PyTDML Tutorial

Publication Date: <2024-07-05>

Editor: Peng Yue, Ruixiang Liu, Yongyi Mi, Yang Zhang, Ming Zhao, Shuaiqi Liu

OGC Training Data Markup Language for Artificial Intelligence (TrainingDML-AI) Python Package Extension: PyTDML

License Agreement

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

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

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

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

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

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications. This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

Table of Contents

Chapter 1. Introduction

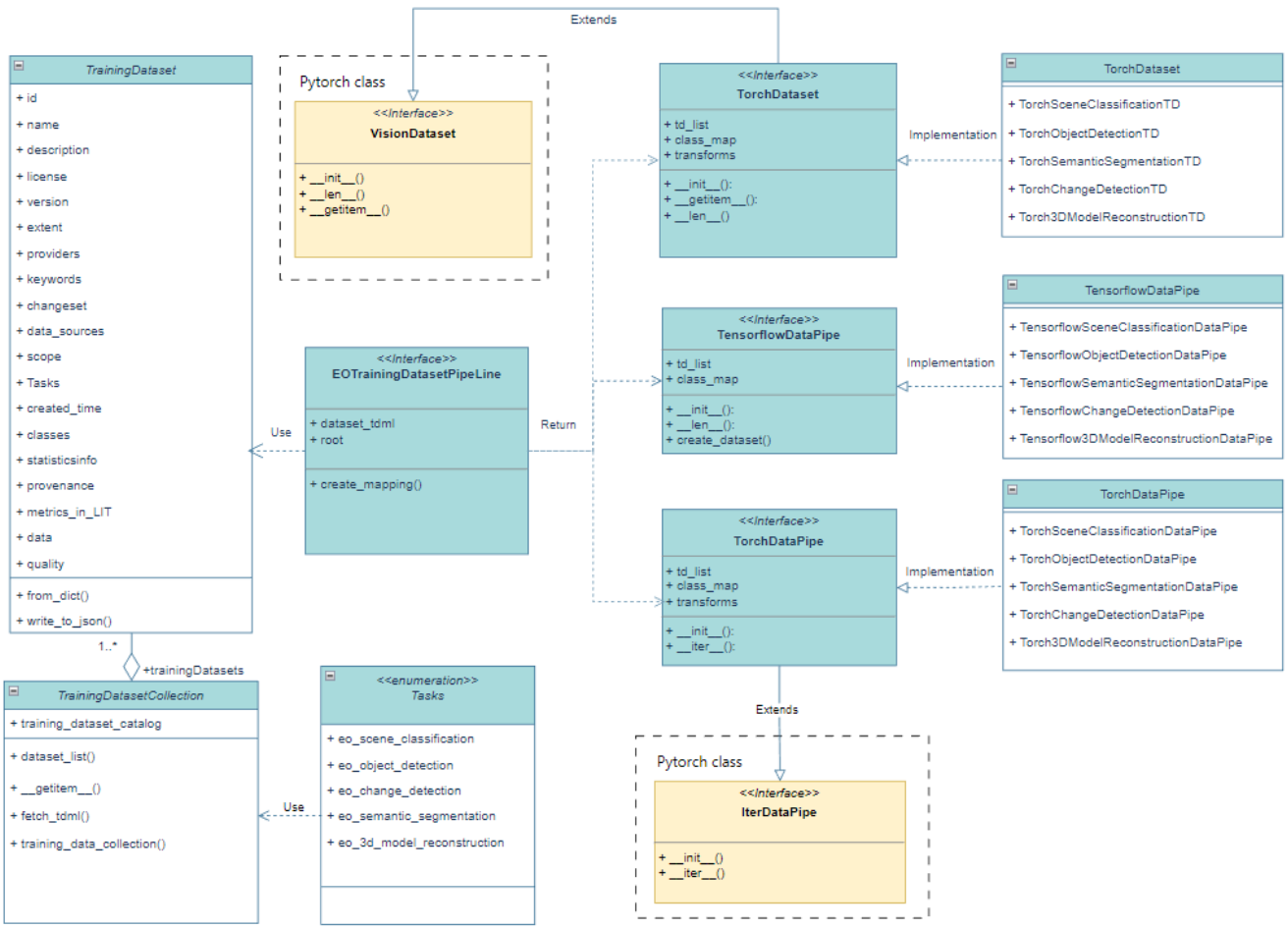
PyTDML is a pure Python parser and encoder for training datasets based on the OGC Training Data Markup Language for AI standard. Welcome to the PyTDML tutorial, your guide to working with Training Data Markup Language (TDML) for Artificial Intelligence.

1.1. What is TDML?

The Training Data Markup Language for Artificial Intelligence (TrainingDML-AI) SWG is chartered to develop the UML model and encodings for geospatial machine learning training data. In machine learning, training data is the dataset used for the training and validation of machine learning models. The geospatial training data categories will include, but are not limited to, remote sensing imagery, moving features (e.g., vehicle trajectories), and related spatial content. The SWG will define a UML model and encodings consistent with the OGC standards baseline to exchange and retrieve the training data in the web environment. The SWG will start with the JSON implementation. Once the UML and JSON encoding are well accepted, the SWG will work on the XML encoding as per the current OGC baseline. This standard will provide more detailed metadata for formalizing the information model of training data.

1.2. Main Functions of PyTDML

PyTDML is a framework or library used for processing and managing training datasets. The following figure shows the structure of classes implemented by PyTDML.



The main functions of PyTDML can be summarized as follows:

- Dataset Definition and Management: PyTDML allows users to organize and describe training datasets by defining classes such as *TrainingDataset* and *Task*. Users can manually input metadata information, which can include the ID, name, description, task type, data list, version, creation and update time, provider, keywords, data source, and specific category of the dataset. This enables the integration of training data from different sources into a *TrainingDataset* object.
- Task Definition: In PyTDML, users can define different tasks including "Scene Classification" and "Semantic Segmentation" and provide detailed descriptions for these tasks. This helps clarify the purpose and scenario of using the dataset, facilitating subsequent data processing and analysis.
- Data Pipeline: PyTDML has designed specific data pipeline routes for different Earth observation missions. Each data pipeline is responsible for loading the dataset required for the corresponding task. Within the same task type, data pipelines can use standardized loading logic to parse and process different datasets, thanks to the shared composition patterns of training samples among these datasets.
- Data Annotation and Classification: PyTDML supports the annotation and classification of training data. For example, in remote sensing image datasets, specific categories such as "Airport," "Beach," "Bridge," etc., can be defined, and data can be organized through these categories.
- Data Standardization: By defining a unified format for the dataset (such as image size, band information, etc.), PyTDML helps to standardize data processing, thereby improving data consistency and comparability.
- Metadata Management: PyTDML emphasizes the importance of metadata and provides convenience for the long-term storage and reuse of datasets by detailing the metadata of the dataset, such as data source, creator, update time, etc.

Let’s dive in and start exploring the world of standardized geospatial training data with PyTDML!

Chapter 2. IO Module

2.1. Reading - Dataset Encoding Input

2.1.1. Local File Input (Reading TrainingDML-AI Encoding from Local Files)

The functionality is located in tdml_readers.py within the IO module, primarily used for reading TrainingDML-AI encoded data from local files.

(1) Function: read_from_json(file_path: str)

a) Description:

- Reads a TDML JSON file from the specified path and returns a *TrainingDataset* object.

b) Parameter:

- file_path(string): The path to the JSON file.

(2) Function: parse_json(json_dict)

a) Description:

Parses a JSON dictionary and returns the corresponding *TrainingDataset* object based on its type.

b) Parameter:

- json_dict (dictionary): The dictionary representation of the JSON data.

Example:

Suppose we have a local file named data.json with the following content:

```
{
    "type": "TrainingDataset",
    "name": "example_dataset",
    "description": "This is an example training dataset.",
    "data": [
        {
            "id": "1",
            "label": "cat",
            "image": "cat1.jpg"
        },
        {
            "id": "2",
            "label": "dog",
```

```
            "image": "dog1.jpg"
        }
    ]
}
```

The user runs the following code:

```
from tdml_reader import read_from_json
file_path = "data.json"
training_dataset = read_from_json(file_path)
print(training_dataset)
```

After executing the above code, data.json will be read and its content will be parsed into a *TrainingDataset* object. The detailed information of this object will then be printed.

2.1.2. S3 Path Reading (Reading TrainingDML-AI Encoding from S3 Object Files)

The functionality is located in S3_reader.py within the IO module, primarily used to read TrainingDML-AI encoded data from S3 object storage.

(1) Function: parse_s3_path(s3_path)

a) Description:

- Parses an S3 path to extract the bucket name and object key.

b) Parameter:

- s3_path(string): The S3 path.

(2) Class *LibraryNotInstalledError*

a) Description:

- A custom exception class that is raised when a required library is not installed.

(3) Class *S3Client*

a) Description:

- The class is used for interacting with the S3 service.

b) Constructor Parameters

- resource(string): The type of resource (e.g., 's3').

- server(string): The S3 server address.
- access_key(string): The access key.
- secret_key(string): The secret key.

c) Methods

- list_buckets()

```
Functionality: Lists all buckets.
Return: A list of bucket names.
```

- list_objects(bucket_name, prefix)

```
Functionality: Lists all objects in a specified bucket.
Parameters:
    bucket_name (string): The name of the bucket.
    prefix (string): The object prefix.
Return: A list of object keys.
```

- get_object(bucket, key)

```
Functionality: Retrieves the content of a specified object.
Parameters:
    bucket (string): The name of the bucket.
    key (string): The object key.
Return: A BytesIO stream of the object's content.
```

- download_file(bucket_name, object_key, file_path)

```
Functionality: Downloads a specified object to a local file.
Parameters:
    bucket_name (string): The name of the bucket.
    object_key (string): The object key.
    file_path (string): The local file path.
    Exception Handling: If an exception occurs during download, an error
message will be printed.
```

Example:

Suppose we have an S3 path such as s3://my-bucket/my-folder/my-object.json which stores TrainingDML-AI encoded data. The user can run the following code:

```
from S3_reader import S3Client, parse_s3_path
resource = 's3'
```

```
server = 'https://s3.amazonaws.com'
access_key = 'your_access_key'
secret_key = 'your_secret_key'
s3_path = 's3://my-bucket/my-folder/my-object.json'
s3_client = S3Client(resource, server, access_key, secret_key)
bucket_name, object_key = parse_s3_path(s3_path)
s3_object = s3_client.get_object(bucket_name, object_key)
import json
json_dict = json.load(s3_object)
# Print the JSON data
print(json_dict)
```

In the above code, the user needs to provide S3 service configuration details, including the server address, access key, and secret key. Then, an *S3Client* instance is created. The S3 path is parsed to obtain the bucket name and object key. The *S3Client* instance is then used to retrieve the object content from S3, which is parsed as JSON and printed.

2.2. Organizing and Generating TrainingDML-AI Code Based on Local Dataset

Organizing and generating TrainingDML-AI code based on the local dataset requires two ways to obtain information: manual input and function calls. The specific steps are as follows:

2.2.1. Organizing Data and Metadata Information in the Dataset to Generate a *TrainingData*

Introduce classes such as *TrainingData* and *SceneLabel*, obtain information including data directory and file location, and organize them into *TrainingData*'s parameters.

Example:

```
import os
from pytdml.type import EOTrainingData, SceneLabel
td_list = []
image_path = r"TrainingDatasets\WHU-RS19\image"
for root, dirs, files in os.walk(image_path):
    for file in files:
        tdml = EOTrainingData(
            id=file.split(".")[0],
            labels=[SceneLabel(label_class=os.path.relpath(root, image_path))],
            data_url=os.path.join(root, file),
            date_time="2010"
        )
        td_list.append(tdml)
```

2.2.2. Organizing the *TrainingData* and Other Metadata Information into a *TrainingDataset*

Introduce classes such as *TrainingDataset* and *Task*, input metadata information manually, and combine the *TrainingData* from the previous step. Organize them into a *TrainingDataset*.

Example:

```
from pytdml.type import EOTrainingDataset, EOTask

whu_rs19 = EOTrainingDataset(
    id='whu_rs19',
    name='WHU_RS19',
    description="WHU-RS19 has 19 classes of remote sensing images scenes obtained from
Google Earth",
    tasks=[EOTask(task_type="Scene Classification", description="Structural high-
resolution satellite image indexing")],
    data=td_list,
    version="1.0",
    amount_of_training_data=len(td_list),
    created_time="2010",
    updated_time="2010",
    providers=["Wuhan University"],
    keywords=["Remote Sensing", "Scene Classification"],
    data_sources=["https://earth.google.com/"],
    classes=["Airport", "Beach", "Bridge", "Commercial", "Desert", "Farmland",
"footballField", "Forest", "Industrial", "Meadow", "Mountain", "Park", "Parking",
"Pond", "Port", "railwayStation", "Residential", "River", "Viaduct"],
    number_of_classes=19,
    bands=["red", "green", "blue"],
    image_size="600x600"
)
```

2.2.3. Writing a *TrainingDataset* as a JSON File and Outputting it Locally

The functionality is located in tdml_writers.py within the IO module, primarily used to write a *TrainingDataset* as a JSON file and output it locally.

(1) Function: write_to_json(td: TrainingDataset, file_path: str, indent: Union[None, int, str] = 4)

a) Description:

- Writes a *TrainingDataset* to a JSON file.

b) Parameter:

- td: Basic training dataset type
- file_path: The path where the file will be stored.

- indent: If “indent” is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. “None” is the most compact representation.

Example:

```
from pytdml.io import write_to_json
write_to_json(td, file_path)
```

2.3. Transform YAML to TDML

YAML (Yet Another Markup Language) is a commonly used data serialization format aimed at providing a data serialization standard that is easy for humans to read and write. The YAML configuration file schema is described in the YAML configuration file schema section. The functionality is located in yaml_to_tdml.py.

(1) Function: yaml_to_eo_tdml(yaml_path)

a) Description:

- Converts a YAML file to an *EOTrainingDataset*.

b) Parameter:

- yaml_path: The path where the YAML file is stored.

c) Return:

- *EOTrainingDataset*: Extended training dataset type for EO training dataset.

Example:

```
from pytdml.yaml_to_tdml import yaml_to_eo_tdml
training_datasets = yaml_to_eo_tdml(yaml_path)
```

2.4. Format Conversion

The functionality is located in convert_utils.py, primarily used to convert different formats to TrainingDML-AI encoding.

(1) Function: convert_coco_to_tdml(coco_dataset_path, output_json_path)

a) Description:

- Converts COCO format to TrainingDML-AI encoding. Reads data from a COCO-formatted

JSON file and saves it after converting it to a new JSON document.

b) Parameter:

- coco_dataset_path: The path to the JSON file in COCO format.
- output_json_path: The path to output the JSON file after conversion.

Example:

```
from pytdml.convert_utils import convert_coco_to_tdml
convert_coco_to_tdml(coco_dataset_path, output_json_path)
```

(2) Function: convert_stac_to_tdml(stac_dataset_path, output_json_path)

a) Description:

- Converts STAC format to TrainingDML-AI encoding. Reads JSON data in STAC format from a given path.

b) Parameter:

- stac_dataset_path: The path to the JSON file in STAC format.
- output_json_path: The path to output the JSON file after conversion.

Example:

```
from pytdml.convert_utils import convert_coco_to_tdml
convert_stac_to_tdml(stac_dataset_path, output_json_path)
```

Chapter 3. Machine Learning Module

3.1. Scene Classification Task

3.1.1. Dataset Approach

In the scene classification task, the *TorchSceneClassificationTD* inheriting from *VisionDataset* is a custom dataset class used for handling Earth Observation (EO) image scene classification training datasets. This class can load datasets in batches by index into the model.

Methods:

(1) __init__ (self, td_list, root, class_map, transform=None)

a) Description:

- Initialize the dataset object with the provided dataset entries, root directory, class map, and optional transformations.

b) Parameters:

- td_list: A list of dataset entries.
- root: The root directory of the dataset.
- class_map: A dictionary mapping class labels to numerical indices.
- transform: Optional transformations to apply to the images.

(2) _load_img_label(self)

a) Description:

- Extract image paths and labels from td_list.

b) Returns:

- imgs: A list of image paths.
- labels: A list of image labels.

(3) __len__(self)

a) Description:

- Return the size of the dataset.

b) Returns:

- The length of the dataset (i.e., the number of dataset entries).

(4) __getitem__(self, item)

a) Description:

- Retrieve the image and label corresponding to the given index, apply necessary preprocessing and transformations.

b) Parameters:

- item: The index of the dataset entry.
- Returns: The image and label.

(5) class_to_idx(self)

a) Description:

- Return the dictionary mapping classes to indices.

b) Returns:

- The class-to-index mapping dictionary.

Example:

```
import torch
from torch.utils.data import DataLoader
from torchvision.transforms import transforms
from datalibrary.datasetcollection import EOTrainingDatasetCollection, Task
from datalibrary.pipeline import PipeLine
import pytdml.ml.object_transforms as transform_target
from pytdml.ml.tdml_torch import TorchSceneClassificationTD

# Define data transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.CenterCrop(224),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    # Optional normalization
    # transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Define target transformations
target_transform = transform_target.Compose([
    transform_target.ToTensor(),
```

```
    transform_target.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
    transform_target.RandomResize((512, 512))
])

# Dataset root directory path
path = "."

def datasetsForSceneTask():
# Load the EOTrainingDatasetCollection library
ds_lib = EOTrainingDatasetCollection()
ds_lib.dataset_list(Task.scene_classification, ["Airport"])

    whurs19_ml = ds_lib["WHU-RS19"]
    print("Load training dataset: " + str(whurs19_ml.name))
    print("Number of training samples: " + str(whurs19_ml.amount_of_training_data))
    print("Number of classes: " + str(whurs19_ml.number_of_classes))

    # Load across datasets
    rsd46_ml = ds_lib["RSD46-WHU"]
    my_dataset_TD = ds_lib.training_data_collection(Task.scene_classification,
[rsd46_ml, whurs19_ml], ["Airport"])

    my_pipeline = PipeLine(my_dataset_TD, path)

    # Load dataset using torch_dataset
    my_dataset = my_pipeline.torch_dataset(download=False, transform=transform)

    # Create DataLoader
    dataloader = DataLoader(my_dataset, batch_size=4, num_workers=4, shuffle=True)

    for batch in dataloader:
        inputs, labels = batch
        print(f"Inputs: {inputs.size()}, Labels: {labels}")

# Run the dataset loading function
datasetsForSceneTask()
```

3.1.2. Data Pipeline Approach

In the scene classification task, the *TorchSceneClassificationDataPipe* inheriting from *IterDataPipe* is a custom dataset class that implements an iterator-based method for loading datasets in batches into the model. This class is suitable for handling large-scale datasets and enables efficient data loading and preprocessing.

Methods:

(1) __init__(self, td_list, root, cache_path, class_map, transform=None)

a) Description:

- Initialize the dataset object and set necessary attributes.

b) Parameters:

- td_list: A list of dataset entries.
- root: The root directory of the dataset.
- cache_path: The path to the cache file.
- class_map: A dictionary mapping class labels to numerical indices.
- transform: Optional transformations to apply to the images.

(2) __iter__(self)

a) Description:

- Iteratively load images and labels, apply necessary preprocessing and transformations.
- If a cache file exists, data is loaded from the cache file.
- If a cache file does not exist, images are loaded from remote or local sources, preprocessed, and cached.

b) Returns:

- A generator yielding images and labels.

Example:

```
import os
import torch
from torchdata.datapipes.iter import IterDataPipe
from torch.utils.data import DataLoader
from torchvision.transforms import transforms
from datalibrary.pipeline import PipeLine
from datalibrary.datasetcollection import EOTrainingDatasetCollection, Task
import pytdml.ml.object_transforms as transform_target
from pytdml.ml.ml_operators import collate_fn
from pytdml.ml.tdml_torch_data_pipe import TorchSceneClassificationDataPipe

# Define data transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.CenterCrop(224),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    # Optional normalization
    # transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

```
# Define target transformations
target_transform = transform_target.Compose([
    transform_target.ToTensor(),
    transform_target.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
    transform_target.RandomResize((512, 512))
])

# Dataset root directory path
path = "."

def datasetsForSceneTask():
    # Load the EOTrainingDatasetCollection library
    ds_lib = EOTrainingDatasetCollection()
    ds_lib.dataset_list(Task.scene_classification, ["Airport"])

    whurs19_ml = ds_lib["WHU-RS19"]
    print("Load training dataset: " + str(whurs19_ml.name))
    print("Number of training samples: " + str(whurs19_ml.amount_of_training_data))
    print("Number of classes: " + str(whurs19_ml.number_of_classes))

    # Load across datasets
    rsd46_ml = ds_lib["RSD46-WHU"]
    my_dataset_TD = ds_lib.training_data_collection(Task.scene_classification,
[rsd46_ml, whurs19_ml], ["Airport"])

    my_pipeline = PipeLine(my_dataset_TD, path)

    # Load data using torch_data_pipe
    my_data_pipe = my_pipeline.torch_data_pipe(transform=transform)

    # Create DataLoader
    dataloader = DataLoader(my_data_pipe, batch_size=4, num_workers=4,
collate_fn=collate_fn)

    for batch in dataloader:
        inputs, labels = batch
        print(f"Inputs: {inputs.size()}, Labels: {labels}")

# Run the dataset loading function
datasetsForSceneTask()
```

3.2. Semantic Segmentation Task

3.2.1. Dataset Approach

The *TorchSemanticSegmentationTD* inheriting from *IterDataPipe* is a custom dataset class designed for semantic segmentation tasks, specifically for Earth Observation (EO) images. This class is compatible with PyTorch's data loading utilities, handling the loading of image and label paths, applying transformations, and providing data in a format suitable for training deep learning

models.

Methods:

(1) __init__(self, td_list, root, classes, transform=None)

a) Description:

- Initialize the dataset with the provided parameters.

b) Parameters:

- td_list (list): List of training data objects, each containing URLs of images and labels.
- root (str): Root directory for storing images and labels.
- classes (list): List of class names for segmentation.
- transform (callable, optional): A function/transform that takes in an image and a label and returns the transformed version. Default is None.

(2) __len__(self)

a) Description:

- Return the number of items in the dataset.

b) Returns:

- int - The number of items in the dataset.

(3) __getitem__(self, item)

a) Description:

- Retrieve the image and label at the specified index, apply any transformations, and returns them.

b) Parameters:

- item (int): Index of the item to retrieve.

c) Returns:

- image (PIL.Image or torch.Tensor): Transformed image.
- label (PIL.Image or torch.Tensor): Transformed label.

(4) _load_data(self, td_list)

a) Description:

- Load data from the provided list. This method is used internally.

b) Parameters:

- td_list (list): List of training data objects.

c) Returns:

- list: The unchanged input list.

(5) _load_img_label(self, td_list)

a) Description:

- Processe the provided list to extract image and label paths.

b) Parameters:

- td_list (list): List of training data objects.

c) Returns:

- tuple: A tuple containing two lists:
- img_paths (list of str): Paths to images.
- label_paths (list of str): Paths to labels.

Example:

To use this dataset, you need a list of training data objects containing URLs of images and labels. Additionally, specify the root directory for storing these files and provide a list of class names for segmentation. Optionally, transformations can be applied to the images and labels.

```
import torchvision.transforms as transforms

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# List of training data objects
td_list = [
    # Example training data objects containing data_url and labels
]

# Initialize dataset
```

```
dataset = TorchSemanticSegmentationTD(td_list, root='path/to/data',
classes=['background', 'object'], transform=transform)

# Access an item
image, label = dataset[0]
```

This setup will load the data, apply the defined transformations, and prepare the images and labels for training a segmentation model.

3.2.2. Data Pipeline Approach

The *TorchSemanticSegmentationDataPipe* is a data pipeline class designed for semantic segmentation tasks. It inherits from *IterDataPipe* and *ABC*. This class can load and preprocess Earth Observation (EO) images, providing the data as an iterator. It supports caching data for efficient loading and can perform image cropping and transformations as required.

Methods:

(1) __init__(self, td_list, root, cache_path, class_list=None, crop=None, transform=None)

a) Description:

- Initialize the data pipeline with the provided parameters.

b) Parameters:

- td_list (list): List of training data objects, each containing URLs of images and labels.
- root (str): Root directory for storing images and labels.
- cache_path (str): Cache file path for saving and loading processed data.
- class_list (list, optional): List of class names for segmentation. Default is None.
- crop (tuple, optional): Crop parameters defining the size of the cropped images. Default is None.
- transform (callable, optional): A function/transform that takes in an image and a label and returns the transformed version. Default is None.

(2) __iter__(self)

a) Description:

- Iterator method for iterating over each item in the data pipeline.

b) Returns:

- iterator - An iterator yielding transformed image and label pairs.

Example:

To use this data pipeline, you need a list of training data objects containing URLs of images and labels. You also need to specify the root directory for storing these files, the cache file path, and provide a list of class names for segmentation and crop parameters. Optionally, you can apply transformations to the images and labels.

```
import torchvision.transforms as transforms

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# List of training data objects
td_list = [
    # Example training data objects containing data_url and labels
]

# Initialize data pipeline
data_pipe = TorchSemanticSegmentationDataPipe(
    td_list,
    root='path/to/data',
    cache_path='path/to/cache.pkl',
    class_list=['background', 'object'],
    crop=(256, 256),
    transform=transform
)

# Iterate over data
for image, label in data_pipe:
    # Process image and label
```

This configuration will load the data, apply the defined transformations, and prepare the images and labels for training a segmentation model. If crop parameters are specified, the images and labels will be cropped.

3.3. Object Detection task

3.3.1. Dataset Approach

The *TorchObjectDetectionTD* inheriting from *VisionDataset* is a custom dataset for EO image object detection training dataset. This class is designed to be compatible with PyTorch's data loading tool.

(1) __init__(self, td_list, root, class_map, transform=None)

a) Description:

- Initialize the dataset using the provided parameters.

b) Parameter:

- td_list (list): A list of training data objects, each containing the URL of the image and label.
- root (str): The root directory where images and labels are stored.
- class_map (list): A list of category names used for segmentation.
- transform (callable, optional): A function/transformation that inputs images and labels and returns the transformed version. The default value is None.

(2) __len__(self)

a) Description:

- Return the number of items in the dataset.

b) Return:

- length: Number of items in the dataset.

(3) __getitem__(self, index)

a) Description:

- Retrieve the specified index of images and labels, apply any transformations, and return them.

b) Parameter:

- index(int): The index of the item to be obtained.

c) Return:

- image (PIL.Image or torch.Tensor): The transformed image.
- targets (Dictionary): The transformed label.

Example:

```
from pytdml.ml.tdml_torch import TorchObjectDetectionTD
import torchvision.transforms as transforms

# Define transformation
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

```
# Training Data Object List
td_list = [
    # Sample training data object containing data_URL and labels
]

# Initialize dataset
dataset = TorchObjectDetectionTD(td_list, root='path/to/data',
class_map=['background', 'object'], transform=transform)

# Find the length of the dataset
length = len(dataset)
# Accessing a project
image, label = dataset[0]
```

This configuration will load data, apply defined transformations, and prepare images and labels for training segmentation models.

3.3.2. Data Pipeline Approach

The *TorchObjectDetectionDataPipe* is a data pipeline class used for object detection tasks, inheriting from *IterDataPipe*. This class can load and preprocess Earth Observation (EO) images, provide data in the form of iterators, crop and transform images as needed.

(1) __init__(self, td_list, root, cache_path, class_map=None, crop=None, transform=None)

a) Description:

- Initialize the function and initialize the data pipeline using the provided parameters.

b) Parameter:

- td_list (list): List of training data objects, each containing URLs of images before and after changes and labels.
- root (str): Root directory for storing images and labels.
- cache_path (str): Cache file path for saving and loading processed data.
- crop (tuple, optional): Crop parameters defining the size of the cropped images. Default is None.
- transform (callable, optional): A function/transform that takes in an image and a label and returns the transformed version. Default is None.

(2) __iter__(self)

a) Description:

- Iterator method for iterating over each item in the data pipeline.

b) Returns:

- iterator - An iterator yielding transformed image pairs and labels.

Example:

```
import torchvision.transforms as transforms
from pytdml.ml.tdml_torch_data_pipe import TorchObjectDetectionDataPipe

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# List of training data objects
td_list = [
    # # Example training data objects containing data_url and labels
]

# Initialize data pipeline
data_pipe = TorchObjectDetectionDataPipe(
    td_list,
    root='path/to/data',
    cache_path='path/to/cache.pkl',
    class_list=['background', 'object'],
    crop=(256, 256),
    transform=transform
)

# Iterate over data
for image, label in data_pipe:
    # Process image pairs and labels
```

3.4. Change Detection Task

3.4.1. Dataset Approach

The *TorchChangeDetectionTD* is a custom dataset designed for change detection tasks, inheriting from *VisionDataset*. This class can load and preprocess Earth Observation (EO) images, including pairs of images before and after changes, and provide them for model training or evaluation. It supports applying transformations to images and labels.

Methods:

(1) __init__(self, td_list, root, transform)

a) Description:

- Initialize the dataset with the provided parameters.

b) Returns:

- td_list (list): List of training data objects, each containing URLs of images before and after changes and labels.
- root (str): Root directory for storing images and labels.
- transform (callable): A function/transform that takes in an image and a label and returns the transformed version.

(2) __len__(self)

a) Description:

- Return the number of samples in the dataset.

b) Returns:

- int - The number of samples in the dataset.

(3) __getitem__(self, index)

a) Description:

- Retrieve the image pair and label at the specified index and applies any transformations.

b) Parameters:

- index (int): Index of the sample to retrieve.

c) Returns:

- tuple - A tuple containing the image before change, image after change, and label.

(4) _load_sample(self)

a) Description:

- Load sample data, generate paths for images before and after change and label paths.

b) Returns:

- list - List of samples containing paths to images and labels.

Example:

To use this dataset, you need a list of training data objects containing URLs of images before and after changes and labels. You also need to specify the root directory for storing these files and provide transformations.

```
import torchvision.transforms as transforms

# List of training data objects
td_list = [
    {
        'data_url': ['path/to/before_image.png', 'path/to/after_image.png'],
        'labels': [{'image_url': 'path/to/label.png'}]
    },
    # More data objects
]

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# Initialize dataset
dataset = TorchChangeDetectionTD(td_list=td_list, root='path/to/data',
transform=transform)

# Access a sample
before_img, after_img, label = dataset[0]
print(before_img.shape, after_img.shape, label.shape)
```

This approach allows the dataset to efficiently load and provide the images and labels needed for change detection tasks.

3.4.2. Data Pipeline Approach

The *TorchChangeDetectionDataPipe* is a data pipeline class for change detection tasks, inheriting from *IterDataPipe* and *ABC*. This class can load and preprocess Earth Observation (EO) images, including pairs of images before and after changes, and provide the data as an iterator. It supports loading data from a cache and can crop and transform images as needed.

Methods:

(1) __init__(self, td_list, root, cache_path, crop=None, transform=None)

a) Description:

- Initialize the data pipeline with the provided parameters.

b) Parameters:

- td_list (list): List of training data objects, each containing URLs of images before and after changes and labels.
- root (str): Root directory for storing images and labels.
- cache_path (str): Cache file path for saving and loading processed data.
- crop (tuple, optional): Crop parameters defining the size of the cropped images. Default is None.
- transform (callable, optional): A function/transform that takes in an image and a label and returns the transformed version. Default is None.

(2) __iter__(self)

a) Description:

- Iterator method for iterating over each item in the data pipeline.

b) Returns:

- iterator - An iterator yielding transformed image pairs and labels.

Example:

To use this data pipeline, you need a list of training data objects containing URLs of images before and after changes, along with labels. Additionally, specify the root directory for storing these files, the cache file path, and provide crop parameters. Optionally, transformations can be applied to the images and labels.

```
import torchvision.transforms as transforms

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# List of training data objects
td_list = [
    # Example training data objects containing data_url and labels
]

# Initialize data pipeline
data_pipe = TorchChangeDetectionDataPipe(
    td_list,
    root='path/to/data',
    cache_path='path/to/cache.pkl',
    crop=(256, 256),
```

```
    transform=transform
)

# Iterate over data
for before_img, after_img, label in data_pipe:
    # Process image pairs and labels
```

This configuration will load the data, apply the defined transformations, and prepare the image pairs and labels for training a change detection model. If crop parameters are specified, the images and labels will be cropped accordingly.

Chapter 4. PyTDML Use Case

Here we take the object detection datasets DOTA 2.0 and RSOD as examples to demonstrate the process using PyTDML.

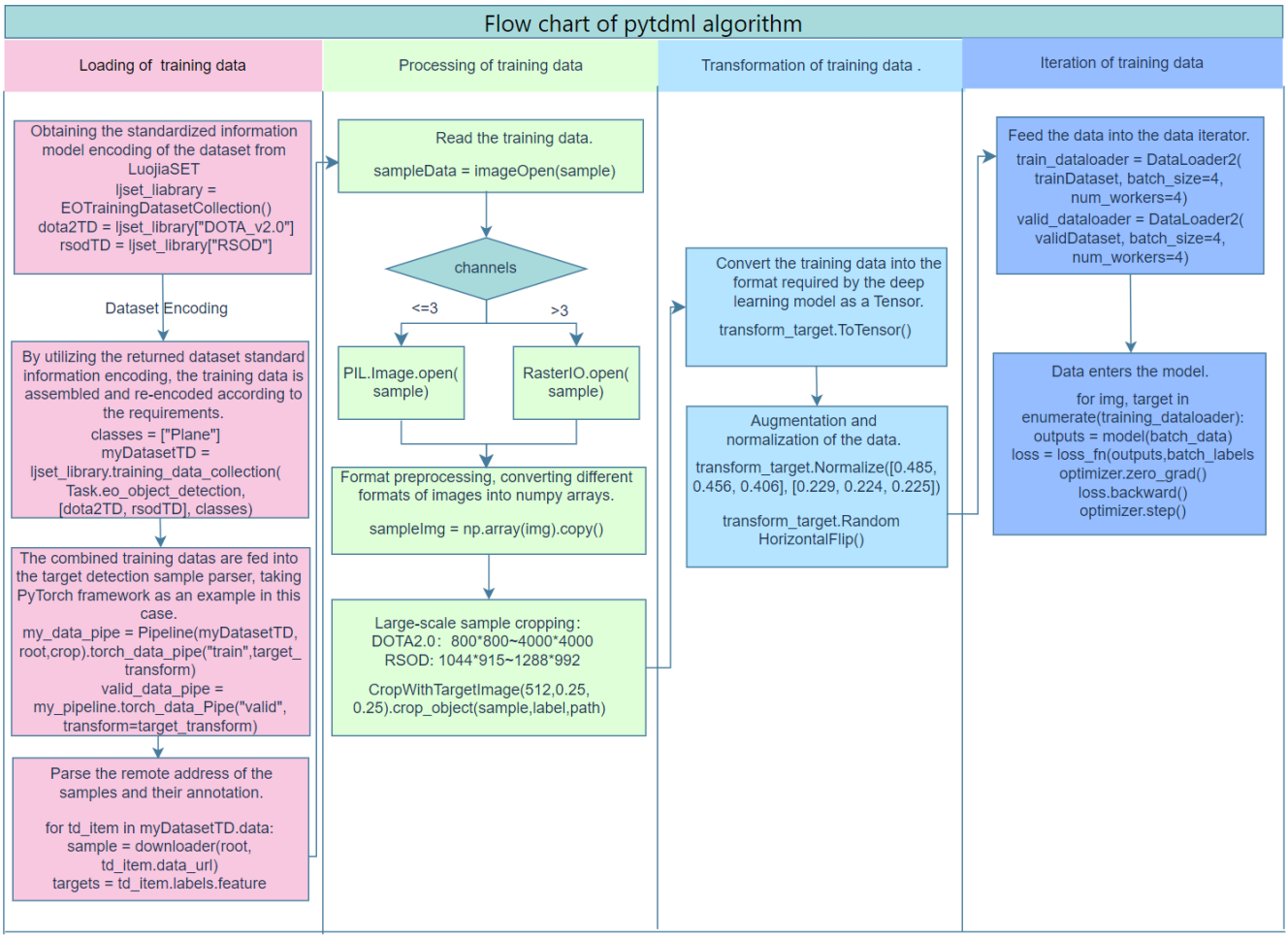
DOTA2.0:

- Number of plane instances: 11365
- Image format: jpg
- Image channels: 1 or 3
- Image Size: 800 * 800 to 4000 * 4000

RSOD:

- Number of plane instances: 5374
- Image format: .jpg
- Image channels: 3
- Image Size: 1044 * 915 to 1288 * 992

The process of the case is illustrated in the following figure, which includes four steps: Loading of training data, Processing of training data, Transformation of training data, and Iteration of training data.



The code for implementing this case is shown below.

```
DOTA2_TD = EOTrainingDatasetcollection()["DOTA-v2.0"]
# Obtain metadata information of the DOTA-v2.0 dataset.
print("Load training dataset: " + DOTA2_TD.name)
print("Number of training samples: " + str(DOTA2_TD.amount_of_training_data))
print ("Number of classes: " + str(DOTA2_TD.number_of_classes))

RSOD_TD = EOTrainingDatasetcollection()["RSOD"]
my_dataset =
EOTrainingDatasetcollection().training_data_collection(Task.eo_object_detection,
[DOTA2_TD_TD, RSOD_TD], ["Plane"])

# Encapsulate the PyTorch dataset class.
my_pipeline = PipeLine(mydataset, path, crop=(800, 0.25, 0.25))
train_data_pipe = my_pipeline.torch_data_Pipe("train", transform=target_transform)
train_dataloader = DataLoader2(train_data_pipe, batch_size=4, num_workers=4,
collate_fn=collate_fn)
valid_data_pipe = my_pipeline.torch_data_Pipe("valid", transform=target_transform)
valid_dataloader = DataLoader2(valid_data_pipe, batch_size=4, num_workers=4,
collate_fn=collate_fn)

# Model training.
for epoch in range(num_epochs):
    for batch_data, batch_labels in train_dataloader:
        # Forward propagation.
        outputs = model(batch_data)
        loss = loss_fn(outputs, batch_labels)
        # Backward propagation and optimization.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # Print the loss value for each epoch.
    print(f"Epoch{epoch+1}/{num_epochs}, Loss: {loss.item()}")

with torch.no_grad():
    for images, labels in valid_dataloader:
    # Compute accuracy and other evaluation metrics.
```