Contributing

Multiparametric Imaging: CODEX

View on GitHub

In this tutorial, we will analyze the CODEX images provided by Schürch et. al: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7479520/.

Here, we will use PathML to process a large number of CODEX slides (140 slides) simultaneously through the SlideDataset class. We also show how the generated count matrices can be used to investigate the complex spatial architecture of the iTME in colorectal cancer.

[1]:

```
from os import listdir, path, getcwd
import glob
import re
import pandas as pd
from pathml.core import SlideDataset
from pathml.core.slide_data import VectraSlide
from pathml.core.slide_data import CODEXSlide
from pathml.preprocessing.pipeline import Pipeline
from pathml.preprocessing.transforms import SegmentMIF, QuantifyMIF, CollapseRunsCODEX
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import rc_context
from dask.distributed import Client, LocalCluster
from deepcell.utils.plot_utils import make_outline_overlay
from deepcell.utils.plot_utils import create_rgb_image
import scanpy as sc
import squidpy as sq
import anndata as ad
import bbknn
from joblib import parallel_backend
```

```
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The text.latex.preview rcparam was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The mathtext.fallback_to_cm rcparam was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle: Support for sett
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The validate_bool_maybe_none function was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The savefig.jpeg_quality rcparam was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The keymap.all_axes rcparam was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The animation.avconv_path rcparam was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
In /Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/matplotlib/mpl-data/stylelib/_classic_test.mplstyle:
The animation.avconv_args rcparam was deprecated in Matplotlib 3.3 and will be removed two minor releases later.
/Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/tensorflow/python/keras/optimizer_v2/optimizer_v2.py:374: UserWarning: T
  warnings.warn(
```

[2]:

```
# set working directory
%cd "~/Documents/Research/Projects/CRC_TMA"
```

```
/Users/mohamedomar/Documents/Research/Projects/CRC_TMA
```

[3]:

```
## Read channel names
channelnames = pd.read_csv(
    "data/channelNames.txt", header=None, dtype=str, low_memory=False
)
channelnames
```

[3]:

	0
0	HOECHST1
1	blank
2	blank
3	blank
4	HOECHST2
...	...
87	empty-Cy5-22
88	HOECHST23
89	empty-A488-23
90	empty-Cy3-23
91	DRAQ5

92 rows × 1 columns

Reading the slides

[21]:

```
dirpath = r"/Volumes/Mohamed/CRC"

# assuming that all slides are in a single directory, all with .tif file extension
for A, B in [listdir(dirpath)]:
    vectra_list_A = [
        CODEXSlide(p, stain="IF") for p in glob.glob(path.join(dirpath, A, "*.tif"))
    ]
    vectra_list_B = [
        CODEXSlide(p, stain="IF") for p in glob.glob(path.join(dirpath, B, "*.tif"))
    ]
    # Fix the slide names and add origin labels (A, B)
    for slide_A, slide_B in zip(vectra_list_A, vectra_list_B):
        slide_A.name = re.sub("X.*", "A", slide_A.name)
        slide_B.name = re.sub("X.*", "B", slide_B.name)
    # Store all slides in a SlideDataSet object
    dataset = SlideDataset(vectra_list_A + vectra_list_B)
```

Define and run the preprocessing pipeline

[8]:

```
# Here, we use DAPI (channel 0) and vimentin (channel 29) for segmentation
# Z=0 since we are processing images with a single slice (best focus)
pipe = Pipeline(
    [
        CollapseRunsCODEX(z=0),
        SegmentMIF(
            model="mesmer",
            nuclear_channel=0,
            cytoplasm_channel=29,
            image_resolution=0.377442,
        ),
        QuantifyMIF(segmentation_mask="cell_segmentation"),
    ]
)

# Initialize a dask cluster using 10 workers. PathML pipelines can be run in distributed mode on
# cloud compute or a cluster using dask.distributed.
cluster = LocalCluster(n_workers=10, threads_per_worker=1, processes=True)
client = Client(cluster)

# Run the pipeline
dataset.run(
    pipe, distributed=True, client=client, tile_size=(1920, 1440), tile_pad=False
)

# Write the processed datasets to disk
dataset.write("data/dataset_processed.h5")
```

Extract and concatenate the resulting count matrices

Combine the count matrices into a single adata object:

[ ]:

```
##  Combine the count matrices into a single adata object:
adata = ad.concat(
    [x.counts for x in dataset.slides], join="outer", label="Region", index_unique="_"
)
# Fix and replace the regions names
origin = adata.obs["Region"]
origin = origin.astype(str).str.replace("[^a-zA-Z0-9 \n\.]", "")
origin = origin.astype(str).str.replace("[\n]", "")
origin = origin.str.replace("SlideDataname", "")
adata.obs["Region"] = origin

# save the adata object
adata_combined.write(filename="./data/adata_combined.h5ad")
```

[6]:

```
# Rename the variable names (channels) in the adata object
adata.var_names = channelnames[0]
adata.var_names_make_unique()
```

Filter the cells using the DAPI and DRAQ5 intensity

[7]:

```
# Plot the DAPI intensity distribution:
sc.pl.violin(adata, keys=["HOECHST1", "DRAQ5"], multi_panel=True)
```

```
/Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/anndata/_core/anndata.py:1220: FutureWarning: The `inplace` parameter in
  c.reorder_categories(natsorted(c.categories), inplace=True)
... storing 'coords' as categorical
/Users/mohamedomar/opt/anaconda3/envs/pathml2/lib/python3.8/site-packages/anndata/_core/anndata.py:1220: FutureWarning: The `inplace` parameter in
  c.reorder_categories(natsorted(c.categories), inplace=True)
... storing 'slice' as categorical
```

../_images/examples_link_codex_13_1.png

[8]:

```
# Remove cells with low DAPI intensity (most likely artifacts)
adata = adata[adata[:, "HOECHST1"].X > 60, :]
adata = adata[adata[:, "DRAQ5"].X > 100, :]
```

[9]:

```
# Remove the empty and nuclear channels
keep = [
    "CD44 - stroma",
    "FOXP3 - regulatory T cells",
    "CD8 - cytotoxic T cells",
    "p53 - tumor suppressor",
    "GATA3 - Th2 helper T cells",
    "CD45 - hematopoietic cells",
    "T-bet - Th1 cells",
    "beta-catenin - Wnt signaling",
    "HLA-DR - MHC-II",
    "PD-L1 - checkpoint",
    "Ki67 - proliferation",
    "CD45RA - naive T cells",
    "CD4 - T helper cells",
    "MUC-1 - epithelia",
    "CD30 - costimulator",
    "CD2 - T cells",
    "Vimentin - cytoplasm",
    "CD20 - B cells",
    "LAG-3 - checkpoint",
    "Na-K-ATPase - membranes",
    "CD5 - T cells",
    "IDO-1 - metabolism",
    "Cytokeratin - epithelia",
    "CD11b - macrophages",
    "CD56 - NK cells",
    "aSMA - smooth muscle",
    "BCL-2 - apoptosis",
    "CD25 - IL-2 Ra",
    "PD-1 - checkpoint",
    "Granzyme B - cytotoxicity",
    "EGFR - singling",
    "VISTA - costimulator",
    "CD15 - granulocytes",
    "ICOS - costimulator",
    "Synaptophysin - neuroendocrine",
    "GFAP - nerves",
    "CD7 - T cells",
    "CD3 - T cells",
    "Chromogranin A - neuroendocrine",
    "CD163 - macrophages",
    "CD45RO - memory cells",
    "CD68 - macrophages",
    "CD31 - vasculature",
    "Podoplanin - lymphatics",
    "CD34 - vasculature",
    "CD38 - multifunctional",
    "CD138 - plasma cells",
]
adata = adata[:, keep]
```

[10]:

```
adata_combined
```

[10]:

```
View of AnnData object with n_obs × n_vars = 272829 × 47
    obs: 'Region', 'coords', 'euler_number', 'filled_area', 'slice', 'tile', 'x', 'y', 'TMA'
    obsm: 'spatial'
    layers: 'max_intensity', 'min_intensity'
```

[11]:

```
# Rename the markers
adata.var_names = [
    "CD44",
    "FOXP3",
    "CD8",
    "p53",
    "GATA3",
    "CD45",
    "T-bet",
    "beta-cat",
    "HLA-DR",
    "PD-L1",
    "Ki67",
    "CD45RA",
    "CD4",
    "MUC-1",
    "CD30",
    "CD2",
    "Vimentin",
    "CD20",
    "LAG-3",
    "Na-K-ATPase",
    "CD5",
    "IDO-1",
    "Cytokeratin",
    "CD11b",
    "CD56",
    "aSMA",
    "BCL-2",
    "CD25-IL-2Ra",
    "PD-1",
    "Granzyme B",
    "EGFR",
    "VISTA",
    "CD15",
    "ICOS",
    "Synaptophysin",
    "GFAP",
    "CD7",
    "CD3",
    "ChromograninA",
    "CD163",
    "CD45RO",
    "CD68",
    "CD31",
    "Podoplanin",
    "CD34",
    "CD38",
    "CD138",
]
```

[12]:

```
# store the raw data for further use (differential expression or other analysis that uses raw counts)
adata.raw = adata
```

[13]:

```
## Add patients and groups info:
# Annotation dict for patients (the public data does not provide an easy way to map samples to phenotypes)
regions_to_patients = dict(
    reg001_A="1",
    reg001_B="1",
    reg002_A="1",
    reg002_B="1",
    reg003_A="2",
    reg003_B="2",
    reg004_A="2",
    reg004_B="2",
    reg005_A="3",
    reg005_B="3",
    reg006_A="3",
    reg006_B="3",
    reg007_A="4",
    reg007_B="4",
    reg008_A="4",
    reg008_B="4",
    reg009_A="5",
    reg009_B="5",
    reg010_A="5",
    reg010_B="5",
    reg011_A="6",
    reg011_B="6",
    reg012_A="6",
    reg012_B="6",
    reg013_A="7",
    reg013_B="7",
    reg014_A="7",
    reg014_B="7",
    reg015_A="8",
    reg015_B="8",
    reg016_A="8",
    reg016_B="8",
    reg017_A="9",
    reg017_B="9",
    reg018_A="9",
    reg018_B="9",
    reg019_A="10",
    reg019_B="10",
    reg020_A="10",
    reg020_B="10",
    reg021_A="11",
    reg021_B="11",
    reg022_A="11",
    reg022_B="11",
    reg023_A="12",
    reg023_B="12",
    reg024_A="12",
    reg024_B="12",
    reg025_A="13",
    reg025_B="13",
    reg026_A="13",
    reg026_B="13",
    reg027_A="14",
    reg027_B="14",
    reg028_A="14",
    reg028_B="14",
    reg029_A="15",
    reg029_B="15",
    reg030_A="15",
    reg030_B="15",
    reg031_A="16",
    reg031_B="16",
    reg032_A="16",
    reg032_B="16",
    reg033_A="17",
    reg033_B="17",
    reg034_A="17",
    reg034_B="17",
    reg035_A="18",
    reg035_B="18",
    reg036_A="18",
    reg036_B="18",
    reg037_A="19",
    reg037_B="19",
    reg038_A="19",
    reg038_B="19",
    reg039_A="20",
    reg039_B="20",
    reg040_A="20",
    reg040_B="20",
    reg041_A="21",
    reg041_B="21",
    reg042_A="21",
    reg042_B="21",
    reg043_A="22",
    reg043_B="22",
    reg044_A="22",
    reg044_B="22",
    reg045_A="23",
    reg045_B="23",
    reg046_A="23",
    reg046_B="23",
    reg047_A="24",
    reg047_B="24",
    reg048_A="24",
    reg048_B="24",
    reg049_A="25",
    reg049_B="25",
    reg050_A="25",
    reg050_B="25",
    reg051_A="26",
    reg051_B="26",
    reg052_A="26",
    reg052_B="26",
    reg053_A="27",
    reg053_B="27",
    reg054_A="27",
    reg054_B="27",
    reg055_A="28",
    reg055_B="28",
    reg056_A="28",
    reg056_B="28",
    reg057_A="29",
    reg057_B="29",
    reg058_A="29",
    reg058_B="29",
    reg059_A="30",
    reg059_B="30",
    reg060_A="30",
    reg060_B="30",
    reg061_A="31",
    reg061_B="31",
    reg062_A="31",
    reg062_B="31",
    reg063_A="32",
    reg063_B="32",
    reg064_A="32",
    reg064_B="32",
    reg065_A="33",
    reg065_B="33",
    reg066_A="33",
    reg066_B="33",
    reg067_A="34",
    reg067_B="34",
    reg068_A="34",
    reg068_B="34",
    reg069_A="35",
    reg069_B="35",
    reg070_A="35",
    reg070_B="35",
)
```

Annotation dict for clinical groups: - CLR: Crohns like reaction - DII: Diffuse inflammatory infiltrate

[14]:

```
regions_to_groups = dict(
    reg001_A="CLR",
    reg001_B="CLR",
    reg002_A="CLR",
    reg002_B="CLR",
    reg003_A="DII",
    reg003_B="DII",
    reg004_A="DII",
    reg004_B="DII",
    reg005_A="DII",
    reg005_B="DII",
    reg006_A="DII",
    reg006_B="DII",
    reg007_A="DII",
    reg007_B="DII",
    reg008_A="DII",
    reg008_B="DII",
    reg009_A="DII",
    reg009_B="DII",
    reg010_A="DII",
    reg010_B="DII",
    reg011_A="CLR",
    reg011_B="CLR",
    reg012_A="CLR",
    reg012_B="CLR",
    reg013_A="DII",
    reg013_B="DII",
    reg014_A="DII",
    reg014_B="DII",
    reg015_A="DII",
    reg015_B="DII",
    reg016_A="DII",
    reg016_B="DII",
    reg017_A="DII",
    reg017_B="DII",
    reg018_A="DII",
    reg018_B="DII",
    reg019_A="CLR",
    reg019_B="CLR",
    reg020_A="CLR",
    reg020_B="CLR",
    reg021_A="CLR",
    reg021_B="CLR",
    reg022_A="CLR",
    reg022_B="CLR",
    reg023_A="CLR",
    reg023_B="CLR",
    reg024_A="CLR",
    reg024_B="CLR",
    reg025_A="CLR",
    reg025_B="CLR",
    reg026_A="CLR",
    reg026_B="CLR",
    reg027_A="DII",
    reg027_B="DII",
    reg028_A="DII",
    reg028_B="DII",
    reg029_A="DII",
    reg029_B="DII",
    reg030_A="DII",
    reg030_B="DII",
    reg031_A="DII",
    reg031_B="DII",
    reg032_A="DII",
    reg032_B="DII",
    reg033_A="CLR",
    reg033_B="CLR",
    reg034_A="CLR",
    reg034_B="CLR",
    reg035_A="DII",
    reg035_B="DII",
    reg036_A="DII",
    reg036_B="DII",
    reg037_A="CLR",
    reg037_B="CLR",
    reg038_A="CLR",
    reg038_B="CLR",
    reg039_A="CLR",
    reg039_B="CLR",
    reg040_A="CLR",
    reg040_B="CLR",
    reg041_A="CLR",
    reg041_B="CLR",
    reg042_A="CLR",
    reg042_B="CLR",
    reg043_A="DII",
    reg043_B="DII",
    reg044_A="DII",
    reg044_B="DII",
    reg045_A="DII",
    reg045_B="DII",
    reg046_A="DII",
    reg046_B="DII",
    reg047_A="CLR",
    reg047_B="CLR",
    reg048_A="CLR",
    reg048_B="CLR",
    reg049_A="DII",
    reg049_B="DII",
    reg050_A="DII",
    reg050_B="DII",
    reg051_A="DII",
    reg051_B="DII",
    reg052_A="DII",
    reg052_B="DII",
    reg053_A="DII",
    reg053_B="DII",
    reg054_A="DII",
    reg054_B="DII",
    reg055_A="CLR",
    reg055_B="CLR",
    reg056_A="CLR",
    reg056_B="CLR",
    reg057_A="CLR",
    reg057_B="CLR",
    reg058_A="CLR",
    reg058_B="CLR",
    reg059_A="DII",
    reg059_B="DII",
    reg060_A="DII",
    reg060_B="DII",
    reg061_A="DII",
    reg061_B="DII",
    reg062_A="DII",
    reg062_B="DII",
    reg063_A="CLR",
    reg063_B="CLR",
    reg064_A="CLR",
    reg064_B="CLR",
    reg065_A="CLR",
    reg065_B="CLR",
    reg066_A="CLR",
    reg066_B="CLR",
    reg067_A="CLR",
    reg067_B="CLR",
    reg068_A="CLR",
    reg068_B="CLR",
    reg069_A="CLR",
    reg069_B="CLR",
    reg070_A="CLR",
    reg070_B="CLR",
)
```

[15]:

```
# map each slide to its source patient and clinical group (CLR vs DII)
adata.obs["patients"] = adata.obs["Region"].map(regions_to_patients).astype("category")

adata.obs["groups"] = adata.obs["Region"].map(regions_to_groups).astype("category")
```

[18]:

```
# log transform and scale the data
sc.pp.log1p(adata)
sc.pp.scale(adata, max_value=10)
```

[19]:

```
# PCA and batch correction using Harmony
sc.tl.pca(adata)
sc.external.pp.harmony_integrate(adata, key="Region")
```

```
2021-08-14 23:23:40,665 - harmonypy - INFO - Iteration 1 of 10
2021-08-14 23:26:19,392 - harmonypy - INFO - Iteration 2 of 10
2021-08-14 23:29:16,153 - harmonypy - INFO - Iteration 3 of 10
2021-08-14 23:32:22,647 - harmonypy - INFO - Converged after 3 iterations
```

[20]:

```
# save for future use
adata.write(filename="./data/adata_harmony.h5ad")
```

[21]:

```
# Compute neighbors and UMAP embedding
sc.pp.neighbors(adata, n_neighbors=15, n_pcs=30, use_rep="X_pca_harmony")
sc.tl.umap(adata)
```

[22]:

```
# louvain clustering
with parallel_backend("threading", n_jobs=15):
    sc.tl.louvain(adata, resolution=3)
```

[26]:

```
# Plot UMAP
sc.pl.umap(adata, color=["patients", "groups", "louvain"], ncols=1)
```

../_images/examples_link_codex_28_0.png

Annotate the clusters based on the markers intensity

Here, we define a function for cell annotation which takes as input the processed adata object together with a set of thresholds for each marker/channel. The function annotates the cells based on the specified thresholds and return the annotated adata object with the cell coordinates.

[27]:

```
def phenotype(adata, annot_dict):
    """
    Given a dict of dicts including phenotypes and marker gene thresholds, phenotype cells.

    Args:
        adata : the anndata object.
        annot_dict (dict): annotation dictionary each key is a cell type of interest and
            its value is a dictionary indicating protein expression ranges for that cell type.
            Each value should be a tuple (min, max) containing the minimum and maximum thresholds.
    """

    # Get the count matrix
    data = adata.copy()
    countMat = data.to_df()

    # Annotate the cell types
    for label in annot_dict.keys():
        for key, value in annot_dict[label].items():
            cond = np.logical_and.reduce(
                [
                    (
                        (countMat[k] >= countMat[k].quantile(list(v)[0]))
                        & (countMat[k] <= countMat[k].quantile(list(v)[1]))
                    )
                    for k, v in annot_dict[label].items()
                ]
            )
            data.obs.loc[cond, "cell_types"] = label
            # replace nan with unknown
            data.obs.cell_types.fillna("unknown", inplace=True)
    return data
```

[30]:

```
annot_dict = {
    "CD3+ T lymphocytes": {"CD3": (0.85, 1.0), "CD4": (0.0, 0.50), "CD8": (0.00, 0.50)},
    "CD4+ T lymphocytes": {
        "CD3": (0.50, 1.0),
        "CD4": (0.50, 1.0),
        "CD8": (0.0, 0.75),
        "CD45RO": (0.0, 0.75),
    },
    "CD8+ T lymphocytes": {"CD3": (0.50, 1), "CD8": (0.50, 1), "CD4": (0.0, 0.75)},
    "CD4+CD45RO+ T cells": {
        "CD3": (0.50, 1),
        "CD8": (0.0, 0.75),
        "CD4": (0.50, 1),
        "CD45RO": (0.50, 1),
    },
    "Tregs": {
        "CD3": (0.50, 1.0),
        "CD25-IL-2Ra": (0.75, 1),
        "FOXP3": (0.75, 1),
        "CD8": (0.0, 0.50),
    },
    "B cells": {"CD20": (0.50, 1), "CD3": (0.0, 0.75)},
    "plasma cells": {"CD38": (0.50, 1), "CD20": (0.50, 1), "CD3": (0.0, 0.75)},
    "granulocytes": {"CD15": (0.50, 1), "CD11b": (0.50, 1), "CD3": (0.0, 0.85)},
    "CD68+ macrophages": {"CD68": (0.95, 1), "CD3": (0.0, 0.50), "CD163": (0.0, 0.95)},
    "CD163+ macrophages": {"CD163": (0.95, 1), "CD3": (0.0, 0.50), "CD68": (0.0, 0.50)},
    "CD68+CD163 macrophages": {
        "CD68": (0.50, 1),
        "CD163": (0.50, 1),
        "CD3": (0.0, 0.95),
    },
    "CD11b+CD68+ macrophages": {
        "CD68": (0.95, 1),
        "CD11b": (0.50, 1),
        "CD3": (0.0, 0.50),
    },
    "NK cells": {"CD56": (0.75, 1), "CD3": (0.0, 0.50), "Cytokeratin": (0.0, 0.50)},
    "vasculature": {"CD34": (0.50, 1), "CD31": (0.50, 1), "Cytokeratin": (0.0, 0.50)},
    "tumor cells": {"Cytokeratin": (0.50, 1), "p53": (0.50, 1), "aSMA": (0.0, 0.75)},
    "immune cells": {
        "CD20": (0.50, 1),
        "CD38": (0.50, 1),
        "CD3": (0.50, 1),
        "GFAP": (0.50, 1),
        "CD15": (0.50, 1),
        "Cytokeratin": (0.0, 0.50),
        "aSMA": (0.0, 0.75),
    },
    "tumor/immune": {
        "CD20": (0.50, 1),
        "CD3": (0.75, 1),
        "CD38": (0.50, 1),
        "GFAP": (0.80, 1),
        "Cytokeratin": (0.85, 1),
        "p53": (0.50, 1),
        "aSMA": (0.0, 0.75),
    },
    "vascular/immune": {
        "CD20": (0.50, 1),
        "CD3": (0.85, 1),
        "CD38": (0.50, 1),
        "GFAP": (0.80, 1),
        "CD34": (0.75, 1),
        "CD31": (0.75, 1),
        "aSMA": (0.0, 0.75),
    },
    "stromal cells": {"Vimentin": (0.50, 1), "Cytokeratin": (0.0, 0.50)},
    "Adipocytes": {
        "p53": (0.75, 1),
        "Vimentin": (0.75, 1),
        "Cytokeratin": (0.0, 0.50),
        "aSMA": (0.0, 0.50),
        "CD44": (0.0, 0.50),
    },
    "smooth muscles": {"aSMA": (0.70, 1), "Vimentin": (0.50, 1), "CD3": (0.0, 0.50)},
    "nerves": {
        "Synaptophysin": (0.85, 1),
        "Vimentin": (0.50, 1),
        "GFAP": (0.85, 1),
        "CD3": (0.0, 0.50),
    },
    "lymphatics": {"Podoplanin": (0.99, 1), "CD3": (0.0, 0.75)},
    "artifact": {
        "CD20": (0.0, 0.50),
        "CD3": (0.0, 0.50),
        "CD38": (0.0, 0.50),
        "GFAP": (0.0, 0.50),
        "Cytokeratin": (0.0, 0.50),
        "p53": (0.0, 0.50),
        "aSMA": (0.0, 0.50),
        "CD15": (0.0, 0.50),
        "CD68": (0.0, 0.50),
        "CD25-IL-2Ra": (0.0, 0.50),
        "CD34": (0.0, 0.50),
        "CD31": (0.0, 0.50),
        "CD56": (0.0, 0.50),
        "Vimentin": (0.0, 0.50),
    },
}
```

[31]:

```
# Annotate the adata
adata_annot = process_adata(adata, annot_dict=annot_dict)
adata_annot.obs.cell_types.value_counts()
```

[31]:

```
unknown                    42834
tumor cells                37035
stromal cells              31646
CD68+CD163 macrophages     28479
granulocytes               23164
vasculature                20016
smooth muscles             19990
B cells                    13433
CD8+ T lymphocytes         10962
plasma cells                8911
CD4+CD45RO+ T cells         6603
artifact                    4650
CD4+ T lymphocytes          4395
vascular/immune             4004
CD3+ T lymphocytes          3975
immune cells                3075
Adipocytes                  2825
tumor/immune                1925
CD68+ macrophages           1431
Tregs                       1067
CD11b+CD68+ macrophages      869
lymphatics                   865
nerves                       366
CD163+ macrophages           220
NK cells                      89
Name: cell_types, dtype: int64
```

[33]:

```
sc.pl.violin(adata_annot, groupby="cell_types", keys=["CD4", "CD8", "CD3"], rotation=90)
```

../_images/examples_link_codex_33_0.png

[35]:

```
sc.pl.violin(
    adata_annot, groupby="cell_types", keys=["CD68", "CD163", "CD11b"], rotation=90
)
```

../_images/examples_link_codex_34_0.png

[36]:

```
sc.pl.spatial(
    adata_annot[adata_annot.obs.Region == "reg020_A"],
    color="cell_types",
    spot_size=25,
    size=1,
)
```

../_images/examples_link_codex_35_0.png

[37]:

```
sc.pl.spatial(
    adata_annot[adata_annot.obs.Region == "reg020_B"],
    color="cell_types",
    spot_size=25,
    size=1,
)
```

../_images/examples_link_codex_36_0.png

[39]:

```
# Put in a dataframe for further analysis
countData = adata_annot.to_df()
obs = adata_annot.obs
data = pd.concat([countData, obs], axis = 1)
```

```
data['CellID'] = Data.index\# Remove the cells with unknown annotation and the artifacts
data = data.loc[~Data['cell_types'].isin(['unknown'])]
data = data.loc[~Data['cell_types'].isin(['artifact'])]
data['cell_types'] = data['cell_types'].cat.remove_unused_categories()
data['cell_types'] = data['cell_types'].astype('str')
```

[44]:

```
data.shape
```

[44]:

```
(225345, 61)
```

[45]:

```
# save
data.to_csv("./data/CRC_pathml.csv")
```

Identification of cellular neighborhoods

After identifying cell types, the next step is to identify cellular neighborhoods using the same approach described in Schürch et. al. and utilizing the code available from: https://github.com/nolanlab/NeighborhoodCoordination

In summary, for each cell, we are going to idenify the 10 nearest spatial neighbors (windows), then we are going to cluster these windows into distinct neighborhoods based on their cell type composition.

[46]:

```
from sklearn.neighbors import NearestNeighbors
import time
import sys
from sklearn.cluster import MiniBatchKMeans
import seaborn as sns
```

[47]:

```
# Function for identifying the windows
def get_windows(job, n_neighbors):
    """
    For each region and each individual cell in dataset, return the indices of the nearest neighbors.

    'job:  meta data containing the start time,index of region, region name, indices of region in original dataframe
    n_neighbors:  the number of neighbors to find for each cell
    """
    start_time, idx, tissue_name, indices = job
    job_start = time.time()

    print("Starting:", str(idx + 1) + "/" + str(len(exps)), ": " + exps[idx])

    # tissue_group: a grouped data frame with X and Y coordinates grouped by unique tissue regions
    tissue = tissue_group.get_group(tissue_name)

    to_fit = tissue.loc[indices][["x", "y"]].values

    # Unsupervised learner for implementing neighbor searches.
    fit = NearestNeighbors(n_neighbors=n_neighbors).fit(tissue[["x", "y"]].values)

    # Find the nearest neighbors

    m = fit.kneighbors(to_fit)

    m = m[0], m[1]

    ## sort_neighbors
    args = m[0].argsort(axis=1)

    add = np.arange(m[1].shape[0]) * m[1].shape[1]

    sorted_indices = m[1].flatten()[args + add[:, None]]

    neighbors = tissue.index.values[sorted_indices]

    end_time = time.time()

    print(
        "Finishing:",
        str(idx + 1) + "/" + str(len(exps)),
        ": " + exps[idx],
        end_time - job_start,
        end_time - start_time,
    )
    return neighbors.astype(np.int32)
```

[48]:

```
data = pd.read_csv("./data/CRC_pathml.csv")
```

[49]:

```
# make dummy variables
data = pd.concat([Data, pd.get_dummies(Data["cell_types"])], axis=1)
# Extract the cell types with dummy variables
sum_cols = data["cell_types"].unique()
values = data[sum_cols].values
```

find windows for each cell in each tissue region

[51]:

```
# Keep the X and Y coordianates + the tissue regions >> then group by tissue regions (140 unique regions)
tissue_group = data[["x", "y", "Region"]].groupby("Region")

# Create a list of unique tissue regions
exps = list(data["Region"].unique())

# time.time(): current time is seconds
# indices: a list of indices (rownames) of each dataframe in tissue_group
# exps.index(t) : t represents the index of each one of the indices eg, exps.index("reg001_A") is 0 and exps.index("reg001_B") is 1 and so on
# t is the name of tissue regions eg, reg001_A
tissue_chunks = [
    (time.time(), exps.index(t), t, a)
    for t, indices in tissue_group.groups.items()
    for a in np.array_split(indices, 1)
]

# Get the window (the 10 closest cells to each cell in each tissue region)
tissues = [get_windows(job, 10) for job in tissue_chunks]
```

```
Starting: 61/140 : reg001_A
Finishing: 61/140 : reg001_A 0.009814977645874023 0.022227048873901367
Starting: 77/140 : reg001_B
Finishing: 77/140 : reg001_B 0.004416942596435547 0.026772260665893555
Starting: 19/140 : reg002_A
Finishing: 19/140 : reg002_A 0.006669044494628906 0.0336461067199707
Starting: 101/140 : reg002_B
Finishing: 101/140 : reg002_B 0.0061969757080078125 0.04012012481689453
Starting: 58/140 : reg003_A
Finishing: 58/140 : reg003_A 0.005404233932495117 0.045838117599487305
Starting: 85/140 : reg003_B
Finishing: 85/140 : reg003_B 0.002471923828125 0.04852771759033203
Starting: 65/140 : reg004_A
Finishing: 65/140 : reg004_A 0.005899906158447266 0.054521799087524414
Starting: 71/140 : reg004_B
Finishing: 71/140 : reg004_B 0.0063402652740478516 0.06109809875488281
Starting: 40/140 : reg005_A
Finishing: 40/140 : reg005_A 0.0077631473541259766 0.0690910816192627
Starting: 105/140 : reg005_B
Finishing: 105/140 : reg005_B 0.005625247955322266 0.07496023178100586
Starting: 8/140 : reg006_A
Finishing: 8/140 : reg006_A 0.004698991775512695 0.0797572135925293
Starting: 127/140 : reg006_B
Finishing: 127/140 : reg006_B 0.004808902740478516 0.08475184440612793
Starting: 17/140 : reg007_A
Finishing: 17/140 : reg007_A 0.007723093032836914 0.09262681007385254
Starting: 92/140 : reg007_B
Finishing: 92/140 : reg007_B 0.0049021244049072266 0.09776902198791504
Starting: 24/140 : reg008_A
Finishing: 24/140 : reg008_A 0.005953073501586914 0.1038961410522461
Starting: 123/140 : reg008_B
Finishing: 123/140 : reg008_B 0.008786916732788086 0.11287188529968262
Starting: 52/140 : reg009_A
Finishing: 52/140 : reg009_A 0.0069849491119384766 0.12202978134155273
Starting: 130/140 : reg009_B
Finishing: 130/140 : reg009_B 0.0077838897705078125 0.12995290756225586
Starting: 12/140 : reg010_A
Finishing: 12/140 : reg010_A 0.006242036819458008 0.13639497756958008
Starting: 133/140 : reg010_B
Finishing: 133/140 : reg010_B 0.003675222396850586 0.14031219482421875
Starting: 23/140 : reg011_A
Finishing: 23/140 : reg011_A 0.004063129425048828 0.1445319652557373
Starting: 124/140 : reg011_B
Finishing: 124/140 : reg011_B 0.00393986701965332 0.14860916137695312
Starting: 69/140 : reg012_A
Finishing: 69/140 : reg012_A 0.005756855010986328 0.15452790260314941
Starting: 75/140 : reg012_B
Finishing: 75/140 : reg012_B 0.0026569366455078125 0.15730595588684082
Starting: 41/140 : reg013_A
Finishing: 41/140 : reg013_A 0.0051670074462890625 0.16249799728393555
Starting: 108/140 : reg013_B
Finishing: 108/140 : reg013_B 0.00403904914855957 0.1666700839996338
Starting: 44/140 : reg014_A
Finishing: 44/140 : reg014_A 0.003408193588256836 0.17021417617797852
Starting: 98/140 : reg014_B
Finishing: 98/140 : reg014_B 0.005333900451660156 0.1755669116973877
Starting: 54/140 : reg015_A
Finishing: 54/140 : reg015_A 0.006878852844238281 0.18254685401916504
Starting: 88/140 : reg015_B
Finishing: 88/140 : reg015_B 0.004870891571044922 0.1875441074371338
Starting: 29/140 : reg016_A
Finishing: 29/140 : reg016_A 0.006279945373535156 0.19389891624450684
```

```
Starting: 107/140 : reg016_B
Finishing: 107/140 : reg016_B 0.00708317756652832 0.20116114616394043
Starting: 67/140 : reg017_A
Finishing: 67/140 : reg017_A 0.006649971008300781 0.20815491676330566
Starting: 72/140 : reg017_B
Finishing: 72/140 : reg017_B 0.005545854568481445 0.21404600143432617
Starting: 4/140 : reg018_A
Finishing: 4/140 : reg018_A 0.008655071258544922 0.2230057716369629
Starting: 78/140 : reg018_B
Finishing: 78/140 : reg018_B 0.006451129913330078 0.2296450138092041
Starting: 32/140 : reg019_A
Finishing: 32/140 : reg019_A 0.006360054016113281 0.23619771003723145
Starting: 113/140 : reg019_B
Finishing: 113/140 : reg019_B 0.003787994384765625 0.24015522003173828
Starting: 60/140 : reg020_A
Finishing: 60/140 : reg020_A 0.010369062423706055 0.2506411075592041
Starting: 138/140 : reg020_B
Finishing: 138/140 : reg020_B 0.008259773254394531 0.25922179222106934
Starting: 39/140 : reg021_A
Finishing: 39/140 : reg021_A 0.004830837249755859 0.2642190456390381
Starting: 116/140 : reg021_B
Finishing: 116/140 : reg021_B 0.003367900848388672 0.26804184913635254
Starting: 9/140 : reg022_A
Finishing: 9/140 : reg022_A 0.007760047912597656 0.27587199211120605
Starting: 126/140 : reg022_B
Finishing: 126/140 : reg022_B 0.0032799243927001953 0.27936792373657227
Starting: 20/140 : reg023_A
Finishing: 20/140 : reg023_A 0.004238128662109375 0.283825159072876
Starting: 93/140 : reg023_B
Finishing: 93/140 : reg023_B 0.005738973617553711 0.2899298667907715
Starting: 33/140 : reg024_A
Finishing: 33/140 : reg024_A 0.002911806106567383 0.2929408550262451
Starting: 112/140 : reg024_B
Finishing: 112/140 : reg024_B 0.013455867767333984 0.30647897720336914
Starting: 62/140 : reg025_A
Finishing: 62/140 : reg025_A 0.006231069564819336 0.313082218170166
Starting: 76/140 : reg025_B
Finishing: 76/140 : reg025_B 0.009344816207885742 0.3225746154785156
Starting: 50/140 : reg026_A
Finishing: 50/140 : reg026_A 0.0073397159576416016 0.3299558162689209
Starting: 95/140 : reg026_B
Finishing: 95/140 : reg026_B 0.010460853576660156 0.34046483039855957
Starting: 57/140 : reg027_A
Finishing: 57/140 : reg027_A 0.008396148681640625 0.3488960266113281
Starting: 81/140 : reg027_B
Finishing: 81/140 : reg027_B 0.008210897445678711 0.3574059009552002
Starting: 14/140 : reg028_A
Finishing: 14/140 : reg028_A 0.007547855377197266 0.3653140068054199
Starting: 87/140 : reg028_B
Finishing: 87/140 : reg028_B 0.00654292106628418 0.3718750476837158
Starting: 46/140 : reg029_A
Finishing: 46/140 : reg029_A 0.009541988372802734 0.38150620460510254
Starting: 104/140 : reg029_B
Finishing: 104/140 : reg029_B 0.006811857223510742 0.38849782943725586
Starting: 47/140 : reg030_A
Finishing: 47/140 : reg030_A 0.003790140151977539 0.39242005348205566
Starting: 99/140 : reg030_B
Finishing: 99/140 : reg030_B 0.006364107131958008 0.39893221855163574
Starting: 53/140 : reg031_A
Finishing: 53/140 : reg031_A 0.004971981048583984 0.4040391445159912
Starting: 128/140 : reg031_B
Finishing: 128/140 : reg031_B 0.006245851516723633 0.41039395332336426
Starting: 28/140 : reg032_A
Finishing: 28/140 : reg032_A 0.006634950637817383 0.4171791076660156
Starting: 106/140 : reg032_B
Finishing: 106/140 : reg032_B 0.004702329635620117 0.4219231605529785
Starting: 1/140 : reg033_A
Finishing: 1/140 : reg033_A 0.008253097534179688 0.4303441047668457
Starting: 73/140 : reg033_B
Finishing: 73/140 : reg033_B 0.006574869155883789 0.43709278106689453
Starting: 15/140 : reg034_A
Finishing: 15/140 : reg034_A 0.006840944290161133 0.4440948963165283
Starting: 89/140 : reg034_B
Finishing: 89/140 : reg034_B 0.009439229965209961 0.45371532440185547
Starting: 25/140 : reg035_A
Finishing: 25/140 : reg035_A 0.008854866027832031 0.4627108573913574
Starting: 120/140 : reg035_B
Finishing: 120/140 : reg035_B 0.009293794631958008 0.47219085693359375
Starting: 68/140 : reg036_A
Finishing: 68/140 : reg036_A 0.009788990020751953 0.48207616806030273
Starting: 139/140 : reg036_B
Finishing: 139/140 : reg036_B 0.01093912124633789 0.49315786361694336
Starting: 43/140 : reg037_A
Finishing: 43/140 : reg037_A 0.005067110061645508 0.4984712600708008
Starting: 117/140 : reg037_B
Finishing: 117/140 : reg037_B 0.006145000457763672 0.5049071311950684
Starting: 38/140 : reg038_A
Finishing: 38/140 : reg038_A 0.006515026092529297 0.5115442276000977
Starting: 110/140 : reg038_B
Finishing: 110/140 : reg038_B 0.006800174713134766 0.5184721946716309
Starting: 59/140 : reg039_A
```

```
Finishing: 59/140 : reg039_A 0.003709077835083008 0.5222232341766357
Starting: 74/140 : reg039_B
Finishing: 74/140 : reg039_B 0.0035741329193115234 0.5277011394500732
Starting: 18/140 : reg040_A
Finishing: 18/140 : reg040_A 0.004296064376831055 0.5321159362792969
Starting: 96/140 : reg040_B
Finishing: 96/140 : reg040_B 0.004117727279663086 0.536341667175293
Starting: 7/140 : reg041_A
Finishing: 7/140 : reg041_A 0.003995180130004883 0.540438175201416
Starting: 82/140 : reg041_B
Finishing: 82/140 : reg041_B 0.003885030746459961 0.5444629192352295
Starting: 37/140 : reg042_A
Finishing: 37/140 : reg042_A 0.003648042678833008 0.548213005065918
Starting: 109/140 : reg042_B
Finishing: 109/140 : reg042_B 0.003875255584716797 0.552192211151123
Starting: 63/140 : reg043_A
Finishing: 63/140 : reg043_A 0.0053560733795166016 0.5576450824737549
Starting: 79/140 : reg043_B
Finishing: 79/140 : reg043_B 0.0034198760986328125 0.5609321594238281
Starting: 10/140 : reg044_A
Finishing: 10/140 : reg044_A 0.003528118133544922 0.5645391941070557
Starting: 131/140 : reg044_B
Finishing: 131/140 : reg044_B 0.002878904342651367 0.5675170421600342
Starting: 21/140 : reg045_A
Finishing: 21/140 : reg045_A 0.01139378547668457 0.5790097713470459
Starting: 97/140 : reg045_B
Finishing: 97/140 : reg045_B 0.004516124725341797 0.58365797996521
Starting: 6/140 : reg046_A
Finishing: 6/140 : reg046_A 0.003998756408691406 0.5878088474273682
Starting: 134/140 : reg046_B
Finishing: 134/140 : reg046_B 0.01061391830444336 0.5984470844268799
Starting: 34/140 : reg047_A
Finishing: 34/140 : reg047_A 0.005155086517333984 0.6037991046905518
Starting: 118/140 : reg047_B
Finishing: 118/140 : reg047_B 0.007688999176025391 0.6116068363189697
Starting: 30/140 : reg048_A
Finishing: 30/140 : reg048_A 0.007600307464599609 0.6195840835571289
Starting: 115/140 : reg048_B
Finishing: 115/140 : reg048_B 0.011443138122558594 0.6316161155700684
Starting: 2/140 : reg049_A
Finishing: 2/140 : reg049_A 0.00881505012512207 0.6405959129333496
Starting: 135/140 : reg049_B
Finishing: 135/140 : reg049_B 0.011723041534423828 0.6528370380401611
Starting: 3/140 : reg050_A
Finishing: 3/140 : reg050_A 0.0052111148834228516 0.6582260131835938
Starting: 136/140 : reg050_B
Finishing: 136/140 : reg050_B 0.009176015853881836 0.6675820350646973
Starting: 31/140 : reg051_A
Finishing: 31/140 : reg051_A 0.008751153945922852 0.6764531135559082
Starting: 114/140 : reg051_B
Finishing: 114/140 : reg051_B 0.008320093154907227 0.68532395362854
Starting: 51/140 : reg052_A
Finishing: 51/140 : reg052_A 0.0061681270599365234 0.6917729377746582
Starting: 84/140 : reg052_B
Finishing: 84/140 : reg052_B 0.009895801544189453 0.7022178173065186
Starting: 45/140 : reg053_A
Finishing: 45/140 : reg053_A 0.004172086715698242 0.7066078186035156
Starting: 94/140 : reg053_B
Finishing: 94/140 : reg053_B 0.004163980484008789 0.7108440399169922
Starting: 27/140 : reg054_A
Finishing: 27/140 : reg054_A 0.005264997482299805 0.7162370681762695
Starting: 111/140 : reg054_B
Finishing: 111/140 : reg054_B 0.003180980682373047 0.7195639610290527
Starting: 66/140 : reg055_A
Finishing: 66/140 : reg055_A 0.00409388542175293 0.7237598896026611
Starting: 80/140 : reg055_B
Finishing: 80/140 : reg055_B 0.0030677318572998047 0.7269597053527832
Starting: 22/140 : reg056_A
Finishing: 22/140 : reg056_A 0.005424976348876953 0.7325129508972168
Starting: 125/140 : reg056_B
Finishing: 125/140 : reg056_B 0.011126041412353516 0.7440230846405029
Starting: 13/140 : reg057_A
Finishing: 13/140 : reg057_A 0.00240325927734375 0.7465171813964844
Starting: 132/140 : reg057_B
Finishing: 132/140 : reg057_B 0.004606008529663086 0.7512319087982178
Starting: 56/140 : reg058_A
Finishing: 56/140 : reg058_A 0.00801992416381836 0.7594480514526367
Starting: 129/140 : reg058_B
Finishing: 129/140 : reg058_B 0.007853984832763672 0.7673490047454834
Starting: 49/140 : reg059_A
Finishing: 49/140 : reg059_A 0.007686138153076172 0.775170087814331
Starting: 119/140 : reg059_B
Finishing: 119/140 : reg059_B 0.008450984954833984 0.7838461399078369
Starting: 11/140 : reg060_A
Finishing: 11/140 : reg060_A 0.007905006408691406 0.7919058799743652
Starting: 86/140 : reg060_B
Finishing: 86/140 : reg060_B 0.00969386100769043 0.8019630908966064
Starting: 16/140 : reg061_A
Finishing: 16/140 : reg061_A 0.007141828536987305 0.8094010353088379
Starting: 100/140 : reg061_B
Finishing: 100/140 : reg061_B 0.008517742156982422 0.8181126117706299
```

```
Starting: 64/140 : reg062_A
Finishing: 64/140 : reg062_A 0.008826017379760742 0.8272860050201416
Starting: 140/140 : reg062_B
Finishing: 140/140 : reg062_B 0.007968902587890625 0.83530592918396
Starting: 35/140 : reg063_A
Finishing: 35/140 : reg063_A 0.003358125686645508 0.8388819694519043
Starting: 122/140 : reg063_B
Finishing: 122/140 : reg063_B 0.005372285842895508 0.8444433212280273
Starting: 48/140 : reg064_A
Finishing: 48/140 : reg064_A 0.0074350833892822266 0.8521482944488525
Starting: 121/140 : reg064_B
Finishing: 121/140 : reg064_B 0.008898019790649414 0.8611080646514893
Starting: 55/140 : reg065_A
Finishing: 55/140 : reg065_A 0.00514674186706543 0.8664467334747314
Starting: 137/140 : reg065_B
Finishing: 137/140 : reg065_B 0.006044864654541016 0.8725459575653076
Starting: 26/140 : reg066_A
Finishing: 26/140 : reg066_A 0.003931999206542969 0.8765108585357666
Starting: 91/140 : reg066_B
Finishing: 91/140 : reg066_B 0.012857198715209961 0.889430046081543
Starting: 5/140 : reg067_A
Finishing: 5/140 : reg067_A 0.004929780960083008 0.8947200775146484
Starting: 83/140 : reg067_B
Finishing: 83/140 : reg067_B 0.004603862762451172 0.899526834487915
Starting: 70/140 : reg068_A
Finishing: 70/140 : reg068_A 0.003593921661376953 0.9033100605010986
Starting: 90/140 : reg068_B
Finishing: 90/140 : reg068_B 0.008028984069824219 0.9117860794067383
Starting: 42/140 : reg069_A
Finishing: 42/140 : reg069_A 0.004137992858886719 0.9161639213562012
Starting: 102/140 : reg069_B
Finishing: 102/140 : reg069_B 0.005839109420776367 0.9220380783081055
Starting: 36/140 : reg070_A
Finishing: 36/140 : reg070_A 0.0063631534576416016 0.92854905128479
Starting: 103/140 : reg070_B
Finishing: 103/140 : reg070_B 0.004929065704345703 0.9335160255432129
```

for each cell and its nearest neighbors, reshape and count the number of each cell type in those neighbors.

[52]:

```
ks = [10]
out_dict = {}
for k in ks:
    for neighbors, job in zip(tissues, tissue_chunks):
        chunk = np.arange(len(neighbors))  # indices
        tissue_name = job[2]
        indices = job[3]
        window = (
            values[neighbors[chunk, :k].flatten()]
            .reshape(len(chunk), k, len(sum_cols))
            .sum(axis=1)
        )
        out_dict[(tissue_name, k)] = (window.astype(np.float16), indices)
```

concatenate the summed windows and combine into one dataframe for each window size tested.

[53]:

```
keep_cols = ["x", "y", "Region", "cell_types"]
windows = {}
for k in ks:
    window = pd.concat(
        [
            pd.DataFrame(
                out_dict[(exp, k)][0],
                index=out_dict[(exp, k)][1].astype(int),
                columns=sum_cols,
            )
            for exp in exps
        ],
        0,
    )
    window = window.loc[Data.index.values]
    window = pd.concat([Data[keep_cols], window], 1)
    windows[k] = window
```

```
<ipython-input-53-dbc25edfe709>:4: FutureWarning: In a future version of pandas all arguments of concat except for the argument 'objs' will be key
  window = pd.concat(
<ipython-input-53-dbc25edfe709>:8: FutureWarning: In a future version of pandas all arguments of concat except for the argument 'objs' will be key
  window = pd.concat([Data[keep_cols], window], 1)
```

[54]:

```
neighborhood_name = "neighborhood" + str(k)
k_centroids = {}

windows2 = windows[10]
```

Clustering the windows

[64]:

```
km = MiniBatchKMeans(n_clusters=10, random_state=0)

labelskm = km.fit_predict(windows2[sum_cols].values)
k_centroids[10] = km.cluster_centers_
data["neighborhood10"] = labelskm
data[neighborhood_name] = data[neighborhood_name].astype("category")
```

[65]:

```
cell_order = [
    "tumor cells",
    "CD68+CD163 macrophages",
    "CD11b+CD68+ macrophages",
    "CD68+ macrophages",
    "CD163+ macrophages",
    "granulocytes",
    "NK cells",
    "CD3+ T lymphocytes",
    "CD4+ T lymphocytes",
    "CD4+CD45RO+ T cells",
    "CD8+ T lymphocytes",
    "Tregs",
    "B cells",
    "plasma cells",
    "tumor/immune",
    "vascular/immune",
    "immune cells",
    "smooth muscles",
    "stromal cells",
    "vasculature",
    "lymphatics",
    "nerves",
]
```

This plot shows the cell types abundance in the different niches

[66]:

```
niche_clusters = k_centroids[10]
tissue_avgs = values.mean(axis=0)
fc = np.log2(
    (
        (niche_clusters + tissue_avgs)
        / (niche_clusters + tissue_avgs).sum(axis=1, keepdims=True)
    )
    / tissue_avgs
)
fc = pd.DataFrame(fc, columns=sum_cols)
s = sns.clustermap(
    fc.loc[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], cell_order],
    vmin=-3,
    vmax=3,
    cmap="bwr",
    row_cluster=False,
)
```

../_images/examples_link_codex_59_0.png

Visualize the identified neighborhoods on the slides in each clinical group

[67]:

```
# CLR
Data["neighborhood10"] = Data["neighborhood10"].astype("category")
sns.lmplot(
    data=Data[Data["groups"] == "CLR"],
    x="x",
    y="y",
    hue="neighborhood10",
    palette="bright",
    height=8,
    col="Region",
    col_wrap=10,
    fit_reg=False,
)
```

[67]:

```
<seaborn.axisgrid.FacetGrid at 0x1a79e42e0>
```

../_images/examples_link_codex_61_1.png

[68]:

```
# DII
Data["neighborhood10"] = Data["neighborhood10"].astype("category")
sns.lmplot(
    data=Data[Data["groups"] == "DII"],
    x="x",
    y="y",
    hue="neighborhood10",
    palette="bright",
    height=8,
    col="Region",
    col_wrap=10,
    fit_reg=False,
)
```

[68]:

```
<seaborn.axisgrid.FacetGrid at 0x1a78afc40>
```

../_images/examples_link_codex_62_1.png

Plot for each group and each patient the percent of total cells allocated to each neighborhood

[70]:

```
fc = Data.groupby(["patients", "groups"]).apply(
    lambda x: x["neighborhood10"].value_counts(sort=False, normalize=True)
)

fc.columns = range(10)
melt = pd.melt(fc.reset_index(), id_vars=["patients", "groups"])
melt = melt.rename(
    columns={"variable": "neighborhood", "value": "frequency of neighborhood"}
)
melt["neighborhood"] = melt["neighborhood"].map(
    {
        0: "smooth muscles",
        1: "plasma cells-enriched",
        2: "tumor",
        3: "B cells-enriched",
        4: "vasculature",
        5: "stroma",
        6: "TAMs-enriched",
        7: "TILs-enriched",
        8: "granulocytes-enriched",
        9: "vasculature/immune",
    }
)

f, ax = plt.subplots(figsize=(10, 7))
sns.stripplot(
    data=melt,
    hue="groups",
    dodge=True,
    alpha=0.2,
    x="neighborhood",
    y="frequency of neighborhood",
)
sns.pointplot(
    data=melt,
    scatter_kws={"marker": "d"},
    hue="groups",
    dodge=0.5,
    join=False,
    x="neighborhood",
    y="frequency of neighborhood",
)
handles, labels = ax.get_legend_handles_labels()
plt.xticks(rotation=90, fontsize="10", ha="center")
ax.legend(
    handles[:2],
    labels[:2],
    title="Groups",
    handletextpad=0,
    columnspacing=1,
    loc="upper left",
    ncol=3,
    frameon=True,
)
plt.tight_layout()
```

../_images/examples_link_codex_64_0.png

Previous Next

Contributing

`PathML` is an open source project. Consider contributing to benefit the entire community!

There are many ways to contribute to PathML, including:

- Submitting bug reports
- Submitting feature requests
- Writing documentation
- Fixing bugs
- Writing code for new features
- Sharing trained model parameters [coming soon]
- Sharing `PathML` with colleagues, students, etc.

Submitting a bug report

Report bugs or errors by filing an issue on GitHub. Make sure to include the following information:

- Short description of the bug
- Minimum working example to reproduce the bug
- Expected result vs. actual result

- Any other useful information

If a bug cannot be reproduced by someone else on a different machine, it will usually be hard to identify what is causing it.

Requesting a new feature

Request a new feature by filing an issue on GitHub. Make sure to include the following information:

- Description of the feature
- Pseudocode of how the feature might work (if applicable)
- Any other useful information

For developers

Coordinate system conventions

With multiple tools for interacting with matrices/images, conflicting coordinate systems has been a common source of bugs. This is typically caused when mixing up (X, Y) coordinate systems and (i, j) coordinate systems. **To avoid these issues, we have adopted the (i, j) coordinate convention throughout PathML.** This follows the convention used by NumPy and many others, where `A[i, j]` refers to the element of matrix A in the ith row, jth column. Developers should be careful about coordinate systems and make the necessary adjustments when using third-party tools so that users of PathML can rely on a consistent coordinate system when using our tools.

Setting up a local development environment

1. Create a new fork of the `PathML` repository
2. Clone your fork to your local machine
3. Set up the PathML environment: `conda env create -f environment.yml; conda activate pathml`
4. Install PathML: `pip install -e .`
5. Install pre-commit hooks: `pre-commit install`

Running tests

To run the full testing suite (not recommended):

```
python -m pytest
```

Some tests are known to be very slow. Tests for the tile stitching functionality must be ran separately. To skip them, run:

```
python -m pytest -m "not slow and not exclude"
```

Then, run the tilestitching test:

```
python -m pytest tests/preprocessing_tests/test_tilestitcher.py
```

Building documentation locally

```
cd docs                                     # enter docs directory
pip install -r readthedocs-requirements.txt     # install packages to build docs
make html                                   # build docs in html format
```

Then use your favorite web browser to open `pathml/docs/build/html/index.html`

Checking code coverage

```
conda install coverage  # install coverage package for code coverage
COVERAGE_FILE=.coverage_others coverage run -m pytest -m "not slow and not exclude" # run coverage for all files except tile stitching
COVERAGE_FILE=.coverage_tilestitcher coverage run -m pytest tests/preprocessing_tests/test_tilestitcher.py # run coverage for tile stitching
coverage combine .coverage_tilestitcher .coverage_others # combine coverage results
coverage report         # view coverage report
coverage html           # optionally generate HTML coverage report
```

How to contribute code, documentation, etc.

1. Create a new GitHub issue for what you will be working on, if one does not already exist
2. Create a local development environment (see above)
3. Create a new branch from the dev branch and implement your changes

4. Write new tests as needed to maintain code coverage
5. Ensure that all tests pass
6. Push your changes and open a pull request on GitHub referencing the corresponding issue
7. Respond to discussion/feedback about the pull request, make changes as necessary

Versioning and Distributing

We use semantic versioning. The version is tracked in `pathml/_version.py` and should be updated there as required. When new code is merged to the master branch on GitHub, the version should be incremented and a new release should be pushed. Releases can be created using the GitHub website interface, and should be tagged in version format (e.g., “v1.0.0” for version 1.0.0) and include release notes indicating what has changed. Once a new release is created, GitHub Actions workflows will automatically build and publish the updated package on PyPI and TestPyPI, as well as build and publish the Docker image to Docker Hub.

Code Quality

We want PathML to be built on high-quality code. However, the idea of “code quality” is somewhat subjective. If the code works perfectly but cannot be read and understood by someone else, then it can’t be maintained, and this accumulated tech debt is something we want to avoid. Writing code that “works”, i.e. does what you want it to do, is therefore necessary but not sufficient. Good code also demands efficiency, consistency, good design, clarity, and many other factors.

Here are some general tips and ideas:

- Strive to make code concise, but not at the expense of clarity.
- Seek efficient and general designs, but avoid premature optimization.
- Prefer informative variable names.
- Encapsulate code in functions or objects.
- Comment, comment, comment your code.

All code should be reviewed by someone else before merging.

We use Black to enforce consistency of code style.

Documentation Standards

All code should be documented, including docstrings for users AND inline comments for other developers whenever possible! Both are crucial for ensuring long-term usability and maintainability. Documentation is automatically generated using the Sphinx autodoc and napoleon extensions from properly formatted Google-style docstrings. All documentation (including docstrings) is written in reStructuredText format. See this docstring example to get started.

Testing Standards

All code should be accompanied by tests, whenever possible, to ensure that everything is working as intended.

The type of testing required may vary depending on the type of contribution:

- New features should use tests to ensure that the code is working as intended, e.g. comparing output of a function with the expected output.
- Bug fixes should first add a failing test, then make it pass by fixing the bug

No pull request can be merged unless all tests pass. We aim to maintain good code coverage for the testing suite (target >90%). We use the pytest testing framework. To run the test suite and check code coverage:

```
conda install coverage  # install coverage package for code coverage
COVERAGE_FILE=.coverage_others coverage run -m pytest -m "not slow and not exclude" # run coverage for all files except tile stitching
COVERAGE_FILE=.coverage_tilestitcher coverage run -m pytest tests/preprocessing_tests/test_tilestitcher.py # run coverage for tile stitching
coverage combine .coverage_tilestitcher .coverage_others # combine coverage results
coverage report         # view coverage report
coverage html           # optionally generate HTML coverage report
```

We suggest using test-driven development when applicable. I.e., if you’re fixing a bug or adding new features, write the tests first! (they should all fail). Then, write the actual code. When all tests pass, you know that your implementation is working. This helps ensure that all code is tested and that the tests are testing what we want them to.

Thank You!

Thank you for helping make `PathML` better!

Previous

Preprocessing: Transforms Gallery

View on GitHub

In PathML, preprocessing pipelines are created by composing modular `Transforms`.

The following tutorial contains an overview of the PathML pre-processing Transforms, with examples.

We will divide Transforms into three primary categories, depending on their function:

1. Transforms that modify an image
   - Gaussian Blur
   - Median Blur
   - Box Blur
   - Stain Normalization
   - Superpixel Interpolation
2. Transforms that create a mask
   - Nucleus Detection
   - Binary Threshold
3. Transforms that modify a mask
   - Morphological Closing
   - Morphological Opening
   - Foreground Detection

- Tissue Detection

[1]:

```
import matplotlib.pyplot as plt
import copy
import numpy as np

import os

os.environ["JAVA_HOME"] = "/opt/conda/envs/pathml/"

from pathml.core import HESlide, Tile, types
from pathml.utils import plot_mask, RGB_to_GREY
from pathml.preprocessing import (
    BoxBlur,
    GaussianBlur,
    MedianBlur,
    NucleusDetectionHE,
    StainNormalizationHE,
    SuperpixelInterpolation,
    ForegroundDetection,
    TissueDetectionHE,
    BinaryThreshold,
    MorphClose,
    MorphOpen,
)

fontsize = 14
```

Note that a `Transform` operates on `Tile` objects. We must first load a whole-slide image, extract a smaller region, and create a `Tile`:

[2]:

```
wsi = HESlide("./../data/CMU-1-Small-Region.svs.tiff")
region = wsi.slide.extract_region(location=(900, 800), size=(500, 500))


def smalltile():
    # convenience function to create a new tile
    tile = Tile(region, coords=(0, 0), name="testregion", slide_type=types.HE)
    tile.image = np.squeeze(tile.image)
    return tile
```

Transforms that modify an image

Blurring Transforms

We’ll start with the 3 blurring transforms: `GaussianBlur`, `MedianBlur`, and `BoxBlur`

Blurriness can be control with the `kernel_size` parameter. A larger kernel width yields a more blurred result for all blurring transforms:

[3]:

```
blurs = ["Original Image", GaussianBlur, MedianBlur, BoxBlur]
blur_name = ["Original Image", "GaussianBlur", "MedianBlur", "BoxBlur"]
k_size = [5, 11, 21]
fig, axarr = plt.subplots(nrows=4, ncols=3, figsize=(7.5, 10))
for i, blur in enumerate(blurs):
    for j, kernel_size in enumerate(k_size):
        tile = smalltile()
        if blur != "Original Image":
            b = blur(kernel_size=kernel_size)
            b.apply(tile)
        ax = axarr[i, j]
        ax.imshow(tile.image)
        if i == 0:
            ax.set_title(f"Kernel_size = {kernel_size}", fontsize=fontsize)
        if j == 0:
            ax.set_ylabel(blur_name[i], fontsize=fontsize)
for a in axarr.ravel():
    a.set_xticks([])
    a.set_yticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_5_0.png

Superpixel Interpolation

Superpixel interpolation is a method for grouping together nearby similar pixels to form larger "superpixels." The `SuperpixelInterpolation` Transform divides the input image into superpixels using SLIC algorithm, then interpolates each superpixel with average color. The `region_size` parameter controls how big the superpixels are:

[4]:

```
region_sizes = ["original", 10, 20, 30]
fig, axarr = plt.subplots(nrows=1, ncols=4, figsize=(10, 10))
for i, region_size in enumerate(region_sizes):
    tile = smalltile()
    if region_size == "original":
        axarr[i].set_title("Original Image", fontsize=fontsize)
    else:
        t = SuperpixelInterpolation(region_size=region_size)
        t.apply(tile)
        axarr[i].set_title(f"Region Size = {region_size}", fontsize=fontsize)
    axarr[i].imshow(tile.image)
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_7_0.png

Stain Normalization

H&E images are a combination of two stains: hematoxylin and eosin. Stain deconvolution methods attempt to estimate the relative contribution of each stain for each pixel. Each stain can then be pulled out into a separate image, and the deconvolved images can then be recombined to normalize the appearance of the image.

The `StainNormalizationHE` Transform implements two algorithms for stain deconvolution.

[5]:

```
fig, axarr = plt.subplots(nrows=2, ncols=3, figsize=(10, 7.5))
fontsize = 18
for i, method in enumerate(["macenko", "vahadane"]):
    for j, target in enumerate(["normalize", "hematoxylin", "eosin"]):
        tile = smalltile()
        normalizer = StainNormalizationHE(target=target, stain_estimation_method=method)
        normalizer.apply(tile)
        ax = axarr[i, j]
        ax.imshow(tile.image)
        if j == 0:
            ax.set_ylabel(f"{method} method", fontsize=fontsize)
        if i == 0:
            ax.set_title(target, fontsize=fontsize)
for a in axarr.ravel():
    a.set_xticks([])
    a.set_yticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_9_0.png

Transforms that create a mask

Binary Threshold

The `BinaryThreshold` transform creates a mask by classifying whether each pixel is above or below the given threshold. Note that you can supply a `threshold` parameter, or use Otsu's method to automatically determine a threshold:

[6]:

```
thresholds = ["original", 50, 180, "otsu"]
fig, axarr = plt.subplots(nrows=1, ncols=len(thresholds), figsize=(12, 6))
for i, thresh in enumerate(thresholds):
    tile = smalltile()
    if thresh == "original":
        axarr[i].set_title("Original Image", fontsize=fontsize)
        axarr[i].imshow(tile.image)
    elif thresh == "otsu":
        t = BinaryThreshold(mask_name="binary_threshold", inverse=True, use_otsu=True)
        t.apply(tile)
        axarr[i].set_title(f"Otsu Threshold", fontsize=fontsize)
        axarr[i].imshow(tile.masks["binary_threshold"])
    else:
        t = BinaryThreshold(
            mask_name="binary_threshold", threshold=thresh, inverse=True, use_otsu=False
        )
        t.apply(tile)
        axarr[i].set_title(f"Threshold = {thresh}", fontsize=fontsize)
        axarr[i].imshow(tile.masks["binary_threshold"])
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_11_0.png

Nucleus Detection

The `NucleusDetectionHE` transform employs a simple nucleus detection algorithm for H&E stained images. It works by first separating hematoxylin channel, then doing interpolation using superpixels, and finally using Otsu's method for binary thresholding. This is an example of a compound Transform created by combining several other Transforms:

[7]:

```
tile = smalltile()
nucleus_detection = NucleusDetectionHE(mask_name="detect_nuclei")
nucleus_detection.apply(tile)

fig, axarr = plt.subplots(nrows=1, ncols=2, figsize=(8, 8))
axarr[0].imshow(tile.image)
axarr[0].set_title("Original Image", fontsize=fontsize)
axarr[1].imshow(tile.masks["detect_nuclei"])
axarr[1].set_title("Nucleus Detection", fontsize=fontsize)
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_13_0.png

We can also overlay the results on the original image to see which regions were identified as being nuclei:

[8]:

```
fig, ax = plt.subplots(figsize=(7, 7))
plot_mask(im=tile.image, mask_in=tile.masks["detect_nuclei"], ax=ax)
plt.title("Overlay", fontsize=fontsize)
plt.axis("off")
plt.show()
```

../_images/examples_link_gallery_15_0.png

Transforms that modify a mask

For the following transforms, we'll use a Tile containing a larger region extracted from the slide.

[11]:

```
bigregion = wsi.slide.extract_region(location=(800, 800), size=(1000, 1000))
bigregion = np.squeeze(bigregion)


def bigtile():
    # convenience function to create a new tile with a binary mask
    bigtile = Tile(bigregion, coords=(0, 0), name="testregion", slide_type=types.HE)
    BinaryThreshold(
        mask_name="binary_threshold", inverse=True, threshold=100, use_otsu=False
    ).apply(bigtile)
    return bigtile


plt.imshow(bigregion)
plt.axis("off")
plt.show()
```

../_images/examples_link_gallery_17_0.png

Morphological Opening

Morphological opening reduces noise in a binary mask by first applying binary erosion *n* times, and then applying binary dilation *n* times. The effect is to remove small objects from the background. The strength of the effect can be controlled by setting *n*

[12]:

```
ns = ["Original Mask", 1, 3, 5]
fig, axarr = plt.subplots(nrows=1, ncols=4, figsize=(10, 10))
for i, n in enumerate(ns):
    tile = bigtile()
    if n == "Original Mask":
        axarr[i].set_title("Original Mask", fontsize=fontsize)
    else:
        t = MorphOpen(mask_name="binary_threshold", n_iterations=n)
        t.apply(tile)
        axarr[i].set_title(f"n_iter = {n}", fontsize=fontsize)
    axarr[i].imshow(tile.masks["binary_threshold"])
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_19_0.png

Morphological Closing

Morphological closing is similar to opening, but in the opposite order: first, binary dilation is applied *n* times, then binary erosion is applied *n* times. The effect is to reduce noise in a binary mask by closing small holes in the foreground. The strength of the effect can be controlled by setting *n*

[13]:

```
ns = ["Original Mask", 1, 3, 5]
fig, axarr = plt.subplots(nrows=1, ncols=4, figsize=(10, 10))
for i, n in enumerate(ns):
    tile = bigtile()
    if n == "Original Mask":
        axarr[i].set_title("Original Mask", fontsize=fontsize)
    else:
        t = MorphClose(mask_name="binary_threshold", n_iterations=n)
        t.apply(tile)
        axarr[i].set_title(f"n_iter = {n}", fontsize=fontsize)
    axarr[i].imshow(tile.masks["binary_threshold"])
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_21_0.png

Foreground Detection

This transform operates on binary masks and identifies regions that have a total area greater than specified threshold. Supports including holes within foreground regions, or excluding holes above a specified area threshold.

[14]:

```
tile = bigtile()
foreground_detector = ForegroundDetection(mask_name="binary_threshold")
original_mask = tile.masks["binary_threshold"].copy()
foreground_detector.apply(tile)

fig, axarr = plt.subplots(nrows=1, ncols=2, figsize=(8, 8))
axarr[0].imshow(original_mask)
axarr[0].set_title("Original Mask", fontsize=fontsize)
axarr[1].imshow(tile.masks["binary_threshold"])
axarr[1].set_title("Detected Foreground", fontsize=fontsize)
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_23_0.png

Tissue Detection

`TissueDetectionHE` is a Transform for detecting regions of tissue from an H&E image. It is composed by applying a sequence of other Transforms: first a median blur, then binary thresholding, then morphological opening and closing, and finally foreground detection.

[15]:

```
tile = bigtile()

tissue_detector = TissueDetectionHE(mask_name="tissue", outer_contours_only=True)
tissue_detector.apply(tile)

fig, axarr = plt.subplots(nrows=1, ncols=3, figsize=(8, 8))
axarr[0].imshow(tile.image)
axarr[0].set_title("Original Image", fontsize=fontsize)
axarr[1].imshow(tile.masks["tissue"])
axarr[1].set_title("Detected Tissue", fontsize=fontsize)
plot_mask(im=tile.image, mask_in=tile.masks["tissue"], ax=axarr[2])
axarr[2].set_title("Overlay", fontsize=fontsize)

for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_gallery_25_0.png

Previous Next

PathML

-
- Preprocessing: Graph construction
- View PathML on GitHub

Previous Next

Preprocessing: Graph construction

In this notebook, we will demonstrate the ability of the new pathml.graph API to construct cell and tissue graphs. Specifically, we will do the following:

1. Detect cells in the given Whole-Slide Image (WSI)
2. Detect tissues in the given WSI
3. Featurize the detected cell and tissue patches using a pre-trained ResNet-34 model
4. Construct both tissue and cell graphs using k-Nearest Neighbour (k-NN) and Region-Adjacency Graph (RAG) methods and save them as torch tensors

To get the full functionality of this notebook for a real-world dataset, we suggest you download the BRACS ROI set from the BRACS dataset. To do so, you will have to sign up and create an account. Next, you will just have to replace the root folder in the last part of the tutorial to the directory you download the BRACS dataset to. You can use the 'previous_version' or the 'latest_version' dataset.

[1]:

```
import os
from glob import glob
import argparse
from PIL import Image
import numpy as np
from tqdm import tqdm
import torch
import h5py
import warnings
import math
from skimage.measure import regionprops, label
import networkx as nx
import traceback
from glob import glob
import matplotlib.pyplot as plt

from pathml.core import HESlide, Tile, types
from pathml.preprocessing import Pipeline, NucleusDetectionHE
import pathml.core.tile
from pathml.datasets.utils import DeepPatchFeatureExtractor
from pathml.graph import RAGGraphBuilder, KNNGraphBuilder
from pathml.graph import ColorMergedSuperpixelExtractor
from pathml.graph.utils import get_full_instance_map, build_assignment_matrix

fontsize = 14
device = "cuda"  # if using GPU
# device = 'cpu' # if using CPU
```

Data

In this notebook, we will use a representative image `CMU-1-Small-Region.svs.tiff` downloaded from OpenSlide. We will then use a small tile for illustrative purposes.

[2]:

```
wsi = HESlide("../data/CMU-1-Small-Region.svs.tiff")

region = wsi.slide.extract_region(location=(800, 900), size=(500, 500))
region = np.squeeze(region)

def smalltile():
    # convenience function to create a new tile
    return Tile(region, coords=(0, 0), name="testregion", slide_type=types.HE)

tile = smalltile()
```

Nucleus Detection

Next, we will use a built-in `PathML` feature to detect cells in the given tile.

For better results, we suggest using a pre-trained segmentation model like HoVer-Net to do the cell segmentation. HoVer-Net can be trained using `PathML` in this tutorial.

[3]:

```
# create a NucleusDetectionHE object
nucleus_detection = NucleusDetectionHE(
    mask_name="detect_cell", superpixel_region_size=10
)

# apply onto our tile
nucleus_detection.apply(tile)
```

[4]:

```
# plot the original and cell segmented image
fig, axarr = plt.subplots(nrows=1, ncols=2, figsize=(8, 8))
axarr[0].imshow(tile.image)
axarr[0].set_title("Original Image", fontsize=fontsize)
axarr[1].imshow(tile.masks["detect_cell"])
axarr[1].set_title("Cell Detection", fontsize=fontsize)
for ax in axarr.ravel():
    ax.set_yticks([])
    ax.set_xticks([])
plt.tight_layout()
plt.show()
```

../_images/examples_link_construct_graphs_8_0.png

Feature Extraction

Next, we generate features for the detected cell using a pre-trained ResNet-34 model. Altough this is not needed for cell-graph construction, we include them as features in the graph for downstream training of a deep learning model.

For each detected cell, we crop out a patch containing that cell in the center. The patches for all detected cells are fed into the pretrained ResNet-34 model to generate some useful features. Although these features are not directly related to pathology, they can be a useful startining point for helping downstream models learn. Alternatively, one could also use hand-engineered features like cell shape, eccentricity, area, etc as features.

[5]:

```
# extract the image from our tile
image = tile.image

# extract the cell segmented mask from our tile
nuclei_map = tile.masks["detect_cell"]

# uniquely label each cell in the mask and record the centroids for each cell
label_instance_map = label(nuclei_map)
regions = regionprops(label_instance_map)
instance_centroids = np.empty((len(regions), 2))
for i, region in enumerate(regions):
    center_y, center_x = region.centroid  # row, col
    center_x = int(round(center_x))
    center_y = int(round(center_y))
    instance_centroids[i, 0] = center_x
    instance_centroids[i, 1] = center_y

# initialize a feature extractor object and apply
extractor = DeepPatchFeatureExtractor(
    patch_size=8,
    batch_size=32,
    entity="cell",
    architecture="resnet34",
    fill_value=255,
    resize_size=224,
    device=device,
    threshold=0,
)
features = extractor.process(image, label_instance_map)
```

Cell-graph Construction

We now construct the cell graphs using our computed cell segmentation mask using a k-Nearest Neighbour (kNN) graph. In this type of graph, a cell is connected to another cell if they are within the distance specified by the `k` parameter. Here we set `k` to 40 (pixels).

[6]:

```
knn_graph_builder = KNNGraphBuilder(k=5, thresh=40, add_loc_feats=True)
cell_graph = knn_graph_builder.process(label_instance_map, features, target=0)
```

The constructed graph overlaid onto the image can now be visualized.

[7]:

```
def plot_graph_on_image(ax, graph, image):
    from torch_geometric.utils.convert import to_networkx

    pos = graph.node_centroids.numpy()
    G = to_networkx(graph, to_undirected=True)
    ax.imshow(image, cmap="cubehelix")
    nx.draw(
        G,
        pos,
        ax=ax,
        node_size=7,
        with_labels=False,
        font_size=8,
        font_color="white",
        node_color="skyblue",
        edge_color="blue",
    )
    ax.set_facecolor("black")
    ax.set_xticks([])
    ax.set_yticks([])
    return ax


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 10))
plot_graph_on_image(ax1, cell_graph, label_instance_map)
ax1.set_title("Cell graph on nuclei map")
plot_graph_on_image(ax2, cell_graph, image)
ax2.set_title("Cell graph on input image")
plt.show()
```

../_images/examples_link_construct_graphs_14_0.png

Tissue detection and feature extraction

Next, we detect tissues with the help of the Region Adjacency Graphs (RAGs) method. This method iteratively merges neighbouring pixels with similar intensity using RAG and creates superpixels that correspond to tissue regions.

We then use the feature extractor class to compute features for each detected tissue.

[8]:

```
# Initialize tissue detector and apply
tissue_detector = ColorMergedSuperpixelExtractor(
    superpixel_size=150,
    compactness=20,
    blur_kernel_size=1,
    threshold=0.05,
    downsampling_factor=4,
)

superpixels, _ = tissue_detector.process(image)
```

[9]:

```
# initialize a feature extractor object and apply
tissue_feature_extractor = DeepPatchFeatureExtractor(
    architecture="resnet34",
    patch_size=144,
    entity="tissue",
    resize_size=224,
    fill_value=255,
    batch_size=32,
    device=device,
    threshold=0.25,
)

features = tissue_feature_extractor.process(image, superpixels)
```

Tissue-graph Construction

We now construct the tissue graphs using our computed tissue superpixels using the RAGs method.

[10]:

```
rag_graph_builder = RAGGraphBuilder(add_loc_feats=True)
tissue_graph = rag_graph_builder.process(superpixels, features, target=0)
```

[11]:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 10))
plot_graph_on_image(ax1, tissue_graph, superpixels)
ax1.set_title("Tissue graph on tissue map")
plot_graph_on_image(ax2, tissue_graph, image)
ax2.set_title("Tissue graph on input image")
plt.show()
```

../_images/examples_link_construct_graphs_21_0.png

(Optional) Creating an assignment matrix

Finally, we can create an assignment matrix that maps each detected cell to the tissue the cell is a part of. This is useful for trainind deep-learning models that require such hierarchical information.

[12]:

```
assignment = build_assignment_matrix(instance_centroids, superpixels)
```

Putting it all together

For simplicity, we provide a general framework to the detect cells and tissues, compute features, construct and save graphs, build and save assignment matrices for all WSIs in a given directory if you are using the the BRACS dataset.

[13]:

```
# Convert the tumor time given in the filename to a label
TUMOR_TYPE_TO_LABEL = {
    "N": 0,
    "PB": 1,
    "UDH": 2,
    "ADH": 3,
    "FEA": 4,
    "DCIS": 5,
    "IC": 6,
}

# Define minimum and maximum pixels for processing a WSI
MIN_NR_PIXELS = 50000
MAX_NR_PIXELS = 50000000

# Define the patch size for applying pathml.transforms.NucleusDetectionHE
PATCH_SIZE = 512
```

Next, we write the main preprocessing loop as a function.

[14]:

```
def is_valid_image(nr_pixels):
    """
    Checks if image does not exceed maximum number of pixels or exceeds minimum number of pixels.

    Args:
        nr_pixels (int): Number of pixels in given image
    """

    if nr_pixels > MIN_NR_PIXELS and nr_pixels < MAX_NR_PIXELS:
        return True
    return False


def does_exists(cg_out, tg_out, assign_out, overwrite):
    """
    Checks if given input files exist or not

    Args:
        cg_out (str): Cell graph file
        tg_out (str): Tissue graph file
        assign_out (str): Assignment matrix file
        overwrite (bool): Whether to overwrite files or not. If true, this function return false and files are
                          overwritten.
    """
    if overwrite:
        return False
    else:
        if (
            os.path.isfile(cg_out)
            and os.path.isfile(tg_out)
            and os.path.isfile(assign_out)
        ):
            return True
        return False


def process(image_path, save_path, split, plot=True, overwrite=False):
    # 1. get image path
    subdirs = os.listdir(image_path)
    image_fnames = []
    for subdir in subdirs + [""]:
        image_fnames += glob(os.path.join(image_path, subdir, "*.png"))

    image_ids_failing = []

    print("*** Start analysing {} image(s) ***".format(len(image_fnames)))
    for image_path in tqdm(image_fnames):

        # a. load image & check if already there
        _, image_name = os.path.split(image_path)
        image = np.array(Image.open(image_path))

        # Compute number of pixels in image and check the label of the image
        nr_pixels = image.shape[0] * image.shape[1]
        image_label = TUMOR_TYPE_TO_LABEL[image_name.split("_")[2]]

        # Get the output file paths of cell graphs, tissue graphs and assignment matrices
        cg_out = os.path.join(
            save_path, "cell_graphs", split, image_name.replace(".png", ".pt")
        )
        tg_out = os.path.join(
            save_path, "tissue_graphs", split, image_name.replace(".png", ".pt")
        )
        assign_out = os.path.join(
            save_path, "assignment_matrices", split, image_name.replace(".png", ".pt")
        )

        # If file was not already created or not too big or not too small, then process
        if not does_exists(cg_out, tg_out, assign_out, overwrite) and is_valid_image(
            nr_pixels
        ):

            print(f"Image name: {image_name}")
            print(f"Image size: {image.shape[0], image.shape[1]}")

            if plot:
                print("Input ROI:")
                plt.imshow(image)
                plt.show()

            try:
                # Read the image as a pathml.core.SlideData class
                print("Reading image")
                wsi = HESlide(
                    image_path, name=image_path, backend="openslide", stain="HE"
                )

                # Apply our HoverNetNucleusDetectionHE as a pathml.preprocessing.Pipeline over all patches
                print("Detecting nuclei")
                pipeline = Pipeline(
                    [
                        NucleusDetectionHE(
                            mask_name="detect_nuclei", stain_estimation_method="macenko"
                        )
                    ]
                )

                # Run the Pipeline
                wsi.run(
                    pipeline,
                    overwrite_existing_tiles=True,
                    distributed=False,
                    tile_pad=True,
                    tile_size=PATCH_SIZE,
                )

                # Extract the ROI, nuclei instance maps as an np.array from a pathml.core.SlideData object
                image, nuclei_map, nuclei_centroid = get_full_instance_map(
                    wsi, patch_size=PATCH_SIZE, mask_name="detect_nuclei"
                )

                # Use a ResNet-34 to extract the features from each detected cell in the ROI
                print("Extracting features from cells")
                extractor = DeepPatchFeatureExtractor(
                    patch_size=64,
                    batch_size=64,
                    entity="cell",
                    architecture="resnet34",
                    fill_value=255,
                    resize_size=224,
                    device=device,
                    threshold=0,
                )
                features = extractor.process(image, nuclei_map)

                # Build a kNN graph with nodes as cells, node features as ResNet-34 computed features, and edges within
                # a threshold of 50
                print("Building graphs")
                knn_graph_builder = KNNGraphBuilder(k=5, thresh=50, add_loc_feats=True)
                cell_graph = knn_graph_builder.process(
                    nuclei_map, features, target=image_label
                )

                # Plot cell graph on ROI image
                if plot:
                    print("Cell graph on ROI:")
                    plot_graph_on_image(cell_graph, image)

                # Save the cell graph
                torch.save(cell_graph, cg_out)

                # Detect tissue using pathml.graph.ColorMergedSuperpixelExtractor class
                print("Detecting tissue")
                tissue_detector = ColorMergedSuperpixelExtractor(
                    superpixel_size=200,
                    compactness=20,
                    blur_kernel_size=1,
                    threshold=0.05,
                    downsampling_factor=4,
                )

                superpixels, _ = tissue_detector.process(image)

                # Use a ResNet-34 to extract the features from each detected tissue in the ROI
                print("Extracting features from tissues")
                tissue_feature_extractor = DeepPatchFeatureExtractor(
                    architecture="resnet34",
                    patch_size=144,
                    entity="tissue",
                    resize_size=224,
                    fill_value=255,
                    batch_size=32,
                    device=device,
                    threshold=0.25,
                )
                features = tissue_feature_extractor.process(image, superpixels)

                # Build a RAG with tissues as nodes, node features as ResNet-34 computed features, and edges using the
                # RAG algorithm
                print("Building graphs")
                rag_graph_builder = RAGGraphBuilder(add_loc_feats=True)
                tissue_graph = rag_graph_builder.process(
                    superpixels, features, target=image_label
                )

                # Plot tissue graph on ROI image
                if plot:
                    print("Tissue graph on ROI:\n")
                    plot_graph_on_image(tissue_graph, image)

                # Save the tissue graph
                torch.save(tissue_graph, tg_out)

                # Build as assignment matrix that maps each cell to the tissue it is a part of
                assignment = build_assignment_matrix(nuclei_centroid, superpixels)

                # Save the assignment matrix
                torch.save(torch.tensor(assignment), assign_out)

            except:
                print(f"Failed {image_path}")
                image_ids_failing.append(image_path)

    print(
        "\nOut of {} images, {} successful graph generations.".format(
            len(image_fnames), len(image_fnames) - len(image_ids_failing)
        )
    )
    print("Failing IDs are:", image_ids_failing)
```

Finally, we write a main function that calls the process function for a specified root and output directory, along with the name of the split (either train, test or validation if using BRACS).

[15]:

```
def main(base_path, save_path, split=None):
    if split is not None:
        root_path = os.path.join(base_path, split)
    else:
        root_path = base_path

    print(root_path)

    os.makedirs(os.path.join(save_path, "cell_graphs", split), exist_ok=True)
    os.makedirs(os.path.join(save_path, "tissue_graphs", split), exist_ok=True)
    os.makedirs(os.path.join(save_path, "assignment_matrices", split), exist_ok=True)

    process(root_path, save_path, split, plot=False, overwrite=True)
```

[16]:

```
# Folder containing all WSI images
base = "../data/"

# Output path
save_path = "../data/output/"

# Start preprocessing
main(base, save_path, split="train")
```

```
../../data/BRACS_RoI/latest_version/train
*** Start analysing 3657 image(s) ***

  0%|                                                                                                  | 0/3657 [00:00<?, ?it/s]

Image name: BRACS_1238_FEA_56.png
Image size: (751, 1050)
Reading image
Detecting nuclei
Extracting features from cells

Building graphs
Detecting tissue
Extracting features from tissues

  0%|                                                                             | 0/3657 [00:15<?, ?it/s]

Building graphs

Out of 3657 images, 3657 successful graph generations.
Failing IDs are: []
```

References

- Pati, Pushpak, Guillaume Jaume, Antonio Foncubierta-Rodriguez, Florinda Feroce, Anna Maria Anniciello, Giosue Scognamiglio, Nadia Brancati et al. “Hierarchical graph representations in digital pathology.” Medical image analysis 75 (2022): 102264.
- Brancati, Nadia, Anna Maria Anniciello, Pushpak Pati, Daniel Riccio, Giosuè Scognamiglio, Guillaume Jaume, Giuseppe De Pietro et al. “Bracs: A dataset for breast carcinoma subtyping in h&e histology images.” Database 2022 (2022): baac093.

Previous Next

Inference API: Tutorial using ONNX

View on GitHub

Introduction

This notebook is a tutorial on how to use the future ONNX `inference` feature in PathML.

Some notes: - The ONNX inference pipeline uses the existing PathML Pipeline and Transforms infrastructure. - ONNX labels are saved to a `pathml.core.slide_data.SlideData` object as `tiles`. - Users can iterate over the tiles as they would when using this feature for preprocessing. - Preprocessing images before inference - Users will need to create their own bespoke `pathml.preprocessing.transforms.transform` method to preprocess images before inference if necessary. - A guide on how to create preprocessing pipelines is here. - A guide on how to run preprocessing pipelines is here. - ONNX Model Initializers - ONNX models often have neural network initializers stored in the input graph. This means that the user is expected to specify initializer values when running inference. To solve this issue, we have a function that removes the network initializers from the input graph. This functions is adopted from the `onnxruntime` github.
- We also have a function that checks if the initializers have been removed from the input graph before running inference. Both of these functions are described more below. - When using a model stored remotely on HuggingFace, the model is *downloaded locally* before being used. The user will need to delete the model after running `Pipeline` with a method that comes with the model class. An example of how to do this is below.

Quick Sample Code

- Below is an example of how users would use the ONNX inference feature in PathML with a locally stored 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)
```

- Below is an example of how users would use the ONNX inference feature in PathML with a model stored in the public HuggingFace repository.

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

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

from pathml.inference import RemoteTestHoverNet

# Define slide path
slide_path = 'PATH TO SLIDE'

inference = RemoteTestHoverNet()

# Create a transformation list
transformation_list = [
    inference
]

# Initialize pathml.core.slide_data.SlideData object
wsi = SlideData(slide_path)

# Set up PathML pipeline
pipeline = Pipeline(transformation_list)

# Run Inference
wsi.run(pipeline, tile_size = 256)

# DELETE ONNX MODEL DOWNLOADED FROM HUGGINGFACE
inference.remove()
```

Load Packages

NOTE - Please put in your environment name in the following line if you are using a jupyter notebook. If not, you may remove this line. `os.environ["JAVA_HOME"] = "/opt/conda/envs/YOUR ENVIRONMENET NAME"`

[8]:

```
import os

os.environ["JAVA_HOME"] = (
    "/opt/conda/envs/YOUR ENVIRONMENET NAME"  # TO DO: CHANGE THIS TO YOUR ENVIRONMENT NAME
)
import numpy as np
import onnx
import onnxruntime
import requests
import torch

from pathml.core import SlideData, Tile
from dask.distributed import Client
from pathml.preprocessing import Pipeline
import pathml.preprocessing.transforms as Transforms

import matplotlib.pyplot as plt
import matplotlib

from PIL import Image
```

ONNX Inference Class and ONNX Model Fixer

- Here is the raw code for the functions that handle the initializers in the ONNX model and the classes that run the inference.

Functions to remove initializers and check that initializers have been removed.

- `remove_initializer_from_input`
    - This function removes any initializers from the input graph of the ONNX model.
    - Without removing the initializers from the input graph, users will not be able to run inference.
    - Adapted from the `onnxruntime` github.
    - Users specify:
        - `model_path` (str): path to ONNX model,
        - `new_path` (str): path to save adjusted model w/o initializers

    - We will run this function on all models placed in our model zoo, so users will not have to run it unless they are working with their own local models.

- `check_onnx_clean`
    - Checks if the initializers are in the input graph
    - Returns `True` and a `ValueError` if there are initializers in the input graph
    - Adapted from the `onnxruntime` github.
    - Users specify:
        - `model_path` (str): path to ONNX model
- `convert_pytorch_onnx`
    - Converts a PyTorch `.pt` file to `.onnx`
    - Wrapper function of the PyTorch function to handle the conversion.
    - Users specify:
        - model_path (torch.nn.Module Model): Pytorch model to be converted,
        - dummy_tensor (torch.tensor): dummy input tensor that is an example of what will be passed into the model,
        - model_name (str): name of ONNX model created with .onnx at the end,
        - opset_version (int): which opset version you want to use to export
        - input_name (str): name assigned to dummy_tensor
    - Note that the model class must be defined before loading the `.pt` file and set to eval before calling this function.

Inference Classes

- `InferenceBase`
    - This class inherits from `pathml.preprocessing.transforms.transform`, similar to all of the preprocessing transformations. Inheriting from `transforms.transform` allows us to use the existing `Pipeline` function in PathML which users should be familar with.
    - This is the base class for all Inference classes for ONNX modeling
    - Each instance of a class also comes with a `model_card` which specifies certain details of the model in dictionary form. The default parameters are:

        -
          ```
          self.model_card = {
          'name' : None,
          'num_classes' : None,
          'model_type' : None,
          'notes' : None,
          'model_input_notes': None,
          'model_output_notes' : None,
          'citation': None }
          ```
        - Model cards are where important information about the model should be kept. Since they are in dictionary form, the user can add keys and values as they see fit.
        - This class also has getter and setter functions to adjust the `model_card`. Certain functions include `get_model_card`, `set_name`, `set_num_classes`, etc.
- `Inference`
    - This class is for when the user wants to use an ONNX model stored locally.
    - Calls the `check_onnx_clean` function to check if the model is clean.
    - Users specify:
        - `model_path` (str): path to ONNX model,
        - `input_name` (str): name of input for ONNX model, *defaults to ``data``*
        - `num_classes` (int): number of outcome classes,
        - `model_type` (str): type of model (classification, segmentation)
        - `local` (bool): if you are using a local model or a remote model, *defaults to ``True``*
- `HaloAIInference`
    - This class inherits from `Inference`

    - HaloAI ONNX models always return 20 prediction maps: this class will subset and return the necessary ones.

- `RemoteTestHoverNet`
    - This class inherits from `Inference` and is the test class for public models hosted on `HuggingFace`.
    - `local` is automatically set to `False`
    - Our current test model is a HoverNet from TIAToolbox
    - Pocock J, Graham S, Vu QD, Jahanifar M, Deshpande S, Hadjigeorghiou G, Shephard A, Bashir RM, Bilal M, Lu W, Epstein D. TIAToolbox as an end-to-end library for advanced tissue image analytics. Communications medicine. 2022 Sep 24;2(1):120.
    - Its `model_card` is:

        -
          ```
          {'name': 'Tiabox HoverNet Test',
           'num_classes': 5,
           'model_type': 'Segmentation',
           'notes': None,
           'model_input_notes': 'Accepts tiles of 256 x 256',
           'model_output_notes': None,
           'citation': 'Pocock J, Graham S, Vu QD, Jahanifar M, Deshpande S, Hadjigeorghiou G, Shephard A, Bashir RM, Bilal M, Lu W, Epstein
          ```
- `RemoteMesmer`
    - This class inherits from `Inference` and is hosted on `HuggingFace`.
    - `local` is automatically set to `False`
    - This model is from Deepcell
    - Greenwald NF, Miller G, Moen E, Kong A, Kagel A, Dougherty T, Fullaway CC, McIntosh BJ, Leow KX, Schwartz MS, Pavelchek C. Whole-cell segmentation of tissue images with human-level performance using large-scale data annotation and deep learning. Nature biotechnology. 2022 Apr;40(4):555-65.
    - Its `model_card` is:
        - ```python {'name': "Deepcell's Mesmer", 'num_classes': 3, 'model_type': 'Segmentation', 'notes': None, 'model_input_notes': 'Accepts tiles of 256 x 256', 'model_output_notes': None, 'citation': 'Greenwald NF, Miller G, Moen E, Kong A, Kagel A, Dougherty T, Fullaway CC, McIntosh BJ, Leow KX, Schwartz MS, Pavelchek C. Whole-cell segmentation of tissue images with human-level performance using large-scale data annotation and deep learning. Nature biotechnology. 2022 Apr;40(4):555-65.'}

Try it Yourself!

- What you need:
    - An ONNX model stored locally
    - An image with which you want to run inference stored locally
    - PathML already downloaded
- Make sure to define the `Inference` class and `remove_initializer_from_input` above in the previous seciton if you have not downloaded the latest version of PathML.
- You will need to define the following variables:
    - `slide_path`: 'PATH TO SLIDE'
    - `model_path`: 'PATH TO ONNX MODEL'
    - `new_path`: 'PATH TO SAVE FIXED ONNX MODEL'
    - `num_classes`: 'NUMBER OF CLASSES IN YOUR DATASET'
    - `tile_size`: 'TILE SIZE THAT YOUR ONNX MODEL ACCEPTS'
- The code in the cell below assumes you want the images passed in as is. If you need to select channels, you will need to add another `transform` method to do so before the inference transform. The following code provides an example if you want to subset into the first channel of an image. *Remember that PathML reads images in as XYZCT.*

```
class convert_format(Transforms.Transform):
    def F(self, image):
        # orig = (1280, 1280, 1, 6, 1) = (XYZCT)
        image = image[:, :, :, 0, ...] # this will make the tile (1280, 1280, 1, 1)
        return image

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

convert = convert_format()
inference = Inference(
    model_path = 'PATH TO LOCAL MODEL',
    input_name = 'data',
```

```
    num_classes = 'NUMBER OF CLASSES' ,
    model_type = 'CLASSIFICATION OR SEGMENTATION',
    local = True)

transformation_list = [convert, inference]
```

Converting a Pytorch Model to ONNX Using the `convert_pytorch_onnx` Function

Note the following: - Similar to PyTorch, you will need to define and create an instance of you model class before loading the `.pt` file. Then you will need to set it to eval mode before calling the conversion function. The code to do these steps is below.

[ ]:

```
# Define your model class
num_input, num_output, batch_size = 10, 1, 1


class SimpleModel(torch.nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.linear = torch.nn.Linear(num_input, num_output)
        torch.nn.init.xavier_uniform_(self.linear.weight)

    def forward(self, x):
        y = self.linear(x)
        return y


# Define your model var
model = SimpleModel()

# Export model as .pt if you haven't already done so
# If you have already exported a .pt file, you will still need to define a model class, initialize it, and set it to eval mode.
# If you saved your model using `torch.jit.script`, you will not need to define your model class and instead load it using `torch.jit.load` then s
torch.save(model, "test.pt")

# Load .pt file
model_test = torch.load("test.pt")
# Set model to eval mode
model_test.eval()

# Define a dummy tensor (this is an example of what the ONNX should expect during inference)
x = torch.randn(batch_size, num_input)

# Run conversion function
convert_pytorch_onnx(
    model=model_test, dummy_tensor=x, model_name="NAME_OF_OUTPUT_MODEL_HERE.onnx"
)
```

Local ONNX Model Using the `Inference` Class

[ ]:

```
# 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
remove_initializer_from_input(model_path, new_path)

inference = Inference(
    model_path=new_path,
    input_name="data",
    num_classes="NUMBER OF CLASSES",
    model_type="CLASSIFICATION OR SEGMENTATION",
    local=True,
)

transformation_list = [inference]

# Initialize pathml.core.slide_data.SlideData object
wsi = SlideData(slide_path)

# Set up PathML pipeline
pipeline = Pipeline(transformation_list)

# Run Inference
# Level is equal to 0 for highest resolution (Note that this is the default setting)
wsi.run(pipeline, tile_size="TILE SIZE THAT YOUR ONNX MODEL ACCEPTS", level=0)
```

Local ONNX Model Using the `HaloAIInference` Class

[ ]:

```
# 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
remove_initializer_from_input(model_path, new_path)

inference = HaloAIInference(
    model_path=new_path,
    input_name="data",
    num_classes="NUMBER OF CLASSES",
    model_type="CLASSIFICATION OR SEGMENTATION",
    local=True,
)

transformation_list = [inference]

# Initialize pathml.core.slide_data.SlideData object
wsi = SlideData(slide_path)

# Set up PathML pipeline
pipeline = Pipeline(transformation_list)

# Run Inference
# Level is equal to 0 for highest resolution (Note that this is the default setting)
wsi.run(pipeline, tile_size="TILE SIZE THAT YOUR ONNX MODEL ACCEPTS", level=0)
```

Remote ONNX Using our `RemoteTestHoverNet` Class

- Uses a Hovernet from TIAToolbox
- This version of Hovernet was trained on the MoNuSAC dataset.
- Note that the purpose of this model is to illustrate how PathML will handle future remote models. We plan on release more public models to our model zoo on HuggingFace in the future.
- Citation for model:
    - Pocock J, Graham S, Vu QD, Jahanifar M, Deshpande S, Hadjigeorghiou G, Shephard A, Bashir RM, Bilal M, Lu W, Epstein D. TIAToolbox as an end-to-end library for advanced tissue image analytics. Communications medicine. 2022 Sep 24;2(1):120.
- Make sure your image has 3 channels!
- When the `RemoteTestHoverNet` is first initialized, it downloads the HoverNet from HuggingFace and saves it locally on your own system as `temp.onnx`.
    - **You will need to remove it manually by calling the ``remove()`` method** An example of how to call this method is in the last line in the code below.

[ ]:

```
# Define slide path
slide_path = "PATH TO SLIDE"

inference = RemoteTestHoverNet()

# Create a transformation list
transformation_list = [inference]

# Initialize pathml.core.slide_data.SlideData object
wsi = SlideData(slide_path)

# Set up PathML pipeline
pipeline = Pipeline(transformation_list)

# Run Inference
wsi.run(pipeline, tile_size=256)

# DELETE ONNX MODEL DOWNLOADED FROM HUGGINGFACE
inference.remove()
```

Iterate over the tiles

Now that you have your tiles saved to your SlideData object, you can now iterate over them.

For example, if you wanted to check the shape of the tiles you could run the following code:

```
for tile in wsi.tiles:
    print(tile.image.shape)
```

To see how to use these tiles to make visualizations, see below.

Full Example With Vizualization of Output

The `RemoteTestHoverNet()` uses a pretrained HoverNet from TIAToolBox trained on the MoNuSAC dataset. **The model was trained to accept tiles of 256x256 to create a prediction matrix of size 164x164 with 9 channels.** The first 5 channels correspond to the Nuclei Types (TP), the next two channels correspond to the Nuclei Pixels (NP), and the last two channels correspond to the Hover (HV). The documention for these channels can be found here on TIAToolBox's website.

In this example we use an taken from the MoNuSAC dataset. See citation in the `References` section.

Run Code as Demonstrated Above

Note that to run the following code, you will need to download and save the image titled `TCGA-5P-A9K0-01Z-00-DX1_1.svs` in the same directory as the notebook.

[3]:

```
# Define slide path
slide_path = "TCGA-5P-A9K0-01Z-00-DX1_1.svs"

inference = RemoteTestHoverNet()

# Create a transformation list
transformation_list = [inference]

# Initialize pathml.core.slide_data.SlideData object
wsi = SlideData(slide_path)

# Set up PathML pipeline
pipeline = Pipeline(transformation_list)

# Run Inference
wsi.run(pipeline, tile_size=256, tile_stride=164, tile_pad=True)

# DELETE ONNX MODEL DOWNLOADED FROM HUGGINGFACE
inference.remove()
```

Let's look at the first tile which comes from the top left corner (0,0) and Nucleus Pixel predictions.

[4]:

```
for tile in wsi.tiles:
    # Create empty numpy array
    a = np.empty((2, 164, 164), dtype=object)
    # Get Nucleus Predictions
    classes = tile.image[0, 5:7, :, :]
    a = classes
    # Take the argmax to make the predictions binary
    image = np.argmax(a, axis=0)
    # Multiple values by 255 to make the array image friendly
    image = image * (255 / 1)
    # Make a grey scale image
    img = Image.fromarray(image.astype("uint8"), "L")
    # Save Image
    img.save("test_array_1.png")
    # Can break after one iteration since we are using at the tile at (0, 0).
    break
```

Lets visualize the tile vs the tile predictions. Since the model uses a 256x256 tile to create a prediction map of size 164x164, we need to take our tile located at (0,0) and crop it down to the center 164x164 pixes.

[5]:

```
prediction_dim = 164
tile_dim = 256
crop_amount = int((256 - 164) / 2)
wsi = SlideData(slide_path)

generator = wsi.generate_tiles(shape=(tile_dim, tile_dim), level=0)

for tile in generator:
    # Extract array from tile
    image = tile.image
    # Crop tile
    image = image[
        crop_amount : crop_amount + prediction_dim,
        crop_amount : crop_amount + prediction_dim,
    ]
    # Convert array to image
    img = Image.fromarray(image)
    # Save Image
    img.save("raw_tile.png")
    break
```

[12]:

```
# Set figure sice
plt.rcParams["figure.figsize"] = 11, 8

# Read images
```

```
img_A = matplotlib.image.imread("raw_tile.png")
img_B = matplotlib.image.imread("test_array_1.png")

# Set up plots
fig, ax = plt.subplots(1, 2)
plt.xticks([])
plt.yticks([])
ax[0].imshow(img_A)
ax[1].imshow(img_B, cmap="gray")
ax[0].set_title("Original Image")
ax[1].set_title("Model Predictions")
plt.tight_layout()

# Get rid of tick marks
for a in ax.ravel():
    a.set_xticks([])
    a.set_yticks([])

# Show images
plt.show()
```

../_images/examples_link_workflow_Inference_21_0.png

References

- Pocock J, Graham S, Vu QD, Jahanifar M, Deshpande S, Hadjigeorghiou G, Shephard A, Bashir RM, Bilal M, Lu W, Epstein D. TIAToolbox as an end-to-end library for advanced tissue image analytics. Communications medicine. 2022 Sep 24;2(1):120.
- 18. Verma, et al. "MoNuSAC2020: A Multi-organ Nuclei Segmentation and Classification Challenge." IEEE Transactions on Medical Imaging (2021).
- https://github.com/microsoft/onnxruntime/blob/main/tools/python/remove_initializer_from_input.py
- https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html

Previous Next

Loading Images: Quickstart

View on GitHub

`PathML` provides support for loading a wide array of imaging modalities and file formats under a standardized syntax. In this vignette, we highlight code snippets for loading a range of image types ranging from brightfield H&E and IHC to highly multiplexed immunofluorescence and spatial expression and proteomics, from small images to gigapixel scale:

Imaging modality	File format	Source	Image dimensions (X, Y, Z, C, T)
Brightfield H&E	Aperio SVS	OpenSlide example data	(32914, 46000, 1, 3, 1)
Brightfield H&E	Generic tiled TIFF	OpenSlide example data	(32914, 46000, 1, 3, 1)
Brightfield IHC	Hamamatsu NDPI	OpenSlide example data	(73728, 126976, 1, 3, 1)
Brightfield H&E	Hamamatsu VMS	OpenSlide example data	(76288, 102400, 1, 3, 1)
Brightfield H&E	Leica SCN	OpenSlide example data	(153470, 53130, 1, 3, 1)
Fluorescence	MIRAX	OpenSlide example data	(170960, 76324, 1, 3, 1)
	Olympus		(6753, 13196,

Brightfield IHC	VSI	OpenSlide example data	1, 3, 1)
Brightfield H&E	Trestle TIFF	OpenSlide example data	(25408, 61504, 1, 3, 1)
Brightfield H&E	Ventana BIF	OpenSlide example data	(93951, 105813, 1, 3, 1)
Fluorescence	Zeiss ZVI	OpenSlide example data	(1388, 1040, 13, 3, 1)
Brightfield H&E	DICOM	`Orthanc example data <https://wsi.orthanc-server.com/orthanc/app/explorer.html#instance?uuid=83a7f39f-b48bd71c-09856fe8-ecb90e4d-00c58ec2> `__	(30462, 78000, 1, 3, 1)
Fluorescence (CODEX spatial proteomics)	TIFF	Schurch et al., Cell 2020	(1920, 1440, 17, 4, 23)
Fluorescence (time-series + volumetric)	OME-TIFF	OME-TIFF example data	(512, 512, 10, 2, 43)
Fluorescence (MERFISH spatial gene expression)	TIF	Zhuang et al., 2020	(2048, 2048, 7, 1, 40)
Fluorescence (Visium 10x spatial gene expression)	TIFF	10x Genomics	(25088, 26624, 1, 1, 4)

All images used in these examples are publicly available for download at the links listed above.

Note that across the wide diversity of modalities and file formats, the syntax for loading images is consistent (see examples below).

```
[2]:
```

```
# import utilities for loading images
from pathml.core import HESlide, CODEXSlide, VectraSlide, SlideData, types
```

Aperio SVS

[3]:

```
my_aperio_image = HESlide("../../data/data/CMU-1.svs")
```

Generic tiled TIFF

[4]:

```
my_generic_tiff_image = HESlide("../../data/data/CMU-1.tiff", backend="bioformats")
```

Hamamatsu NDPI

The `labels` field can be used to store slide-level metadata. For example, in this case we store the target gene, which is Ki-67:

[5]:

```
my_ndpi_image = SlideData(
    "../../data/data/OS-2.ndpi", labels={"taget": "Ki-67"}, slide_type=types.IHC
)
```

Hamamatsu VMS

[8]:

```
my_vms_image = HESlide(
    "../../data/data/CMU-1-40x - 2010-01-12 13.24.05.vms", backend="openslide"
)
```

Leica SCN

[9]:

```
my_leica_image = HESlide("../../data/data/Leica-1.scn")
```

MIRAX

[12]:

```
my_mirax_image = SlideData(
    "../../data/data/Mirax2-Fluorescence-1.mrxs", slide_type=types.IF
)
```

Olympus VSI

Again, we use the `labels` field to store slide-level metadata such as the name of the target gene.

[13]:

```
my_olympus_vsi = SlideData(
    "../../data/data/OS-3.vsi", labels={"taget": "PTEN"}, slide_type=types.IHC
)
```

Trestle TIFF

[14]:

```
my_trestle_tiff = SlideData("../../data/data/CMU-2.tif")
```

Ventana BIF

[15]:

```
my_ventana_bif = SlideData("../../data/data/OS-1.bif")
```

Zeiss ZVI

Again, we use the `labels` field to store slide-level metadata such as the name of the target gene.

[17]:

```
my_zeiss_zvi = SlideData(
    "../../data/data/Zeiss-1-Stacked.zvi",
    labels={"target": "HER-2"},
    slide_type=types.IF,
)
```

DICOM

[19]:

```
my_dicom = HESlide("../../data/data/orthanc_example.dcm")
```

Volumetric + time-series OME-TIFF

[21]:

```
my_volumetric_timeseries_image = SlideData(
    "../../data/data/ttubhiswt_C1_TP41.ome.tif",
    labels={"organism": "C elegans"},
    volumetric=True,
    time_series=True,
    backend="bioformats",
)
```

CODEX spatial proteomics

The `labels` field can be used to store whatever slide-level metadata the user wants; here we specify the tissue type

[22]:

```
my_codex_image = CODEXSlide(
    "../../data/data/reg031_X01_Y01.tif", labels={"tissue type": "CRC"}
);
```

MERFISH spatial gene expression

[15]:

```
my_merfish_image = SlideData("./data/aligned_images0.tif", backend="bioformats")
```

Visium 10x spatial gene expression

Here we load an image with accompanying expression data in `AnnData` format.

[16]:

```
# load the counts matrix of spatial genomics information
import scanpy as sc

adata = sc.read_10x_h5("./data/Visium_FFPE_Mouse_Brain_IF_raw_feature_bc_matrix.h5")

# load the image, with accompanying counts matrix metadata
my_visium_image = SlideData(
    "./data/Visium_FFPE_Mouse_Brain_IF_image.tif", counts=adata, backend="bioformats"
)
```

```
Variable names are not unique. To make them unique, call `.var_names_make_unique`.
Variable names are not unique. To make them unique, call `.var_names_make_unique`.
```

Vectra Slide

[3]:

```
my_vectra_image = VectraSlide(
    "./data/MISI3542i_W21-04143_bi016966_M394_OVX_LM_Scan1_[14384,29683]_component_data.tif",
    labels={"tissue type": "breast"},
)
```

Previous Next

Multiparametric Imaging: Quickstart

View on GitHub

Pathology imaging experiments commonly produce data where each channel corresponds to a molecular feature, such as the expression level of a protein or nucleic acid. PathML implements MultiparametricSlide, a subclass of SlideData for which we implement special transforms (for more information about transforms, see “Creating Preprocessing Pipelines” in our documentation).

MultiparametricSlide is the appropriate type to analyze low-dimensional techniques including immunofluoresence (protein and in situ hybridization/RNAscope). Recently multiple approaches to higher dimensional imaging of ‘spatial omics’, the simultaneous measurement of a large number of molecular features, have emerged (see https://www.nature.com/articles/s41592-020-01033-y, https://pubmed.ncbi.nlm.nih.gov/30078711/, among many others).

In this notebook we run a pipeline to analyze CODEX data to demonstrate PathML’s support for multiparametric imaging data.

We use the MultiparametricSlide subclass, CODEXSlide, which supports special preprocessing transformations for the CODEX technique. See “Convenience SlideData Classes” (https://pathml.readthedocs.io/en/latest/api_core_reference.html#convenience-slidedata-classes) to see other subclasses that we have implemented.

[1]:

```
import os

os.environ["JAVA_HOME"] = "/opt/conda/envs/pathml/"

# load libraries and data
from pathml.core.slide_data import CODEXSlide
from pathml.preprocessing.pipeline import Pipeline
from pathml.preprocessing.transforms import SegmentMIF, QuantifyMIF, CollapseRunsCODEX

import numpy as np
import matplotlib.pyplot as plt
from dask.distributed import Client
from deepcell.utils.plot_utils import make_outline_overlay
from deepcell.utils.plot_utils import create_rgb_image


import warnings

warnings.filterwarnings("ignore")

%matplotlib inline

slidedata = CODEXSlide("../../data/data/reg031_X01_Y01.tif")
```

Here we analyze a TMA from Schurch et al., *Coordinated Cellular Neighborhoods Orchestrate Antitumoral Immunity at the Colorectal Cancer Invasive Front* (Cell, 2020)

Below are the proteins measured in this TMA and the cell types they label. CODEX images proteins in cycles of 3, so here we list proteins by cycle.

Cycle	Protein 1	Protein 2	Protein 3
HOECHST1	blank	blank	blank

HOECHST2	CD44 - stroma	FOXP3 - regulatory T cells	CDX2 - intestinal epithelia
HOECHST3	CD8 - cytotoxic T cells	p53 - tumor suppressor	GATA3 - Th2 helper T cells
HOECHST4	CD45 - hematopoietic cells	T-bet - Th1 cells	beta-catenin - Wnt signaling
HOECHST5	HLA-DR - MHC-II	PD-L1 - checkpoint	Ki67 - proliferation
HOECHST6	CD45RA - naive T cells	CD4 - T helper cells	CD21 - DCs
HOECHST7	MUC-1 - epithelia	CD30 - costimulator	CD2 - T cells
HOECHST8	Vimentin - cytoplasm	CD20 - B cells	LAG-3 - checkpoint
HOECHST9	Na-K-ATPase - membranes	CD5 - T cells	IDO-1 - metabolism
HOECHST10	Cytokeratin - epithelia	CD11b - macrophages	CD56 - NK cells
HOECHST11	aSMA - smooth muscle	BCL-2 - apoptosis	CD25 - IL-2 Ra
HOECHST12	Collagen IV - bas. memb.	CD11c - DCs	PD-1 - checkpoint
HOCHST13	Granzyme B - cytotoxicity	EGFR - signaling	VISTA - costimulator
HOECHST14	CD15 - granulocytes	CD194 - CCR4 chemokine R	ICOS - costimulator
HOECHST15	MMP9 - matrix metalloproteinase	Synaptophysin - neuroendocrine	CD71 - transferrin R
HOECHST16	GFAP - nerves	CD7 - T cells	CD3 - T cells
HOECHST17	Chromogranin A - neuroendocrine	CD163 - macrophages	CD57 - NK cells
HOECHST18	empty - A488_18	CD45RO - memory cells	CD68 - macrophages
HOECHST19	empty - A488_19	CD31 - vasculature	Podoplanin - lymphatics
HOECHST20	empty-A488-20	CD34 - vasculature	CD38 - multifunctional
HOECHST21	empty-A488-21	CD138 - plasma cells	MMP12 - matrix metalloproteinase
HOECHST22	empty-A488-22	empty-Cy3-22	empty-Cy5-22
HOECHST23	empty-A488-23	empty-Cy3-23	DRAQ5

[2]:

```
# These tif are of the form (x,y,z,c,t) but t is being used to denote cycles
# 17 z-slices, 4 channels per 23 cycles, 70 regions
slidedata.slide.shape
```

[2]:

```
(1440, 1920, 17, 4, 23)
```

Defining a Multiparametric Pipeline

We define a pipeline that chooses a z-slice from our CODEX image, segments cells in the image, then quantifies the expression of each protein in each cell.

[3]:

```
# 31 -> Na-K-ATPase
pipe = Pipeline(
    [
        CollapseRunsCODEX(z=6),
        SegmentMIF(
            model="mesmer",
            nuclear_channel=0,
            cytoplasm_channel=31,
            image_resolution=0.377442,
        ),
        QuantifyMIF(segmentation_mask="cell_segmentation"),
    ]
)
client = Client()
slidedata.run(pipe, distributed=True, client=client, tile_size=1000, tile_pad=False);
```

```
2024-02-14 16:33:05.572790: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror
2024-02-14 16:33:05.572836: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2024-02-14 16:33:05.572863: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (p
2024-02-14 16:33:05.573264: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Netwo
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:tensorflow:No training configuration found in save file, so the model was *not* compiled. Compile it manually.
/opt/conda/envs/pathml/lib/python3.9/site-packages/anndata/_core/anndata.py:183: ImplicitModificationWarning: Transforming to str index.
  warnings.warn("Transforming to str index.", ImplicitModificationWarning)
```

[4]:

```
img = slidedata.tiles[0].image
```

[5]:

```
plt.imshow(img[:, :, 7])
```

[5]:

```
<matplotlib.image.AxesImage at 0x7fe4506fce50>
```

../_images/examples_link_multiplex_if_8_1.png

[6]:

```
def plot(slidedata, tile, channel1, channel2):
    image = np.expand_dims(slidedata.tiles[tile].image, axis=0)
    nuc_segmentation_predictions = np.expand_dims(
        slidedata.tiles[tile].masks["nuclear_segmentation"], axis=0
    )
    cell_segmentation_predictions = np.expand_dims(
        slidedata.tiles[tile].masks["cell_segmentation"], axis=0
    )
    # nuc_cytoplasm = np.expand_dims(np.concatenate((image[:,:,:,channel1,0], image[:,:,:,channel2,0]), axis=2), axis=0)
    nuc_cytoplasm = np.stack(
        (image[:, :, :, channel1], image[:, :, :, channel2]), axis=-1
    )
    rgb_images = create_rgb_image(nuc_cytoplasm, channel_colors=["blue", "green"])
    overlay_nuc = make_outline_overlay(
        rgb_data=rgb_images, predictions=nuc_segmentation_predictions.astype("uint8")
    )
    overlay_cell = make_outline_overlay(
        rgb_data=rgb_images, predictions=cell_segmentation_predictions.astype("uint8")
    )
    fig, ax = plt.subplots(1, 2, figsize=(15, 15))
    ax[0].imshow(rgb_images[0, ...])
    ax[1].imshow(overlay_cell[0, ...])
    ax[0].set_title("Raw data")
    ax[1].set_title("Cell Predictions")
    plt.show()
```

Let's check the quality of our segmentations in a 1000x1000 pixel tile by looking at DAPI, Syp, and CD44.

[7]:

```
# DAPI + Syp
plot(slidedata, tile=0, channel1=0, channel2=60)

# DAPI + CD44
plot(slidedata, tile=0, channel1=0, channel2=24)
```

../_images/examples_link_multiplex_if_11_0.png

../_images/examples_link_multiplex_if_11_1.png

AnnData Integration and Spatial Single Cell Analysis

Now let's explore the single-cell quantification of our imaging data. Our pipeline produced a single-cell matrix of shape (cell x protein) where each cell has attached additional information including location on the slide and the size of the cell in the image. This information is stored in slidedata.counts as an anndata object (https://anndata.readthedocs.io/en/latest/anndata.AnnData.html).

[8]:

```
adata = slidedata.counts.to_memory()
```

[9]:

```
adata
```

[9]:

```
AnnData object with n_obs × n_vars = 945 × 92
    obs: 'y', 'x', 'label', 'filled_area', 'euler_number'
    obsm: 'spatial'
    layers: 'min_intensity', 'max_intensity'
```

[10]:

```
adata.X
```

[10]:

```
array([[ 8.11290323, 10.98387097, 11.72580645, ...,  0.        ,
         0.        , 21.77419355],
       [10.15277778, 10.90277778, 11.33333333, ...,  0.        ,
         0.        , 23.61111111],
       [20.5       , 17.19642857, 18.51785714, ...,  0.        ,
         0.        , 24.25      ],
       ...,
       [ 9.68345324, 12.20143885, 12.55395683, ...,  0.        ,
         0.        , 12.07913669],
       [ 4.31666667,  8.48333333,  8.65      , ...,  0.        ,
         0.        ,  7.31666667],
       [10.06896552,  8.89655172,  8.79310345, ...,  0.        ,
         0.        ,  8.55172414]])
```

[11]:

```
adata.obs
```

[11]:

	y	x	label	filled_area	euler_number
0	2.048387	219.290323	1	62.0	1
1	2.666667	291.402778	2	72.0	1
2	2.535714	873.107143	3	56.0	1
3	3.729885	550.339080	4	174.0	1
4	3.242105	853.084211	5	95.0	1
...	...	...	...	...	...
940	994.813853	842.056277	941	231.0	1
941	995.741935	336.379032	942	124.0	1
942	995.589928	812.503597	943	139.0	1
943	997.083333	444.583333	944	60.0	1
944	997.517241	966.482759	945	29.0	1

945 rows × 5 columns

[12]:

```
adata.var
```

[12]:

0
1
2
3
4
...
87
88
89
90
91

92 rows × 0 columns

This anndata object gives us access to the entire python (or Seurat) single cell analysis ecosystem of tools. We follow a single cell analysis workflow described in https://scanpy-tutorials.readthedocs.io/en/latest/pbmc3k.html and https://www.embopress.org/doi/full/10.15252/msb.20188746.

[13]:

```
import scanpy as sc

sc.pl.violin(adata, keys=["0", "24", "60"])
sc.pp.log1p(adata)
sc.pp.scale(adata, max_value=10)
sc.tl.pca(adata, svd_solver="arpack")
sc.pp.neighbors(adata, n_neighbors=10, n_pcs=10)
sc.tl.umap(adata)
sc.pl.umap(adata, color=["0", "24", "60"])
```

../_images/examples_link_multiplex_if_19_0.png

../_images/examples_link_multiplex_if_19_1.png

[16]:

```
sc.tl.leiden(adata, resolution=0.15)
sc.pl.umap(adata, color="leiden")
sc.tl.rank_genes_groups(adata, "leiden", method="t-test")
sc.pl.rank_genes_groups_dotplot(adata, groupby="leiden", vmax=5, n_genes=5)
```

../_images/examples_link_multiplex_if_20_0.png

../_images/examples_link_multiplex_if_20_1.png

We can also use spatial analysis tools such as https://github.com/theislab/squidpy.

[19]:

```
import scanpy as sc
import squidpy as sq

sc.pl.spatial(adata, color="leiden", spot_size=15)
sc.pl.spatial(adata, color="leiden", groups=["2", "4"], spot_size=15)
```

[15]:

```
sq.gr.co_occurrence(adata, cluster_key="leiden")
sq.pl.co_occurrence(adata, cluster_key="leiden")
```

../_images/examples_link_multiplex_if_23_1.png

References

- Schürch, C.M., Bhate, S.S., Barlow, G.L., Phillips, D.J., Noti, L., Zlobec, I., Chu, P., Black, S., Demeter, J., McIlwain, D.R. and Samusik, N., 2020. Coordinated cellular neighborhoods orchestrate antitumoral immunity at the colorectal cancer invasive front. Cell, 182(5), pp.1341-1359.

Previous Next

Preprocessing: H&E Stain Normalization

View on GitHub

This notebook gives examples of the stain deconvolution and normalization tools available in `PathML`.

H&E images are the result of applying two stains to a tissue sample: hematoxylin and eosin. Hematoxylin binds to the cell nuclei and colors them purple, while eosin binds to the cytoplasm and extracellular matrix, coloring them pink. In this context, stain normalization refers to the process of standardizing or normalizing the appearance of tissue samples across different slides or imaging systems. Stain deconvolution is the process of untangling these two superimposed stains from an H&E image.

Digital pathology images can vary for many reasons, including:

- variation in stain intensity due to inconsistencies of technicians while applying stains to specimens
- variation in image qualities due to differences in slide scanners
- variation due to differences in lighting conditions when slide is scanned
- etc.

For these reasons, stain normalization is a crucial part of any computational pathology workflow.

Stain deconvolution can also be used in other ways, due to the different biological properties of the stains. For example, we can apply stain separation and use the hematoxylin channel as input to a nucleus detection algorithm (see nucleus detection example notebook).

`PathML` comes with two stain normalization algorithms out of the box: the Macenko and Vahadane methods (Macenko et al. 2009; Vahadane et al. 2016). As more stain deconvolution methods are incorporated into `PathML`, they will be added here.

[1]:

```
import numpy as np
import matplotlib.pyplot as plt
import os

from pathml.core import HESlide
from pathml.preprocessing import StainNormalizationHE
```

[2]:

```
fontsize = 20
```

Data

This example notebook uses publicly available images from:

1. OpenSlide: Download `CMU-1-Small-Region.svs.tiff` from here.
2. BRACS: Download `target.png` from here.

You can also change the filepaths to use any whole-slide images that you have locally.

We will pull out some tiles to use as an example. The **source** image is what you want to normalize, and the **target** is the high-quality image which provides the reference for normalizing. In many worflows, you have multiple **source** images that you want to normalize with the help of one representative **target** image.

[3]:

```
source_wsi = HESlide("../data/CMU-1-Small-Region.svs.tiff")
source_region = source_wsi.slide.extract_region(location=(0, 0), size=(2000, 2000))
source_region = np.squeeze(source_region)

target_wsi = HESlide("../data/target.png")
target_region = target_wsi.slide.extract_region(location=(0, 0), size=(500, 500))
target_region = np.squeeze(target_region)
```

[4]:

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 7.5))

ax[0].imshow(source_region)
ax[0].set_title("Source image", fontsize=fontsize)

ax[1].imshow(target_region)
ax[1].set_title("Target image", fontsize=fontsize)

for a in ax.ravel():
    a.set_xticks([])
    a.set_yticks([])

plt.tight_layout()
plt.show()
```

../_images/examples_link_stain_normalization_6_0.png

Stain normalization

[5]:

```
fig, axarr = plt.subplots(nrows=2, ncols=2, figsize=(10, 7.5))
axarr[0, 0].imshow(source_region)
axarr[0, 0].set_title("Source image", fontsize=fontsize)

axarr[0, 1].imshow(target_region)
axarr[0, 1].set_title("Target image", fontsize=fontsize)

for j, method in enumerate(["Macenko", "Vahadane"]):
    # initialize stain normalization object
    normalizer = StainNormalizationHE(
        target="normalize", stain_estimation_method=method
    )

    # fit to the target image
    normalizer.fit_to_reference(target_region)

    # apply on example source image
    im = normalizer.F(source_region)

    # plot results
    ax = axarr[1, j]
    ax.imshow(im)
    ax.set_title(f"{method} normalization", fontsize=fontsize)

for a in axarr.ravel():
    a.set_xticks([])
    a.set_yticks([])

plt.tight_layout()
plt.show()
```

../_images/examples_link_stain_normalization_8_0.png

Stain Deconvolution

[6]:

```
fig, axarr = plt.subplots(nrows=2, ncols=3, figsize=(10, 7.5))

for i, method in enumerate(["macenko", "vahadane"]):
    for j, target in enumerate(["normalize", "hematoxylin", "eosin"]):
        # initialize stain normalization object
        normalizer = StainNormalizationHE(target=target, stain_estimation_method=method)
        # apply on example image
        im = normalizer.F(source_region)

        # plot results
        ax = axarr[i, j]
        ax.imshow(im)
        if j == 0:
            ax.set_ylabel(f"{method} method", fontsize=fontsize)
        if i == 0:
            ax.set_title(target, fontsize=fontsize)
```

```
for a in axarr.ravel():
    a.set_xticks([])
    a.set_yticks([])

plt.tight_layout()
plt.show()
```

../_images/examples_link_stain_normalization_10_0.png

References

- Macenko, M., Niethammer, M., Marron, J.S., Borland, D., Woosley, J.T., Guan, X., Schmitt, C. and Thomas, N.E., 2009, June. A method for normalizing histology slides for quantitative analysis. In 2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro (pp. 1107-1110). IEEE.
- Vahadane, A., Peng, T., Sethi, A., Albarqouni, S., Wang, L., Baust, M., Steiger, K., Schlitter, A.M., Esposito, I. and Navab, N., 2016. Structure-preserving color normalization and sparse stain separation for histological images. IEEE transactions on medical imaging, 35(8), pp.1962-1971.

Previous Next

Contributing

Preprocessing: Tile Stitching

View on GitHub

In the rapidly evolving field of digital pathology, handling and processing high-resolution histopathology images is a critical task. This is especially true in the context of whole-slide imaging (WSI), a technique that has revolutionized the analysis of tissue samples by digitizing entire microscope slides at a gigapixel scale. However, the large size of these images presents a significant challenge in terms of data management and analysis. To address this, the images are often segmented into smaller, manageable, and overlapping segments known as tiles. The real challenge, and the focus of this tutorial, is in the accurate reconstruction of these tiles to reform the complete image—a process known as tile stitching.

PathML offers the TileStitcher class, which is the Python adaptation of an existing Groovy script used in QuPath which is available here. The TileStitcher class reimplements the functionality of its Groovy counterpart, allowing for the extraction of tile coordinates from the baseline TIFF tags followed by seamlessly stitching them and writing the stitched image as a pyramidal OME-TIFF file.

This tutorial will walk you through the process of using TileStitcher class to collect, parse, and stitch large sets of tiled TIFF images then saving the reconstructed image.

Prerequisites

Before using the `TileStitcher` class, we need to install the necessary software and configure the environment.

Software Installation

The `TileStitcher` class requires QuPath and OpenJDK. Here is how to install them:

1. Download and install QuPath from its GitHub release page. Here we are using version 0.3.1.

```
wget https://github.com/qupath/qupath/releases/download/v0.4.3/QuPath-0.4.3-Linux.tar.xz
```

Unzip

```
tar -xvf QuPath-0.4.3-Linux.tar.xz
```

Make executable

```
chmod u+x /path/to/QuPath/bin/QuPath
```

2. Download and Install OpenJDK 17

```bash wget https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.deb sudo apt install ./jdk-17_linux-x64_bin.deb

Set the Java path according to your installation method. If you have set up your environment using PathML, set the Java path to `/opt/conda/envs/pathml`. Otherwise, adjust it to the appropriate path on your system.

### Environment Configuration

To use `TileStitcher`, we need to set the correct paths to the QuPath library and OpenJDK. For this, we need to add the paths to the environment variables `JAVA_HOME`, `CLASSPATH`, and `LD_LIBRARY_PATH`.

The `JAVA_HOME` environment variable should be set to the path where the JDK is installed. The `CLASSPATH` environment variable should include paths to all the QuPath library jar files. In the initialization of TileStitcher, these environment variables are used to start the Java Virtual Machine (JVM) and import the necessary QuPath classes.
```

Best Practices and Considerations for Using the TileStitcher Module

1. JVM Session Management

The TileStitcher module utilizes jpype to manage the JVM sessions, a departure from the python-javabridge used in other parts of the package. This difference can cause conflicts when trying to run modules concurrently within the same Python environment. Hence, it is advisable to avoid running TileStitcher operations in the same notebook where python-javabridge dependent modules are running.

2. Restarting Kernel to Re-initialize JVM

The jpype does not allow the JVM to be restarted within the same Python session once it has been terminated. As a result, to run the TileStitcher module again or to switch back to modules that use python-javabridge, a kernel restart might be necessary.

3. Segregating Workflows

To mitigate potential conflicts, consider segregating workflows based on the JVM management package they depend on. It is recommended to use separate notebooks or scripts for operations involving TileStitcher and for those involving modules that are dependent on python-javabridge.

Using TileStitcher

Ensure QuPath and JDK installations are complete before proceeding.

Initialization

The class is initialized with several parameters:

- `qupath_jarpath`: List of paths to QuPath JAR files.
- `java_path`: Custom path to Java installation. If set, `JAVA_HOME` will be overridden.
- `memory`: Allocated memory for the JVM (default: "40g").
- `bfconvert_dir`: Directory for Bio-Formats conversion tools.

During initialization, `TileStitcher` sets up the Java Virtual Machine (JVM) and imports necessary QuPath classes. It also includes error handling for Java path configurations and JVM startup issues.

JVM Startup

The `_start_jvm` method initiates the JVM with specified memory and classpath settings. It imports necessary QuPath classes upon successful startup, ensuring compatibility with Java 17.

[12]:

```
import glob
import os
from pathml.preprocessing.tilestitcher import TileStitcher
from pathml.utils import setup_qupath


# Set the path to the JDK
os.environ["JAVA_HOME"] = "/usr/lib/jvm/jdk-17/"

# Use setup_qupath to get the QuPath installation path
qupath_home = setup_qupath("../../tools1/tools1/")

if qupath_home is not None:
    os.environ["QUPATH_HOME"] = qupath_home

    # Construct the path to QuPath jars based on qupath_home
    qupath_jars_dir = os.path.join(qupath_home, "lib", "app")
    qupath_jars = glob.glob(os.path.join(qupath_jars_dir, "*.jar"))
    qupath_jars.append(os.path.join(qupath_jars_dir, "libopenslide-jni.so"))

    # Create an instance of TileStitcher
    stitcher = TileStitcher(qupath_jars)
else:
    print("QuPath installation not found. Please check the installation path.")
```

```
./tools/bftools/bfconvert ./tools/bftools/bf.sh
bfconvert version: Version: 7.0.1
Build date: 16 October 2023
VCS revision: 20e58cef1802770cc56ecaf1ef6f323680e4cf65
Setting Environment Paths
Java Home is already set
JVM was already started
```

[13]:

```
import jpype
```

[14]:

```
jpype.isJVMStarted()
```

[14]:

```
False
```

Image Stitching with TileStitcher

Once `TileStitcher` is initialized, it's capable of stitching together tiled images.

- Method: `run_image_stitching`
- Inputs:
  - A list of TIFF files or a directory containing TIFF files.
  - Output file path.
- Optional Parameters:
  - `downsamples`: Specify the number of downsample levels (e.g., `[1,4,8]`). Defaults to levels read from the tiles.
  - `separate_series`: If set to `True`, it downloads bftools and extracts the base level image from the stitched image.

[4]:

```
input_files = glob.glob("path/to/tiles/*.tif")
output_file = "path/to/output.ome.tif"
stitcher.run_image_stitching(input_files, output_file)
```

Demo

[1]:

```
import jpype
```

[2]:

```
jpype.isJVMStarted(), jpype.getJVMVersion()
```

[2]:

```
(False, (0, 0, 0))
```

[3]:

```
import glob
import os

# Set the path to the JDK
os.environ["JAVA_HOME"] = "/opt/conda/envs/pathml"
os.environ["PATH"] += os.pathsep + os.path.join("/opt/conda/envs/pathml", "bin")
```

[4]:

```
from pathml.preprocessing.tilestitcher import TileStitcher
from pathml.utils import setup_qupath


# Use setup_qupath to get the QuPath installation path
qupath_home = setup_qupath("./tools/")

if qupath_home is not None:
    os.environ["QUPATH_HOME"] = qupath_home

    # Construct the path to QuPath jars based on qupath_home
    qupath_jars_dir = os.path.join(qupath_home, "lib", "app")
    qupath_jars = glob.glob(os.path.join(qupath_jars_dir, "*.jar"))
    qupath_jars.append(os.path.join(qupath_jars_dir, "libopenslide-jni.so"))

    # Create an instance of TileStitcher
    stitcher = TileStitcher(qupath_jars)
else:
    print("QuPath installation not found. Please check the installation path.")
```

```
Using JVM version: (17, 0, 10) from /opt/conda/envs/pathml/lib/jvm/lib/server/libjvm.so
Importing required QuPath classes
```

[5]:

```
jpype.isJVMStarted(), jpype.getJVMVersion()
```

[5]:

```
(True, (17, 0, 10))
```

[6]:

```
# Specify the folder path where the list of .tif files are present, here we are using a folder path that has single tif file for demo purposes.
infile_path = "../tests/testdata/tilestitching_testdata/"
outfile_path = "./output/tile_stitching_demo.ome.tif"
```

[7]:

```
import time

start = time.time()
# Run the image stitching process
stitcher.run_image_stitching(
    infile_path, outfile_path, downsamples=[1], separate_series=True
)
end = time.time()
```

```
19:07:21.270 [main] [INFO ] q.l.i.s.b.BioFormatsServerOptions - Setting max Bio-Formats readers to 8
19:07:21.900 [main] [ERROR] q.l.i.s.o.OpenslideServerBuilder - Could not load OpenSlide native libraries
java.lang.UnsatisfiedLinkError: no openslide-jni in java.library.path: /opt/conda/envs/pathml/lib/python3.9/site-packages/cv2/../../lib64:/usr/loc
        at java.base/java.lang.ClassLoader.loadLibrary(ClassLoader.java:2434)
        at java.base/java.lang.Runtime.loadLibrary0(Runtime.java:818)
        at java.base/java.lang.System.loadLibrary(System.java:1993)
        at org.openslide.OpenSlideJNI.<clinit>(OpenSlideJNI.java:55)
        at org.openslide.OpenSlide.<clinit>(OpenSlide.java:53)
        at qupath.lib.images.servers.openslide.OpenslideServerBuilder.<clinit>(OpenslideServerBuilder.java:90)
        at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:77)
        at java.base/jdk.internal.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
        at java.base/java.lang.reflect.Constructor.newInstanceWithCaller(Constructor.java:499)
        at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:480)
        at java.base/java.util.ServiceLoader$ProviderImpl.newInstance(ServiceLoader.java:789)
        at java.base/java.util.ServiceLoader$ProviderImpl.get(ServiceLoader.java:729)
        at java.base/java.util.ServiceLoader$3.next(ServiceLoader.java:1403)
        at qupath.lib.images.servers.ImageServerProvider.getServerBuilders(ImageServerProvider.java:191)
        at qupath.lib.images.servers.ImageServerProvider.getPreferredUriImageSupport(ImageServerProvider.java:223)
19:07:21.901 [main] [INFO ] q.l.i.s.o.OpenslideServerBuilder - If you want to use OpenSlide, you'll need to get the native libraries (either build
and add them to your system PATH, including openslide-jni.
19:07:24.717 [main] [WARN ] q.l.i.writers.ome.OMEPyramidWriter - Deleting existing file /home/jupyter/sreekar/projects/tilestitching/pathml/exampl
19:07:24.733 [main] [INFO ] q.l.i.writers.ome.OMEPyramidWriter - Writing Sparse image (1 regions) to /home/jupyter/sreekar/projects/tilestitching/
19:07:24.734 [main] [INFO ] q.l.i.writers.ome.OMEPyramidWriter - Setting series 0 compression to zlib
19:07:24.734 [main] [INFO ] q.l.i.writers.ome.OMEPyramidWriter - Writing resolution 1 of 1 (downsample=1.0, 12 tiles)
19:07:24.736 [main] [INFO ] q.l.i.writers.ome.OMEPyramidWriter - Writing plane 1/1
19:07:35.528 [main] [INFO ] q.l.i.writers.ome.OMEPyramidWriter - Plane written in 10792 ms
Image stitching completed. Output file: ./output/tile_stitching_demo.ome.tif
bfconvert version: Version: 7.1.0
Build date: 11 December 2023
VCS revision: 05c7b2413cfad19a73b619c61ddf77ca2d038ce7
./output/tile_stitching_demo.ome.tif
OMETiffReader initializing ./output/tile_stitching_demo.ome.tif
[OME-TIFF] -> ./output/tile_stitching_demo_separated.ome.tif [OME-TIFF]
Reading IFDs
Populating metadata
Reading IFDs
Populating metadata
        Converted 1/7 planes (14%)
        Converted 7/7 planes (100%)
Overwriting existing Creator attribute: OME Bio-Formats 6.12.0
[done]
2.023s elapsed (162.28572+47.857143ms per plane, 507ms overhead)
bfconvert completed. Output file: ./output/tile_stitching_demo_separated.ome.tif
Original stitched image deleted: ./output/tile_stitching_demo.ome.tif
```

Previous Next

Machine Learning: Training a HACTNet model

View on GitHub

In this notebook, we will train the HACTNet graph neural network (GNN) model on input cell and tissue graphs using the new `pathml.graph` API.

To run the notebook and train the model, you will have to first download the BRACS Regions of Interest (ROI) set from the BRACS dataset. To do so, you will have to sign up and create an account. Next, you will have to construct the cell and tissue graphs using the tutorial in `examples/construct_graphs.ipynb`. Use the output directory specified there as the input to the main function in this tutorial.

NOTE: The actual HACTNet model uses HoVer-Net, an ML model, to detect cells. In `examples/construct_graphs.ipynb`, we used a manual method for simplicity. Hence the performance of the model trained in this notebook will be lesser.

[22]:

```
import os
from glob import glob
import argparse
from PIL import Image
import numpy as np
from tqdm import tqdm
import torch
import h5py
import warnings
import math
from skimage.measure import regionprops, label
import networkx as nx
import traceback
from glob import glob
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch_geometric.data import Batch
from torch_geometric.data import Data
from torch.utils.data import Dataset
from torch_geometric.loader import DataLoader
from torch.optim.lr_scheduler import StepLR
from sklearn.metrics import f1_score

from pathml.core import HESlide
from pathml.datasets import EntityDataset
from pathml.ml.utils import get_degree_histogram, get_class_weights
from pathml.ml import HACTNet

# If using GPU
device = "cuda"

# If using CPU
# device = "cpu"
```

Data visualization

First, let us take a look at the inputs to our model. The dataset comrpises of approximately 3600 training ROIs, 310 validation ROIs and 560 testing ROIs. In each of these, the ROIs can belong to one out of seven possible labels, that correspond to breast cancer subtypes. Refer to Brancati et al., 2022 for more information about the dataset. Our task is to train a model that can classify the given ROI to the correct cancer subtype.

We will now visualize one ROI from each of these seven subtypes to see the similarities and differences between them.

[21]:

```
# PATH to the train split of the BRACS dataset
base_path = '../../data/BRACS_RoI/latest_version/train/'

# We manually choose a random ROI to visualize along with information about its corresponding label
image_info = [('0_N/BRACS_1231_N_27.png','Normal'),
              ('1_PB/BRACS_1003671_PB_1.png', 'Benign'),
              ('2_UDH/BRACS_1003707_UDH_1.png', 'Usual Ductal Hyperplasia'),
              ('3_FEA/BRACS_1003693_FEA_1.png', 'Flat Epithelial Atypia'),
              ('4_ADH/BRACS_1003728_ADH_1.png', 'Atypical Ductal Hyperplasia'),
              ('5_DCIS/BRACS_1003697_DCIS_1.png', 'Ductal Carcinoma in Situ'),
              ('6_IC/BRACS_1003699_IC_1.png', 'Invasive Carcinoma')]

# Plot the figure
fig, axarr = plt.subplots(nrows=7, ncols=2, figsize=(7.5, 15))

for i, (image_path, label) in enumerate(image_info):
    wsi = HESlide(base_path + image_path)
    region1 = wsi.slide.extract_region(location=(0, 0), size=(500, 500))
    region2 = wsi.slide.extract_region(location=(500, 500), size=(500, 500))

    axarr[i,0].imshow(np.squeeze(region1))
    axarr[i,1].imshow(np.squeeze(region2))

    axarr[i,0].set_ylabel(label, fontsize=10)

for a in axarr.ravel():
    a.set_xticks([])
    a.set_yticks([])

plt.tight_layout()
plt.show()
```

../_images/examples_link_train_hactnet_4_0.png

Model Training

Now that we know the input dataset and the objective, we can proceed to training the model. The model takes constructed graphs as input, so make sure you have run `examples/construct_graphs.ipynb`.

We can define the main training loop for loading the constructed graphs, initializing the model and training.

[12]:

```
def train_hactnet(
    train_dataset,
    val_dataset,
    test_dataset,
    batch_size=4,
    load_histogram=True,
    histogram_dir=None,
    calc_class_weights=True,
):

    # Print the lengths of each dataset split
    print(f"Length of training dataset: {len(train_dataset)}")
    print(f"Length of validation dataset: {len(val_dataset)}")
    print(f"Length of test dataset: {len(test_dataset)}")

    # Define the torch_geometric.DataLoader object for each dataset split
    train_batch = DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        follow_batch=["x_cell", "x_tissue"],
        drop_last=True,
    )
    val_batch = DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        follow_batch=["x_cell", "x_tissue"],
        drop_last=True,
    )
    test_batch = DataLoader(
        test_dataset,
        batch_size=batch_size,
        shuffle=False,
        follow_batch=["x_cell", "x_tissue"],
        drop_last=True,
    )

    # The GNN layer we use in this model, PNAConv, requires the computation of a node degree histogram of the
    # train dataset. We only need to compute it once. If it is precomputed already, set the load_histogram=True.
    # Else, the degree histogram is calculated and saved.
    if load_histogram:
        histogram_dir = "./"
        cell_deg = torch.load(os.path.join(histogram_dir, "cell_degree_norm.pt"))
        tissue_deg = torch.load(os.path.join(histogram_dir, "tissue_degree_norm.pt"))
    else:
        train_batch_hist = DataLoader(
            train_dataset,
            batch_size=20,
            shuffle=True,
            follow_batch=["x_cell", "x_tissue"],
        )
        print("Calculating degree histogram for cell graph")
        cell_deg = get_degree_histogram(train_batch_hist, "edge_index_cell", "x_cell")
        print("Calculating degree histogram for tissue graph")
        tissue_deg = get_degree_histogram(
            train_batch_hist, "edge_index_tissue", "x_tissue"
        )
        torch.save(cell_deg, "cell_degree_norm.pt")
        torch.save(tissue_deg, "tissue_degree_norm.pt")

    # Since the BRACS dataset has unbalanced data, it is important to calculate the class weights in the training set
    # and provide that as an argument to our loss function.
    if calc_class_weights:
        train_w = get_class_weights(train_batch)
        torch.save(torch.tensor(train_w), "loss_weights_norm.pt")

    # Here we define the keyword arguments for the PNAConv layer in the model for both cell and tissue processing
    # layers.
    kwargs_pna_cell = {
        "aggregators": ["mean", "max", "min", "std"],
        "scalers": ["identity", "amplification", "attenuation"],
        "deg": cell_deg,
    }
    kwargs_pna_tissue = {
        "aggregators": ["mean", "max", "min", "std"],
        "scalers": ["identity", "amplification", "attenuation"],
        "deg": tissue_deg,
    }
```

```
cell_params = {
    "layer": "PNAConv",
    "in_channels": 514,
    "hidden_channels": 64,
    "num_layers": 3,
    "out_channels": 64,
    "readout_op": "lstm",
    "readout_type": "mean",
    "kwargs": kwargs_pna_cell,
}

tissue_params = {
    "layer": "PNAConv",
    "in_channels": 514,
    "hidden_channels": 64,
    "num_layers": 3,
    "out_channels": 64,
    "readout_op": "lstm",
    "readout_type": "mean",
    "kwargs": kwargs_pna_tissue,
}

classifier_params = {
    "in_channels": 128,
    "hidden_channels": 128,
    "out_channels": 7,
    "num_layers": 2,
}

# Initialize the pathml.ml.HACTNet model
model = HACTNet(cell_params, tissue_params, classifier_params)

# Set up optimizer
opt = torch.optim.Adam(model.parameters(), lr=0.0005)

# Learning rate scheduler to reduce LR by factor of 10 each 25 epochs
scheduler = StepLR(opt, step_size=25, gamma=0.1)

# Send the model to GPU
model = model.to(device)

# Define number of epochs
n_epochs = 60

# Keep a track of best epoch and metric for saving only the best models
best_epoch = 0
best_metric = 0

# Load the computed class weights if calc_class_weights = True
if calc_class_weights:
    loss_weights = torch.load("loss_weights.pt")

# Define the loss function
loss_fn = nn.CrossEntropyLoss(
    weight=loss_weights.float().to(device) if calc_class_weights else None
)

# Define the evaluate function to compute metrics for validation and test set to keep track of performance.
# The metrics used are per-class and weighted F1 score.
def evaluate(data_loader):
    model.eval()
    y_true = []
    y_pred = []
    with torch.no_grad():
        for data in tqdm(data_loader):
            data = data.to(device)
            outputs = model(data)
            y_true.append(
                torch.argmax(outputs.detach().cpu().softmax(dim=1), dim=-1).numpy()
            )
            y_pred.append(data.target.cpu().numpy())
        y_true = np.array(y_true).ravel()
        y_pred = np.array(y_pred).ravel()
        per_class = f1_score(y_true, y_pred, average=None)
        weighted = f1_score(y_true, y_pred, average="weighted")
    print(f"Per class F1: {per_class}")
```

```
        print(f"Weighted F1: {weighted}")
        return np.append(per_class, weighted)

    # Start the training loop
    for i in range(n_epochs):
        print(f"\n>>>>>>>>>>>>>>>>Epoch number {i}>>>>>>>>>>>>>>>>")
        minibatch_train_losses = []

        # Put model in training mode
        model.train()

        print("Training")

        for data in tqdm(train_batch):

            # Step optimizer and scheduler
            opt.step()

            # Send the data to the GPU
            data = data.to(device)

            # Zero out gradient
            opt.zero_grad()

            # Forward pass
            outputs = model(data)

            # Compute loss
            loss = loss_fn(outputs, data.target)

            # Compute gradients
            loss.backward()

            # Track loss
            minibatch_train_losses.append(loss.detach().cpu().numpy())

        print(f"Loss: {np.array(minibatch_train_losses).ravel().mean()}")

        # Print performance metrics on validation set
        print("\nEvaluating on validation")
        val_metrics = evaluate(val_batch)

        # Save the model only if it is better than previous checkpoint in validation metrics
        if val_metrics[-1] > best_metric:
            print("Saving checkpoint")
            torch.save(model.state_dict(), "hact_net.pt")
            best_metric = val_metrics[-1]

        # Print performance metrics on test set
        print("\nEvaluating on test")
        _ = evaluate(test_batch)

        # Step LR scheduler
        scheduler.step()
```

[13]:

```
# Read the train, validation and test dataset into the pathml.datasets.EntityDataset class
root_dir = "./data/BRACS_RoI/latest_version/output/"
train_dataset = EntityDataset(
    os.path.join(root_dir, "cell_graphs/train/"),
    os.path.join(root_dir, "tissue_graphs/train/"),
    os.path.join(root_dir, "assignment_matrices/train/"),
)
val_dataset = EntityDataset(
    os.path.join(root_dir, "cell_graphs/val/"),
    os.path.join(root_dir, "tissue_graphs/val/"),
    os.path.join(root_dir, "assignment_matrices/val/"),
)
test_dataset = EntityDataset(
    os.path.join(root_dir, "cell_graphs/test/"),
    os.path.join(root_dir, "tissue_graphs/test/"),
    os.path.join(root_dir, "assignment_matrices/test/"),
)
```

[14]:

```
train_hactnet(
    train_dataset,
    val_dataset,
    test_dataset,
    batch_size=8,
    load_histogram=True,
    calc_class_weights=False,
)
```

```
Length of training dataset: 3627
Length of validation dataset: 311
Length of test dataset: 563

>>>>>>>>>>>>>>>>Epoch number 0>>>>>>>>>>>>>>>>
Training

100%|██████████████████████████████████████████████████| 453/453 [16:33<00:00,  2.19s/it]

Loss: 1.681248664855957

Evaluating on validation

100%|██████████████████████████████████████████████████| 38/38 [01:19<00:00,  2.10s/it]

Per class F1: [0.14285714 0.23404255 0.         0.         0.         0.27802691
 0.71287129]
Weighted F1: 0.34292555902950034
Saving checkpoint

Evaluating on test

100%|██████████████████████████████████████████████████| 70/70 [01:44<00:00,  1.49s/it]

Per class F1: [0.29906542 0.34177215 0.         0.         0.         0.30291262
 0.32323232]
Weighted F1: 0.30912310035688134
```

```
[ ]:
```

After training the model for 40-50 epochs, you should see performance similar to the table below, depending on the dataset version you used.

Dataset	Weighted F-1 score
BRACS (Previous version)	60.14
BRACS (Latest Version)	55.96

References

- Pati, Pushpak, Guillaume Jaume, Antonio Foncubierta-Rodriguez, Florinda Feroce, Anna Maria Anniciello, Giosue Scognamiglio, Nadia Brancati et al. "Hierarchical graph representations in digital pathology." Medical image analysis 75 (2022): 102264.
- Brancati, Nadia, Anna Maria Anniciello, Pushpak Pati, Daniel Riccio, Giosuè Scognamiglio, Guillaume Jaume, Giuseppe De Pietro et al. "Bracs: A dataset for breast carcinoma subtyping in h&e histology images." Database 2022 (2022): baac093.

Session info

```
[15]:
```

```
import IPython

print(IPython.sys_info())
print(f"torch version: {torch.__version__}")
```

```
{'commit_hash': '8b1204b6c',
 'commit_source': 'installation',
 'default_encoding': 'utf-8',
 'ipython_path': '/home/jupyter/miniforge3/envs/pathml_cuda/lib/python3.10/site-packages/IPython',
 'ipython_version': '8.21.0',
 'os_name': 'posix',
 'platform': 'Linux-4.19.0-26-cloud-amd64-x86_64-with-glibc2.28',
 'sys_executable': '/home/jupyter/miniforge3/envs/pathml_cuda/bin/python',
 'sys_platform': 'linux',
 'sys_version': '3.10.13 | packaged by conda-forge | (main, Dec 23 2023, '
                '15:36:39) [GCC 12.3.0]'}
torch version: 1.13.1+cu116
```

Previous Next

Machine Learning: Training a HoVer-Net model

View on GitHub

In this notebook, we will train HoVer-Net model to perform nucleus detection and classification, using data from PanNuke dataset.

This notebook should be a good reference for how to do a full machine learning workflow using `PathML` and `PyTorch`

[1]:

```
import numpy as np
from tqdm import tqdm
import copy
import matplotlib.pyplot as plt
from matplotlib import cm
import torch
from torch.optim.lr_scheduler import StepLR
import albumentations as A
```

[2]:

```
from pathml.datasets.pannuke import PanNukeDataModule
from pathml.ml.hovernet import HoVerNet, loss_hovernet, post_process_batch_hovernet
from pathml.ml.utils import wrap_transform_multichannel, dice_score
from pathml.utils import plot_segmentation
```

Data augmentation

Data augmentation is the process of applying random transformations to the data before feeding it to the network. This introduces some noise and can help improve model performance by reducing overfitting. For example, each image can be randomly rotated by 90 degrees - the idea is that this would force the network to learn representations which are robust to rotation.

Importantly, whatever transform is applied to the image also needs to be applied to the corresponding mask!

We'll use the Albumentations library to handle data augmentation. You can also write custom data augmentations, but albumentations and other similar libraries (e.g. torchvision.transforms) are convenient because they automatically handle masks in the augmentation pipeline.

However, because our masks have multiple channels, they are not natively supported by Albumentations. So we'll wrap each transform in the `wrap_transform_multichannel()` utility function which will make it compatible.

[3]:

```
n_classes_pannuke = 6

# data augmentation transform
hover_transform = A.Compose(
    [
        A.VerticalFlip(p=0.5),
        A.HorizontalFlip(p=0.5),
        A.RandomRotate90(p=0.5),
        A.GaussianBlur(p=0.5),
        A.MedianBlur(p=0.5, blur_limit=5),
    ],
    additional_targets={f"mask{i}": "mask" for i in range(n_classes_pannuke)},
)

transform = wrap_transform_multichannel(hover_transform)
```

Load PanNuke dataset

[4]:

```
pannuke = PanNukeDataModule(
    data_dir="../data/pannuke/",
    download=False,
    nucleus_type_labels=True,
    batch_size=8,
    hovernet_preprocess=True,
    split=1,
    transforms=transform,
)

train_dataloader = pannuke.train_dataloader
valid_dataloader = pannuke.valid_dataloader
test_dataloader = pannuke.test_dataloader
```

Let's visualize what the inputs to HoVer-Net model look like:

[7]:

```
images, masks, hvs, types = next(iter(train_dataloader))

n = 4
fig, ax = plt.subplots(nrows=n, ncols=4, figsize=(8, 8))

cm_mask = copy.copy(cm.get_cmap("tab10"))
cm_mask.set_bad(color="white")

for i in range(n):
    im = images[i, ...].numpy()
    ax[i, 0].imshow(np.moveaxis(im, 0, 2))
    m = masks.argmax(dim=1)[i, ...]
    m = np.ma.masked_where(m == 5, m)
    ax[i, 1].imshow(m, cmap=cm_mask)
    ax[i, 2].imshow(hvs[i, 0, ...], cmap="coolwarm")
    ax[i, 3].imshow(hvs[i, 1, ...], cmap="coolwarm")

for a in ax.ravel():
    a.axis("off")
for c, v in enumerate(["H&E Image", "Nucleus Types", "Horizontal Map", "Vertical Map"]):
    ax[0, c].set_title(v)

plt.tight_layout()
plt.show()
```

../_images/examples_link_train_hovernet_8_0.png

Model Training

Now we are ready to train the HoVer-Net model.
To train a model in PyTorch, we need to write a training loop ourselves to specify exactly how we want to train the model. This gives us precise control over exactly what we are doing, at the expense of somewhat verbose code. This tutorial is a good reference.

Training with multi-GPU

When using GPUs with PyTorch, there are a few things to keep in mind when writing the training loop. For example, we need to explicitly move data to the GPU by calling `.to(device)`.
For multi-GPU, we also need to wrap our model object with `torch.nn.DataParallel()`. PyTorch will then take care of all the tricky parts of distributing the computation across the GPUs.

[5]:

```
print(f"GPUs used:\t{torch.cuda.device_count()}")
device = torch.device("cuda:0")
print(f"Device:\t\t{device}")
```

```
GPUs used:      4
Device:         cuda:0
```

[6]:

```
n_classes_pannuke = 6

# load the model
```

```
hovernet = HoVerNet(n_classes=n_classes_pannuke)

# wrap model to use multi-GPU
hovernet = torch.nn.DataParallel(hovernet)
```

[7]:

```
# set up optimizer
opt = torch.optim.Adam(hovernet.parameters(), lr=1e-4)
# learning rate scheduler to reduce LR by factor of 10 each 25 epochs
scheduler = StepLR(opt, step_size=25, gamma=0.1)
```

[8]:

```
# send model to GPU
hovernet.to(device);
```

Main training loop

This contains all our logic for looping over batches, doing a forward pass through the network, computing the loss, and then stepping the model parameters to minimize the loss. We also add some code to evaluate the model on the validation set as we train, and to track the performance metrics throughout the training process.

[ ]:

```
n_epochs = 50

# print performance metrics every n epochs
print_every_n_epochs = None

# evaluating performance on a random subset of validation mini-batches
# this saves time instead of evaluating on the entire validation set
n_minibatch_valid = 50

epoch_train_losses = {}
epoch_valid_losses = {}
epoch_train_dice = {}
epoch_valid_dice = {}

best_epoch = 0

# main training loop
for i in tqdm(range(n_epochs)):
    minibatch_train_losses = []
    minibatch_train_dice = []

    # put model in training mode
    hovernet.train()

    for data in train_dataloader:
        # send the data to the GPU
        images = data[0].float().to(device)
        masks = data[1].to(device)
        hv = data[2].float().to(device)
        tissue_type = data[3]

        # zero out gradient
        opt.zero_grad()

        # forward pass
        outputs = hovernet(images)

        # compute loss
        loss = loss_hovernet(outputs=outputs, ground_truth=[masks, hv], n_classes=6)

        # track loss
        minibatch_train_losses.append(loss.item())

        # also track dice score to measure performance
        preds_detection, preds_classification = post_process_batch_hovernet(
            outputs, n_classes=n_classes_pannuke
        )
        truth_binary = masks[:, -1, :, :] == 0
        dice = dice_score(preds_detection, truth_binary.cpu().numpy())
        minibatch_train_dice.append(dice)

        # compute gradients
        loss.backward()

        # step optimizer and scheduler
        opt.step()

    # step LR scheduler
    scheduler.step()

    # evaluate on random subset of validation data
    hovernet.eval()
    minibatch_valid_losses = []
    minibatch_valid_dice = []
    # randomly choose minibatches for evaluating
    minibatch_ix = np.random.choice(
        range(len(valid_dataloader)), replace=False, size=n_minibatch_valid
    )
    with torch.no_grad():
        for j, data in enumerate(valid_dataloader):
            if j in minibatch_ix:
                # send the data to the GPU
                images = data[0].float().to(device)
                masks = data[1].to(device)
                hv = data[2].float().to(device)
                tissue_type = data[3]

                # forward pass
                outputs = hovernet(images)

                # compute loss
                loss = loss_hovernet(
                    outputs=outputs, ground_truth=[masks, hv], n_classes=6
                )

                # track loss
                minibatch_valid_losses.append(loss.item())

                # also track dice score to measure performance
                preds_detection, preds_classification = post_process_batch_hovernet(
                    outputs, n_classes=n_classes_pannuke
                )
                truth_binary = masks[:, -1, :, :] == 0
                dice = dice_score(preds_detection, truth_binary.cpu().numpy())
                minibatch_valid_dice.append(dice)

    # average performance metrics over minibatches
    mean_train_loss = np.mean(minibatch_train_losses)
    mean_valid_loss = np.mean(minibatch_valid_losses)
    mean_train_dice = np.mean(minibatch_train_dice)
    mean_valid_dice = np.mean(minibatch_valid_dice)

    # save the model with best performance
    if i != 0:
        if mean_valid_loss < min(epoch_valid_losses.values()):
            best_epoch = i
            torch.save(hovernet.state_dict(), f"hovernet_best_perf.pt")

    # track performance over training epochs
    epoch_train_losses.update({i: mean_train_loss})
    epoch_valid_losses.update({i: mean_valid_loss})
    epoch_train_dice.update({i: mean_train_dice})
    epoch_valid_dice.update({i: mean_valid_dice})

    if print_every_n_epochs is not None:
        if i % print_every_n_epochs == print_every_n_epochs - 1:
            print(f"Epoch {i+1}/{n_epochs}:")
            print(
                f"\ttraining loss: {np.round(mean_train_loss, 4)}\tvalidation loss: {np.round(mean_valid_loss, 4)}"
            )
            print(
                f"\ttraining dice: {np.round(mean_train_dice, 4)}\tvalidation dice: {np.round(mean_valid_dice, 4)}"
            )

# save fully trained model
torch.save(hovernet.state_dict(), f"hovernet_fully_trained.pt")
print(f"\nEpoch with best validation performance: {best_epoch}")
```

```
 36%|███▌      | 18/50 [4:22:48<7:46:23, 874.50s/it]
```

[23]:

```
fix, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ax[0].plot(epoch_train_losses.keys(), epoch_train_losses.values(), label="Train")
ax[0].plot(epoch_valid_losses.keys(), epoch_valid_losses.values(), label="Validation")
ax[0].scatter(
    x=best_epoch,
    y=epoch_valid_losses[best_epoch],
    label="Best Model",
    color="green",
    marker="*",
)
ax[0].set_title("Training: Loss")
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Loss")
ax[0].legend()

ax[1].plot(epoch_train_dice.keys(), epoch_train_dice.values(), label="Train")
ax[1].plot(epoch_valid_dice.keys(), epoch_valid_dice.values(), label="Validation")
ax[1].scatter(
    x=best_epoch,
    y=epoch_valid_dice[best_epoch],
    label="Best Model",
    color="green",
    marker="*",
)
ax[1].set_title("Training: Dice Score")
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Dice Score")
ax[1].legend()
plt.show()
```

../_images/examples_link_train_hovernet_17_0.png

Evaluate Model

Now that we have trained the model, we can evaluate performance on the held-out test set.

First we load the weights for the best model:

[30]:

```
# load the best model
checkpoint = torch.load("hovernet_best_perf.pt")
hovernet.load_state_dict(checkpoint)
```

[30]:

```
<All keys matched successfully>
```

Next, we loop through the test set and store the model predictions:

[69]:

```
hovernet.eval()

ims = None
```

```
mask_truth = None
mask_pred = None
tissue_types = []

with torch.no_grad():
    for i, data in tqdm(enumerate(test_dataloader)):
        # send the data to the GPU
        images = data[0].float().to(device)
        masks = data[1].to(device)
        hv = data[2].float().to(device)
        tissue_type = data[3]

        # pass thru network to get predictions
        outputs = hovernet(images)
        preds_detection, preds_classification = post_process_batch_hovernet(
            outputs, n_classes=n_classes_pannuke
        )

        if i == 0:
            ims = data[0].numpy()
            mask_truth = data[1].numpy()
            mask_pred = preds_classification
            tissue_types.extend(tissue_type)
        else:
            ims = np.concatenate([ims, data[0].numpy()], axis=0)
            mask_truth = np.concatenate([mask_truth, data[1].numpy()], axis=0)
            mask_pred = np.concatenate([mask_pred, preds_classification], axis=0)
            tissue_types.extend(tissue_type)
```

```
341it [16:56,  2.98s/it]
```

Now we can compute the Dice score for each image in the test set:

[70]:

```
# collapse multi-class preds into binary preds
preds_detection = np.sum(mask_pred, axis=1)

dice_scores = np.empty(shape=len(tissue_types))

for i in range(len(tissue_types)):
    truth_binary = mask_truth[i, -1, :, :] == 0
    preds_binary = preds_detection[i, ...] != 0
    dice = dice_score(preds_binary, truth_binary)
    dice_scores[i] = dice
```

[124]:

```
dice_by_tissue = pd.DataFrame({"Tissue Type": tissue_types, "dice": dice_scores})
dice_by_tissue.groupby("Tissue Type").mean().plot.bar()
plt.title("Dice Score by Tissue Type")
plt.ylabel("Averagae Dice Score")
plt.gca().get_legend().remove()
plt.show()
```

../_images/examples_link_train_hovernet_25_0.png

[72]:

```
print(f"Average Dice score in test set: {np.mean(dice_scores)}")
```

```
Average Dice score in test set: 0.7850396088887557
```

Examples

Let's take a look at some example predictions from the network to see how it is performing.

[100]:

```
# change image tensor from (B, C, H, W) to (B, H, W, C)
# matplotlib likes channels in last dimension
ims = np.moveaxis(ims, 1, 3)
```

[119]:

```
n = 8
ix = np.random.choice(np.arange(len(tissue_types)), size=n)
fig, ax = plt.subplots(nrows=n, ncols=2, figsize=(8, 2.5 * n))

for i, index in enumerate(ix):
    ax[i, 0].imshow(ims[index, ...])
    ax[i, 1].imshow(ims[index, ...])
    plot_segmentation(ax=ax[i, 0], masks=mask_pred[index, ...])
    plot_segmentation(ax=ax[i, 1], masks=mask_truth[index, ...])
    ax[i, 0].set_ylabel(tissue_types[index])

for a in ax.ravel():
    a.get_xaxis().set_ticks([])
    a.get_yaxis().set_ticks([])

ax[0, 0].set_title("Prediction")
ax[0, 1].set_title("Truth")
plt.tight_layout()
plt.show()
```

../_images/examples_link_train_hovernet_29_0.png

We can see that the model is doing quite well at nucleus detection, although there are some discrepancies in nucleus classification.

Conclusion

We trained HoVer-Net from scratch on the public PanNuke dataset to perform simulataneous nucleus segmentation and classification. We wrote model training and evaluation loops in PyTorch, including code to distribute training across 4 GPUs. The trained model performs well, with an average Dice coefficient of 0.785 on held-out test set. We also evaluated performance across tissue types, finding that the model performs best in Stomach tissue and worst in Head & Neck tissue.

Load this pre-trained model and test it out yourself!

References

- 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.
- 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.

Session info

[26]:

```
import IPython

print(IPython.sys_info())
print(f"torch version: {torch.__version__}")
```

```
{'commit_hash': '223e783c4',
 'commit_source': 'installation',
 'default_encoding': 'utf-8',
 'ipython_path': '/opt/conda/envs/pathml/lib/python3.8/site-packages/IPython',
 'ipython_version': '7.19.0',
 'os_name': 'posix',
 'platform': 'Linux-4.19.0-12-cloud-amd64-x86_64-with-glibc2.10',
 'sys_executable': '/opt/conda/envs/pathml/bin/python',
 'sys_platform': 'linux',
 'sys_version': '3.8.6 | packaged by conda-forge | (default, Dec 26 2020, '
                '05:05:16) \n'
                '[GCC 9.3.0]'}
torch version: 1.7.1
```

[29]:

```
# hash for PathML commit:
!git rev-parse HEAD
```

```
3f68d77d0c7b324acce74214e713a0bf79e60d84
```

Previous Next

PathML

-
- Brightfield Imaging: Quickstart
- View PathML on GitHub

Previous Next

Brightfield Imaging: Quickstart

View on GitHub

Here we demonstrate a typical workflow for preprocessing of H&E images. The image used in this example is publicly avilalable for download: http://openslide.cs.cmu.edu/download/openslide-testdata/Aperio/

a. Load the image

[3]:

```
import os

os.environ["JAVA_HOME"] = "/opt/conda/envs/pathml/"

from pathml.core import SlideData, types

# load the image
wsi = SlideData("../../data/CMU-1.svs", name="example", slide_type=types.HE)
```

b. Define a preprocessing pipeline

Pipelines are created by composing a sequence of modular transformations; in this example we apply a blur to reduce noise in the image followed by tissue detection

[5]:

```
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,
        ),
    ]
)
```

c. Run preprocessing

Now that we have constructed our pipeline, we are ready to run it on our WSI. PathML supports distributed computing, speeding up processing by running tiles in parallel among many workers rather than processing each tile sequentially on a single worker. This is supported by Dask.distributed on the backend, and is highly scalable for very large datasets.

The first step is to create a `Client` object. In this case, we will use a simple cluster running locally; however, Dask supports other setups including Kubernetes, SLURM, etc. See the PathML documentation for more information.

[6]:

```
from dask.distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=6)
client = Client(cluster)

wsi.run(pipeline, distributed=True, client=client);
```

[7]:

```
print(f"Total number of tiles extracted: {len(wsi.tiles)}")
```

```
Total number of tiles extracted: 150
```

e. Save results to disk

The resulting preprocessed data is written to disk, leveraging the HDF5 data specification optimized for efficiently manipulating larger-than-memory data.

[8]:

```
wsi.write("./data/CMU-1-preprocessed.h5path")
```

f. Create PyTorch DataLoader

The `DataLoader` provides an interface with any machine learning model built on the PyTorch ecosystem

[9]:

```
from pathml.ml import TileDataset
from torch.utils.data import DataLoader

dataset = TileDataset("./data/CMU-1-preprocessed.h5path")
dataloader = DataLoader(dataset, batch_size=16, num_workers=4)
```

Previous Next