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

Table of Contents

Chapter 1. Introduction

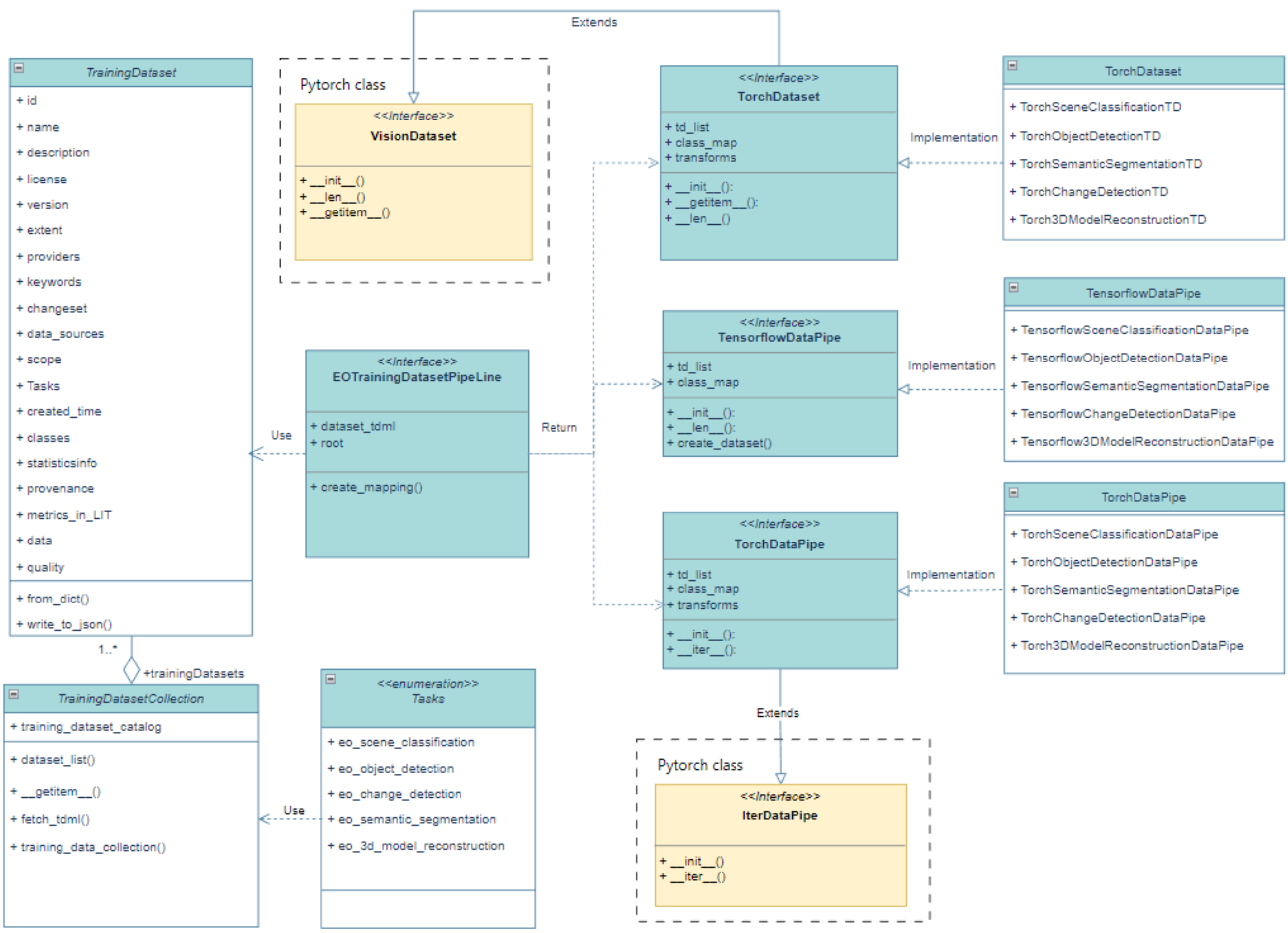
PyTDML is a pure python parser and encoder for training datasets based on OGC Training Data Markup Language for AI standard. Welcome to the PyTDML tutorial, your guide to work with Training Data Markup Language (TDML) for Artificial Intelligence using Python.

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 training and validation of machine learning models. The geospatial training data categories will include, but are not restricted 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 from the JSON implementation. Once the UML and JSON encoding are well accepted, the SWG will work on the XML encoding as 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 organizational structure of abstract classes implemented by PyTDML.



The main functions of PyTDML can be summarized as follows:

- Datasets 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 and integrate training data from different sources into a *TrainingDataset* object. This includes information such as the ID, name, description, task type, data list, version, creation and update time, provider, keywords, data source, and specific category of the dataset.
- Task definition: In PyTDML, users can define different tasks including "Scene Classification" and "Semantic Segmentation", and provide detailed descriptions for these tasks. This helps to 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, which is due to the shared training sample composition patterns among these datasets. Therefore, cross dataset training sample utilization becomes feasible as data pipelines can consistently process and interpret training samples.
- Data annotation and classification: PyTDML supports 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 local file input functionality is located in the tdml_readers.py file within the IO module. This module is primarily used for reading TrainingDML-AI encoded data from local files.

(1) Function: read_from_json(file_path: str)

a) Functionality:

- 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: def parse_json(json_dict)

a) Functionality:

- Parses a JSON dictionary and returns the corresponding training dataset 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:

```
# Import the reading function
from tdml_reader import read_from_json

# Specify the local file path
file_path = "data.json"
# Read the JSON file and parse it into a TrainingDataset object
training_dataset = read_from_json(file_path)
# Print the contents of the TrainingDataset object
print(training_dataset)
```

After executing the above code, the data.json file 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 S3 path reading functionality is located in the S3_reader.py file within the IO module. This module is primarily used to read TrainingDML-AI encoded data from S3 object storage.

(1) Function parse_s3_path(s3_path)

a) Functionality:

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

b) Parameter:

- s3_path (string): The S3 path.

(2) Class LibraryNotInstalledError

a) Functionality:

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

(3) Class S3Client

a) Functionality:

- The S3 client 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 s3://my-bucket/my-folder/my-object.json which stores TrainingDML-AI encoded data. The user can run the following code:

```
# Import necessary modules and classes
from S3_reader import S3Client, parse_s3_path

# S3 storage configuration
resource = 's3'
server = 'https://s3.amazonaws.com'  # Corresponding S3 service address
access_key = 'your_access_key'
secret_key = 'your_secret_key'
s3_path = 's3://my-bucket/my-folder/my-object.json'

# Create an S3 client
s3_client = S3Client(resource, server, access_key, secret_key)

# Parse the S3 path to get the bucket and key
bucket_name, object_key = parse_s3_path(s3_path)

# Retrieve the object content from S3
s3_object = s3_client.get_object(bucket_name, object_key)

# Read the object content and parse it as JSON
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 using this information. 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. Organize and generate TrainingDML-AI code based on the local dataset

Organizing and generating TrainingDML-AI code based on the local dataset requires two ways to obtain information: manual input and function calls, including the following steps:

2.2.1. Organize the 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 a *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. Organize 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. 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) Functionality:

- 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. YAML configuration file schema is described in encoding YAML configuration file schema. Here is provided to convert the YAML file encoding of the dataset to TrainingDML-AI encoding.

(1) Function: yaml_to_eo_tdml(yaml_path)

a) Functionality:

- Convert the YAML file to *EOTrainingDataset*.

b) Parameter:

- yaml_path: The path where the YAML file 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

(1) Function: convert_coco_to_tdml(coco_dataset_path, output_json_path)

a) Functionality:

- Object recognition task: Convert 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) Functionality:

- Semantic segmentation task: Convert 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* class is used to handle Earth Observation (EO) image scene classification training datasets. This class inherits from *VisionDataset* and can load datasets in batches by index into the model. Below is a detailed introduction to this class, its functions, and specific parameters.

Methods:

(1) *init*(self, td_list, root, class_map, transform=None)

a) Functionality:

- Initializes 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) Functionality:

- Extracts 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) Functionality:

- Returns the size of the dataset.

b) Returns:

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

(4) *getitem*(self, item)

a) Functionality:

- Retrieves the image and label corresponding to the given index, applies necessary preprocessing and transformations.

b) Parameters:

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

(5) class_to_idx(self)

a) Functionality:

- Returns 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* class implements an iterator-based method for loading datasets in batches into the model by inheriting from *IterDataPipe*. This method is suitable for handling large-scale datasets and enables efficient data loading and preprocessing. Below is a detailed introduction to this class, its functions, and specific parameters.

Methods:

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

a) 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.
- Functionality: Initializes the dataset object and sets necessary attributes.

(2) *iter*(self)

a) Functionality:

- Iteratively loads images and labels, applies 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

TorchSemanticSegmentationTD is a custom dataset class designed for semantic segmentation tasks, specifically for Earth Observation (EO) images. This class extends *VisionDataset* and is compatible with PyTorch's data loading utilities. It handles the loading of image and label paths, applies transformations, and provides the data in a format suitable for training deep learning models.

Methods:

(1) *init*(self, td_list, root, classes, transform=None)

a) Functionality:

- Initializes 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) Functionality:

- Returns the number of items in the dataset.

b) Returns:

- int - The number of items in the dataset.

(3) *getitem*(self, item)

a) Functionality:

- Retrieves the image and label at the specified index, applies 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) Functionality:

- Loads 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) Functionality:

- Processes 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.

Usage:

To use this dataset, 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 and provide a list of class names for segmentation. 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 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

TorchSemanticSegmentationDataPipe is a data pipeline class for semantic segmentation tasks, inheriting from *IterDataPipe* and *ABC*. This class can load and preprocess Earth Observation (EO) images 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, class_list=None, crop=None, transform=None)

a) Functionality:

- Initializes 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) Functionality:

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

b) Returns:

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

Usage:

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 Recognition task

3.3.1. Dataset Approach

TorchObjectDetectionTD is a Torch Dataset for EO image object detection training dataset. This class inherits from *VisionDataset* and is designed to be compatible with PyTorch's data loading tool.

(1) *init*(self, td_list, root, class_map, transform=None)

a) Functionality:

- 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) Functionality:

- Returns the number of items in the dataset.

b) Return:

- length: Number of items in the dataset.

(3) *getitem*(self, index)

a) Functionality:

- 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

TorchObjectDetectionDataPipe is a data pipeline class used for target recognition tasks, inherited from *IterDataPipe*. This class can load and preprocess Earth Observation (EO) images and provide data in the form of iterators. It supports loading data from cache and can crop and transform images as needed.

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

a) Functionality:

- 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) Functionality:

- 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

TorchChangeDetectionTD is a dataset class designed for change detection tasks, extending *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) Functionality:

- Initializes 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) Functionality:

- Returns the number of samples in the dataset.

b) Returns:

- int - The number of samples in the dataset.

(3) *getitem*(self, index)

a) Functionality:

- Retrieves 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) Functionality:

- Loads sample data, generating paths for images before and after changes 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

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) Functionality:

- Initializes 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) Functionality:

- 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 and labels. You also need to specify the root directory for storing these files, the cache file path, and provide 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 = 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.

Chapter 4. PyTDML Use Case

Here we take the object detection dataset DOTA2.0 and RSOD dataset as examples to demonstrates the process of PyTDML.

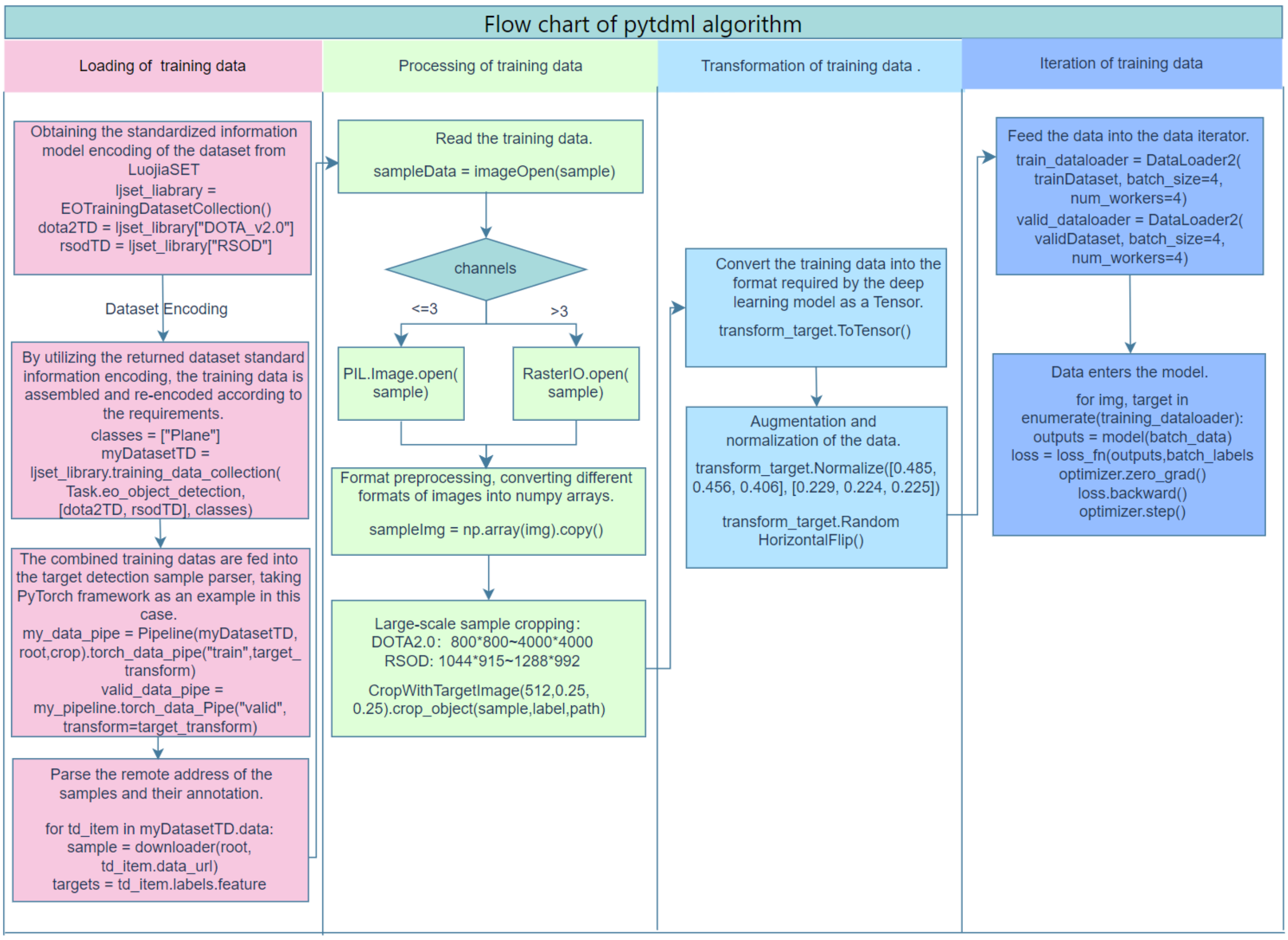
DOTA2.0:

- 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 shown in the following figure. It contains 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.
```