- Testing Standards
- Thank You!

PathML

- Creating Preprocessing Pipelines
- View PathML on GitHub

Previous Next

Creating Preprocessing Pipelines

Preprocessing pipelines define how raw images are transformed and prepared for downstream analysis. The `pathml.preprocessing` module provides tools to define modular preprocessing pipelines for whole-slide images.

In this section we will walk through how to define a `Pipeline` object by composing pre-made `Transform` objects, and how to implement a new custom `Transform`.

What is a Transform?

The `Transform` is the building block for creating preprocessing pipelines.

Each `Transform` applies a specific operation to a `Tile` which may include modifying an input image, creating or modifying pixel-level metadata (i.e., masks), or creating or modifying image-level metadata (e.g., image quality metrics or an AnnData counts matrix).

Schematic diagram of a `Transform` operating on a tile. In this example, several masks are created (represented by stacked rectangles) as well as several labels (depicted here as cubes).

examples of Transforms

Examples of several types of `Transform`

What is a Pipeline?

A preprocessing pipeline is a set of independent operations applied sequentially. In `PathML`, a `Pipeline` is defined as a sequence of `Transform` objects. This makes it easy to compose a custom `Pipeline` by mixing-and-matching:

schematic diagram of modular pipeline composition

Schematic diagram of `Pipeline` composition from a set of modular components

In the PathML API, this is concise:

```
from pathml.preprocessing import Pipeline, BoxBlur, TissueDetectionHE

pipeline = Pipeline([
    BoxBlur(kernel_size=15),
    TissueDetectionHE(mask_name = "tissue", min_region_size=500,
                      threshold=30, outer_contours_only=True)
])
```

In this example, the preprocessing pipeline will first apply a box blur kernel, and then apply tissue detection.

Creating custom Transforms❑

Note

For advanced users

In some cases, you may want to implement a custom `Transform`. For example, you may want to apply a transformation which is not already implemented in `PathML`. Or, perhaps you want to create a new transformation which combines several others.

To define a new custom `Transform`, all you need to do is create a class which inherits from `Transform` and implements an `apply()` method which takes a `Tile` as an argument and modifies it in place. You may also implement a functional method `F()`, although that is not strictly required.

For example, let's take a look at how `BoxBlur` is implemented:

```
class BoxBlur(Transform):
    """Box (average) blur kernel."""
    def __init__(self, kernel_size=5):
        self.kernel_size = kernel_size

    def F(self, image):
        return cv2.boxFilter(image, ksize = (self.kernel_size, self.kernel_size), ddepth = -1)

    def apply(self, tile):
        tile.image = self.F(tile.image)
```

Once you define your custom `Transform`, you can plug it in with any of the other `Pipeline`, etc.

Previous Next

- Coordinate system conventions
- Setting up a local development environment
- Running tests
- Building documentation locally
- Checking code coverage
- How to contribute code, documentation, etc.
- Versioning and Distributing
- Code Quality
- Documentation Standards
- Testing Standards

Thank You!

PathML

-
- DataLoaders
- View PathML on GitHub

Previous Next

DataLoaders

After running a preprocessing pipeline and writing the resulting `.h5path` file to disk, the next step is to create a DataLoader for feeding tiles into a machine learning model in PyTorch.

To do this, use the `TileDataset` class and then wrap it in a PyTorch DataLoader:

```
dataset = TileDataset("/path/to/file.h5path")
dataloader = torch.utils.data.DataLoader(dataset, batch_size = 16, shuffle = True, num_workers = 4)
```

Note

Label dictionaries are not standardized, as users are free to store whatever labels they want. For that reason, PyTorch cannot automatically stack labels into batches. It may therefore be necessary to create a custom `collate_fn` to specify how to create batches of labels. See here.

This provides an interface between PathML and the broader ecosystem of machine learning tools built on PyTorch. For more information on how to use Datasets and DataLoaders, please see the PyTorch documentation and tutorials.

Previous Next

- Documentation Standards
- Testing Standards
- Thank You!

PathML

- Datasets
- View PathML on GitHub

Previous Next

Datasets❑

The `pathml.datasets` module provides easy access to common datasets for standardized model evaluation and comparison.

DataModules❑

`PathML` uses `DataModules` to encapsulate datasets. DataModule objects are responsible for downloading the data (if necessary) and formatting the data into `DataSet` and `DataLoader` objects for use in downstream tasks. Keeping everything in a single object is easier for users and also facilitates reproducibility.

Inspired by PyTorch Lightning.

Using public datasets❑

PathML has built-in support for several public datasets:

Datasets❑

Dataset	Description	Image type	Size
`PanNukeDataModule`	Pixel-level nucleus classification, with 6 nucleus types and 19 tissue types. Images are 256px RGB. [PanNuke1] [PanNuke2]	H&E	n=7901 (37.33 GB)
`DeepFocusDataModule`	Patch-level focus classification with 3 IHC and 1 H&E histologies. [DeepFocus]	H&E, IHC	n=204k (10.0 GB)

References❑

[PanNuke1]

Gamper, J., Koohbanani, N.A., Benet, K., Khuram, A. and Rajpoot, N., 2019, April. PanNuke: an open pan-cancer histology dataset for nuclei instance segmentation and classification. In European Congress on Digital Pathology (pp. 11-19). Springer, Cham.

[PanNuke2]

Gamper, J., Koohbanani, N.A., Graham, S., Jahanifar, M., Khurram, S.A., Azam, A., Hewitt, K. and Rajpoot, N., 2020. PanNuke Dataset Extension, Insights and Baselines. arXiv preprint arXiv:2003.10778.

[DeepFocus]

Senaras, C., Niazi, M., Lozanski, G., Gurcan, M., 2018, October. Deepfocus: Detection of out-of-focus regions in whole slide digital images using deep learning. PLOS One 13(10): e0205387.

Previous Next

Graphs

PathML provides a Graph API to construct cell or tissue graphs from Whole-Slide Images (WSIs).

Note

Graphs are a data structure comprised of nodes connected by edges, which allow for explicit modeling of spatial relationships. In computational pathology, nodes may represent tissue regions or individual nuclei, and the resulting graph structure can be used to study the spatial organization of the specimen.

We provide template code below for cell graph construction.

```
# load packages
from pathml.core import HESlide

from pathml.preprocessing import Pipeline, NucleusDetectionHE

from pathml.graph import KNNGraphBuilder
from pathml.graph.utils import get_full_instance_map

# Define slide path
slide_path = 'PATH TO SLIDE'

# Initialize pathml.core.slide_data.HESlide object
wsi = HESlide(slide_path, name = slide_path, backend = "openslide", stain = 'HE')

# Set up PathML pipeline for nuclei detection
pipeline = Pipeline([NucleusDetectionHE(mask_name = "detect_nuclei")])

# Run pipeline to get nuclei segmentation masks
wsi.run(pipeline, overwrite_existing_tiles=True, distributed=False, tile_pad=True, tile_size=PATCH_SIZE)

# Extract the nuclei segmentation masks
image, nuclei_map, nuclei_centroid = get_full_instance_map(wsi, patch_size = PATCH_SIZE, mask_name="detect_nuclei")

# Initialize a pathml.graph.KNNGraphBuilder object
knn_graph_builder = KNNGraphBuilder(k=5, thresh=50, add_loc_feats=True)

# Build the cell graph
cell_graph = knn_graph_builder.process(nuclei_map, return_networkx=True)
```

For a full example that considers tissue graph construction and feature extraction for machine learning, please refer to the Graph construction and processing tab under Examples.

Previous Next

- Documentation Standards
- Testing Standards
- Thank You!

PathML

- HDF5 Integration
- View PathML on GitHub

Previous Next

HDF5 Integration❑

Note

For advanced users

Overview❑

A single whole-slide image may contain on the order of 10^{10} pixels, making it infeasible to process entire images in RAM. `PathML` supports efficient manipulation of large-scale imaging data via the **h5path** format, a hierarchical data structure which allows users to access small regions of the processed WSI without loading the entire image. This feature reduces the RAM required to run a `PathML` workflow (pipelines can be run on a consumer laptop), simplifies the reading and writing of processed WSIs, improves data exploration utilities, and enables fast reading for downstream tasks (e.g. PyTorch Dataloaders). Since slides are managed on disk, your drive must have sufficient storage. Performance will benefit from storage with fast read/write (SSD, NVMe).

How it Works❑

Each `SlideData` object is backed by an `.h5path` file on disk. All interaction with the `.h5path` file is handled automatically by the `h5pathManager`. For example, when a user calls `slidedata.tiles[tile_key]`, the `h5pathManager` will retrieve the tile from disk and return it, without the user needing to worry about accessing the HDF5 file themself. As tiles are extracted and passed to a preprocessing pipeline, the `h5pathManager` also handles aggregating the processed tiles into the `.h5path` file. At the conclusion of preprocessing, the h5py object can optionally be permanently written to disk in `.h5path` format via the `SlideData.write()` method.

About HDF5❑

The internals of `PathML` as well as the `.h5path` file format are based on the hierarchical data format HDF5, implemented by h5py.

HDF5 format consists of 3 types of elements:

Groups	A “container,” similar to a directory in a filesystem. Groups may contain Datasets, Attributes, or other Groups.
Datasets	Rectangular collection of data elements. Wraps `np.ndarray` .
Attributes	Small named metadata elements. Each attribute is attached to a Group or Dataset.

`Groups` are container-like and can be queried like dictionaries:

```
import h5py
root = h5py.File('path/to/file.h5path', 'r')
masks = root['masks']
```

`Datasets` can be treated like `numpy.ndArray` objects:

Important

To retrieve a `numpy.ndArray` object from `h5py.Dataset` you must slice the Dataset with NumPy fancy-indexing syntax: for example […] to retrieve the full array, or [a:b, …] to return the array with first dimension sliced to the interval [a, b].

```
import h5py
root = h5py.File('path/to/file.h5path', 'r')
im = root['tiles']['(0, 0)']['array'][...]
im_slice = root['tiles']['(0, 0)']['array'][0:100, 0:100, :]
```

`Attributes` are stored in a `.attrs` object which can be queried like a dictionary:

```
import h5py
root = h5py.File('path/to/file.h5path', 'r')
tile_shape = root['tiles'].attrs['tile_shape']
```

`.h5path` File Format

h5path utilizes a self-describing hierarchical file system similar to `SlideData`.

Here we examine the **h5path** file format in detail:

```
root/                           (Group)
├── fields/                     (Group)
│   ├── name                    (Attribute, str)
│   ├── shape                   (Attribute, tuple)
│   ├── labels                  (Group)
```

```
│   │   ├── label1              (Attribute, [str, int, float, array])
│   │   ├── label2              (Attribute, [str, int, float, array])
│   │   └── etc...
│   └── slide_type              (Group)
│       ├── stain               (Attribute, str)
│       ├── tma                 (Attribute, bool)
│       ├── rgb                 (Attribute, bool)
│       ├── volumetric          (Attribute, bool)
│       └── time_series         (Attribute, bool)
├── masks/                      (Group)
│   ├── mask1                   (Dataset, array)
│   ├── mask2                   (Dataset, array)
│   └── etc...
├── counts                      (Group)
│   └── `.h5ad` format
└── tiles/                      (Group)
    ├── tile_shape              (Attribute, tuple)
    ├── tile_stride             (Attribute, tuple)
    ├── tile_key1/              (Group)
    │   ├── array               (Dataset, array)
    │   ├── masks/              (Group)
    │   │   ├── mask1           (Dataset, array)
    │   │   ├── mask2           (Dataset, array)
    │   │   └── etc...
    │   ├── coords              (Attribute, tuple)
    │   ├── name                (Attribute, str)
    │   └── labels/             (Group)
    │       ├── label1          (Attribute, [str, int, float, array])
    │       ├── label2          (Attribute, [str, int, float, array])
    │       └── etc...
    ├── tile_key2/              (Group)
    │   └── etc...
    └── etc...
```

Slide-level metadata is stored in the `fields/` group.

Slide-level counts matrix metadata is stored in the `counts/` group.

The `tiles/` group stores tile-level data. Each tile occupies its own group, and tile coordinates are used as keys for indexing tiles within the `tiles/` group. Within each tile's group, the `array` dataset contains the tile image, the `masks/` group contains tile-level masks, and other metadata including name, labels, and coords are stored as attributes. Slide-level metadata about tiling, including tile shape and stride, are stored as attributes in the `tiles/` group.

Whole-slide masks are stored in the `masks/` Group. All masks are enforced to be the same shape as the image array. However, when running a pipeline, these masks are moved to the tile-level and stored within the tile groups. The slide-level masks are therefore not saved when calling `SlideData.write()`.

We use `float16` as the data type for all Datasets.

Note

Be aware that the `h5path` format specification may change between major versions

Reading and Writing

SlideData objects are easily written to **h5path** format by calling SlideData.write(). All files with .h5 or .h5path extensions are loaded to SlideData objects automatically.

Previous Next

- Setting up a local development environment
- Running tests
- Building documentation locally
- Checking code coverage
- How to contribute code, documentation, etc.
- Versioning and Distributing
- Code Quality
- Documentation Standards
- Testing Standards

Thank You!

PathML

- Inference
- View PathML on GitHub

Previous Next

Inference

PathML comes with an API to run inference on ONNX models. This API relies on PathML's existing preprocessing pipeline.

Below is an example of how to run inference on a locally stored ONNX model.

```
# load packages
from pathml.core import SlideData

from pathml.preprocessing import Pipeline
import pathml.preprocessing.transforms as Transforms

from pathml.inference import Inference, remove_initializer_from_input

# Define slide path
slide_path = 'PATH TO SLIDE'

# Set path to model
model_path = 'PATH TO ONNX MODEL'
# Define path to export fixed model
new_path = 'PATH TO SAVE NEW ONNX MODEL'

# Fix the ONNX model by removing initializers. Save new model to `new_path`.
remove_initializer_from_input(model_path, new_path)

inference = Inference(model_path = new_path, input_name = 'data', num_classes = 8, model_type = 'segmentation')

# Create a transformation list
transformation_list = [
    inference
]

# Initialize pathml.core.slide_data.SlideData object
wsi = SlideData(slide_path, stain = 'Fluor')

# Set up PathML pipeline
pipeline = Pipeline(transformation_list)

# Run Inference
wsi.run(pipeline, tile_size = 1280, level = 0)
```

For an end to end example and explaination of the code, please see the PathML ONNX Tutorial tab under Examples.

        - HaloAIInference.apply()
    - RemoteTestHoverNet Class
        - RemoteTestHoverNet
            - RemoteTestHoverNet.apply()
            - RemoteTestHoverNet.remove()
    - RemoteMesmer Class
        - RemoteMesmer
            - RemoteMesmer.F()
            - RemoteMesmer.apply()
            - RemoteMesmer.inference()
            - RemoteMesmer.remove()
        - Helper functions
            - remove_initializer_from_input()
            - check_onnx_clean()
            - convert_pytorch_onnx()

Contributing

- Contributing
    - Submitting a bug report
    - Requesting a new feature
    - For developers
        - Coordinate system conventions
        - Setting up a local development environment
        - Running tests
        - Building documentation locally
        - Checking code coverage
        - How to contribute code, documentation, etc.
        - Versioning and Distributing
        - Code Quality
        - Documentation Standards
        - Testing Standards
    - Thank You!

PathML

-
- Loading Images
- View PathML on GitHub

Previous Next

Loading Images

Individual Images

The first step in any computational pathology workflow is to load the image from disk. In `PathML` this can be done in one line:

```
wsi = HESlide("../data/CMU-1.svs", name = "example")
```

Datasets of Images

Using “in-house” datasets from the local filesystem is also supported.

Simply initialize a `SlideDataset` object by passing a list of individual `SlideData` objects:

```
from pathlib import Path
from pathml.core import HESlide, SlideDataset

# assuming that all WSIs are in a single directory, all with .svs file extension
data_dir = Path("/path/to/data/")
wsi_paths = list(data_dir.glob("*.svs"))

# create a list of SlideData objects by loading each path
wsi_list = [HESlide(p) for p in wsi_paths]

# initialize a SlideDataset
dataset = SlideDataset(wsi_list)
```

Supported slide types

All slides are represented as `SlideData` objects.

We provide several convenience classes for loading common types of slides:

Slide Classes

Slide Class	Description
`HESlide`	H&E stained images.
`IHCSlide`	IHC stained images
`MultiparametricSlide`	Generic multidimensional, multichannel, time-series images (e.g. multiplexed immunofluorescence).
`VectraSlide`	Multiplex images from Vectra platform.
`CODEXSlide`	Multiplex images from CODEX platform.

It is also possible to load a slide by using the generic `SlideData` class and specifying explicitly the slide_type and which backend to use (refer to table below):

```
wsi = SlideData("../data/CMU-1.svs", name = "example", slide_backend = "openslide", slide_type = types.HE)
```

For more information on specifying `slide_type`, see full documentation at `SlideType`

Supported file formats

Whole-slide images can come in a variety of file formats, depending on the type of image and the scanner used. `PathML` has several backends for loading images, enabling support for a wide variety of data formats. All backends use the same API for interfacing with other parts of `PathML`. Choose the appropriate backend for the file format:

PathML Backends

Backend	Supported file types
`OpenSlideBackend`	`.svs`, `.tif`, `.tiff`, `.bif`, `.ndpi`, `.vms`, `.vmu`, `.scn`, `.mrxs`, `.svslide` Complete list of file types supported by OpenSlide
`DICOMBackend`	`.dcm`, `.dicom` Digital Imaging and Communications in Medicine (DICOM)
`BioFormatsBackend`	Supports almost all commonly used file formats, including multiparametric and volumetric TIFF files. `.1sc`, `.2fl`, `.acff`, `.afi`, `.afm`, `.aim`, `.al3d`, `.ali`, `.am`, `.amiramesh`, `.apl`, `.arf`, `.avi`, `.bif`, `.bin`, `.bip`, `.bmp`, `.c01`, `.cfg`, `.ch5`, `.cif`, `.cr2`, `.crw`, `.cxd`, `.czi`, `.dat`, `.dat`, `.db`, `.dib`, `.dm2`, `.dm3`, `.dm4`, `.dti`, `.dv`, `.eps`, `.epsi`, `.exp`, `.fdf`, `.fff`, `.ffr`, `.fits`, `.fli`, `.frm`, `.gel`, `.grey`, `.hdr`, `.hdr`, `.hdr`, `.hdr`, `.hed`, `.his`, `.htd`, `.htd`, `.hx`, `.i2i`, `.ics`, `.ids`, `.im3`, `.img`, `.img`, `.ims`, `.inr`, `.ipl`, `.ipm`, `.ipw`, `.j2k`, `.jp2`, `.jpf`, `.jpk`, `.jpx`, `.klb`, `.l2d`, `.labels`, `.lei`, `.lif`, `.liff`, `.lim`, `.lms`, `.lsm`, `.map`, `.mdb`, `.mnc`, `.mng`, `.mod`, `.mov`, `.mrc`, `.mrcs`, `.mrw`, `.msr`, `.msr`, `.mtb`, `.mvd2`, `.naf`, `.nd`, `.nef`, `.nhdr`, `.nii`, `.nii.gz`, `.nrrd`, `.obf`, `.obsep`, `.oib`, `.oif`, `.oir`, `.ome`, `.ome.btf`, `.ome.tf2`, `.ome.tf8`, `.ome.tif`, `.ome.tiff`, `.ome.xml`, `.par`, `.pbm`, `.pcoraw`, `.pcx`, `.pds`, `.pgm`, `.pic`, `.pict`, `.png`, `.pnl`, `.ppm`, `.pr3`, `.ps`, `.psd`, `.qptiff`, `.r3d`, `.raw`, `.rcpnl`, `.rec`, `.rec`, `.scn`, `.scn`, `.sdt`, `.seq`, `.sif`, `.sld`, `.sld`, `.sm2`, `.sm3`, `.spc`, `.spe`, `.spi`, `.st`, `.stk`, `.stk`, `.stp`, `.sxm`, `.tfr`, `.tga`, `.tif`, `.tiff`, `.tnb`, `.top`, `.vff`, `.vsi`, `.vws`, `.wat`, `.wlz`, `.wpi`, `.xdce`, `.xml`, `.xqd`, `.xqf`, `.xv`, `.xys`, `.zfp`, `.zfr`, `.zvi` Complete list of file types supported by Bio-Formats

Previous Next

- Documentation Standards
- Testing Standards
- Thank You!

PathML

- Models
- View PathML on GitHub

Previous Next

Models❑

`PathML` comes with model architectures ready to use out of the box.

Model	Reference	Description
`HoVerNet`	[HoVerNet]	A model for nucleus segmentation and classification in H&E images
`HACTNet`	[HACTNet]	A graph neural network (GNN) for cancer subtyping

You can also use models from fantastic resources such as torchvision.models and pytorch-image-models (timm).

References❑

[HoVerNet]

Graham, S., Vu, Q.D., Raza, S.E.A., Azam, A., Tsang, Y.W., Kwak, J.T. and Rajpoot, N., 2019. Hover-Net: Simultaneous segmentation and classification of nuclei in multi-tissue histology images. Medical Image Analysis, 58, p.101563.

[HACTNet]

Pati, P., Jaume, G., Foncubierta-Rodriguez, A., Feroce, F., Anniciello, A.M., Scognamiglio, G., Brancati, N., Fiche, M., Dubruc, E., Riccio, D. and Di Bonito, M., 2022. Hierarchical graph representations in digital pathology. Medical image analysis, 75, p.102264.

- Documentation Standards
- Testing Standards
- Thank You!

PathML

- Overview
- View PathML on GitHub

Previous Next

Overview

`PathML` is a toolkit supporting each step in the computational pathology research workflow.

_images/schematic_design.jpg

We aim to accelerate research, reduce barriers to entry for new researchers, and promote open science in computational pathology.

Note

We provide pre-built tools, public datasets, and documentation to allow anyone with beginner proficiency in Python to get started.

For advanced users, we provide a modular set of tools which can be composed into custom workflows, as well as complete API documentation to enable implementation of new features or tools on top of `PathML`.

License

The GNU GPL v2 version of PathML is made available via Open Source licensing. The user is free to use, modify, and distribute under the terms of the GNU General Public License version 2.

Commercial license options are available also. Please contact us at PathML@dfci.harvard.edu

Previous Next

- Documentation Standards
- Testing Standards
- Thank You!

PathML

- Running Preprocessing Pipelines
- View PathML on GitHub

Previous Next

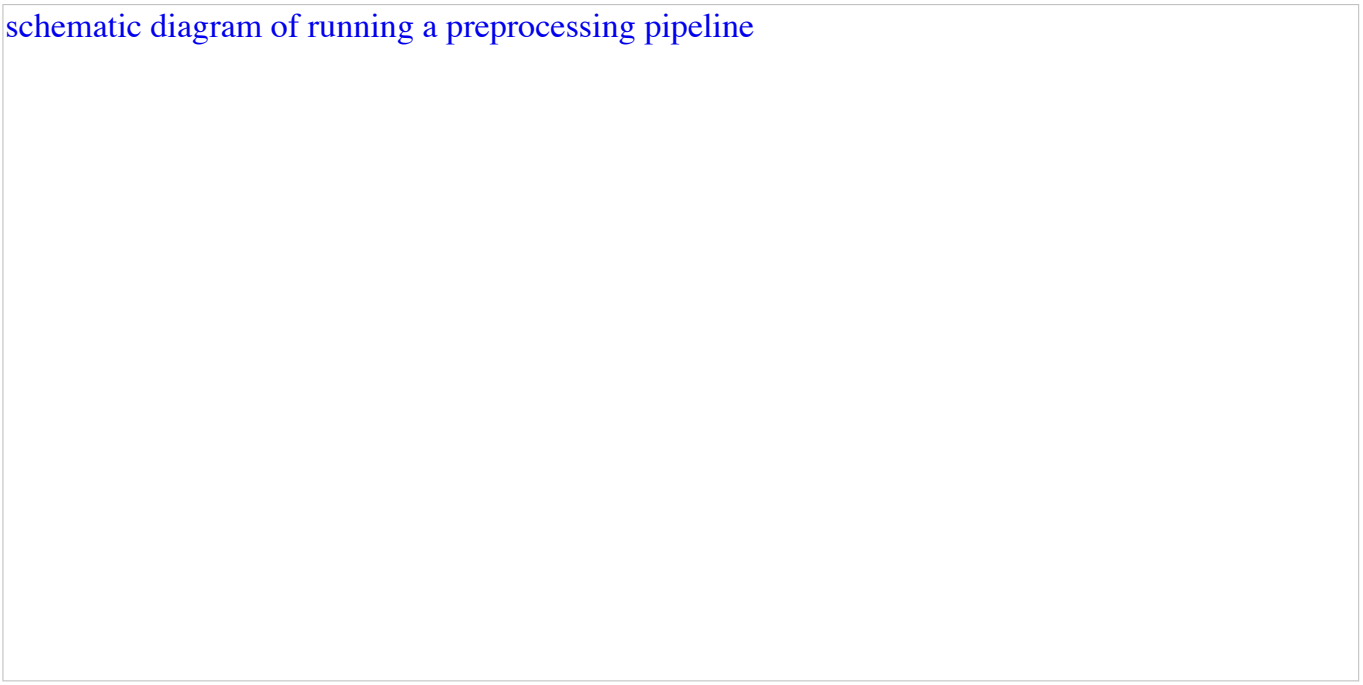
Running Preprocessing Pipelines

How it works

Whole-slide images are typically too large to load in memory, and computational requirements scale poorly in image size. `PathML` therefore runs preprocessing on smaller regions of the image which can be held in RAM, and then aggregates the results at the end.

Preprocessing pipelines are defined in `Pipeline` objects. When `SlideData.run()` is called, `Tile` objects are lazily extracted from the slide by the `SlideData.generate_tiles()` method and passed to the `Pipeline.apply()` method, which modifies the tiles in place. Finally, all processed tiles are aggregated into a single `h5py.Dataset` array and a PyTorch Dataset is generated.

Each tile is processed independently, and this data-parallel design makes it easy to utilize computational resources and scale up to large datasets of gigapixel-scale whole-slide images.



Schematic diagram of running a preprocessing pipeline. Tiles are extracted from the WSI and processed in parallel, before finally being aggregated into a HDF5 file on disk.

Preprocessing a single WSI

Get started by loading a WSI from disk and running a preprocessing pipeline:

```
from pathml.core import HESlide
from pathml.preprocessing import Pipeline, BoxBlur, TissueDetectionHE

wsi = HESlide("../data/CMU-1.svs", name = "example")

pipeline = Pipeline([
    BoxBlur(kernel_size=15),
    TissueDetectionHE(mask_name = "tissue", min_region_size=500,
                      threshold=30, outer_contours_only=True)
])

wsi.run(pipeline)
```

Preprocessing a dataset of WSI

Pipelines can also be run on entire datasets, with no change to the code: Here we create a mock `SlideDataset` and run the same pipeline as above:

```
from pathml.core import HESlide, SlideDataset
from pathml.preprocessing import Pipeline, BoxBlur, TissueDetectionHE

# create demo dataset
n = 10
slide_list = [HESlide("../data/CMU-1.svs", name = "example") for _ in range(10)]
slide_dataset = SlideDataset(slide_list)

pipeline = Pipeline([
    BoxBlur(kernel_size=15),
    TissueDetectionHE(mask_name = "tissue", min_region_size=500,
                      threshold=30, outer_contours_only=True)
])

slide_dataset.run(pipeline)
```

Distributed processing

When running a pipeline, `PathML` will use multiprocessing by default to distribute the workload to all available cores. This allows users to efficiently process large datasets by scaling up computational resources (local cluster, cloud machines, etc.) without needing to make any changes to the code. It also makes it feasible to run preprocessing pipelines on less powerful machines, e.g. laptops for quick prototyping.

We use dask.distributed as the backend for multiprocessing. Jobs are submitted to a `Client`, which takes care of sending them to available resources and collecting the results. By default, `PathML` creates a local cluster.

Several libraries exist for creating `Clients` on different systems, e.g.:

- dask-kubernetes for kubernetes
- dask-jobqueue for common job queuing systems including PBS, Slurm, MOAB, SGE, LSF, and HTCondor typically found in high performance supercomputers, academic research institutions, etc.
- dask-yarn for Hadoop YARN clusters

To take full advantage of available computational resources, users must initialize the appropriate `Client` object for their system and pass it as an argument to the `SlideData.run()` or `SlideDataset.run()`. Please refer to the Dask documentation linked above for complete information on creating the `Client` object to suit your needs.

Previous Next
