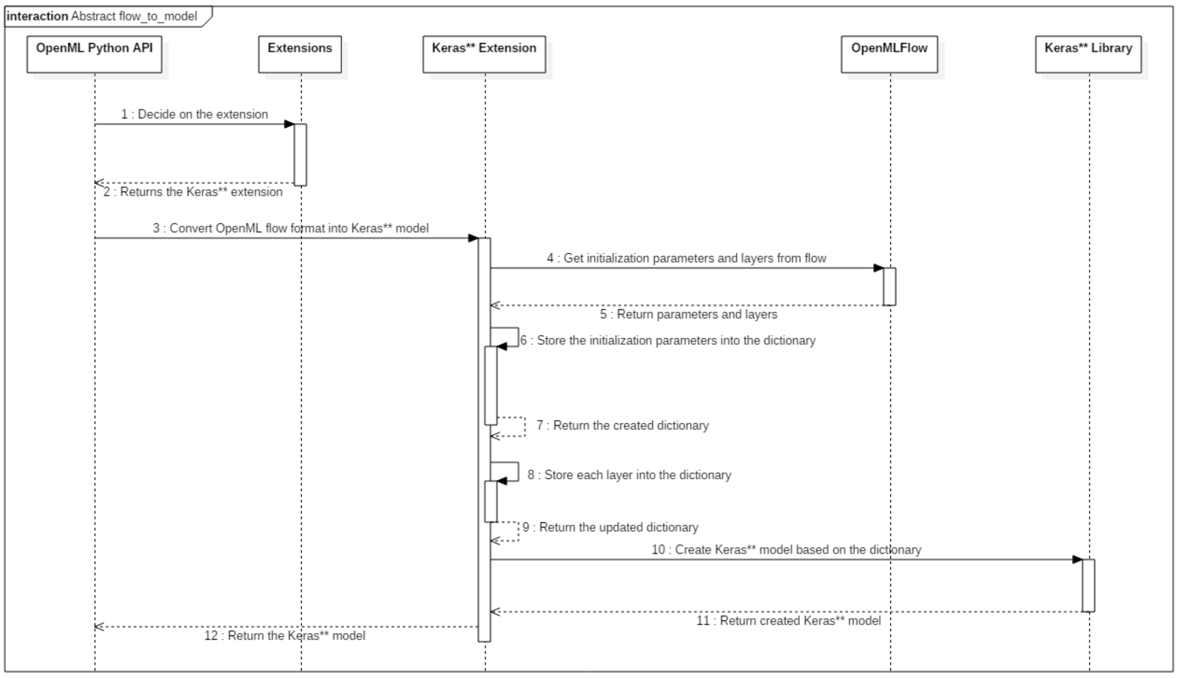

interaction Abstract flow_to_model
OpenML Python API
Extensions
Keras** Extension
OpenMLFlow
Keras** Library
1 : Decide on the extension
2 : Returns the Keras** extension
3 : Convert OpenML flow format into Keras** model
4 : Get initialization parameters and layers from flow
5 : Return parameters and layers
6 : Store the initialization parameters into the dictionary
7 : Return the created dictionary
8 : Store each layer into the dictionary
9 : Return the updated dictionary
10 : Create Keras** model based on the dictionary
11 : Return created Keras** model
12 : Return the Keras** model

Figure 2.14: Abstracted Sequence Diagram flow_to_model (Keras**)

2.7.7.3 Run OpenML flow on task

Use cases fulfilled: UC1, UC3

Figure 2.15 shows the diagram containing the main idea of how an OpenML flow can be run. In order to run a flow, it has to be converted back into the deep learning model it represents. Note that a flow cannot be run. Only models can be executed. A flow represents either the description of a model of a certain deep learning library/framework, such as Keras or PyTorch, or a specification, such as ONNX.

Firstly, the corresponding library for the flow needs to be found. This library is named X in the diagram. Once X is determined, the appropriate extension package for X needs to be called in order to create the model according to the flow.

There are two options to create a model from a given flow, as depicted in the diagram. (1) If the flow represents a deep learning model, then the corresponding library for that model is called in order to create it. (2) If the flow represents a specification, such as ONNX, then the specification needs to be created before converting it into a model. In essence, a flow can represent a specification which represents a deep learning model. More details about how to create a model from a flow/specification (i.e. convert flow to model) can be found in the low level activity diagrams, in section 2.7.8, for the appropriate extension package.

Once the model is received by the OpenML Python API, it is run using the corresponding extension package. Depending on whether the given task is classification or regression, the appropriate functions from the library of the model are called in order to return the run. More about how a model can be run on a task can be found in the low level activity diagrams, in section

2.7.8, for the appropriate extension package.

Corresponding Extension

Keras

PyTorch

MXNet

ONNX

Corresponding model Library

Keras

PyTorch

MXNet

Corresponding specification Library

ONNX

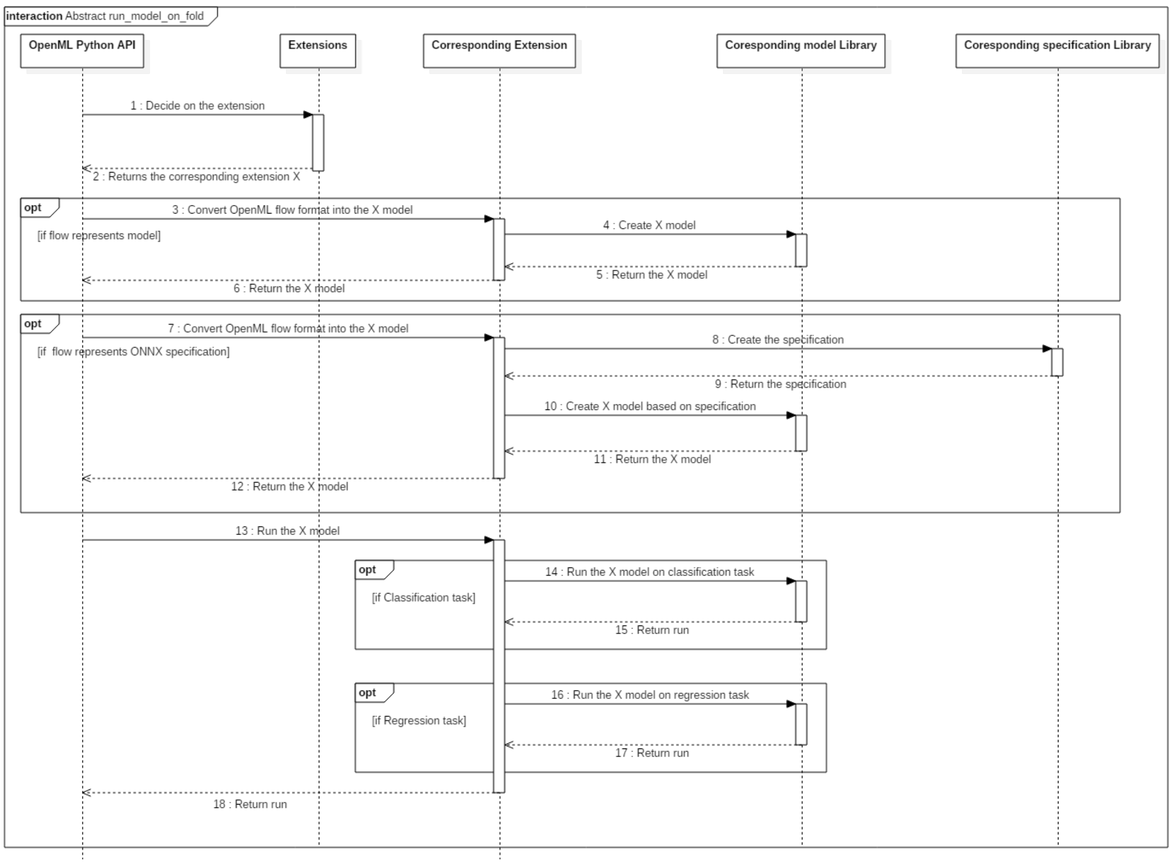

interaction Abstract run_model_on_fold
OpenML Python API
Extensions
Corresponding Extension
Coresponding model Library
Coresponding specification Library
1 : Decide on the extension
2 : Returns the corresponding extension X
opt
3 : Convert OpenML flow format into the X model
[if flow represents model]
4 : Create X model
5 : Return the X model
6 : Return the X model
opt
7 : Convert OpenML flow format into the X model
[if flow represents ONNX specification]
8 : Create the specification
9 : Return the specification
10 : Create X model based on specification
11 : Return the X model
12 : Return the X model
13 : Run the X model
opt
[if Classification task]
14 : Run the X model on classification task
15 : Return run
opt
[if Regression task]
16 : Run the X model on regression task
17 : Return run
18 : Return run

Figure 2.15: Abstracted Sequence Diagram run_model_on_fold

2.7.7.4 Display the learning curve graph of a run

Use cases fulfilled: UC4

Figure 2.16 shows the diagram with the main idea of how an OpenML run can be visualized. The visualization consists of plotting a learning curve graph based on the information from the OpenML run. This graph is created using the Dash framework and is outside of the scope of the project "Sharing deep learning models" since it is not implemented by the OpenML Support Squad.

In order for a user to see the learning curve graph of a run, it needs to find the id of the run from the OpenML website. This id is inserted in the local Dash website. Once the user types the id, he has to press the "Load run" button found in the local Dash website. The Dash website will call the `Visualization` package, which is created by the OpenML Support Squad. The package will retrieve the run from the OpenML website by using the OpenML Python API. The OpenML Python API is outside of the scope of project "Sharing deep learning models" since it is not implemented by the OpenML Support Squad. Once the run is received, the `Visualization` package will parse all the information from the run and will pass the processed information to Dash. Dash will plot the graph based on the received data and display it on the local Dash website of the user. The user can see the graph on the local Dash website. More about how the start the local Dash website in the Software User Manual (SUM).

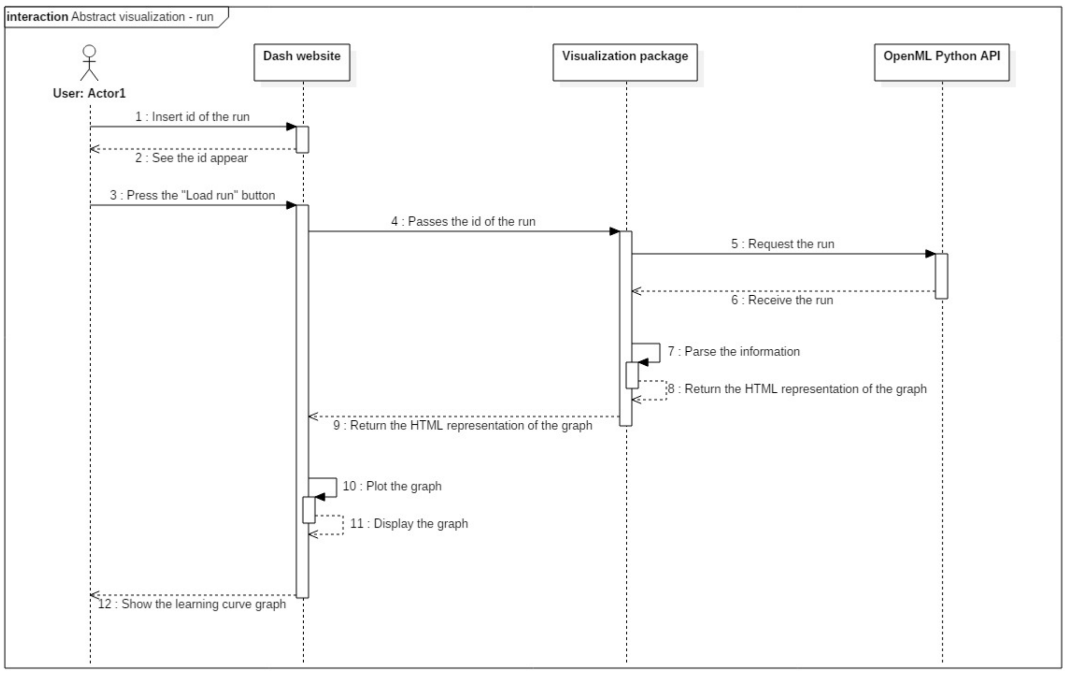

interaction Abstract visualization - run
User: Actor1
Dash website
Visualization package
OpenML Python API
1 : Insert id of the run
2 : See the id appear
3 : Press the "Load run" button
4 : Passes the id of the run
5 : Request the run
6 : Receive the run
7 : Parse the information
8 : Return the HTML representation of the graph
9 : Return the HTML representation of the graph
10 : Plot the graph
11 : Display the graph
12 : Show the learning curve graph

Figure 2.16: Abstracted Sequence Diagram visualization of run

2.7.7.5 Display the neural network structure of a flow

Use cases fulfilled: UC4

Figure 2.17 shows the diagram with the main idea of how the structure of an OpenML flow can be visualized. The visualization consists of plotting the structure of the model that the flow represents. This graph is created using the Dash framework and is outside of the scope of the project "Sharing deep learning models" since it is not implemented by the OpenML Support Squad. Furthermore, the structure of a model is done by using ONNX. The model is converted into an ONNX specification and then the package creates the visual representation of the model.

In order for a user to see the neural network structure of a flow, it needs to find the id of the flow from the OpenML website. This id is inserted in the local Dash website. Once the user types the id, he has to press the "Load flow" button found in the local Dash website. The Dash website will call the `Visualization` package, which is created by the OpenML Support Squad. The package will retrieve the flow together with its corresponding model from the OpenML website by using the OpenML Python API. The OpenML Python API is outside of the scope of project "Sharing deep learning models" since it is not implemented by the OpenML Support Squad.

Once the model is received, the `Visualization` package will convert it into an ONNX specification. The package will use the ONNX specification to create the visualized neural network structure of the model. Afterwards, the package will create the HTML representation of the graph (i.e. structure of the model). Dash will plot the graph based on the received HTML and display it on the local Dash website of the user. The user can see the graph on the local Dash website. More about how the start the local Dash website in the Software User Manual (SUM).

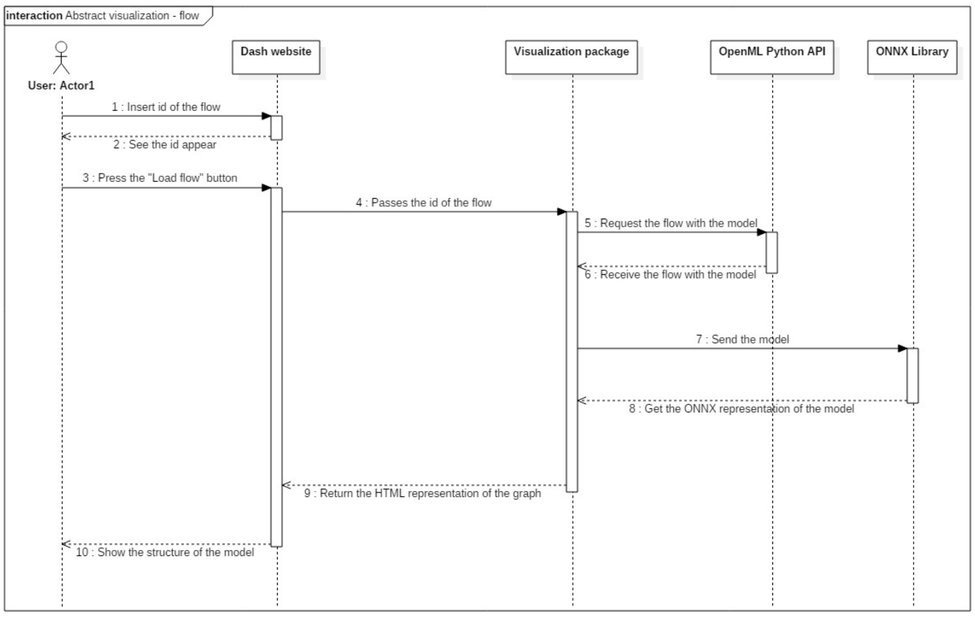

interaction Abstract visualization - flow
User: Actor1
Dash website
Visualization package
OpenML Python API
ONNX Library
1 : Insert id of the flow
2 : See the id appear
3 : Press the "Load flow" button
4 : Passes the id of the flow
5 : Request the flow with the model
6 : Receive the flow with the model
7 : Send the model
8 : Get the ONNX representation of the model
9 : Return the HTML representation of the graph
10 : Show the structure of the model

Figure 2.17: Abstracted Sequence Diagram visualization of a flow

2.7.8 Low level activity diagrams

The following subsections will offer low level implementation details about all the use cases that have been described in the URD. They will go into more detail, while keeping them understandable. Each subsection contains an activity diagram, which illustrates the respective process flow (i.e. the main function calls), while also offering a short description of the steps and the purpose of such calls. The diagrams are abstracted from machine learning technicalities since they are hard to understand and outside of the subsections' scope to be present in a diagram. The diagrams are meant to offer some understanding on how the actual code is structured and implemented as well as what are the main functions that are called. In this way maintaining the code will be easier since all the function dependencies of each of the main use cases are known.

2.7.8.1 Convert from a Keras model to an OpenML flow

Use cases fulfilled: UC1, UC3

Figure 2.18 shows the diagram for conversion from a Keras model into an OpenML flow. The method `model_to_flow`, from the Keras extension package, is used to convert directly from a Keras model into a OpenML flow. The method `model_to_flow` is invoked by the OpenML Python API. Upon invocation, it retrieves the configuration of the Keras model. This is shown in the corresponding activity *"Get configuration of model"* in diagram 2.18. This step is made substantially easier in Keras than in other frameworks, such as PyTorch, since the Keras library provides a function `get_config` that returns a dictionary with all the layers of the model. On a conceptual basis, after getting the config dictionary, the function `_serialize_keras` recurses through the model component by component to check if all parameters are of a supported type as shown in *"Iterate parameters"*. These are then stored in a dictionary in json format. This dictionary can then be translated into the OpenML flow format. This is depicted in the iterative expansion region of the diagram *"Transform to json"*. Once it is done building the dictionary, an OpenML flow is created and returned to the OpenML Python API.

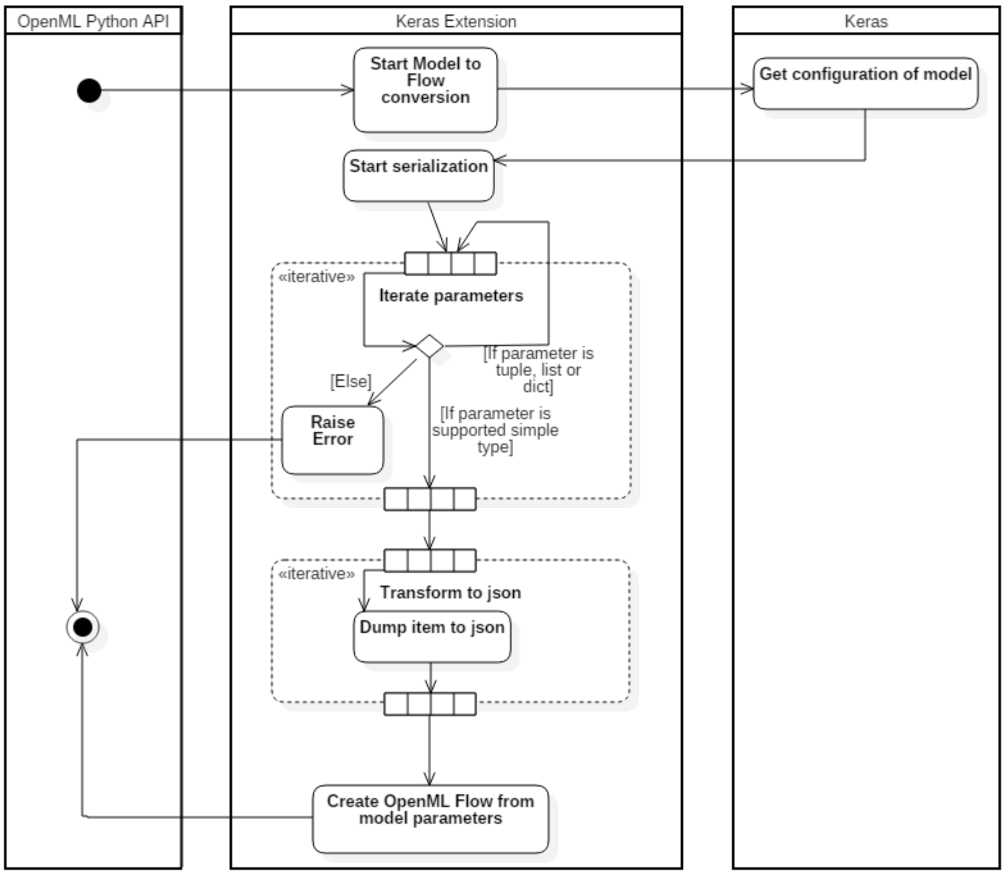

OpenML Python API
Keras Extension
Keras
Start Model to Flow conversion
Get configuration of model
Start serialization
«iterative»
Iterate parameters
[If parameter is tuple, list or dict]
[Else]
Raise Error
[If parameter is supported simple type]
«iterative»
Transform to json
Dump item to json
Create OpenML Flow from model parameters

Figure 2.18: Activity Diagram model_to_flow (Keras)

2.7.8.2 Convert from a PyTorch model to an OpenML flow

Use cases fulfilled: UC1, UC3

Figure 2.19 shows the diagram concerning the conversion from a PyTorch model into an OpenML flow. The method `model_to_flow`, from the PyTorch extension package, is used to convert directly from a PyTorch model into an OpenML flow. The method `model_to_flow` is invoked by the OpenML Python API. Upon invocation, it constructs the configuration from the PyTorch model. This is shown in the corresponding activity *"Get configuration of model"* in diagram 2.19. Unlike with Keras models, obtaining the configuration of PyTorch models needs to be done manually. On a conceptual basis, after getting the config dictionary, the function `_serialize_pytorch` recurses through the model component by component to check if all parameters are of a supported type as shown in *"Iterate parameters"*. In case that a parameter is a function, the function is replaced by a reference to that function, since a function can not be dumped to a json format. These parameters are then stored in a dictionary in json format. This dictionary can then be translated into the OpenML flow format. This is depicted in the iterative expansion region of the diagram *"Transform to json"*. Once it is done building the dictionary, an OpenML flow is created and returned to the OpenML Python API.

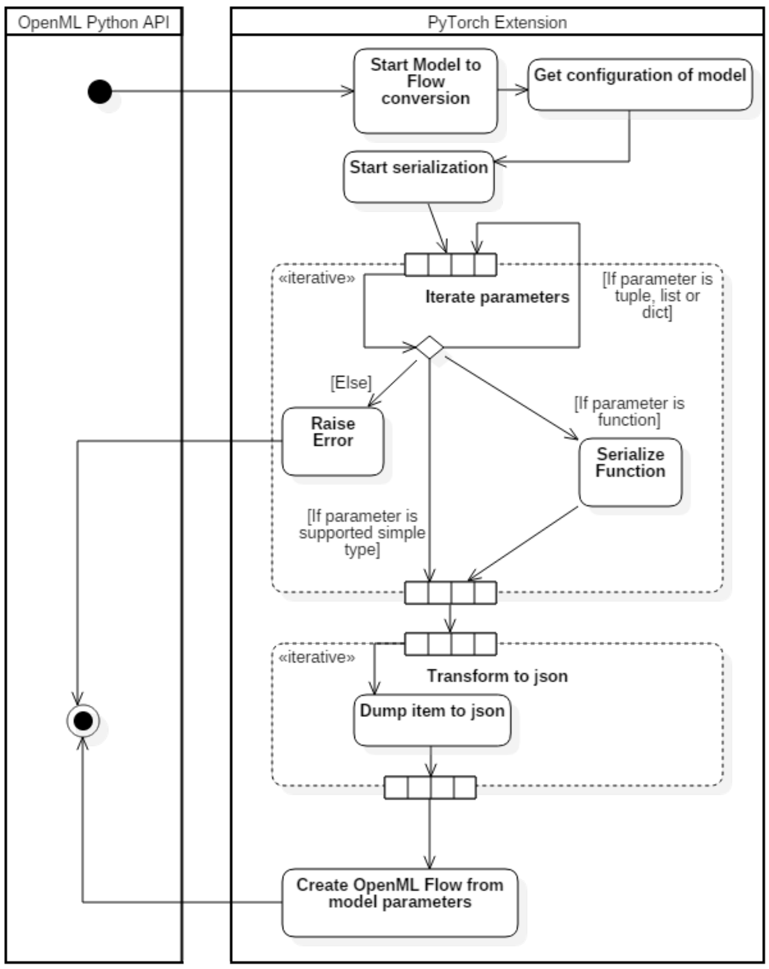

OpenML Python API
PyTorch Extension
Start Model to Flow conversion
Get configuration of model
Start serialization
«iterative»
Iterate parameters
[If parameter is tuple, list or dict]
[Else]
Raise Error
[If parameter is function]
Serialize Function
[If parameter is supported simple type]
«iterative»
Transform to json
Dump item to json
Create OpenML Flow from model parameters

Figure 2.19: Activity Diagram model_to_flow (PyTorch)

2.7.8.3 Convert from an ONNX specification to an OpenML flow

Use cases fulfilled: UC1, UC3

Figure 2.20 displays the diagram with the conversion from an ONNX specification to an OpenML flow. The method `model_to_flow` is called by the OpenML Python API to serialize an ONNX specification. It firstly obtains the configuration from the ONNX specification. Afterwards, the serialization begins. The first activity is extracting the parameters and iteratively create an ordered dictionary from them. At this point, the dictionary still has to be reformatted. Instead of using recursion, like with Keras, we can simply use a for-loop for this. Inside this for-loop, we serialize each parameter using json inside a well-formatted new "parameter dictionary". If this item is a dictionary, it will recursively be sorted by keys using the `_to_ordered` function and then dump it as an ordered dictionary. After this is done for each component in the dictionary, all parameters have been obtained and stored in an OpenML-friendly format. Additional steps are done, i.e. getting the external versions of all sub-components and initializing them. With the parameter values the ONNX extension can be turned into an OpenML flow in trivial manner.

This OpenML flow is then returned to the initial `model_to_flow` call and the method terminates after successfully having serialized the ONNX specification into an OpenML flow.

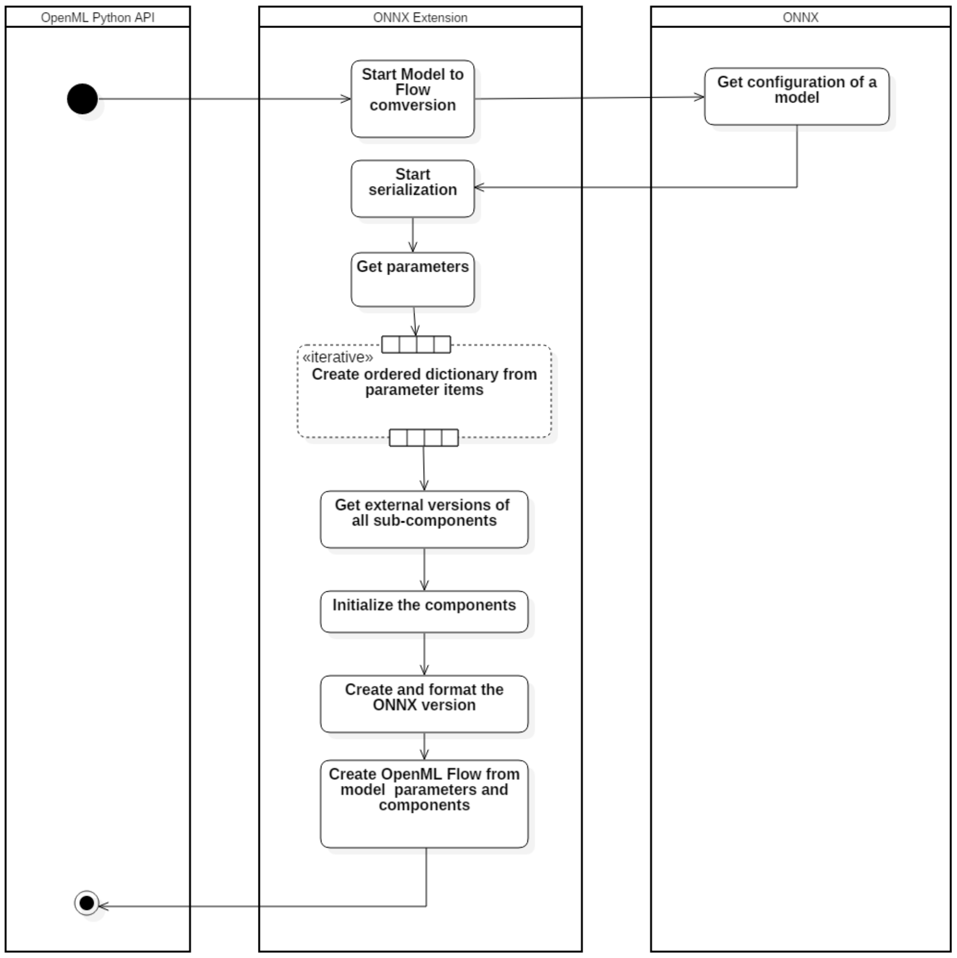

OpenML Python API
ONNX Extension
ONNX
Start Model to Flow conversion
Get configuration of a model
Start serialization
Get parameters
«iterative»
Create ordered dictionary from parameter items
Get external versions of all sub-components
Initialize the components
Create and format the ONNX version
Create OpenML Flow from model parameters and components

Figure 2.20: Activity Diagram model_to_flow (ONNX)

2.7.8.4 Convert from an MXNet model to an OpenML flow

Use cases fulfilled: UC1, UC3

Figure 2.21 displays the diagram with the conversion of an Apache MXNet model to an OpenML flow, which is very similar to converting a PyTorch model to an OpenML flow. The conversion starts by calling `model_to_flow` which belongs to the Apache MXNet extension package of the DL Extension Library. This method is invoked by the OpenML Python API. When the model to flow conversion is initiated the information from the Apache MXNet model is extracted and presented in a JSON format. Afterwards, the serialization is started and the model is serialized component by component, as presented in the iterative expansion region "Iterate components". If there are more components to be serialized then these are serialized. Once all of the components are serialized an OpenML flow is constructed and returned to the original call.

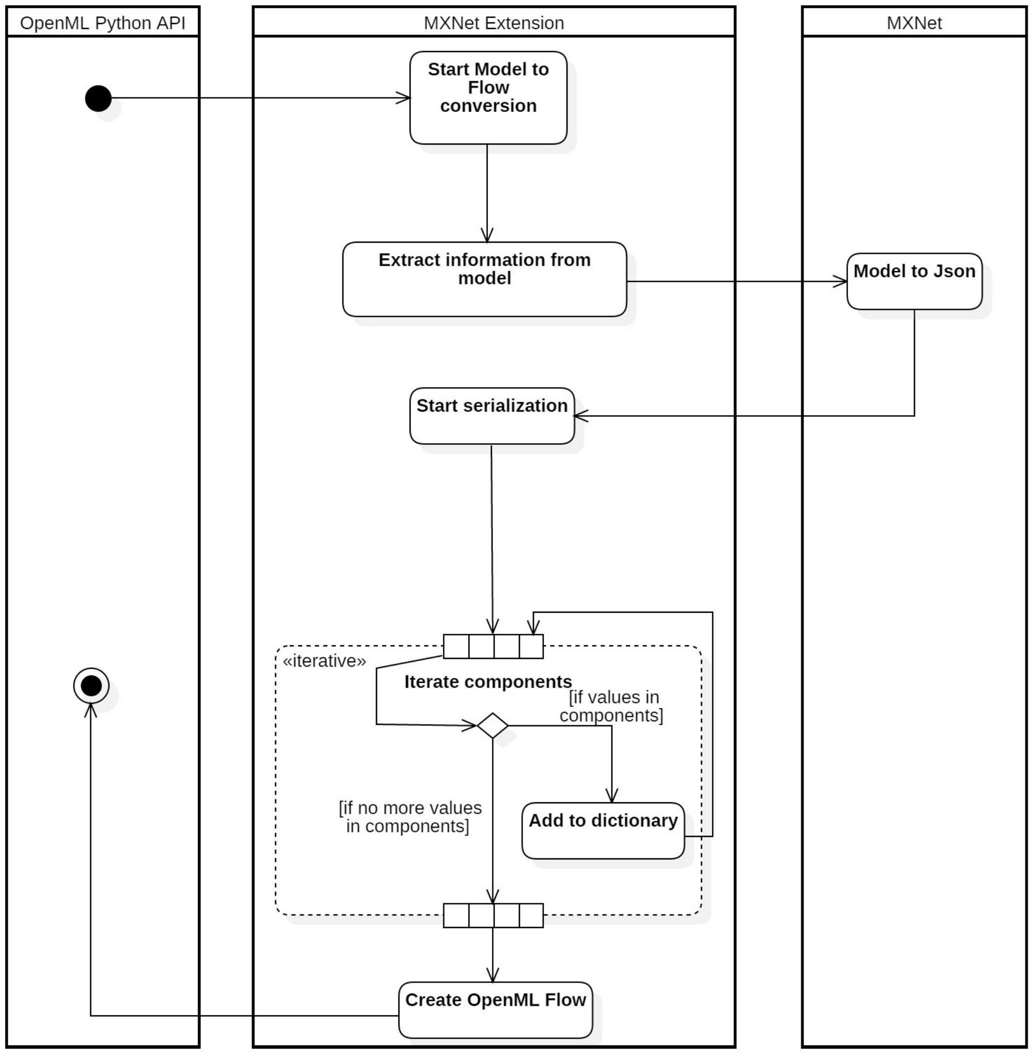


Figure 2.21: Activity Diagram model_to_flow (MXNet)

2.7.8.5 Convert from an OpenML flow to a Keras model

Use cases fulfilled: UC2, UC3

Figure 2.22 shows the diagram with the conversion from a flow representing a Keras model into a Keras model instance. The method `flow_to_model` is invoked by the OpenML Python API. When invoked, it first retrieves the parameters and components from the OpenML flow. This is shown in the corresponding activity *"Get parameters and components from flow"*. This can be retrieved directly from the flow format which is already handled by OpenML itself. When the components and parameters have been retrieved, deserialization can begin. On a conceptual basis, after getting these components and parameters, the function `_deserialize_keras` recurses through the components one by one. For each iteration, the method first checks if it is still in a string format. If this is the case, json tries to load it back into variables. Afterwards it checks what type the parameter is. If the parameter is a list, tuple or dictionary, the method calls itself again. If the parameter is a simple supported type, then it exits the method. This is shown in the expansion region *"Transform from json"*. These are then stored in a dictionary. This dictionary is then transformed into a configuration dictionary which is then used to reinstantiate the model. Finally, the model is returned to the OpenML Python API.

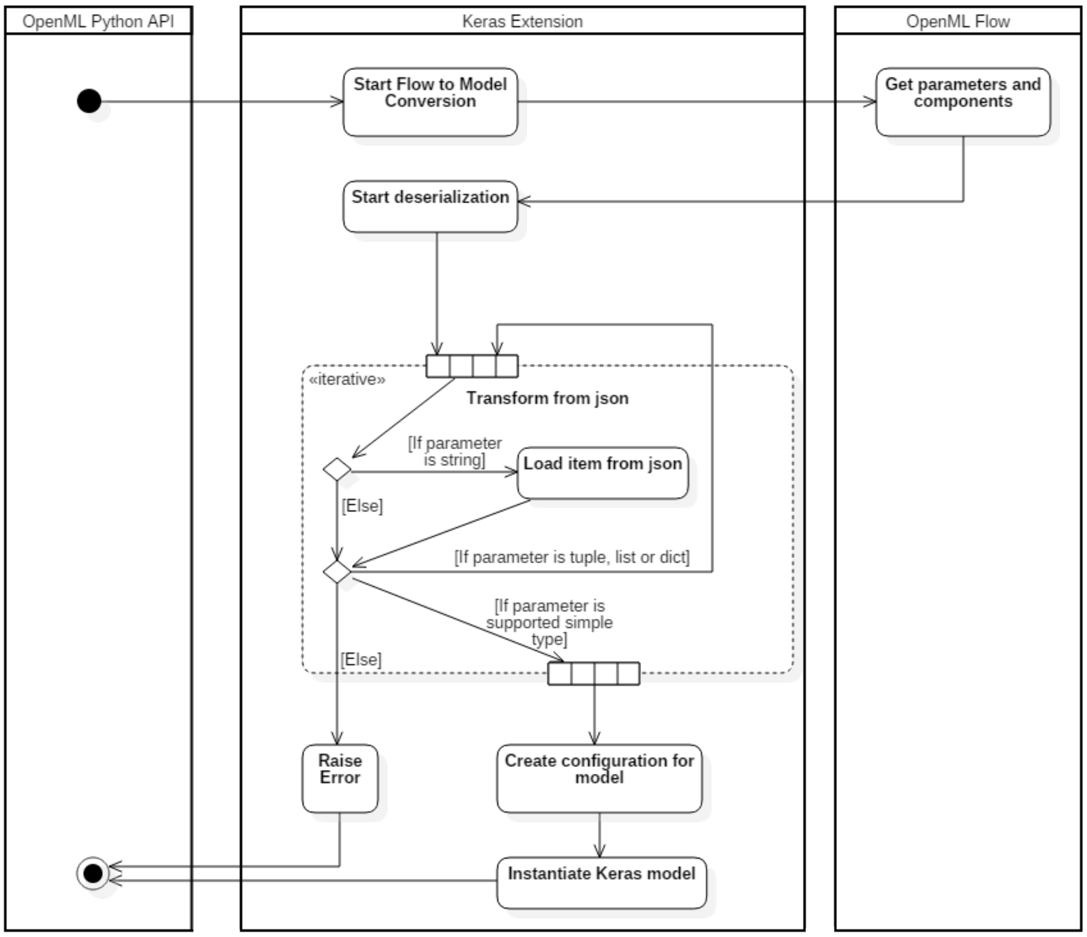


Figure 2.22: Activity Diagram flow_to_model (Keras)

2.7.8.6 Convert from an OpenML flow to a PyTorch model

Use cases fulfilled: UC2, UC3

Figure 2.23 shows the diagram with the conversion from a flow representing a PyTorch model into a PyTorch model instance. The method `flow_to_model` is invoked by the OpenML Python

API. When invoked, it first retrieves the parameters and components from the OpenML flow. This is shown in the corresponding activity *"Get parameters and components from flow"*. This can be retrieved directly from the flow format which is already handled by OpenML itself. When the components and parameters have been retrieved, deserialization can begin. On a conceptual basis, after getting these components and parameters, the function `_deserialize_pytorch` recurses through the components one by one. For each iteration, the method first checks if it is still in a string format. If this is the case, json tries to load it back into variables. Afterwards it checks what type the parameter is. If the parameter is a list or a tuple, the method calls itself again. If the parameter is a simple supported type, then it exits the method. If the object is a dictionary, the method has to check if there is a serialized object inside, such as a function reference. In this case, this object has to be deserialized separately. When this is done, it can exit the method. If it is a normal dictionary the method calls itself again. These choices are shown in the expansion region *"Transform from json"*. These are then stored in a dictionary. This dictionary is then transformed into a configuration dictionary which is then used to reinstantiate the model. Finally, the model is returned to the OpenML Python API.

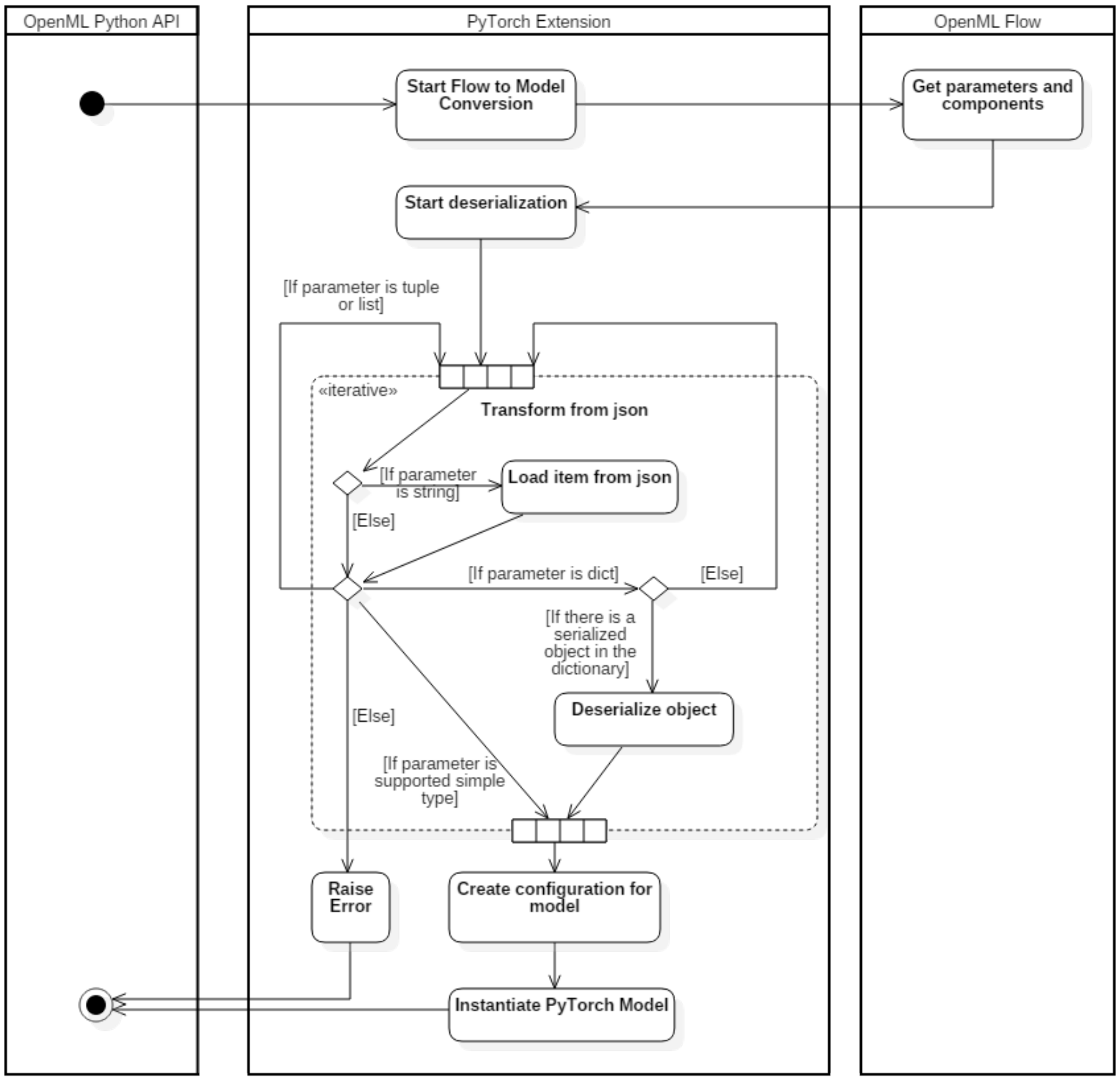

OpenML Python API
PyTorch Extension
OpenML Flow
Start Flow to Model Conversion
Get parameters and components
Start deserialization
[If parameter is tuple or list]
«iterative»
Transform from json
[If parameter is string]
Load item from json
[Else]
[If parameter is dict]
[Else]
[If there is a serialized object in the dictionary]
Deserialize object
[Else]
[If parameter is supported simple type]
Raise Error
Create configuration for model
Instantiate PyTorch Model

Figure 2.23: Activity Diagram flow_to_model (PyTorch)

2.7.8.7 Convert from an OpenML flow to an ONNX specification

Use cases fulfilled: UC2, UC3

As shown in the diagram depicted on figure 2.24, the conversion from an OpenML flow to an ONNX is somewhat different from Keras. Just like for serialization, no recursion is used for ONNX deserialization. Instead multiple for-loops are used to construct the specification from the flow.

First of all the parameters can then be retrieved from the flow in the form of a parameter dictionary. This dictionary has the wrong format, so it has to be reformatted i.e. construct the model dictionary. This happens inside the first iteration. An empty dictionary, `model_dic`, is created to hold the parameters in a format that is usable by ONNX. After this dictionary is built, it can be converted using Google's Protobuf. For this, an empty ONNX specification is created. Then, the Protobuf parses the dictionary and puts its contents into the ONNX specification. Finally, the completed model is returned to the OpenML Python API.

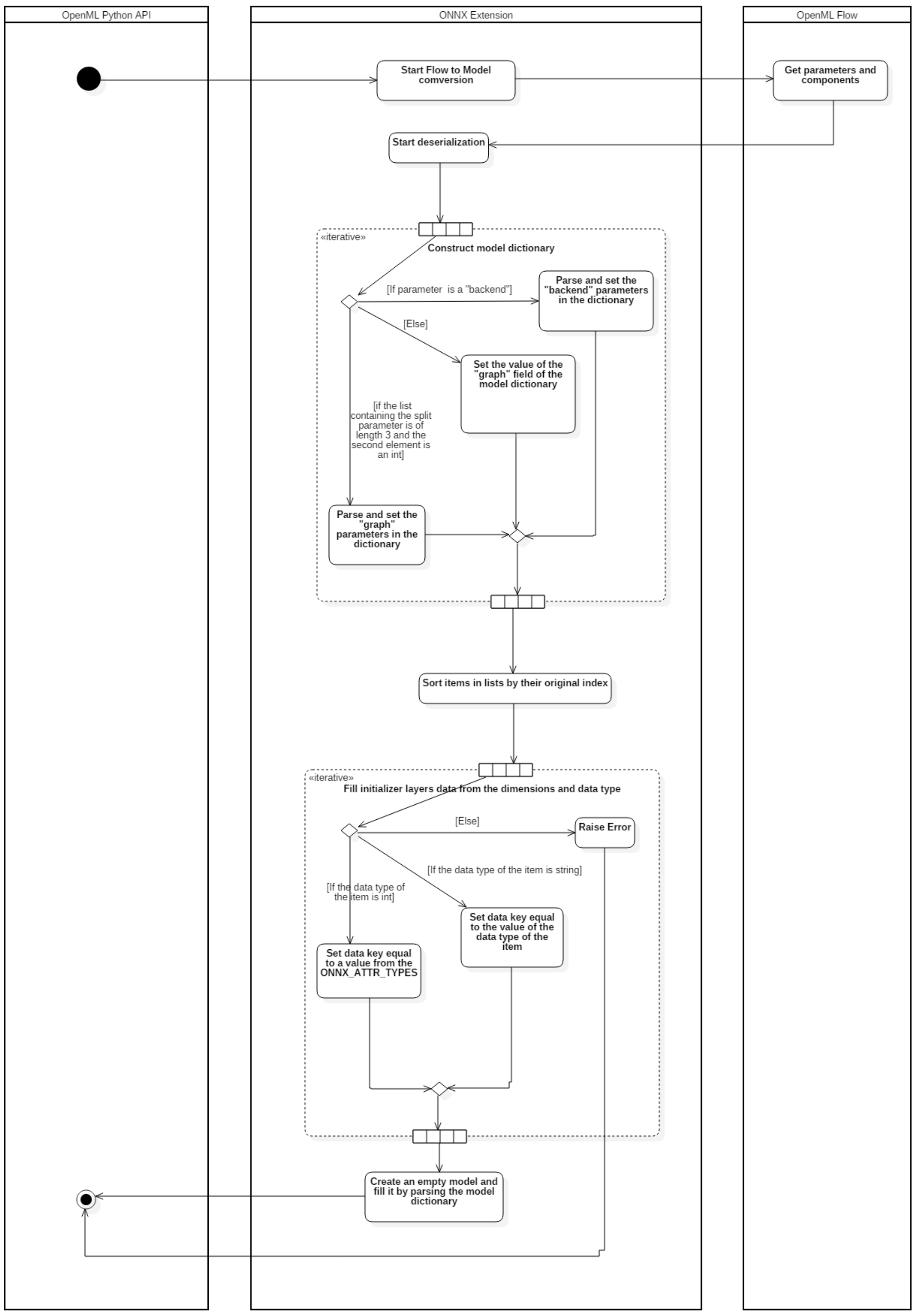

OpenML Python API
ONNX Extension
OpenML Flow
Start Flow to Model conversion
Get parameters and components
Start deserialization
«iterative»
Construct model dictionary
[If parameter is a "backend"]
Parse and set the "backend" parameters in the dictionary
[Else]
Set the value of the "graph" field of the model dictionary
[if the list containing the split parameter is of length 3 and the second element is an int]
Parse and set the "graph" parameters in the dictionary
Sort items in lists by their original index
«iterative»
Fill initializer layers data from the dimensions and data type
[Else]
Raise Error
[If the data type of the item is string]
[If the data type of the item is int]
Set data key equal to the value of the data type of the item
Set data key equal to a value from the ONNX_ATTR_TYPES
Create an empty model and fill it by parsing the model dictionary

Figure 2.24: Activity Diagram flow_to_model (ONNX)

2.7.8.8 Convert from an OpenML flow to a MXNet model

Use cases fulfilled: UC2, UC3

The activity diagram on figure 2.25 shows the conversion from an OpenML flow into a MXNet model. Firstly, the method from the MXNet extension package `flow_to_model` is invoked by the OpenML Python API. If the flow is not an MXNet flow, an error is raised. However, if the flow is an MXNet flow the parameters and the components from the OpenML flow are obtained. Then the deserialization is started. The deserialization is depicted in the iterative expansion region "Iterate parameters". While there are parameters to be serialized the values from these are extracted and loaded in a JSON representation. When all the parameters are extracted in a JSON representation, a configuration of the nodes is created. Afterwards all the nodes are iterated, as shown in the iterative expansion region "Iterate nodes". All these are appended in a dictionary. Later from this dictionary the original MXNet model is reconstructed and transferred back to the original call terminating the process.

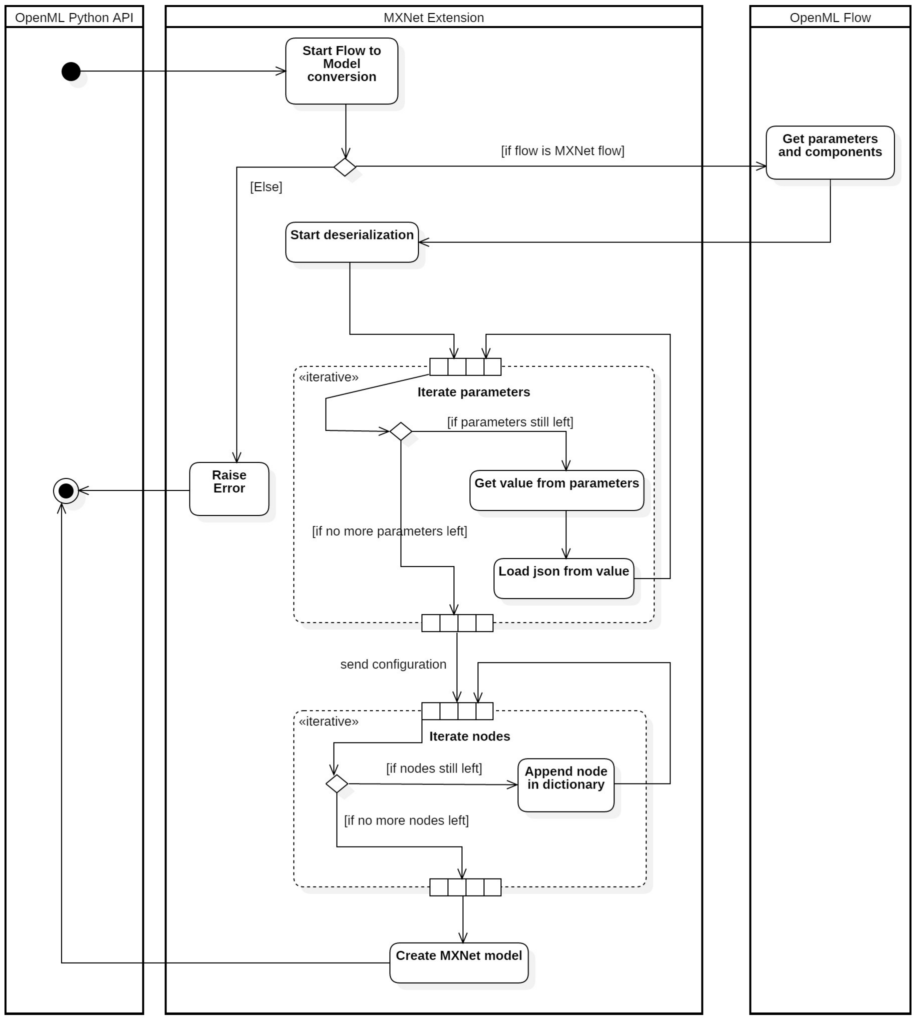


Figure 2.25: Activity Diagram flow_to_model (MXNet)

2.7.8.9 Run a Keras model on fold

Use cases fulfilled: UC1, UC3

Figure 2.26 shows the diagram concernign how a Keras model is run using the method `_run_model_on_fold` from the Keras extension package. This method is invoked by the OpenML Python API. If the provided task is not supervised, an error will be raised. In other cases, the model can be trained on some provided training data and a prediction on some other data can be made. These datasets can be chosen from those provided by OpenML. When training is completed, a distinction is made between classification and regression tasks. For a regression task, the extension can immediately return the predictions and pass them back to the OpenML Python API. If the tasks is classification, it first needs to get the probabilities for each prediction, which will then be included in the results. After this it can also return the results.

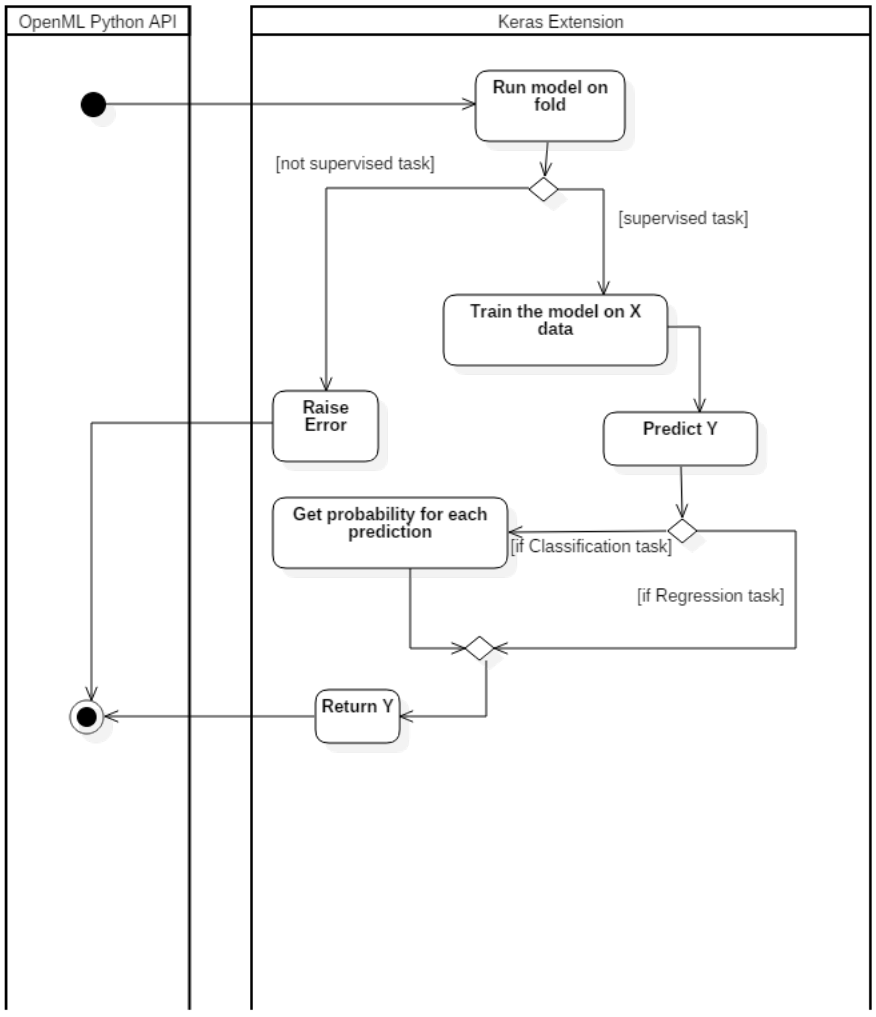


Figure 2.26: Activity Diagram _run_model_on_fold (Keras)

2.7.8.10 Run PyTorch model on fold

Use cases fulfilled: UC1, UC3

Figure 2.27 shows the diagram about how a PyTorch model is run using the method `_run_model_on_fold` from the PyTorch extension package. It has the exact same structure as the equivalent function from the Keras extension. As such, refer to subsection 2.7.8.9 (Run Keras model on fold).

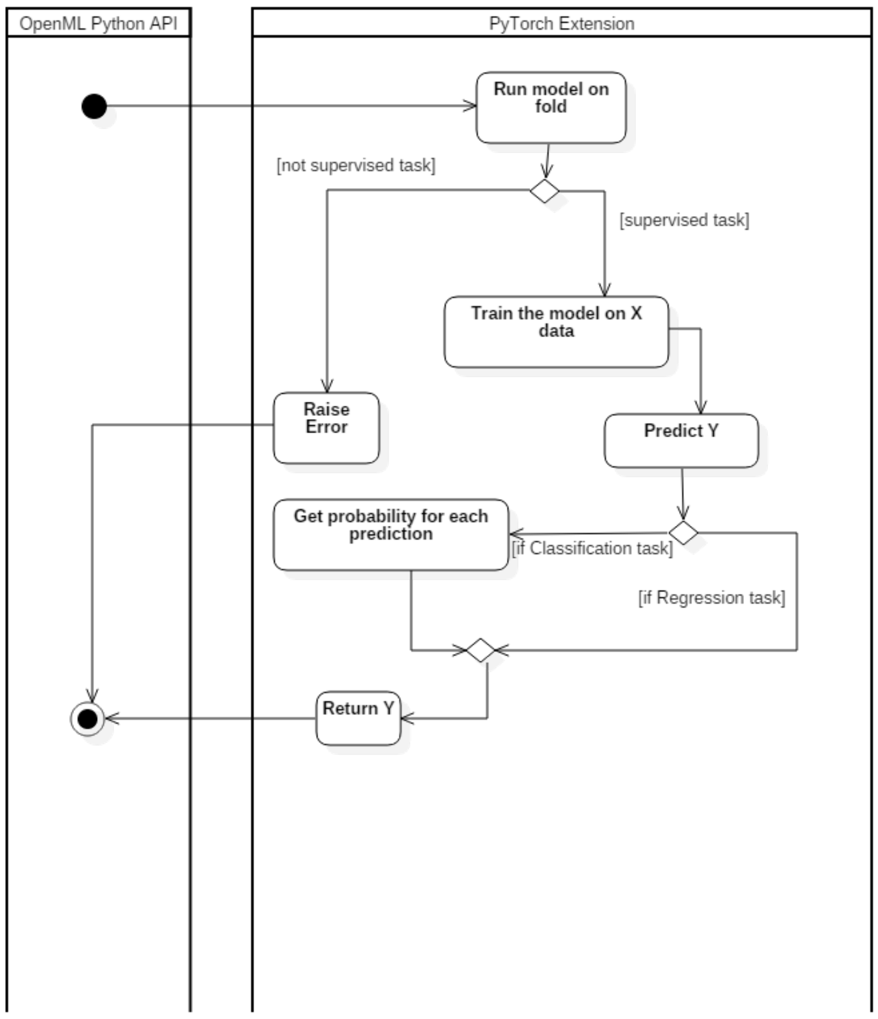


Figure 2.27: Activity Diagram _run_model_on_fold (PyTorch)

2.7.8.11 Run ONNX specification on fold

Use cases fulfilled: UC1, UC3

Figure 2.28 shows the diagram on how a ONNX specification is run using the method `_run_model_on_fold` from the ONNX extension package. The method `_run_model_on_fold` in ONNX is almost identical to the structure of the equivalent method in the Keras or PyTorch extension. Only the differences will be elaborated on. For a full picture of the ONNX variant, the explanation for the Keras equivalent should also be read in section 2.7.8.9 (Run Keras model on fold).

Before checking the task type and fitting the model, a conversion to MXNet has to be done. This is necessary because an ONNX specification is not a runnable model. MXNet is the only library that fully supports conversion from and to ONNX, which is why it is preferred. After conversion to MXNet, the same structure as the Keras equivalent is used to train the model.

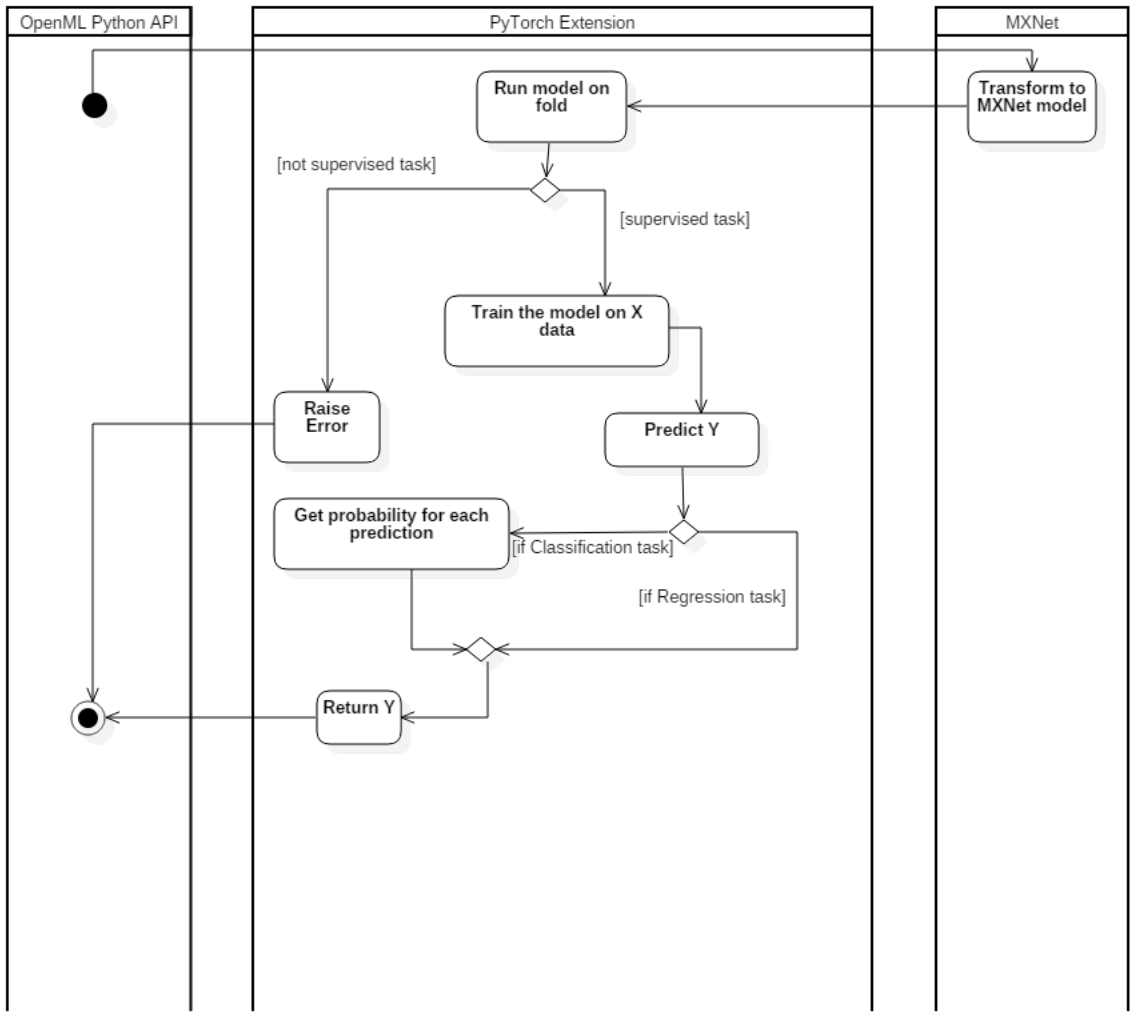


Figure 2.28: Activity Diagram _run_model_on_fold (ONNX)

2.7.8.12 Run MXNet model on fold

Use cases fulfilled: UC1, UC3

Figure 2.29 shows the diagram on how a MXNet model is run using the method `_run_model_on_fold` from the MXNet extension package. It has the exact same structure as the equivalent function from the Keras extension. As such, refer to subsection 2.7.8.9 (Run Keras model on fold).

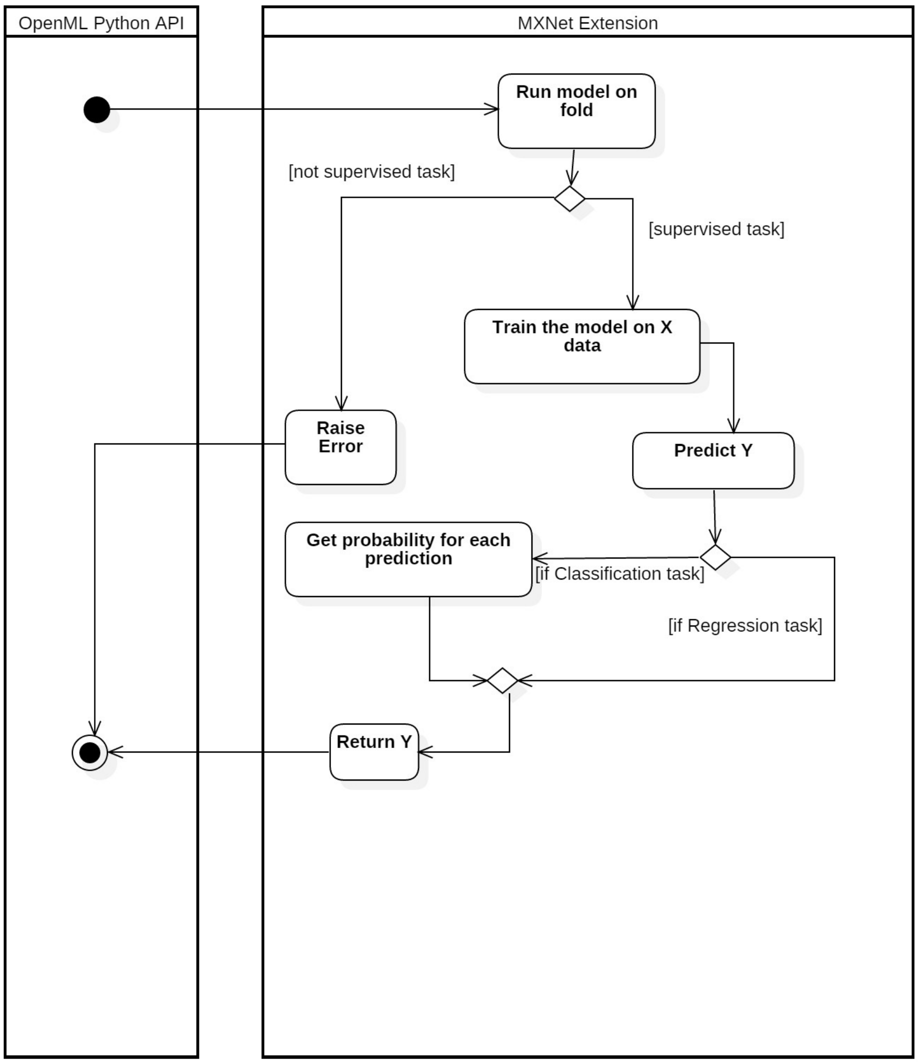


Figure 2.29: Activity Diagram _run_model_on_fold (MXNet)

3 | Specific Requirements

In this chapter a complete list of the software requirements for the project "Sharing deep learning models" is provided. These are *functional* requirements and are denoted by Software Requirements Functional (SRF). They are divided into the following categories: **(to be inserted)**. The MoSCoW method [21] is used to prioritize requirements with the help of four categories. These categories are the following:

- **Must have**: requirements that must be implemented for the product to function properly. These determine whether the product is a Minimum Usable Subset. [22].
- **Should have**: requirements that are important for the project but are not seen as necessary to be implemented for the project to be successfully completed.
- **Could have**: requirements that are seen as optional and would be good to have. These requirements can be implemented if time allows and the more important requirements are already realized.
- **Won't have**: requirements that will not be implemented by the development team in the current project and hence are not part of the scope of this project. They may be fulfilled in future projects.

In total, this chapter contains **101** requirements.

3.1 Extension «interface»

SRF - 1.1 Must have

can_handle_flow(flow): `Boolean`

Checks whether a given OpenML flow is a valid description of a neural network.

Parameters:	*flow*: `OpenMLFlow` - The flow that is being checked.
Returns:	Boolean value indicating whether *flow* is a valid description of a neural network or not.
Precondition:	-
Postcondition:	*True* is returned if *flow* can be handled and *False* otherwise.

SRF - 1.2 Must have

can_handle_model(model): `Boolean`

Checks whether a given machine learning model is an instance of this specific framework model.

Parameters:	*model*: `Any` - The model to be checked.
Returns:	Boolean value indicating whether *model* is an instance of a model of the specific framework.
Precondition:	-
Postcondition:	*True* is returned if *model* is an instance of a model from that specific framework and *False* otherwise.

SRF - 1.3 Must have

flow_to_model(flow, initialize_with_defaults = False): `Any`

Initializes a machine learning model from the given OpenMLFlow object.

Parameters:	*flow*: `OpenMLFlow` - The object to deserialize (can be flow object, or any serialized parameter value that is accepted). *initialize_with_defaults*: Optional[`Boolean`] = *False* - If this flag is set, the hyperparameter values of flows will be ignored and a flow with its defaults is returned.
Returns:	A model of the specific framework.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	The returned model is a correct machine learning model from the corresponding framework.

SRF - 1.4 Must have

model_to_flow(model): `OpenMLFlow`

Transforms a machine learning model to an OpenMLFlow object which can be uploaded to OpenML.

Parameters:	*model*: `Any` - The machine learning model to be serialized to an OpenMLFlow object.
Returns:	Returns an OpenMLFlow object from *model*.
Precondition:	*can_handle_model(model) = True*
Postcondition:	The created *flow* is a valid flow that can be uploaded to OpenML.

SRF - 1.5 Must have

get_version_information(): `List`

Lists the versions of libraries required by an OpenMLFlow object in order for it to be converted into a machine learning model.

Parameters:	-
Returns:	A List object of library versions.
Precondition:	-
Postcondition:	The created list contains the correct version for all necessary libraries.

SRF - 1.6 Must have

create_setup_string(model): `String`

Creates a string which can be used to re-instantiate the given model.

Parameters:	*model*: `Any` - The machine learning model of which the setup string is created.
Returns:	A String containing information about *model*.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The machine learning model can be re-instantiated from the returned string.

SRF - 1.7 Must have

is_estimator(model): `Boolean`

Checks whether the given model is a framework-specific neural network.

Parameters:	*model*: `Any` - The model to be checked.
Returns:	Boolean value indicating whether *model* is an instance of a model of the specific framework.
Precondition:	-
Postcondition:	*True* is returned if *model* is an instance of a model from that specific framework and *False* otherwise.

SRF - 1.8 Must have

seed_model(model, seed = None): `Any`

Sets the random state of all the unseeded components of a model and returns the seeded model. (Not applicable for Keras and ONNX since both have no notion of RandomState)

Parameters:	*model*: `Any` - The machine learning model to be seeded.
	seed: Optional[`int`] = *None* - The seed to initialize the RandomState with. Unseeded sub-components will be seeded with a random number from the RandomState.
Returns:	A model of the specific framework with is unseeded components set with a random state.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The random state of all unseeded components of the model are set correctly.

SRF - 1.9 Must have

obtain_parameter_values(flow, model = None): `List[Dict[String, Any]]`

Extracts all parameter settings required for the OpenMLFlow object from the model.

Parameters:	*flow*: `OpenMLFlow` - OpenMLFlow object (containing flow ids, i.e., it has to be downloaded from the server).
	model: Optional[`Any`] = *None* - The machine learning model from which to obtain the parameter values.
Returns:	A List of dictionaries containing parameter values.
Precondition:	The *model* must match the *flow* signature. If *None*, use the model specified in OpenMLFlow.model.
Postcondition:	The list of dictionaries contains the correct parameter values required by the OpenMLFlow object from the specific *model*.

SRF - 1.10 Must have

instantiate_model_from_hpo_class(model, trace_iteration): `Any`

Instantiates a base model, which can be scanned over by the hyperparameter optimization model.

Parameters:	*model*: `Any` - A hyperparameter optimization model which defines the machine learning model to be instantiated.
	trace_iteration: `OpenMLTraceIteration` - Describes the hyperparameter settings to instantiate.
Returns:	A model of the specific framework that is instantiated from the given *trace_iteration*.
Precondition:	*can_handle_model(model)* = True
Postcondition:	The instantiated base model is valid and can be scanned over through the hyperparameter optimization model.

SRF - 1.11 Must have

_run_model_on_fold(model, task, X_train, rep_no, fold_no, y_train = None, X_test = None): `Tuple`

Run a model on a task and return prediction information.

Parameters:

model: `Any` - The untrained machine learning model to run. The model instance will be copied and not altered.

task: `OpenMLTask` - The task to run the model on.

X_train: `Union[numpy.ndarray, scipy.sparse.spmatrix, pandas.DataFrame]` = *None* - Training data for the given repetition and fold.

rep_no: `int` - The repeat of the experiment (0-based; in case of 1 time Cross-validation (CV), always 0)

fold_no: `int` - The fold number of the experiment (0-based; in case of hold-out, always 0)

y_train: Optional[`numpy.ndarray`] = *None* - Target attributes for supervised tasks. In case of classification, these are integer indices to the potential classes specified by dataset.

X_test: Optional[`Union[numpy.ndarray, scipy.sparse.spmatrix, pandas.DataFrame]`] = *None* - Test attributes to test for generalization in supervised tasks.

Returns: A tuple of: 1.`np.ndarray` - Model predictions; 2.`np.ndarray` - Predicted probabilities (only applicable for supervised classification tasks); 3.`OrderedDict[String, float]` - User defined measures that were generated on this fold; 4.`Optional[OpenMLRunTrace]` - Hyperparameter optimization trace (only applicable for supervised tasks with hyperparameter optimization).

Precondition: *y_train* and *X_test* are set, *can_handle_model(model) = True*, *task* is a classification task or a regression task.

Postcondition: The data returned is sufficient to construct the OpenML run object.

3.2 KerasExtension

SRF - 2.1 Must have

DEPENDENCIES_PATTERN: `Regex`

Regex pattern of the required dependencies.

SRF - 2.2 Must have

SIMPLE_NUMPY_TYPES: `List[Type]`

List that holds numpy types except types in the category "others".

SRF - 2.3 Must have

SIMPLE_TYPES: `Tuple(Type)`

Tuple that holds SIMPLE_NUMPY_TYPES and simple types such as String, int, Boolean.

SRF - 2.4 Must have

LAYER_PATTERN: `Regex`

Regex pattern that describes how a Keras layer is formatted.

SRF - 2.5 Must have

_deserialize_keras(o, components = None, initialize_with_defaults = False, recursion_depth = 0): `Any`

Recursive function to deserialize an OpenMLFlow corresponding to a Keras machine learning model. This function delegates all work to the respective functions to deserialize special data structures etc.

Parameters:	*o*: `Any` - The object to deserialize (can be flow object, or any serialized parameter value that is accepted).
	components: `Dictionary` - Describes the components.
	initialize_with_defaults: Optional[`Boolean`] = False - If this flag is set, the hyperparameter values of flows will be ignored and a flow with its defaults is returned.
	recursion_depth: Optional[`int`] = 0 - The depth at which this flow is called, mostly for debugging purposes.
Returns:	A Keras model.
Precondition:	*o* must be an OpenMLFlow object or any serialized parameter value that is accepted.
Postcondition:	The OpenMLFlow object is correctly deserialized into a Keras model.

SRF - 2.6 Must have

_serialize_keras(o, parent_model): `Any`

Recursive function to serialize a Keras model object.

Parameters:	*o*: `Any` - The object to serialize (can be a model object, or any deserialized parameter value that is accepted).
	parent_model: Optional[`Any`] = None - A reference to the parent model.
Returns:	A serialized model.
Precondition:	*can_handle_model(o) = True*
Postcondition:	The model is correctly serialized and returned.

SRF - 2.7 Must have

_is_keras_flow(flow): `Boolean`

Checks whether the flow was originally a Keras model.

Parameters:	*flow*: `OpenMLFlow` - The OpenMLFlow object which is being checked.
Returns:	A Boolean value depending on whether the *flow* has been originally a Keras model.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	*True* is returned if the flow was originally a Keras model and *False* otherwise.

SRF - 2.8 Must have

_serialize_model(model): `OpenMLFlow`

Calls *model_to_flow* (for Keras) recursively (mutual recursion) to properly serialize the parameters to strings and the components (other models) to OpenML flows.

Parameters:	*model*: `Any` - Keras machine learning model.
Returns:	An OpenMLFlow object
Precondition:	*can_handle_model(model) = True.*
Postcondition:	A valid OpenMLFlow object is returned.

SRF - 2.9 Must have

_get_external_version_string(model, sub_components): `String`

Creates the external version string for a flow, given the model and the already parsed dictionary of sub-components. Retrieves the external version of all sub-components, which themselves already contain all requirements for their sub-components. The external version string is a sorted concatenation of all modules which are present in the run.

Parameters:	*model*: `Any` - The model from which the string is being created. *sub_components*: `Dict[String, OpenMLFlow]` - A dictionary containing the sub-components of the model.
Returns:	A String containing the external version of all sub-components of the *model*.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The string is correctly created for the flow from the model and its sub-components.

SRF - 2.10 Must have

_get_parameters(model): `OrderedDict`

Constructs a dictionary with the parameters of the Keras model.

Parameters:	*model*: `Any` - The Keras machine learning model from which the parameters are obtained.
Returns:	An OrderedDict containing the parameters of the *model*.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The parameters of the serialized Keras model are returned in an ordered dictionary format.

SRF - 2.11 Must have

_extract_information_from_model(model): `Tuple`

Extracts all necessary information (parameters and sub-components) for the OpenMLFlow object from the model.

Parameters:	*model*: `Any` - The Keras machine learning model from which the information is being extracted.
Returns:	A Tuple of 4 elements: 1) Ordered dictionary with string key and optional string value, 2) ordered dictionary with string key and optional dictionary value, 3)ordered dictionary with string key and OpenMLFlow, 4) a Set.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The information returned is sufficient for the OpenMLFlow object creation or model to flow conversion.

SRF - 2.12 Must have

_deserialize_model(flow, keep_defaults, recursion_depth): `Any`

Deserializes an OpenML flow into a Keras model.

Parameters:	*flow*: `OpenMLFlow` - OpenML flow being deserialized. *keep_defaults*: `Boolean` - Indicates whether default values should be kept or not. *recursion_depth*: `int` - Stores the recursion depth.
Returns:	A Keras model.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	*flow* is correctly deserialized to a Keras model.

SRF - 2.13 Must have

_check_dependencies(dependencies): `None`

Checks whether the dependencies required for the deserialization of an OpenMLFlow object are met.

Parameters:	*dependencies*: `String` - A string representing the required dependencies.
Returns:	None
Precondition:	-
Postcondition:	Raises an error if the dependencies cannot be read, the current versions are lower than required or the dependencies are not present or installed.

SRF - 2.14 Must have

_format_external_version(model_package_name, model_package_version_number): `String`

Returns a formatted string representing the necessary dependencies for a flow.

Parameters:	*model_package_name*: `String` - The name of the required package. *model_package_version_number*: `String` - The version of the required package.
Returns:	A String representing the necessary dependencies for an OpenMLFlow object.
Precondition:	-
Postcondition:	The returned string contains the required dependencies for a flow.

SRF - 2.15 Must have

_openml_param_name_to_keras(openml_parameter, flow): `String`

Converts the name of an OpenMLParameter into the Keras name, given a flow.

Parameters:	*openml_parameter*: `OpenMLParameter` - The parameter under consideration.
	flow: `OpenMLFlow` - The flow that provides context.
Returns:	A String containing the name of the structure of the *flow* and the name of the *openml_parameter*.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	The returned string is a Keras name, converted from the OpenMLParameter and the flow.

SRF - 2.16 Must have

_from_parameters(parameters): `Any`

Get a Keras model from flow parameters. Create a dictionary and recursively fill it with model components. First this is done for non-layer items, then layer items.

Parameters:	*parameters*: `Ordered dictionary` - Flow parameters from which the Keras model is retrieved.
Returns:	A Keras model
Precondition:	-
Postcondition:	A Keras model is generated from the passed OpenMLFlow parameters.

3.3 PyTorchExtension

SRF - 3.1 Must have

DEPENDENCIES_PATTERN: `Regex`

Regex pattern of the required dependencies.

SRF - 3.2 Must have

SIMPLE_NUMPY_TYPES: `List[Type]`

List that holds numpy types except types in the category "others".

SRF - 3.3 Must have

SIMPLE_TYPES: `Tuple(Type)`

Tuple that holds SIMPLE_NUMPY_TYPES and simple types such as String, int, Boolean.

SRF - 3.4 Must have

_deserialize_pytorch(o, components = None, initialize_with_defaults = False, recursion_depth = 0): `Any`

Recursive function to deserialize an OpenMLFlow corresponding to a PyTorch machine learning model. This function delegates all work to the respective functions to deserialize special data structures etc.

Parameters:	*o*: `Any` - The object to deserialize (can be flow object, or any serialized parameter value that is accepted).
	components: `Dictionary` - Describes the components.
	initialize_with_defaults: Optional[`Boolean`] = False - If this flag is set, the hyperparameter values of flows will be ignored and a flow with its defaults is returned.
	recursion_depth: Optional[`int`] = 0 - The depth at which this flow is called, mostly for debugging purposes.
Returns:	A PyTorch model.
Precondition:	*o* must be an OpenMLFlow object or any serialized parameter value that is accepted.
Postcondition:	The OpenMLFlow object is correctly deserialized into a PyTorch model.

SRF - 3.5 Must have

_serialize_pytorch(o, parent_model): `Any`

Recursive function to serialize a PyTorch model object.

Parameters:	*o*: `Any` - The object to serialize (can be a model object, or any deserialized parameter value that is accepted). *parent_model*: Optional[`Any`] = None - A reference to the parent model.
Returns:	A serialized model.
Precondition:	*can_handle_model(o) = True*
Postcondition:	The model is correctly serialized and returned.

SRF - 3.6 Must have

_is_pytorch_flow(flow): `Boolean`

Checks whether the flow was originally a PyTorch model.

Parameters:	*flow*: `OpenMLFlow` - The OpenMLFlow object which is being checked.
Returns:	A Boolean value depending on whether the *flow* has been originally a PyTorch model.
Precondition:	*can_handle_flow(flow) = True.*
Postcondition:	*True* is returned if the flow was originally a PyTorch model and *False* otherwise.

SRF - 3.7 Must have

_serialize_model(model): `OpenMLFlow`

Calls *model_to_flow* (for PyTorch) recursively (mutual recursion) to properly serialize the parameters to strings and the components (other models) to OpenML flows.

Parameters:	*model*: `Any` - PyTorch estimator.
Returns:	An OpenMLFlow object.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	A valid OpenMLFlow object is returned.

SRF - 3.8 Must have

_get_external_version_string(model, sub_components): `String`

Creates external version string for a flow, given the model and the already parsed dictionary of sub-components. Retrieves the external version of all sub-components, which themselves already contain all requirements for their sub-components. The external version string is a sorted concatenation of all modules which are present in the run.

Parameters:	*model*: `Any` - The model from which the string is being created. *sub_components*: `Dict[String, OpenMLFlow]` - A dictionary containing the sub-components of the model.
Returns:	A String containing the external version of all sub-components of the *model*.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The string is correctly created for the flow from the model and its sub-components.

SRF - 3.9 Must have

_check_multiple_occurence_of_component_in_flow(model, sub_components): `None`

Checks if there is more than one occurrence of a component in a flow.

Parameters:	*model*: `Any` - The model whose components are checked. *sub_components*: `Dict[String, OpenMLFlow]` - A dictionary containing the sub-components of the model.
Returns:	-
Precondition:	*can_handle_model(model) = True.*
Postcondition:	There are no double occurrences of any of the components, otherwise a value error is raised.

SRF - 3.10 Must have

_is_container_module(module): `Boolean`

Checks if a module is a Sequential container, ModuleDict container or a ModuleList container.

Parameters:	*module*: `torch.nn.Module` - The module being checked.
Returns:	A Boolean value indicating if *module* is a Sequential container, ModuleDict container or a ModuleList container.
Precondition:	-
Postcondition:	*True* is returned if module is a Sequential container, ModuleDict container or a ModuleList container, otherwise *False* is returned.

SRF - 3.11 Must have

_get_module_hyperparameters(module, parameters): `Dict[String, Any]`

Finds the hyperparameters of a module.

Parameters:	*module*: `torch.nn.Module` - The module whose hyper-parameters are obtained. *parameters*: `Dict[String, torch.nn.Parameter]` - The parameters of the module.
Returns:	A Dictionary of the hyperparameters of *module*.
Precondition:	-
Postcondition:	The returned dictionary has all the hyperparameters of the module.

SRF - 3.12 Must have

_get_module_descriptors(module, deep = True): `Dict[String, Any]`

Finds the descriptors of a module.

Parameters:	*module*: `torch.nn.Module` - The module whose descriptors are needed. *deep*: `Boolean` = True - This flag is set to *True* only if the hyper-parameters of the *module* need to be included. If the hyper-parameters and the sub-modules are included (used for verificatrion performed by OpenML) it is set to *False*.
Returns:	A Dict of the module descriptors.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The returned dictionary has all the required descriptors of the module.

SRF - 3.13 Must have

_extract_information_from_model(model): `Tuple`

Extracts all necessary information (parameters and sub-components) for the OpenMLFlow object from the model.

Parameters:	*model*: `Any` - The PyTorch machine learning model from which the information is being extracted.
Returns:	A Tuple of 4 elements: Ordered dictionary with string key, and optional string value, ordered dictionary with string key and optional dictionary value, ordered dictionary with string key and OpenMLFlow and a Set.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The information returned is sufficient for the OpenMLFlow object creation or model to flow conversion.

SRF - 3.14 Must have

_get_fn_arguments_with_defaults(fn_name): `Tuple[Dict, Set]`

Obtains all the parameter names of a function together with their default values.

Parameters:	*fn_name*: `Callable` - The function of which the defaults are obtained.
Returns:	A Tuple consisting of a dictionary mapping a parameter name to the default value and a set with all parameters that do not have a default value.
Precondition:	-
Postcondition:	All the parameter names are found and their defaults are set correctly.

SRF - 3.15 Must have

_deserialize_model(flow, keep_defaults, recursion_depth): `Any`

Deserializes an OpenML flow into a PyTorch model.

Parameters:	*flow*: `OpenMLFlow` - The OpenML flow being serialized.
	keep_defaults: `Boolean` - Indicates whether default values should be kept or not.
	recursion_depth: `int` - Stores the recursion depth.
Returns:	A PyTorch model.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	*flow* is correctly deserialized into a PyTorch model.

SRF - 3.16 Must have

_check_dependencies(dependencies): `None`

Checks whether the dependencies required for the deserialization of an OpenMLFlow object are met.

Parameters:	*dependencies*: `String` - A string representing the required dependencies.
Returns:	None
Precondition:	-
Postcondition:	Raises an error if the dependencies cannot be read, the current versions are lower than required or the dependencies are not present or installed.

SRF - 3.17 Must have

_serialize_type(o): `OrderedDict[String, String]`

Converts a given type into a string representation of it.

Parameters:	*o*: `Any` - The type being serialized.
Returns:	An OrderedDict containing the serialized type as a string.
Precondition:	-
Postcondition:	The type matches the string.

SRF - 3.18 Must have

_deserialize_type(o): `Any`

Converts a string representation of a type into an instance of that type.

Parameters:	*o*: `String` - The type being serialized.
Returns:	A type deserialized from *o*.
Precondition:	-
Postcondition:	The string matches the type.

SRF - 3.19 Must have

_serialize_function(o): `OrderedDict[String, String]`

Converts a given function *o* into a string representation of that function.

Parameters:	*o*: `Callable` - Function to be serialized.
Returns:	An OrderedDict containing the serialized function as a string.
Precondition:	-
Postcondition:	The serialized string matches the function *o*.

SRF - 3.20 Must have

_deserialize_function(name): `Callable`

Converts a given string name of a function into the function.

Parameters:	*o*: `Callable` - Function to be serialized.
Returns:	A Callable object representing the deserialized function.
Precondition:	-
Postcondition:	The deserialized function matches the string representation.

SRF - 3.21 Must have

_serialize_methoddescriptor(o): `OrderedDict[str, str]`

Creates a string representation of a method descriptor.

Parameters:	*o*: `Any` - Method descriptor.
Returns:	An OrderedDict of the serialized method descriptor *o*.
Precondition:	*inspect.ismethoddescriptor(o) = True*
Postcondition:	The serialized version of the method descriptor can be used by the function *_deserialize_methoddescriptor(name)* to recover the method descriptor.

SRF - 3.22 Must have

_deserialize_methoddescriptor(name): `Any`

Turns a string representation of a method descriptor to a valid method descriptor.

Parameters:	*name*: `String` - A string representing a method descriptor
Returns:	A a method descriptor.
Precondition:	*name* is a valid string representation of a method descriptor.
Postcondition:	The *name* is correctly deserialized into a valid method descriptor.

SRF - 3.23 Must have

_format_external_version(model_package_name, model_package_version_number): `String`

Returns a formatted string representing the necessary dependencies for a flow.

Parameters:	*model_package_name*: `String` - The name of the required package.
	model_package_version_number: `String` - The version of the required package.
Returns:	A String representing the necessary dependencies for an OpenMLFlow object.
Precondition:	-
Postcondition:	The returned string contains the required dependencies for a flow.

SRF - 3.24 Must have

_get_parameter_values_recursive(param_grid, parameter_name): `List[Any]`

Creates a list of values for a given hyperparameter, encountered recursively throughout the flow. (e.g., n_jobs can be defined for various flows).

Parameters:	*param_grid*: `Union[Dict, List[Dict]]` - Dictionary mapping from a hyperparameter list to value, to a list of such dictionaries.
	parameter_name: `String` - The hyperparameter that needs to be inspected.
Returns:	A list of all values of hyperparameters with the name of *parameter_name*.
Precondition:	-
Postcondition:	All the values of the hyperparameters with the given *parameter_name* are included in the list.

SRF - 3.25 Must have

_openml_param_name_to_pytorch(openml_parameter, flow): `String`

Converts the name of an OpenMLParameter into the PyTorch name, given a flow.

Parameters:	*openml_parameter*: `OpenMLParameter` - The parameter under consideration.
	flow: `OpenMLFlow` - The flow that provides context.
Returns:	A String containing the name of the structure of the *flow* and the name of the *openml_parameter*.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	The returned string is a PyTorch name, converted from the OpenMLParamter and the flow.

SRF - 3.26 Must have

_default_criterion_gen(task): `torch.nn.Module`

Returns a criterion based on the task type.

Parameters:	*task*: `OpenMLTask` - The task whose criterion is returned.
Returns:	A torch.nn.Module as a criterion of the task.
Precondition:	-
Postcondition:	If *task* is a regression task - torch.nn.SmoothL1Loss is returned, if *task* is a classification task - torch.nn.CrossEntropyLoss is returned.

SRF - 3.27 Must have

_default_optimizer_gen(model, _): `torch.optim.Optimizer`

Returns an optimizer for the given model.

Parameters:	*model*: `torch.nn.Module` - The model whose optimizer is returned. _: `OpenMLTask` - Task which the user can pass as a parameter if creating their own implementation of this function.
Returns:	Returns the torch.optim.Adam optimizer for *model*.
Precondition:	-
Postcondition:	The returned optimizer is correct given the parameters of *model*.

SRF - 3.28 Must have

_default_scheduler_gen(optim, _): `Any`

Returns a scheduler for the given optimizer.

Parameters:	*optim*: `torch.optim.Optimizer` - The model whose optimizer is returned. _: `OpenMLTask` - Task which the user can pass as a parameter if creating their own implementation of this function.
Returns:	Returns the torch.optim.lr_scheduler.ReduceLROnPlateau optimizer for *optim*.
Precondition:	-
Postcondition:	The returned scheduler is correct given the optimizer.

SRF - 3.29 Must have

_default_predict(output, task): `torch.Tensor`

Returns predictions given the outputs of the model.

Parameters:	*output*: `torch.Tensor` - The output tensor. *task*: `OpenMLTask` - The task that needs to be performed.
Returns:	A torch.Tensor representing the prediction.
Precondition:	-
Postcondition:	If *task* is a classification task, the *argmax* of the *output* is returned, if *task* is a regression task the prediction is flattened.

SRF - 3.30 Must have

_default_predict_proba(output): `torch.Tensor`

Returns predictions given the outputs of the model as a tensor of probabilities for each class.

Parameters:	*output*: `torch.Tensor` - The output tensor.
Returns:	A torch.Tensor representing the prediction, where all the entries are probabilities in the interval [0,1].
Precondition:	-
Postcondition:	The returned tensor is obtained by the *output* using "softmax".

SRF - 3.31 Must have

_default_sanitize(tensor): `torch.Tensor`

Sanitizes the input data in order to ensure that models can be trained safely.

Parameters:	*tensor*: `torch.Tensor` - The input tensor.
Returns:	A torch.Tensor representing the sanitized version of *tensor*.
Precondition:	-
Postcondition:	The returned tensor has no unassigned values.

SRF - 3.32 Must have

_default_retype_labels(tensor, task): `torch.Tensor`

Changes the types of the labels in order to ensure type compatibility.

Parameters:	*tensor*: `torch.Tensor` - The input tensor.
	task: `OpenMLTask` - The task that needs to be performed.
Returns:	A torch.Tensor with compatible label types.
Precondition:	-
Postcondition:	If *task* is a classification task the labels are turned into a torch.(cuda.)LongTensor, if *task* is a regression task the labels are turned into a torch.(cuda.)FloatTensor.

SRF - 3.33 Must have

_default_progress_callback(fold, rep, epoch, step, loss, accuracy): `None`

Reports the current progress when a training step is finished.

Parameters:	*fold*: `int` - The current fold.
	rep: `int` - Number of repetitions done so far.
	epoch: `int` - The current epoch.
	step: `int` - The current step.
	loss: `float` - The loss of this training step.
	accuracy: `float` - The accuracy of this training step.
Returns:	None
Precondition:	A training step is finished.
Postcondition:	The progress is reported correctly.

3.4 ONNXExtension

SRF - 4.1 Must have

DEPENDENCIES_PATTERN: `Regex`

Regex pattern of the required dependencies.

SRF - 4.2 Must have

ONNX_FILE_PATH: `String`

String that contains the path to the ONNX library.

SRF - 4.3 Must have

ONNX_ATTR_TYPES: `Dictionary`

A dictionary containing the ONNX attribute tasks.

SRF - 4.4 Must have

_deserialize_onnx(flow, initialize_with_defaults = False): `Any`

A function to deserialize an OpenMLFlow corresponding to an ONNX specification.

Parameters:	*flow*: `OpenMLFlow` - The object to deserialize.
	initialize_with_defaults: Optional[`Boolean`] = False - If this flag is set, the hyperparameter values of flows will be ignored and a flow with its defaults is returned.
Returns:	An ONNX specification.
Precondition:	*flow* must be an OpenMLFlow object.
Postcondition:	The OpenMLFlow object is correctly deserialized into an ONNX specification.

SRF - 4.5 Must have

_serialize_onnx(model): `Any`

A function to serialize an ONNX specification object.

Parameters:	*model*: `Any` - The object to serialize (can be a specification object, or any deserialized parameter value that is accepted).
Returns:	A serialized specification.
Precondition:	*can_handle_model(o) = True*
Postcondition:	The specification is correctly serialized and returned.

SRF - 4.6 Must have

_is_onnx_flow(flow): `Boolean`

Checks whether the flow was originally a ONNX specification.

Parameters:	*flow*: `OpenMLFlow` - The OpenMLFlow object which is being checked.
Returns:	A Boolean value depending on whether the *flow* has been originally an ONNX specification.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	*True* is returned if the flow was originally a ONNX specification and *False* otherwise.

SRF - 4.7 Must have

_get_external_version_string(model, sub_components): `String`

Creates external version string for a flow, given the specification and the already parsed dictionary of sub-components. Retrieves the external version of all sub-components, which themselves already contain all requirements for their sub-components. The external version string is a sorted concatenation of all specifications which are present in the run.

Parameters:	*model*: `Any` - The specification from which the string is being created. *sub_components*: `Dict[String, OpenMLFlow]` - A dictionary containing the sub-components of the specification.
Returns:	A String containing the external version of all sub-components of the specification.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The string is correctly created for the flow from the specification and its sub-components.

SRF - 4.8 Must have

_get_parameters(model): `Ordered dictionary`

Constructs a dictionary with the parameters of the ONNX specification.

Parameters:	*model*: `Any` - The ONNX machine learning specification from which the parameters are obtained.
Returns:	An OrderedDictionary containing the parameters of the specification.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The parameters of the serialized ONNX specification are returned in an ordered dictionary format.

SRF - 4.9 Must have

_check_dependencies(dependencies): `None`

Checks whether the dependencies required for the deserialization of an OpenMLFlow object are met.

Parameters:	*dependencies*: `String` - A string representing the required dependencies.
Returns:	None
Precondition:	-
Postcondition:	Raises an error if the dependencies cannot be read, the current versions are lower than required or the dependencies are not present or installed.

SRF - 4.10 Must have

_format_external_version(model_package_name, model_package_version_number): `String`

Returns a formatted string representing the necessary dependencies for a flow.

Parameters:	*model_package_name*: `String` - The name of the required package. *model_package_version_number*: `String` - The version of the required package.
Returns:	A String representing the necessary dependencies for an OpenMLFlow object.
Precondition:	-
Postcondition:	The returned string contains the required dependencies for a flow.

SRF - 4.11 Must have

_default_criterion_gen(task): `gluon.loss.Loss`

Returns a criterion based on the task type.

Parameters:	*task*: `OpenMLTask` - The task whose criterion is returned.
Returns:	A gluon.loss.Loss as a criterion of the task.
Precondition:	-
Postcondition:	If *task* is a regression task - gluon.loss.L2Loss is returned, if *task* is a classification task - gluon.loss.SoftmaxCrossEntropyLoss is returned.

3.5 MXNetExtension

SRF - 5.1 Must have

DEPENDENCIES_PATTERN: `Regex`

Regex pattern of the required dependencies.

SRF - 5.2 Must have

SIMPLE_NUMPY_TYPES: `List[Type]`

List that holds numpy types except types in the category "others".

SRF - 5.3 Must have

SIMPLE_TYPES: `Tuple(Type)`

Tuple that holds SIMPLE_NUMPY_TYPES and simple types such as String, int, Boolean.

SRF - 5.4 Must have

_serialize_mxnet(o, parent_model): `Any`

Recursive function to serialize a MXNet model object.

Parameters:	*o*: `Any` - The object to serialize (can be a model object, or any deserialized parameter value that is accepted).
	parent_model: Optional[`Any`] = None - A reference to the parent model.
Returns:	A serialized model.
Precondition:	*can_handle_model(o) = True*
Postcondition:	The model is correctly serialized and returned.

SRF - 5.5 Must have

_is_mxnet_flow(flow): `Boolean`

Checks whether the *flow* was originally a MXNet model.

Parameters:	*flow*: `OpenMLFlow` - The OpenMLFlow object which is being checked.
Returns:	A Boolean value depending on whether the *flow* has been originally a MXNet model.
Precondition:	*can_handle_flow(flow) = True.*
Postcondition:	*True* is returned if the flow was originally a MXNet model and *False* otherwise.

SRF - 5.6 Must have

_serialize_model(model): `OpenMLFlow`

Calls *model_to_flow* (for MXNet) recursively (mutual recursion) to properly serialize the parameters to strings and the components (other models) to OpenML flows.

Parameters:	*model*: `Any` - MXNet network.
Returns:	An OpenMLFlow object.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	A valid OpenMLFlow object is returned.

SRF - 5.7 Must have

_get_external_version_string(model, sub_components): `String`

Creates external version string for a flow, given the model and the already parsed dictionary of sub-components. Retrieves the external version of all sub-components, which themselves already contain all requirements for their sub-components. The external version string is a sorted concatenation of all modules which are present in the run.

Parameters:	*model*: `Any` - The model from which the string is being created.
	sub_components: `Dict[String, OpenMLFlow]` - A dictionary containing the sub-components of the model.
Returns:	A String containing the external version of all sub-components of the *model*.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The string is correctly created for the flow from the model and its sub-components.

SRF - 5.8 Must have

_get_symbolic_configuration(model): `OrderedDict[str, Any]`

Extracts the nodes of an MXNet model and their order and creates a configuration of the model in the form of an ordered dictionary.

Parameters:	*model*: `mxnet.gluon.HybridBlock` - The model from which the string is being created.
Returns:	An OrderedDict representing the configuration of the *model*.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The MXNet model can be reconstructed from the returned configuration.

SRF - 5.9 Must have

_from_symbolic_configuration(configuration): `mxnet.gluon.HybridBlock`

Retrieves the MXNet model from its configuration.

Parameters:	*configuration*: `OrderedDict[str, Any]` - The configuration from which the model is retrieved.
Returns:	A mxnet.gluon.HybridBlock representing the MXNet model.
Precondition:	-
Postcondition:	The returned model is correctly retrieved from the configuration.

SRF - 5.10 Must have

_extract_information_from_model(model): `Tuple`

Extracts all necessary information (parameters and sub-components) for the OpenMLFlow object from the model.

Parameters:	*model*: `Any` - The MXNet machine learning model from which the information is being extracted.
Returns:	A Tuple of 4 elements: Ordered dictionary with string key, and optional string value, ordered dictionary with string key and optional dictionary value, ordered dictionary with string key and OpenMLFlow and a Set.
Precondition:	*can_handle_model(model) = True.*
Postcondition:	The information returned is sufficient for the OpenMLFlow object creation or model to flow conversion.

SRF - 5.11 Must have

_deserialize_model(flow, keep_defaults): `Any`

Deserializes an OpenML flow into a MXNet model.

Parameters:	*flow*: `OpenMLFlow` - The OpenML flow being serialized.
	keep_defaults: `Boolean` - Indicates whether default values should be kept or not.
Returns:	A MXNet model.
Precondition:	*can_handle_flow(flow) = True*
Postcondition:	*flow* is correctly deserialized into a MXNet model.

SRF - 5.12 Must have

_check_dependencies(dependencies): `None`

Checks whether the dependencies required for the deserialization of an OpenMLFlow object are met.

Parameters:	*dependencies*: `String` - A string representing the required dependencies.
Returns:	None
Precondition:	-
Postcondition:	Raises an error if the dependencies cannot be read, the current versions are lower than required or the dependencies are not present or installed.

SRF - 5.13 Must have

_format_external_version(model_package_name, model_package_version_number): `String`

Returns a formatted string representing the necessary dependencies for a flow.

Parameters:	*model_package_name*: `String` - The name of the required package. *model_package_version_number*: `String` - The version of the required package.
Returns:	A String representing the necessary dependencies for an OpenMLFlow object.
Precondition:	-
Postcondition:	The returned string contains the required dependencies for a flow.

SRF - 5.14 Must have

_default_criterion_gen(task): `mxnet.gluon.loss.Loss`

Returns a criterion based on the task type.

Parameters:	*task*: `OpenMLTask` - The task whose criterion is returned.
Returns:	A mxnet.gluon.loss.Loss as a criterion of the task.
Precondition:	-
Postcondition:	If *task* is a regression task - mxnet.gluon.loss.L1Loss is returned, if *task* is a classification task - mxnet.gluon.loss.SoftmaxCrossEntropyLoss is returned.

SRF - 5.15 Must have

_default_optimizer_gen(lr_scheduler, _): `torch.optim.Optimizer`

Returns an optimizer for the given task.

Parameters:	*lr_scheduler*: `mxnet.lr_scheduler.LRScheduler` - The scheduler that can be used for a given task. _: `OpenMLTask` - Task which the user can pass as a parameter if creating their own implementation of this function.
Returns:	Returns the mxnet.optimizer.Adam optimizer for *model*.
Precondition:	-
Postcondition:	The returned optimizer is correct for the given task.

SRF - 5.16 Must have

_default_scheduler_gen(_): `Optional[mxnet.lr_scheduler.LRScheduler]`

Returns the None scheduler for a given task.

Parameters:	_: `OpenMLTask` - Task which the user can pass as a parameter if creating their own implementation of this function.
Returns:	Returns a None mxnet.lr_scheduler.LRScheduler scheduler for a given task.
Precondition:	-
Postcondition:	The returned scheduler is None.

SRF - 5.17 Must have

_default_predict(output, task): `Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol]`

Returns predictions given the outputs of the model.

Parameters:	*output*: `Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol]` - The output of the model. *task*: `OpenMLTask` - The task that needs to be performed.
Returns:	A Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol] representing the prediction.
Precondition:	-
Postcondition:	If *task* is a classification task, the *argmax* of the *output* is returned, if *task* is a regression task the prediction is flattened.

SRF - 5.18 Must have

_default_predict_proba(output): `Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol]`

Returns predictions given the outputs of the model as probabilities for each class.

Parameters:	*output*: `Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol]` - The output of the model.
Returns:	A Union[mxnet.ndarray.NDArray, mxnet.symbol.Symbol] representing the prediction, where all the entries are probabilities in the interval [0,1].
Precondition:	-
Postcondition:	The returned output is obtained by the *output* using "softmax".

SRF - 5.19 Must have

_default_sanitize(output): `mxnet.ndarray.NDArray`

Sanitizes the input data in order to ensure that models can be trained safely.

Parameters:	*output*: `mxnet.ndarray.NDArray` - The input data.
Returns:	A mxnet.ndarray.NDArray representing the sanitized version of the input data.
Precondition:	-
Postcondition:	The returned output has no unassigned values.

SRF - 5.20 Must have

_default_metric_gen(task): `mxnet.metric.EvalMetric`

Returns a composite metric for the given task.

Parameters:	*task*: `OpenMLTask` - The task that needs to be performed.
Returns:	A mxnet.metric.EvalMetric with the metrics for the task.
Precondition:	-
Postcondition:	If *task* is a classification task, the metric is composed of accuracy, if *task* is a classification task the metric is composed of mean squared error, mean absolute error and root mean squared error.

SRF - 5.21 Must have

_default_progress_callback(fold, rep, epoch, step, loss, metric): `None`

Reports the current progress when a training step is finished.

Parameters:	*fold*: `int` - The current fold.
	rep: `int` - Number of repetitions done so far.
	epoch: `int` - The current epoch.
	step: `int` - The current step.
	loss: `mxnet.ndarray.NDArray` - The loss of this training step.
	metric: `mxnet.metric.EvalMetric` - The metric of this training step.
Returns:	None
Precondition:	A training step is finished.
Postcondition:	The progress is reported correctly.

3.6 Visualization

SRF - 6.1 Must have

run: `OpenMLRun`

Stores the run to be visualized.

SRF - 6.2 Must have

flow: `OpenMLFlow`

Stores the flow to be visualized.

SRF - 6.3 Must have

view: `View`

The corresponding HTML rendered page that is shown for the visualization of the flow and run.

SRF - 6.4 Must have

load_run(*run_id*): `Tuple[String, List, List, String]`

This function gets the *run* from the OpenML website given the *run_id*. Then it extracts and stores all the necessary data that is needed to visualize the *run* by using *get_training_data(url, run_id)*.

Parameters:	*run_id*: `int` - The id of the *run* that is to be visualized.
Returns:	Returns a tuple containing the data extracted from the *run*, a list indicating the presence of errors, a list of visualizable metrics for the run depending on the type of task, and the default metric to be displayed.
Precondition:	There exists a run on the OpenML website with an id = *run_id*. The run contains the associated training data.
Postcondition:	The data extracted from the downloaded *run* is correct and stored correctly.

SRF - 6.5 Must have

load_flow(*flow_id*): `Tuple[String, List]`

This function gets the *flow* from the OpenML website given the *flow_id*. Then it extracts and stores all the necessary data that is needed to visualize the *flow*. Furthermore, this function uses *get_onnx_model(flow.model) to create the ONNX specification which is later visualized.*

Parameters:	*flow_id*: `int` - The id of the *flow* that is to be visualized.
Returns:	Returns a tuple containing the data extracted from the *flow* and a list indicating the presence of errors.
Precondition:	There exists a OpenML flow on the OpenML website with an id = *flow_id*. The model associated with the flow can be converted into an ONNX representation.
Postcondition:	The data extracted from the downloaded *flow* is correct and stored correctly.

SRF - 6.6 Must have

update_flow_graph(*n_clicks*, *flow_data_json*, *nr_loads*): `Tuple[html.Iframe, int]`

Displays the flow in the window for visualization.

Parameters:	*n_clicks*: `int` - Represents the number of times the "Load flow" button has been clicked.
	flow_data_json: `String` - Contains the data to be visualized.
	nr_loads: `int` - How many times a *flow* has been loaded.
Returns:	A tuple containing the `html.Iframe` that contains the visualized layers of the flow and *flow* id.
Precondition:	The *flow_data_json* contains the information needed to visualize the flow.
Postcondition:	The flow is correctly visualized.

SRF - 6.7 Must have

update_run_graph(*n_clicks*, *metric*, *flow_data_json*, *nr_loads*): `Tuple[Figure, String]`

Displays a graph for the selected metric in the window for visualization.

Parameters:	*n_clicks*: `int` - Represents the number of times the "Load run" button has been clicked.
	metric: `String` - The metric to be visualized.
	run_data_json: `String` - Contains the data to be visualized.
	nr_loads: `int` - How many times a *run* has been loaded.
Returns:	Return a tuple with the figure and metric to be displayed.
Precondition:	The *run_data_json* is not empty and there is a valid selected metric.
Postcondition:	The run is correctly visualized.

SRF - 6.8 Must have

get_training_data(*url*, *run_id*): `pandas.DataFrame`

Used in **SRF** - 6.4 to extract the training data for the *run*.

Parameters:	*url*: `String` - Represents the *url* from where the training data for the run can be found.
	run_id: `int` - The id of the *run* that is to be visualized.
Returns:	Return the training data of the *run* with run id = *run_id*.
Precondition:	There exists a run on the OpenML website with an id = *run_id*. The run contains the associated training data and the *url* must be a valid url leading to the training data of the *run*.
Postcondition:	The training data is correctly extracted.

SRF - 6.9 Must have

get_onnx_model(*model*): `onnx.ModelProto`

Used in **SRF** - 6.5 to create an ONNX representation from the given *model*.

Parameters:	*model*: `Any` - The *model* to be represented as an ONNX representation.
Returns:	Returns an ONNX representation of the *model*.
Precondition:	The *model* is of type `keras.model.Model`, `torch.nn.Module`, `mx.gluon.nn.HybridBlock`, `onnx.ModelProtol`.
Postcondition:	The ONNX specification is correctly instantiated and corresponds to the model.

4 | Requirements Traceability Matrix

4.1 Software Requirements to User Requirements

SRF	**URF**
SRF - 1.1	**URF** - 02, **URF** - 04, **URF** - 06, **URF** - 08, **URF** - 10, **URF** - 12, **URF** - 14, **URF** - 16
SRF - 1.2	**URF** - 01, **URF** - 03, **URF** - 05, **URF** - 07, **URF** - 09, **URF** - 11, **URF** - 13, **URF** - 15
SRF - 1.3	**URF** - 02, **URF** - 04, **URF** - 06, **URF** - 08, **URF** - 10, **URF** - 12, **URF** - 14, **URF** - 16, **URF** - 17
SRF - 1.4	**URF** - 01, **URF** - 03, **URF** - 05, **URF** - 07, **URF** - 09, **URF** - 11, **URF** - 13, **URF** - 15, **URF** - 17
SRF - 1.5	**URF** - 20, **URF** - 21, **URF** - 22, **URF** - 23, **URF** - 24, **URF** - 25
SRF - 1.6	**URF** - 20, **URF** - 21, **URF** - 22, **URF** - 23, **URF** - 24, **URF** - 25
SRF - 1.7	**URF** - 01, **URF** - 03, **URF** - 05, **URF** - 07, **URF** - 09, **URF** - 11, **URF** - 13, **URF** - 15, **URF** - 20, **URF** - 21, **URF** - 22, **URF** - 23, **URF** - 24, **URF** - 25
SRF - 1.8	**URF** - 22, **URF** - 23
SRF - 1.9	**URF** - 20, **URF** - 21, **URF** - 22, **URF** - 23, **URF** - 24, **URF** - 25
SRF - 1.10	**URF** - to be added in URD
SRF - 1.11	**URF** - 20, **URF** - 21, **URF** - 22, **URF** - 23, **URF** - 24, **URF** - 25
SRF - 2.1	**URF** - 04
SRF - 2.2	**URF** - 03
SRF - 2.3	**URF** - 03
SRF - 2.4	**URF** - 04
SRF - 2.5	**URF** - 04
SRF - 2.6	**URF** - 03
SRF - 2.7	**URF** - 04
SRF - 2.8	**URF** - 03
SRF - 2.9	**URF** - 03
SRF - 2.10	**URF** - 03
SRF - 2.11	**URF** - 03
SRF - 2.12	**URF** - 04
SRF - 2.13	**URF** - 04

SRF - 2.14	**URF** - 03
SRF - 2.15	**URF** - 04
SRF - 2.16	**URF** - 04
SRF - 3.1	**URF** - 06
SRF - 3.2	**URF** - 05
SRF - 3.3	**URF** - 05
SRF - 3.4	**URF** - 06
SRF - 3.5	**URF** - 05
SRF - 3.6	**URF** - 06
SRF - 3.7	**URF** - 05
SRF - 3.8	**URF** - 05
SRF - 3.9	**URF** - 05
SRF - 3.10	**URF** - 05, **URF** - 06
SRF - 3.11	**URF** - 05
SRF - 3.12	**URF** - 05
SRF - 3.13	**URF** - 05
SRF - 3.14	**URF** - 06
SRF - 3.15	**URF** - 06
SRF - 3.16	**URF** - 06
SRF - 3.17	**URF** - 05
SRF - 3.18	**URF** - 06
SRF - 3.19	**URF** - 05
SRF - 3.20	**URF** - 06
SRF - 3.21	**URF** - 05
SRF - 3.22	**URF** - 06
SRF - 3.23	**URF** - 05
SRF - 3.24	**URF** - 05
SRF - 3.25	**URF** - 06
SRF - 3.26	**URF** - 22, **URF** - 23
SRF - 3.27	**URF** - 22, **URF** - 23
SRF - 3.28	**URF** - 22, **URF** - 23
SRF - 3.29	**URF** - 22, **URF** - 23
SRF - 3.30	**URF** - 22, **URF** - 23
SRF - 3.31	**URF** - 22, **URF** - 23
SRF - 3.32	**URF** - 22, **URF** - 23
SRF - 3.33	**URF** - 22, **URF** - 23

SRF - 4.1	**URF** - 02
SRF - 4.2	**URF** - 01, **URF** - 02
SRF - 4.3	**URF** - 01, **URF** - 02
SRF - 4.4	**URF** - 02
SRF - 4.5	**URF** - 01
SRF - 4.6	**URF** - 02
SRF - 4.7	**URF** - 01
SRF - 4.8	**URF** - 01
SRF - 4.9	**URF** - 02
SRF - 4.10	**URF** - 01
SRF - 4.11	**URF** - for ONNX classification tasks (ADD IN URD), **URF** - for ONNX regression tasks (ADD IN URD)
SRF - 5.1	**URF** - 10
SRF - 5.2	**URF** - 09
SRF - 5.3	**URF** - 09
SRF - 5.4	**URF** - 09
SRF - 5.5	**URF** - 10
SRF - 5.6	**URF** - 09
SRF - 5.7	**URF** - 09
SRF - 5.8	**URF** - 09
SRF - 5.9	**URF** - 10
SRF - 5.10	**URF** - 09
SRF - 5.11	**URF** - 10
SRF - 5.12	**URF** - 10
SRF - 5.13	**URF** - 09
SRF - 5.14	**URF** - 24, **URF** - 25
SRF - 5.15	**URF** - 24, **URF** - 25
SRF - 5.16	**URF** - 24, **URF** - 25
SRF - 5.17	**URF** - 24, **URF** - 25
SRF - 5.18	**URF** - 24, **URF** - 25
SRF - 5.19	**URF** - 24, **URF** - 25
SRF - 5.20	**URF** - 24, **URF** - 25
SRF - 5.21	**URF** - 24, **URF** - 25
SRF - 6.1	**URF** - 18
SRF - 6.2	**URF** - 19
SRF - 6.3	**URF** - 18, **URF** - 19
SRF - 6.4	**URF** - 18

SRF - 6.5	**URF** - 19
SRF - 6.6	**URF** - 19
SRF - 6.7	**URF** - 18
SRF - 6.8	**URF** - 18
SRF - 6.9	**URF** - 19

4.2 User Requirements to Software Requirements

URF	**SRF**
URF - 01	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7, **SRF** - 4.2, **SRF** - 4.3, **SRF** - 4.5, **SRF** - 4.7, **SRF** - 4.8, **SRF** - 4.10
URF - 02	**SRF** - 1.1, **SRF** - 1.3, **SRF** - 4.1, **SRF** - 4.2, **SRF** - 4.3, **SRF** - 4.4, **SRF** - 4.6, **SRF** - 4.9
URF - 03	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7, **SRF** - 2.2, **SRF** - 2.3, **SRF** - 2.6, **SRF** - 2.8, **SRF** - 2.9, **SRF** - 2.10, **SRF** - 2.11, **SRF** - 2.14
URF - 04	**SRF** - 1.1, **SRF** - 1.3, **SRF** - 2.1, **SRF** - 2.4 **SRF** - 2.5, **SRF** - 2.7, **SRF** - 2.12, **SRF** - 2.13, **SRF** - 2.15, **SRF** - 2.16
URF - 05	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7, **SRF** - 3.2, **SRF** - 3.3, **SRF** - 3.5, **SRF** - 3.7, **SRF** - 3.8, **SRF** - 3.9, **SRF** - 3.10, **SRF** - 3.11, **SRF** - 3.12, **SRF** - 3.13, **SRF** - 3.17, **SRF** - 3.19, **SRF** - 3.21, **SRF** - 3.23, **SRF** - 3.24
URF - 06	**SRF** - 1.1, **SRF** - 1.3, **SRF** - 3.1, **SRF** - 3.4, **SRF** - 3.6, **SRF** - 3.10, **SRF** - 3.14, **SRF** - 3.15, **SRF** - 3.16, **SRF** - 3.18, **SRF** - 3.20, **SRF** - 3.22, **SRF** - 3.25
URF - 07	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7
URF - 08	**SRF** - 1.1, **SRF** - 1.3
URF - 09	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7, **SRF** - 5.2, **SRF** - 5.3, **SRF** - 5.4, **SRF** - 5.6, **SRF** - 5.7, **SRF** - 5.8, **SRF** - 5.10, **SRF** - 5.13
URF - 10	**SRF** - 1.1, **SRF** - 1.3, **SRF** - 5.1, **SRF** - 5.5, **SRF** - 5.9, **SRF** - 5.11, **SRF** - 5.12
URF - 11	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7
URF - 12	**SRF** - 1.1, **SRF** - 1.3
URF - 13	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7
URF - 14	**SRF** - 1.1, **SRF** - 1.3
URF - 15	**SRF** - 1.2, **SRF** - 1.4, **SRF** - 1.7
URF - 16	**SRF** - 1.1, **SRF** - 1.3
URF - 17	**SRF** - 1.3, **SRF** - 1.4
URF - 18	**SRF** - 6.1, **SRF** - 6.3, **SRF** - 6.4, **SRF** - 6.7, **SRF** - 6.8
URF - 19	**SRF** - 6.2, **SRF** - 6.3, **SRF** - 6.5, **SRF** - 6.6, **SRF** - 6.9

URF - 20	**SRF** - 1.5, **SRF** - 1.6, **SRF** - 1.7, **SRF** - 1.9, **SRF** - 1.11
URF - 21	**SRF** - 1.5, **SRF** - 1.6, **SRF** - 1.7, **SRF** - 1.9, **SRF** - 1.11
URF - 22	**SRF** - 1.5, **SRF** - 1.6, **SRF** - 1.7, **SRF** - 1.8, **SRF** - 1.9, **SRF** - 1.11, **SRF** - 3.26, **SRF** - 3.27, **SRF** - 3.28, **SRF** - 3.29, **SRF** - 3.30, **SRF** - 3.31, **SRF** - 3.32, **SRF** - 3.33
URF - 23	**SRF** - 1.5, **SRF** - 1.6, **SRF** - 1.7, **SRF** - 1.8, **SRF** - 1.9, **SRF** - 1.11, **SRF** - 3.26, **SRF** - 3.27, **SRF** - 3.28, **SRF** - 3.29, **SRF** - 3.30, **SRF** - 3.31, **SRF** - 3.32, **SRF** - 3.33
URF - 24	**SRF** - 1.5, **SRF** - 1.6, **SRF** - 1.7, **SRF** - 1.9, **SRF** - 1.11, **SRF** - 5.14, **SRF** - 5.15, **SRF** - 5.16, **SRF** - 5.17, **SRF** - 5.18, **SRF** - 5.19, **SRF** - 5.20, **SRF** - 5.21
URF - 25	**SRF** - 1.5, **SRF** - 1.6, **SRF** - 1.7, **SRF** - 1.9, **SRF** - 1.11, **SRF** - 5.14, **SRF** - 5.15, **SRF** - 5.16, **SRF** - 5.17, **SRF** - 5.18, **SRF** - 5.19, **SRF** - 5.20, **SRF** - 5.21

A | User Interface Mockups

In this appendix the user interface for the visualization module is represented.

A.1 Visualizing a run

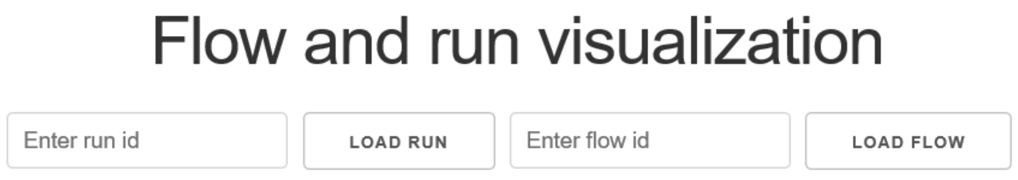


Figure A.1: Flow and run visualization starting screen

To visualize a run the user firstly needs to go to the starting screen of the visualization page, which looks as depicted on figure A.1. Then the user has to type a valid run id into the input field for run id as shown on figure A.2.

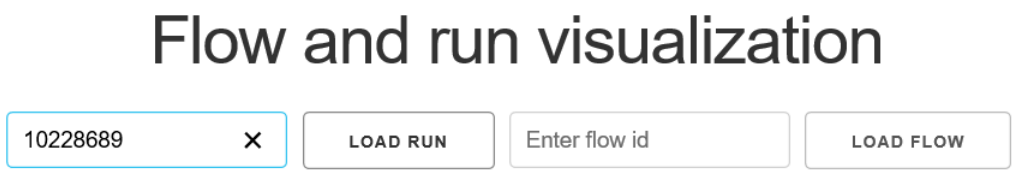


Figure A.2: Typing a valid run id

After pressing the button "LOAD RUN" the user receives a feedback message that the run is loading. This message is shown on figure A.3.

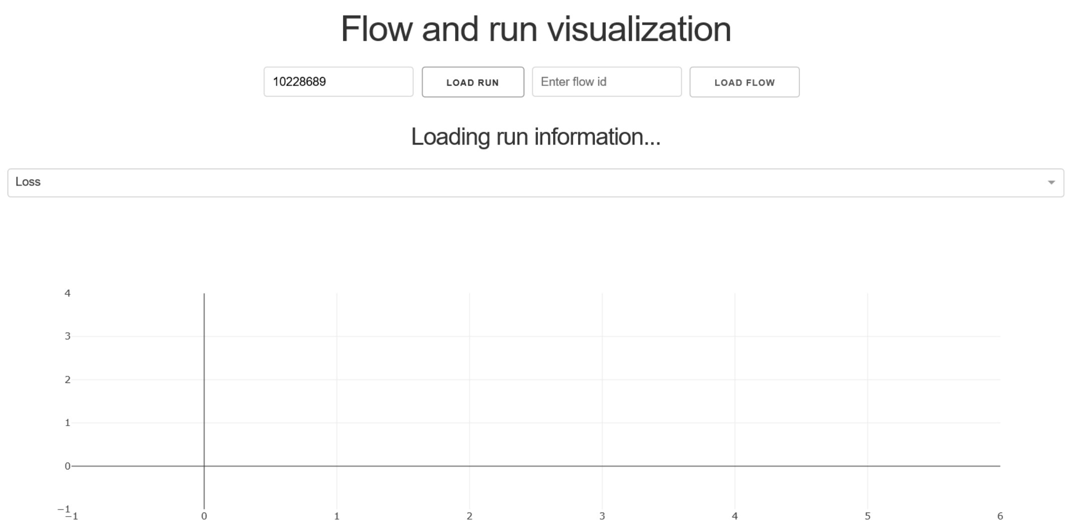


Figure A.3: Feedback message for loading a run

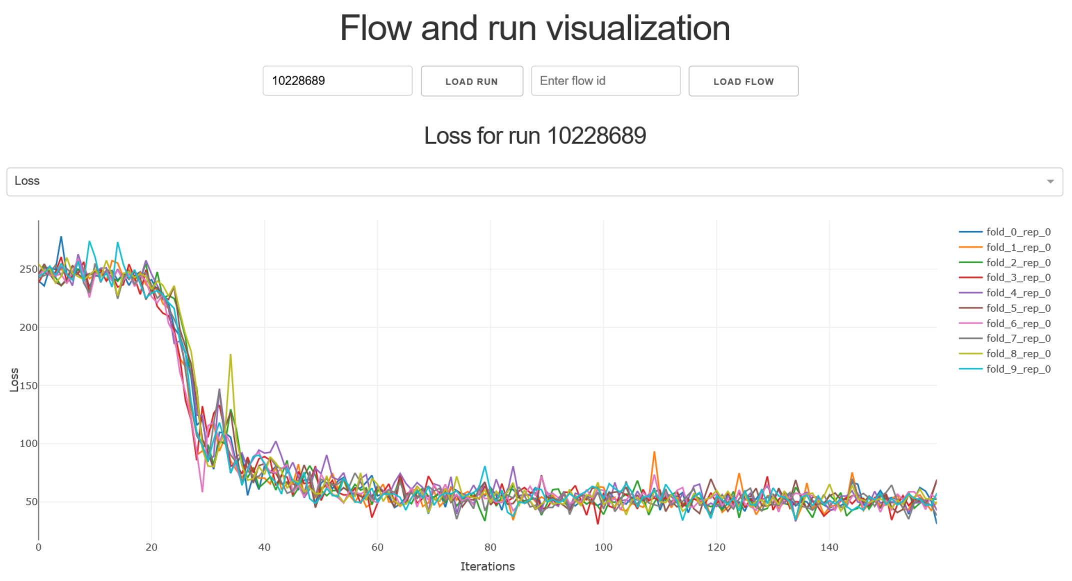


Figure A.4: Loss of a run for a regression task

If the run is associated with a regression task a graph similar to the one on figure A.4 will be shown, which by default depicts the loss of the given run for all the given folds. The user can select which folds they want to see on the graph by clicking on the lines at the right side of the graph on the figure. This can result in a graph like the one shown on figure A.5, which is the loss for one particular fold.

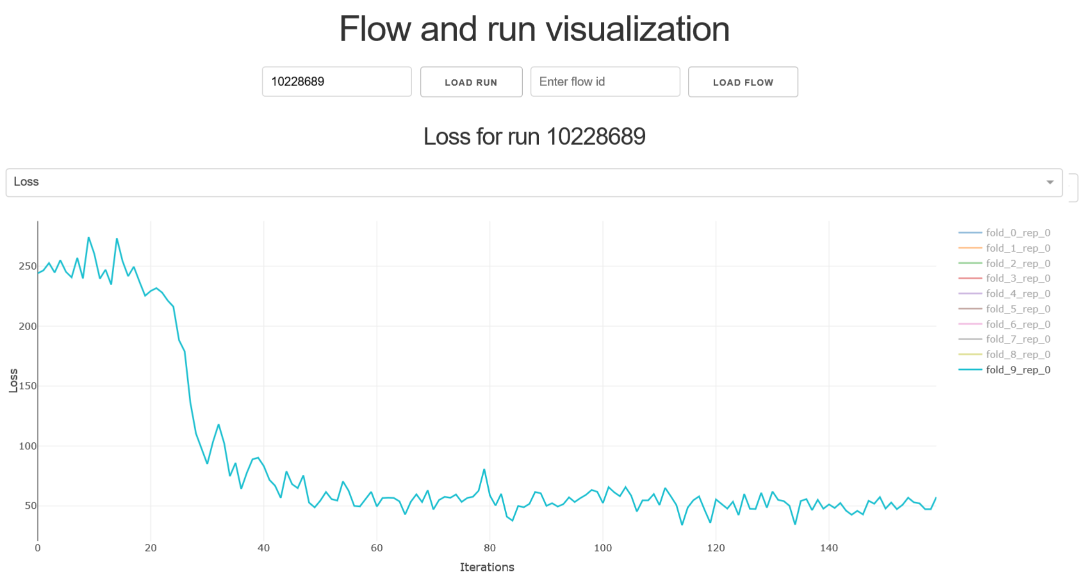


Figure A.5: Loss of a run for one fold for a regression task

If the user wants to choose another metric to plot, they can select one from the drop down menu, which can be seen on figure A.6. Regression tasks can have their loss, mean square error, mean absolute error and root mean square error plotted.

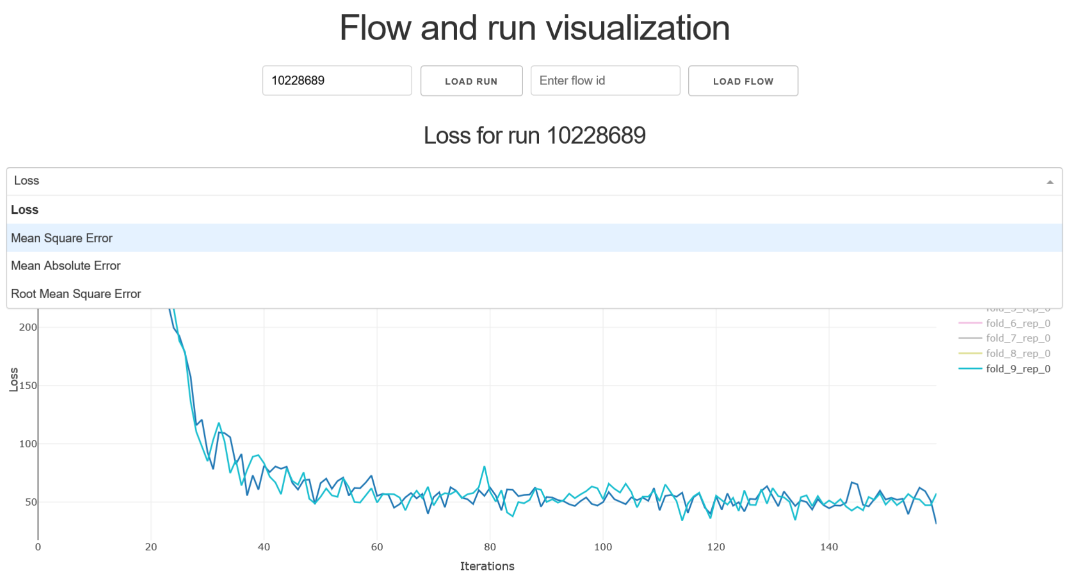


Figure A.6: Loss of a run for one fold for a regression task

For example the mean squared error can be selected from the drop down menu for the last fold and this will result in the graph, depicted in figure A.7.

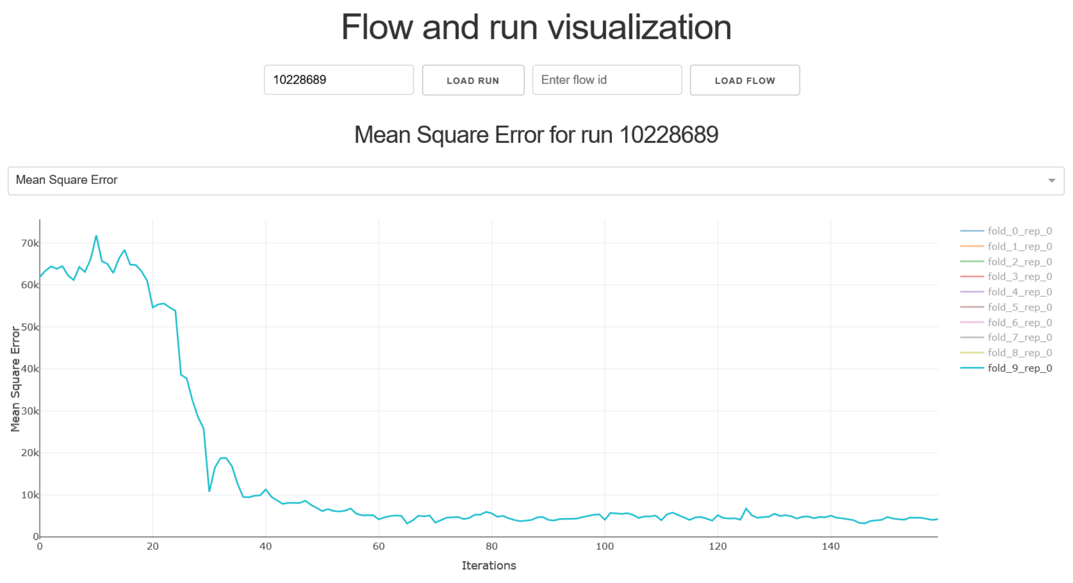


Figure A.7: Mean square error for one fold

If the user selects to visualize a run associated with a classification task, there are two metrics that can be plotted - loss and accuracy. The default one is the loss and will be displayed in a similar manner as on figure A.8. From a drop down men as the one shown previously on figure A.6 the user can select to have the accuracy plotted, resulting in figure A.9.

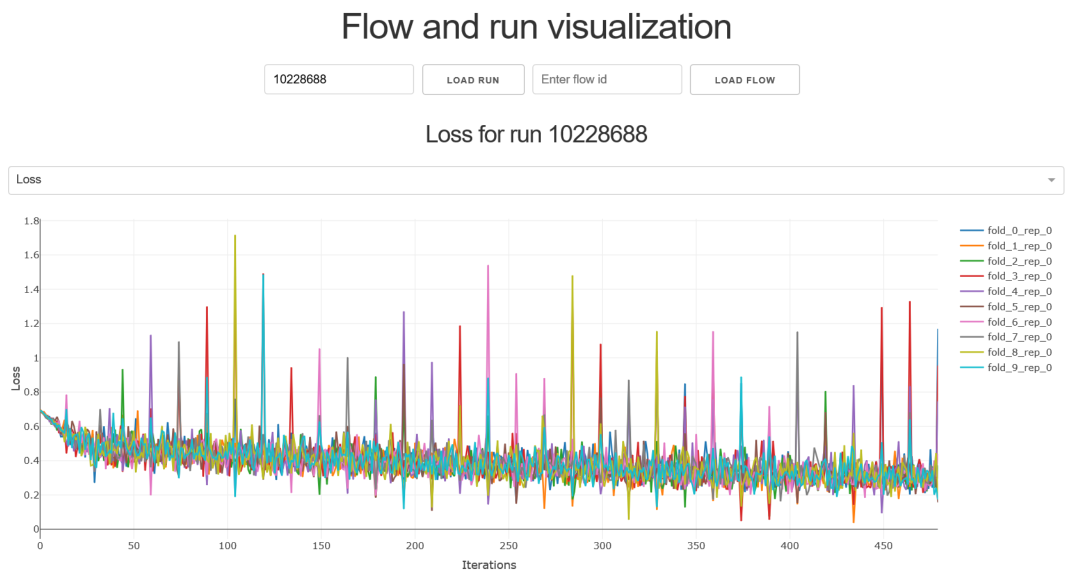


Figure A.8: Loss of a run for a classification task

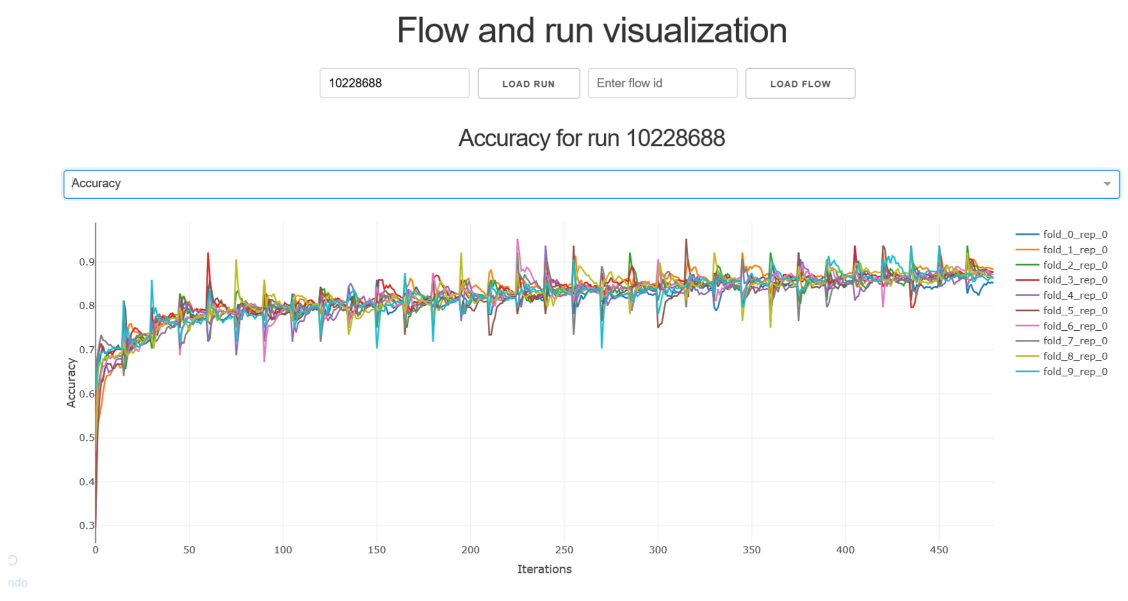


Figure A.9: Accuracy of a run for a classification task

The user can also select that they want to visualize only the mean of all the folds for a certain metric. This can be done by double clicking on "mean" at the right sight of figure A.10.

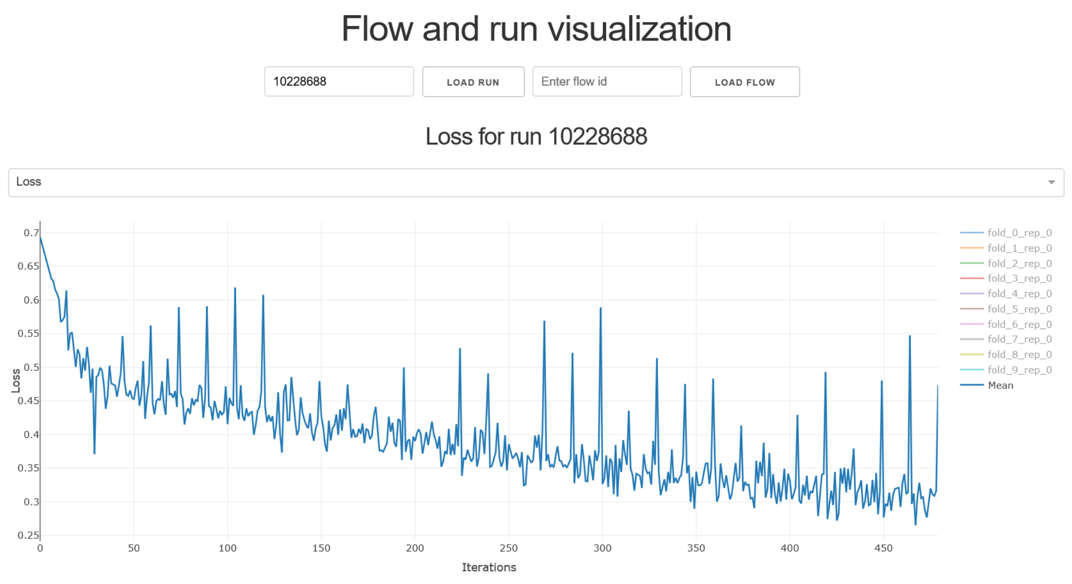
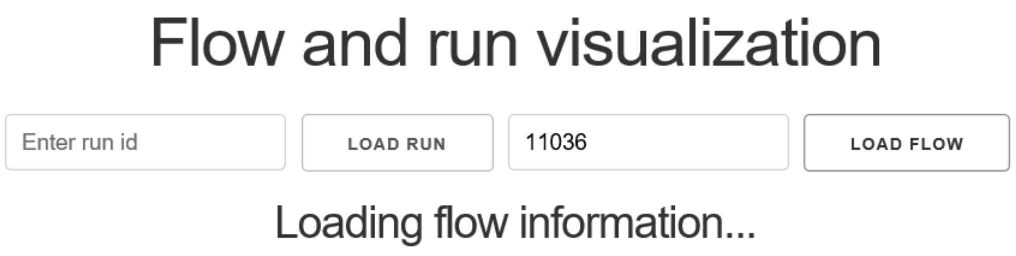


Figure A.10: Mean for loss

A.2 Visualizing a flow

If the user wants to visualize a flow, they need to type a valid OpenML flow id in the input field as shown on figure A.1. If this is done and the "LOAD FLOW" button is pressed a loading message will be shown as figure A.11 depicts.


Flow and run visualization

Loading flow information...

Figure A.11: Loading a flow

The visualization of the flow will look in a similar way as on figure A.12, depending on the given flow.

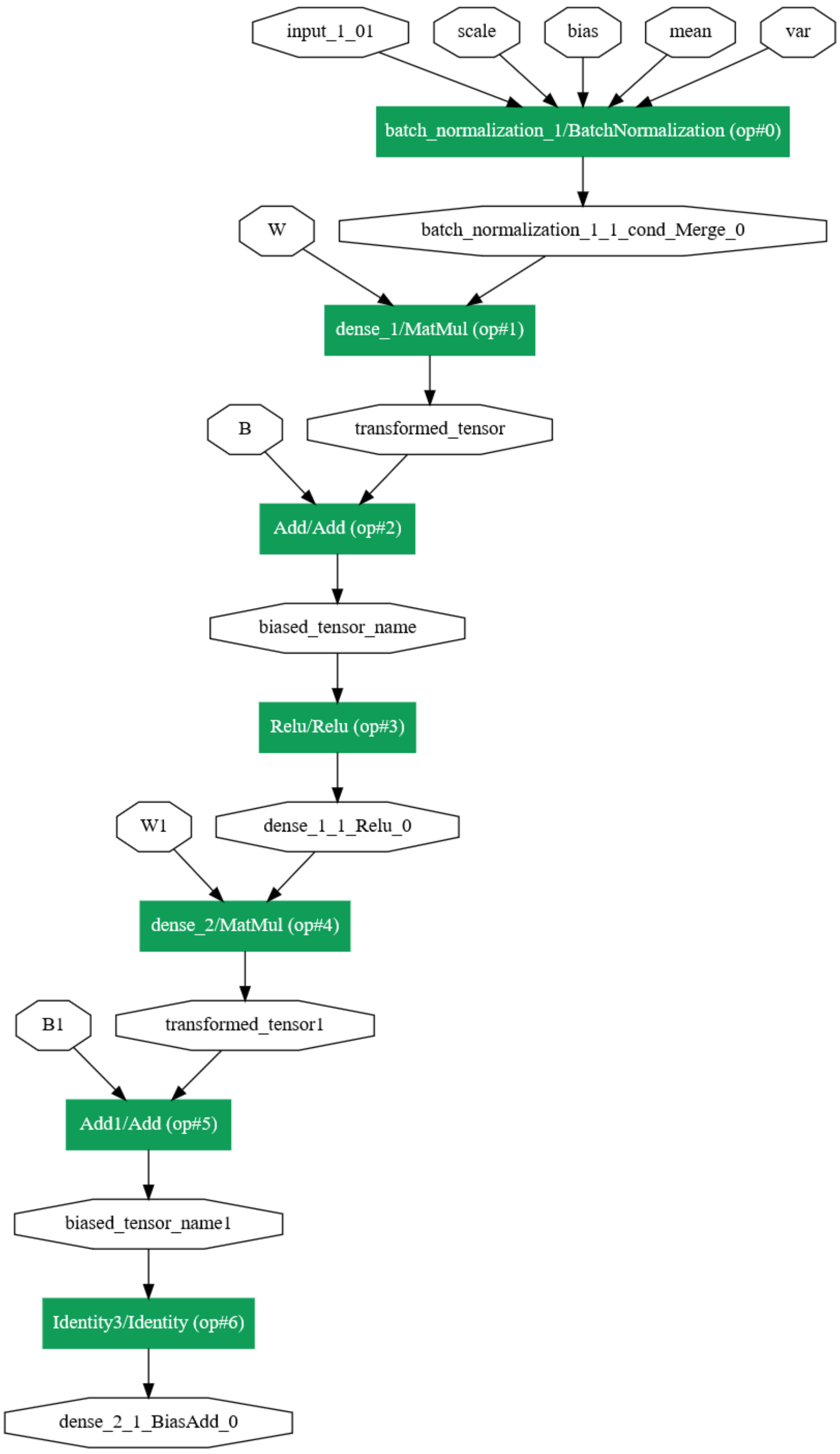


Figure A.12: Visualization of a flow

B | Transitions

The DL Extension Library's user interface contains only a **single** page as shown in A which can plot different graphs depending on the users choice. There are no transitions between different pages, thus defeating the purpose of this appendix.

Bibliography

[1] OpenML Support Squad. *DL Extension Library, URD User Requirement Document version 1.0.0.*

[2] Software Standardisation and Control. *ESA software engineering standards. 1991.*

[3] OpenML. *OpenML Documentation.* URL: `https://docs.openml.org/` (visited on 04/26/2019).

[4] Wikipedia. *Apache MXNet.* URL: `https://en.wikipedia.org/wiki/Apache_MXNet` (visited on 04/26/2019).

[5] Wikipedia. *Caffe (software.* URL: `https://en.wikipedia.org/wiki/Caffe_(software)` (visited on 04/26/2019).

[6] Caffe2. *Caffe2 homepage.* URL: `https://caffe2.ai/` (visited on 04/26/2019).

[7] Wikipedia. *Statistical classification.* URL: `https://en.wikipedia.org/wiki/Statistical_classification` (visited on 05/01/2019).

[8] Eijaz Allibhai. *Hold-out vs. Cross-validation in Machine Learning.* URL: `https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f` (visited on 05/17/2019).

[9] Kyle Kelley. *Introducing Dash.* URL: `https://medium.com/@plotlygraphs/introducing-dash-5ecf7191b503` (visited on 06/05/2019).

[10] JSON. *JSON homepage.* URL: `https://www.json.org/` (visited on 05/28/2019).

[11] Keras. *Keras homepage.* URL: `https://keras.io/` (visited on 04/26/2019).

[12] Wikipedia. *Machine Learning.* URL: `https://en.wikipedia.org/wiki/Machine_learning` (visited on 04/26/2019).

[13] MLflow. *MLflow Models.* URL: `https://mlflow.org/docs/latest/models.html` (visited on 04/26/2019).

[14] Joydeep Bhattacharjee. *Some Key Machine Learning Definitions.* URL: `https://medium.com/technology-nineleaps/some-key-machine-learning-definitions-b524eb6cb48` (visited on 05/01/2019).

[15] Wikipedia. *Artificial Neural Network.* URL: `https://en.wikipedia.org/wiki/Artificial_neural_network` (visited on 04/30/2019).

[16] Numpy Developers. *Numpy Homepage.* URL: `https://www.numpy.org/` (visited on 05/13/2019).

[17] *Protocol Buffers.* URL: `https://en.wikipedia.org/wiki/Protocol_Buffers` (visited on 05/29/2019).

[18] PyTorch. *PyTorch homepage.* URL: `https://pytorch.org/` (visited on 04/26/2019).

[19] Brian Beers. *Regression Definition.* URL: `https://www.investopedia.com/terms/r/regression.asp` (visited on 05/01/2019).

[20] PyTorch. *A Gentle Introduction to k-fold Cross-Validation.* URL: `https://machinelearningmastery.com/k-fold-cross-validation/` (visited on 05/24/2019).

[21] Agile Business Consortium. *MoSCoW Prioritisation*. URL: `https://www.agilebusiness.org/content/moscow-prioritisation-0` (visited on 04/26/2019).

[22] Wikipedia. *MoSCoW method*. URL: `https://en.wikipedia.org/wiki/MoSCoW_method` (visited on 04/26/2019).