CSB Software Engineering Course 2025: Final Exam

Goals

1. Work with existing API by creating plugins for a well documented software tool (Napari)
2. Use Dask for lazy evaluation of large datasets
3. Find existing open source projects that implement functionality of interest and expand upon them.

Exercises:

1. Intro: Familiarize yourself with the Napari image viewer and its framework for building plugins
2. Plugin 1: Build a Napari "viewer" plugin for SVS files that preloads a file into memory before displaying.
3. Plugin 2: Build a Napari "viewer" plugin for SVS files that loads relevant parts of the image on the fly.
4. Plugin 3: Extend the 'one-the-fly' image "viewer" plugin to perform nuclear thresholding in real time.
5. Plugin 4: Extend the 'one-the-fly' image "viewer" plugin to use deep learning nuclear segmenter in real time.

Test Data

You can find folder with two test images here. Use the smaller one for your initial tests, especially for Plugin 1 which loads slowly.

Part I: Introduction to Napari

A) Install Napari

Create a new conda environment(Python=3.10) and install TensorFlow (recommended version=2.10) and then Napari as indicated here.

B) Install Openslide

In this environment install openslide using the following commands:

```
conda install -c conda-forge openslide
conda install -c conda-forge openslide-python
```

C) Create your first plugin

Follow through the tutorial here to get a "Hello World" plugin working. Make sure you understand the structure of the various files and the naming conventions. Deviations from these could give rise to errors when you are writing your own plugins.

After you are done with this part, please uninstall this hello world plugin before proceeding.

D) Learn how to use cookie cutter to avoid boiler plate code

Copier is a package that creates empty project templates to save you the trouble of writing standard repeated code. Install Copier as follows.

```
conda install -c conda-forge copier jinja2-time npe2
```

Now that you have Copier installed you can create an empty plugin for Napari as follows:

```
copier copy --trust https://github.com/napari/napari-plugin-template new-plugin-name
```

This will prompt a long series of questions which determines the exact nature of the code created. It doesn't matter what you choose for most of these options, but please ensure that you specify that your plugins are just Readers. Look at the Napari documentation for this type of plugin and read through their example. Based on these, make sure you understand how you would need to modify the files created by copier.

Part II: Build a pre-loading SVS Reader

Your goal is to write a plugin that when pointed to an SVS file, pre-loads the whole image, with all pyramid layers, into memory and then passes these to Napari for display.

A) Learn how to use OpenSlide to load an SVS file in the format expected by Napari

Each SVS file contains a pyramid of the image at different zoom levels. We want to load this entire pyramid a list of numpy arrays (with each array representing a single zoom level). Make use of the

provided Test Data slides for testing your code.

1. Take a look at OpenSlide's (not so easy to read) documentation and look for examples online about how to use openslide to read images using it.
2. Create an empty list `myPyramid`. Write a loop to go through the different zoom levels in the svs file, and for each level use the `read_region` function to read the data at that zoom level, convert to a numpy array and append to the list. This is the input expected by napari.
3. If you have done this correctly, a modified version of this code should display the pyramid:

```
import napari as npr
viewer=npr.Viewer()
viewer.add_image(myPyramid,contrast_limits=[0,255])
```

B) Implement this reading function as a plugin

1. Create an empty Reader plugin using Copier
2. Update the manifest file to use your plugin by default for SVS files
3. Insert your code above into the main python file (likely named _reader.py). This file should already be pre-populated with code to load up numpy files. You need to replace it with the svs reading code you just wrote while sticking to the basic conventions for variables returned etc.
4. Install the plugin and make sure it shows up as active in your plugins list inside napari. Now drag an svs file into Napari to see if it works. Note: especially for the larger file this could take a while to preload, so it may look like the plugin is frozen.

Part III: Build an On-The-Fly SVS Reader

Your goal is to write an SVS reader plugin that doesn't need to pre-load the whole image pyramid into memory, but instead reads the part of the image it needs to display on the fly. To achieve this you take advantage of Dask's lazy (i.e., delayed) evaluation ability. You will essentially replace the pyramid based on a list of numpy arrays in the previous plugin with one made of dask arrays instead.

A) Familiarize yourself with Dask and it's lazy evaluation.

Read a little bit about dask arrays here and watch the embedded video. You should have a rough idea of how they are similar and different from numpy arrays, and what is meant by lazy/delayed evaluation.

B) Learn about how Dask and Napari can be used to deal with large data sets.

In this napari tutorial the different parts of an image (equivalent to our pyramid) are stored in set of png files. The tutorial shows you how dask can be used to load just the part of the image being displayed in real time. You don't need to work through this example in detail, but do need to understand the basic principles.

C) Implement delayed reading in the context of SVS files

1. Instead of loading individual tif files as in the previous example, you will need to load individual tiles from an SVS file using OpenSlide's deep zoom functionality. Familiarize yourself with this, and using `get_tile` to load individual tiles.
2. Write a loop over zoom levels and tiles within each level to load up the whole image. Now wrap this within the `from_delayed` functionality to enable lazy evaluation. Note: this is hard to figure out from scratch. May be a good idea to look online for code that has already implemented parts of it, for example PathFlowAI. Make sure your code gets all the zoom levels! Also note that napari expects the layers to be ordered monotonically in terms of size, and if the output generated does not conform you will get an error. Note: if you choose to use an alternate pre-existing implementation using additional libraries, be prepared to explain the functionality in detail during your exam.
3. Confirm this works, by passing the lazy pyramid to napari and viewing, similar for the previous plugin.
4. Now that you've figure this out, create a new plugin using CookieCutter replacing the reading with your new lazy reader.

Part IV: Add On-The-Fly Image Processing to your lazy SVS Reader

The goal of this part is to run some image calculation on the fly as you are loading up the image. In this specific case, we will use `rgb2hed` to identify nuclei when we are zoomed in enough.

A) Learn how SciKit Image's rgb2hed works

Look through the github code for the `rgb2hed` function in scikit-image. This currently calls a sequence of functions. Implement this as a single standalone function that does not require `skimage`. This should really contain just a handful of lines. It is okay to hardcode the stain vectors.

B) Re-implement using dask rather than numpy functions

To enable lazy evaluation rewrite your `rgb2hed` function using Dask's array operations.

C) Implement on the fly thresholding of rgb2hed as an additional layer

In addition to displaying layers in form of images, Napari is also capable of displaying overlays. These overlays can be in the form of another image or a so called Labels layer, which may be specified in terms of Dask arrays. Implement either a dask based image or label pyramid, such that only at the highest zoom level it invokes your `rgb2hed` function, and returns 1 if it is above a certain threshold (for simplicity you may hardcode this value) and zero otherwise. Confirm this works correctly by passing both the lazy image pyramid from the previous exercise and this new image/label pyramid to Napari and confirming they produce the expected behavior.

D) Implement as a plugin

Readers are capable of returning multiple layers, including Label layers. To your previous plugin code add the Label layer you just devised, and save as a new plugin. It is also possible to use two Image layers as needed.

Part IV: Add On-The-Fly deep learning model to your lazy SVS Reader

The goal of this part is to understand how you can implement existing deep learning models that allow for image analysis as you are loading up the image. For this specific case, we will use a published pre-trained `2D_versatile_he` model from StarDist. The model allows for nuclear segmentation, and was trained on histopathological images from the MoNuSeg 2018 challenge and TNBC dataset from Naylor et al (2018).

A) Learn how to use StarDist nuclear segmenter

Look through the github code to see how you can call the `2D_versatile_he` model from StarDist. Clone the repository and integrate it with your environment to be able to use its libraries with Dask and Napari. StarDist uses TensorFlow to run, which, for simplicity, we suggest to use running on CPU.

B) Implement on the fly nuclei segmentation as an additional layer

Similarly as in the previous exercise, we want to create a dask based pyramid, such that only at

the highest zoom level it invokes the nuclei segmentation. To do so, explore how to run a deep learning model on dask array. Save its output so that 0 is background and 1 are nuclei. Save the new dask array pyramid and visualize it with the H&E image. Pay attention to the results, and think what might have caused the model to make mistakes in detection.

D) Implement as a plugin

Now, add the Label layer (or another Image layer) you just devised, and save as a new plugin.

In Person Group Presentation On Friday (June 13th):

1. You will work on this exercise in groups of three and present as a group on Friday. We encourage discussion among everyone to get through the exercise.
2. Show up on Friday 5 minutes before your specified time slot (we will decide this on Thursday).
3. Your oral exam will last 20 minutes, 14 minutes for a prepared presentation from you and 6 minutes for questions. During your group's presentation:
    i. Demo your plugins
    ii. Ensure all members of the group take turns to present.
    iii. Prepare a couple of slides explaining walking us through how you implemented the plugins. If you could not complete the exercise, explain what went wrong and what you would do next to fix it.
    iv. Be prepared to answer questions about the logic, code, and other details of the plugin. We may ask to see your code, so have it on hand. Also questions about anything covered during the course is fair game.
    v. Pitch us your dream plugin: expect to have a high level discussion about it