Hochschule Neu-Ulm
University of Applied Sciences

Bachelor Thesis

In the bachelor program

**Game-Produktion und Management**

at University of Applied Sciences Neu-Ulm

# Implementation of a pipeline for segmenting histological images using machine learning

| | |
|---|---|
| 1st examiner: | Prof. Dr. Johannes Schobel |
| 2nd examiner: | Prof. Dr. Peter Kuhn |

Author: Fabio, Maier (Enrolment number: 294207)

Topic received: 01.12.2023
Date of submission: 02.04.2024

Hochschule Neu-Ulm
University of Applied Sciences

## Erklärung

Ich versichere, dass ich die vorliegende Abschlussarbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe und die Überprüfung mittels Anti-Plagiatssoftware dulde.

| | |
|---|---|
| Mengen, 02.04.2024 | [illegible] |
| Ort, Datum | Unterschrift |

Hochschule Neu-Ulm
University of Applied Sciences

## Abstract

The present thesis delves into the implementation of a pipeline for segmenting histological images using machine learning. The intended purpose of the pipeline is to enable quick and reliable preparation of test data to be used in the training of machine learning models as well as the evaluation of the prepared data by fully trained models. The whole slide images used by such models generally require some manor of pre-processing before they can be evaluated by a machine learning algorithm, thus this project aims to enable the researcher to complete all necessary steps by using the Pre-processed Image Preparator and Tiler (PIPET).
Given an input image, PIPET can generate tiled images for any given size and resize these, apply tissue masks to the given image and stitch evaluated tiles back into the original image size.

Keywords: machine learning, pipeline, whole-slide images, tissue

## Table of Contents

Hochschule Neu-Ulm
University of Applied Sciences

## List of Figures

Hochschule Neu-Ulm
University of Applied Sciences

## List of Code

## List of Tables

Hochschule Neu-Ulm
University of Applied Sciences

## List of Abbreviations

| | |
|---|---|
| **PIPET** | Pre-processed Image Preparator and Tiler |
| **ML** | Machine Learning |
| **API** | Application Programming Interface |
| **WSI** | Whole Slide Image |
| **TIFF** | Tagged Image File Format |
| **FOSS** | Free Open-Source Software |
| **IDE** | Integrated Development Environment |
| **ENTE** | Extended Neuronal-Networks for Tumour Exploration |
| **RGB** | Red Green Blue |
| **ONNX** | Open Neural Network Exchange |

# 1 Introduction

Machine Learning (ML) has become increasingly popular over the last few years, especially with the advancement of semiconductor technology and thus with the rise of dedicated ML-Accelerators, like Google’s Tensor Processing Units (TPU) [1], ML-Models have become ever more robust and reliable due to faster and more efficient training.
In the medical sector, ML has long been a subject of extensive research [2], but many projects in the field have been confined to proprietary technologies or theoretical research papers.
This thesis attempts to create a long-lasting basis for preparing data for use in training ML-models as well as a user-friendly tool to evaluate datasets using fully trained models.

## 1.1 Motivation

With Cancer being one of the most prevalent and varied causes of death on the Planet [3], researchers have developed a variety of methods and ways to detect and categorize cancers using ML [4]. One such method uses whole slide images, also known as virtual slides or digital slides, referring to high-resolution digital representations of complete histological slides, and a ML-Model that is trained to recognize cancer cells in such images. This approach of segmenting cancers requires the evaluated image to be pre-processed and to be split into smaller images. Manually pre-processing the amount of data required to train a ML-model is time consuming and repetitive.
The present work focuses on developing a robust algorithm that can be used to speed up and automate tasks, such as slicing and masking a WSI.
Furthermore, the developed project will be fully open source.

## 1.2 Challenges

The challenge in developing an algorithm that can reliably accomplish the goal of this thesis lies mostly in the varied formats, sizes, and resolutions WSIs come in, which will be described in the following.

### 1.2.1 The TIFF Format

One of the most common image formats used in WSIs is the Tagged Image File Format (TIFF), the format has a few advantages, which makes it suitable for the characteristically large WSI. The advantages TIFF provides in this context are the capability to be uncompressed, or have a lossless compression applied [5].
Furthermore, TIFF files can be comprised of multiple image files, enabling the TIFF to become pyramidal if the subfiles are the same image at multiple resolutions (Fig.1).

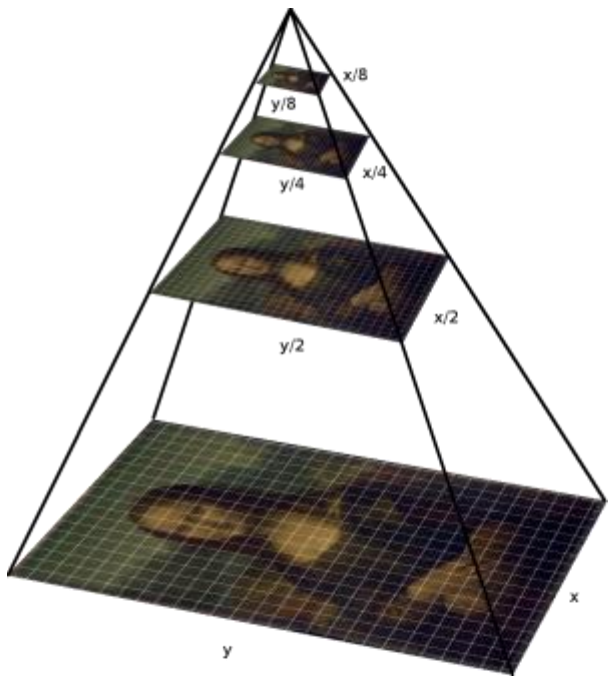


*Figure 1: Tiled Pyramidal TIFF, x and y specify the length and width of the image*

While the TIFF is one of the most flexible image formats in use today, that also brings problems with it. With the ability to be pyramidal, and tiled, or non-pyramidal and striped, or many other combinations of the name-giving tags, a TIFF itself may be entirely different to a program based on a few changes, even if visually there is no discernible difference. The added complexity that can be contained in a TIFF requires the program interacting with a given TIFF to be able to support a wide variety of TIFF tag-combinations, or the ability to parse a given TIFF into the required format depending on the operations being done on the image.

Another common format used in WSI is SVS. SVS is a proprietary extension of the TIFF format, making it incompatible with many standard applications, fortuitously, there exist a number of open source libraries, like OpenSlide [6], that are capable of operating with SVS images.

### 1.2.2 File Size

With WSIs being upwards of 93000 x 80000 pixels in size, an uncompressed image of that size can be larger than one gigabyte. This makes working with such files incredibly memory intensive. The program working with such files needs to strike a balance between having parts, or the whole image in memory, and reading or writing other parts from and to the disk.

Additionally, some operations on such large images can become bottlenecked by the compute performance of the device running the application. For example, generating a tissue mask using PIPETs tissue masking functionality, is of $O(n^2)$ complexity, which means the runtime scales with image size.

1.2.3 Packages and Libraries

Inter-package compatibility is an important aspect of this project. With the amount of varied operation and inputs PIPET deals with, it is important for image packages like Pillow, pyVips, OpenCV2 and OpenSlide to be able to perform their operations without having to read the base image from disk at each step.

### 1.2 Objective

The objective of this thesis is to develop a program that is capable of handling a broad spectrum of WSIs and prepare them for use with a ML-Model, as well as releasing the program as Free Open-Source Software (FOSS).
PIPET is intended to be used as a standalone tool used in the command line or bash, the user should only need to supply a ML-model, a WSI and the preferred input parameters. Additionally, PIPET will be able to be used like a package to access its component functions in conjunction with other programs or projects.

## 2.0 Technology

As PIPET is intended to be used by scientists and be FOSS, special care had to be placed in selecting the technologies and applications used in its creation.

### 2.1 Language

For the development of PIPET, the programming language Python was chosen. Python is a highly portable, OS-Agnostic language capable on running on most hardware in use today, thus making it ideal for reaching the widest possible audience [7]. Furthermore, Python is very popular and thus has an extensive library of modules, so called packages[8], available for a wide variety of applications. Lastly, Python syntax is very human readable, allowing beginners and newcomers to a project to quickly understand what a given piece of code does.

For PIPET, the Python version 3.11 has been chosen to maximize compatibility with existing packages, as one drawback of Python is that newer versions of Python may not be compatible with older packages.

## 2.2 Packages

Taking advantage of the rich Python ecosystem, PIPET relies on a few packages to function.

### 2.2.1 OpenCV

OpenCV is, in its current version [9] 2.X, a C++ based collection of computer vision algorithms, PIPET uses OpenCV's python wrapper, opencv-python.
OpenCV is a powerful tool for image processing, it provides a host of different algorithms to transform images, videos, detect objects and even basic signal processing.
While OpenCV is not the sole image-processing package in use in PIPET, it provides core functionality that is indispensable for PIPET, such as thresholding and noise removal. Additionally, OpenCV is the de-facto standard for image-processing in python.

### 2.2.2 Pillow

Pillow is the continuation of the deprecated Python Image Library (PIL), Pillow [10] is another image handling package which is extremely flexible and one of the de-facto standard packages used to handle images with Python.
The central part of Pillow is formed by the Image class [11], which provides extensive functions to modify and handle a multitude of image formats, used in PIPET to post-process the image after inference.
As one of the most popular image handling packages, Pillow has ample support from other libraries to interact with Image objects, such as NumPy, OpenSlide and the aforementioned OpenCV.

### 2.2.3 OpenSlide

OpenSlide is the third pillar of PIPETs image handling capabilities.
As the standard package for working with WSIs, OpenSlide provides functionality missing in some other packages, for example opening proprietary TIFF image files or other proprietary formats [6].

### 2.2.4 NumPy

NumPy is the most popular and comprehensive Mathematics package for python, it is powerful, fast, and easy to use.
The popularity of NumPy comes from its high efficiency, attributed to the optimized C-Code which contains its functions, in addition to its syntax which wraps around said functions [12].
NumPy is used for different operations on the image arrays within PIPET.

### 2.2.6 PyVips

PyVips is a python binding for libvips, an image processing framework with a focus on efficiency. Furthermore, PyVips handles TIFF files using libTIFF, enabling very solid handling of large TIFF files, where Pillow falls short. Because of the efficiency PyVips provides, most of the file handling in PIPET is done by PyVips.

2.2.6 ML-Frameworks
As PIPET is aimed at usage with ML-models it needs to support integration thereof. For such tasks, the Python environment has a few options. Pytorch [13] is one of the most popular choices for anything ML-related in Python thus making it the primary choice of running an inference on a given ML-model in PIPET. To facilitate a wider compatibility of ML-Models, ONNX (Open Neural Network Exchange), is used as the format which PIPET uses to load ML-models.

### 2.3 IDE & Version Control

PIPET is written in Python version 3.11 using the JetBrains Integrated Development Environment (IDE) PyCharm. PyCharm is a full-fledged IDE with support for all features needed to develop PIPET. PyCharm offers a full Python terminal, version control using Git, powerful refactoring tools, support for remote development, as mobile devices may not have the required hardware to execute PIPET on a full sized WSI, as well as support for automated tests and a powerful debugger. Images are represented as arrays, as such the built-in array-viewer of PyCharm is very useful to debug operations on arrays.
PIPET is hosted and versioned on GitHub, using the integrated git client of PyCharm.

# 3.0 Goals

The primary Goal of this thesis is to develop a robust pipeline for processing whole slide images (WSIs) to facilitate subsequent segmentation. PIPET aims to transform raw WSI data into a format suitable for analysis by machine learning models, thereby enabling automated segmentation of histopathological structures. The goals for the loading and preparation are as follow:

### 3.1 Efficient WSI Loading

Implement mechanisms to efficiently load large-scale whole slide images into memory, ensuring optimal utilization of system resources while maintaining data integrity and accuracy.

### 3.2 Data Preprocessing (Optional)

Develop preprocessing techniques to improve the suitability of WSIs for segmentation. This involves normalization, colour standardization, and artifact removal to mitigate noise and variability inherent in histopathological images.

### 3.3 Interoperability

Ensure interoperability with ML-based segmentation algorithms by providing functions and options to tailor the in- and outputs for the respective model used in conjunction with PIPET.

By achieving these goals, PIPET will lay the foundation for reliable segmentation of histopathological structures in whole slide images within the Extended Neuronal-Networks for Tumour Exploration (ENTE) project.

## 4.0 Existing works

### PyHIST[14]

PyHIST is an open source WSI pipeline that is similar to PIPET in scope. It offers functionality to generate tissue masks, evaluate singular tiles for useful data and allows export of selected tiles. While PyHIST offers a solid command line program to prepare WSIs, it is lacking a way to incorporate an inference step using a given ML-model, as well as a function to output a fully processed WSI. Furthermore, PyHIST has been not actively maintained as of the date of writing for four years. [15]

### end2end-WSI-preprocessing[16]

End2end-WSI-preprocessing is an open-source tool designed for comprehensive preprocessing of whole slide images (WSIs), offering an integrated solution for image enhancement and analysis. Similar in scope to PyHIST, it provides functionalities for preprocessing WSIs, including colour normalization, stain separation, and artifact removal. Additionally, end2end-WSI-preprocessing incorporates deep learning-based algorithms for feature extraction and classification, allowing users to leverage machine learning for WSI analysis. However, it should be noted that the tool lacks support for exporting processed tiles or fully processed WSIs, limiting its utility for downstream analysis tasks.

### WSITools[17]

WSITools is a Python library tailored for efficient handling of WSIs in digital pathology. Notable features include robust patch extraction capabilities, enabling users to systematically extract patches from WSIs for downstream analysis tasks. Additionally, WSITools offers functionalities to detect tissue regions within WSIs, facilitating automated tissue segmentation for pathology analysis. The library also provides support for exporting and parsing annotations from popular platforms such as QuPath and Aperio ImageScope, enabling seamless integration with existing annotation workflows. WSITools also includes advanced features such as WSI registration for aligning image pairs. Furthermore, WSITools enables users to reconstruct WSIs from processed image patches, enabling visualization and analysis of the entire slide after preprocessing.

## 5.0 Implementation

In the following part of the Thesis, the goals and technologies mentioned above, have been brought together into working solutions. Subsequent paragraphs will illustrate how the aforementioned goals have been reached and how PIPET works in detail. An overview of PIPETs structure can be seen in Figure 2.

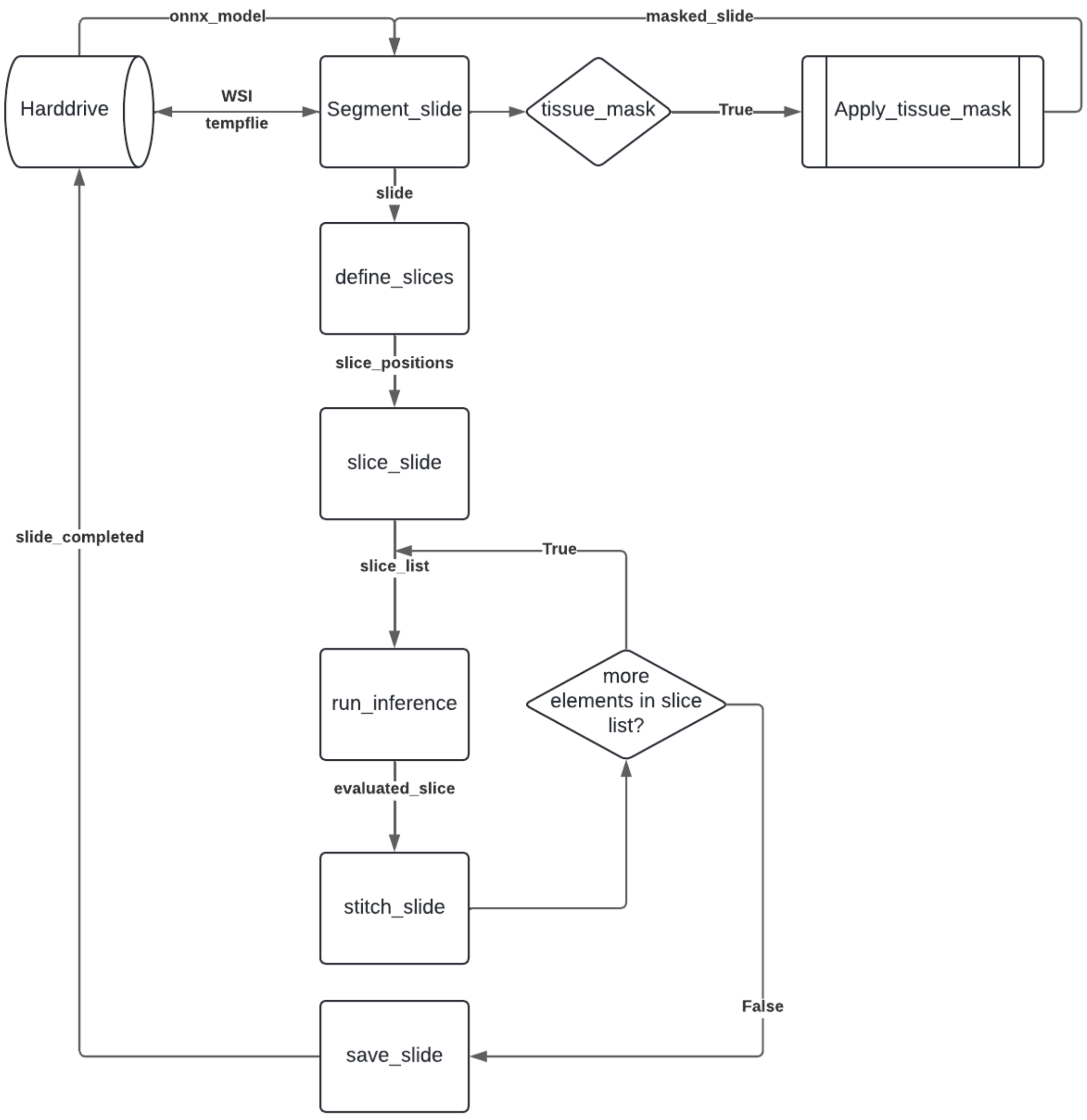


*Figure 2: Flow diagram of PIPET.*

Hochschule Neu-Ulm
University of Applied Sciences

### 5.1 WSI Loading

As OpenSlide has been chosen as the basis for reading images from storage, special care had to be taken to implement a way for OpenSlide to deal with unexpected variations in TIFF or other formats that can occur from programs like Photoshop.

```python
try:
    # Attempt to open the image with openslide
    print("Opening slide...")
    if not tissue_mask:
        slide = openslide.open_slide(slide_path)
    else:
        temp_slide = pyvips.Image.new_from_file(slide_path)
        temp_slide = Preprocessing.apply_tissue_mask(temp_slide, thresholding_tech)
        temp_slide = pyvips.Image.new_from_array(temp_slide)
        # write a tempfile to disk
        with tempfile.NamedTemporaryFile(delete=False, suffix='.tiff') as temp_file:
            temp_slide.write_to_file(temp_file.name, pyramid=True, tile=True, compression="jpeg")

        del temp_slide

        slide = openslide.open_slide(temp_file.name)

except openslide.OpenSlideUnsupportedFormatError:
    print("Converting file...")
    # If openslide cannot open the image format, catch the error
    # and then open the image with pyvips to convert it to a format openslide can read
    temp_slide = pyvips.Image.new_from_file(slide_path)

    if tissue_mask:

        temp_slide = Preprocessing.apply_tissue_mask(temp_slide, thresholding_tech)
        temp_slide = pyvips.Image.new_from_array(temp_slide)

    # write a tempfile to disk
    with tempfile.NamedTemporaryFile(delete=False, suffix='.tiff') as temp_file:
        temp_slide.write_to_file(temp_file.name, pyramid=True, tile=True, compression="jpeg")

    del temp_slide
    slide = openslide.open_slide(temp_file.name)
```

*Code block 1: WSI Loading and exception handling if a unsupported image is loaded.*

To ensure maximum compatibility, PIPETs algorithm to load images with OpenSlide utilizes PyVips as a surrogate to convert ingested WSIs into a Pyramidal tiled TIFF with jpeg compression. As the Code block (Code block 1) shows, PIPET differentiates between two scenarios, segmenting WSIs with, or without tissue masks, it also tries to handle WSIs that are compatible directly with OpenSlide to save a small amount of resources. The algorithm works as follows.

First, PIPET checks whether a tissue mask should be applied to the WSI (Line 4 & 24), this is important as the tissue mask is applied to the WSI before it is finally loaded by OpenSlide. If a tissue mask should be applied to the WSI, PIPET will load the WSI using PyVips (Line 7-14), in both scenarios where OpenSlide could or could not load the image by itself (Line 24 - 27). If no tissue mask is to be applied, the file will be passed to OpenSlide (Line 5), if OpenSlide throws an OpenSlideUnsupportedFormatError, PIPET will default to the PyVips conversion algorithm (Line 18).
The conversion algorithm opens the WSI using PyVips (Line 22), it will then parse the image that is now in memory to a standard pyramidal tiled TIFF with jpeg compression (Line 31), subsequently PIPET uses the Python tempfile module to write a temporary version of the converted WSI to disk (Line 30). Afterward it is opened using OpenSlide by passing the tempfile reference to OpenSlide (Line 35).
If a Tissue mask is applied, the algorithm works largely the same. Except it passes the read image to PIPETs Utils module, which will apply a tissue mask before it is returned (Line 8 & 26).

PyVips has been chosen in this scenario due to its breadth of supported formats, as well as its efficient method in which it stores large images in memory. Additionally, PyVips has the ability to save such large images to disk again, for example, Pillow has been considered for this use case to limit conversions between image libraries, but Pillow exhibited issues saving images in the necessary sizes required for WSI processing.

## 5.2 Preprocessing

In order to efficiently segment a WSI, it is necessary to discern whether a given slice contains tissue or not, with this information it is possible to skip empty slices to speed up processing of large WSIs. The first step is applying a Tissue mask to the image.

```
1. def apply_tissue_mask(image, thresholding_tech, threshold=127, filter=True, rm_noise=True,
noise_filter_level=50, ):
2.     print("Starting masking process")
3.     image = image.numpy()
4.     original_image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
5.     image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
6.
7.     if rm_noise:
8.         print("Removing noise")
9.         kernel = numpy.ones((noise_filter_level, noise_filter_level), numpy.uint8)
10.        image = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
11.        image = cv2.medianBlur(image, 5)
12.        image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
```

*Code block 2: Preprocessing using morphological OpenCV operations.*

In preprocessing the WSI, we remove the artifacts from scanning or other artifacts that occur in WSIs, this is done with a combination of thresholding algorithms and noise reductions techniques. Thresholding describes algorithms to create binary images of black and white by comparing a given pixel to a threshold value and deciding whether the pixel should be black or white, these threshold values can range from being a hardcoded value to being calculated by a mathematical formula [18].

The Preprocessing class from the Utils module contains the thresholding algorithm, the apply_tissue_mask function is setup to be easily expandable to cover more complex thresholding techniques if necessary. In the scope of this thesis, it contains Otsus binarization, a simple thresholding algorithm, and adaptive thresholding, all provided by OpenCV. It functions as follows.

apply_tissue_mask expects an image from the PyVips module, as the loading of WSIs is done by PyVips in the main class. Additionally, the function requires the parameter thresholding_tech, which is a string that corresponds to one of the three available thresholding techniques implemented. The remaining parameters are for more fine-grained control, like skipping the de-noising step, or not applying a blur filter before thresholding, in addition to the threshold for simple thresholding (Line 1&2). The preprocessing steps are done on NumPy arrays that contain the image in greyscale. In addition, the original image is stored in a NumPy array as well, this is to facilitate merging in a later step (Line 5-7).
When noise reduction is chosen, OpenCV's morphological operations are employed to remove noise from the image (Line 7-12). In detail, the image undergoes a three-step process, first the image is being "opened", in OpenCV *opening* describes the process of eroding, and then dilating the image. Erosion helps in removing small,

isolated pixels (salt noise) in the image but reduces the size of objects and dilation increases the size of foreground objects.
Followed by a *medianBlur* operation, this operation is highly effective against small, isolated pixels that may have been missed on the first operation, here a pixel is replaced with the median value of its surrounding pixels.
Lastly, in a *closing* operation, which is the inverse of the *opening* operation is used to close any holes that may have developed in the previous steps, it is in essence dilation followed by erosion.

```
1. def otsus_binarization(image, filter):
2.     print("Applying Otsu's binarization")
3.     if filter:
4.         blurred_image = cv2.GaussianBlur(image, (5, 5), 0)
5.         ret, mask = cv2.threshold(blurred_image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
6.     else:
7.         ret, mask = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
8.
9.     return mask
```
*Code block 3: Thresholding using Otsus method.*

After the preliminary steps are finished, the tissue mask is created with the thresholding technique chosen, in the example above (Code block 3) Otsus method is chosen.
Otsus_binarization can be executed either with a gaussian blur operation or without. This can be chosen by specifying the filter parameter of apply_tissue_mask (Line 3). The application of the filter further reduces the impact noise has on the thresholding, in some cases it may still be beneficial to skip this step to preserve sharper detail, thus the user can choose the optimal procedure for their use case.
Otsus method has been chosen as the default algorithm for PIPET as it requires the least amount of user-input while providing good results in testing. It works by analysing the histogram of a grayscale image and finding the threshold that maximizes the separation between foreground and background pixel intensities [19]. This threshold is then applied to the image, resulting in a binary image where pixels are categorized as foreground or background based on their intensity values. Otsu's method is widely used for tasks such as object detection and image segmentation because it requires no manual input and effectively separates objects from the background.
After the image has been binarized, the ret variable now contains the chosen threshold and is discarded, while mask contains the actual mask and is returned to the previous function.

The last step before the masked image is returned, is applying the generated mask to the original image. The steps are illustrated in Figure3.

```
def merge(image, mask):
2.    print("Merging mask with source.")
3.    kernel = numpy.ones((20, 20), numpy.uint8)
4.    mask = cv2.bitwise_not(mask)
5.    mask = cv2.dilate(mask, kernel, 1)
6.    combined_image = cv2.bitwise_and(image, image, mask=mask)
7.
8.    black_pixels = numpy.where(
9.        (combined_image[:, :, 0] == 0) &
10.        (combined_image[:, :, 1] == 0) &
11.        (combined_image[:, :, 2] == 0)
12.    )
13.    combined_image[black_pixels] = [255, 255, 255]
14.    combined_image = cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB)
15.    return combined_image
```

*Code block 4: Merging by arithmetic operations*

- The black and white image from the thresholding step is inverted as the areas from thresholding where tissue is are black. (Line 4)
- The mask is dilated to ensure no tissue is removed when merging. (Line 5)
- The mask and the original image are arithmetically combined, if the mask value at a particular pixel is zero, the corresponding pixel value in the output image will be set to zero, regardless of the pixel values in the original image. (Line 6)
- An array is created in which indices are stored of the black pixels in the *combined_image* array, those will then be set to [255, 255, 255] which corresponds to white. (Line 8-14)
- Finally, the image will be converted to Red Green Blue (RGB) colour space, as OpenSlide would read OpenCV’s BGR colours as RGB anyway thus falsifying the colours of the image.

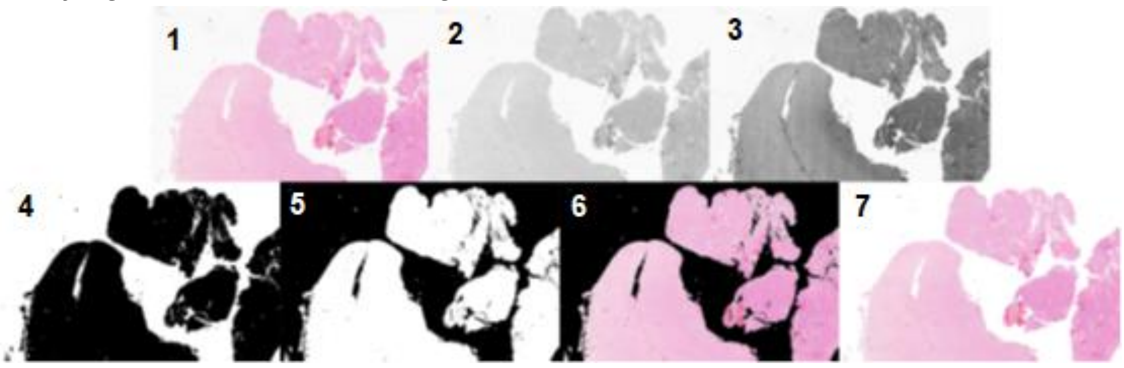


*Figure 3: WSI Tissue masking process*

### 5.3 Slice Processing

When the WSI has been successfully loaded into memory, PIPET will calculate the slices by iterating through the dimensions of the slide and dividing it into smaller sections.

```python
1.  def define_slices(slide, slice_height, slice_width):
2.      print("Defining slices.")
3.      slide_width, slide_height = slide.dimensions
4.      vertical_slices = math.ceil(slide_height / slice_height)
5.      horizontal_slices = math.ceil(slide_width / slice_width)
6.      slice_positions = []
7.
8.      for i in range(horizontal_slices):
9.          for y in range(vertical_slices):
10.             slice_positions.append((i * slice_height, y * slice_width))
11.     print("Defined: ", len(slice_positions), " Slices.")
12.
13.     return slice_positions
```

*Code block 5: Defining WSI slices*

The function *define_slices* takes in three parameters: *slide*, representing the WSI; *slice_height* and *slice_width*, denoting the dimensions of each slice. It starts by retrieving the dimensions of the slide using its *dimensions* attribute (Line 3). Then, it calculates the number of vertical and horizontal slices required to cover the entire slide (Line 4-5). It then iterates over the horizontal and vertical divisions, calculating the coordinates for each slice based on the provided slice height and width (Line 8-10).

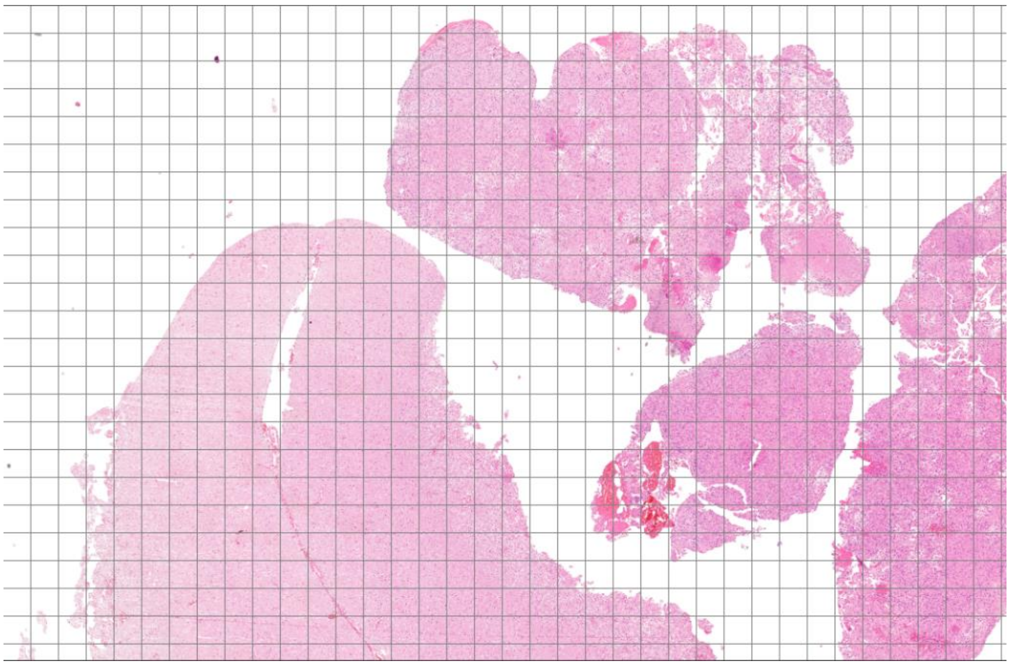


*Figure 4: WSI slices*

As is visible in Figure 4, the chosen slice size does not divide evenly across the resolution of the sample WSI, leading to slices that extend further out than the WSI.

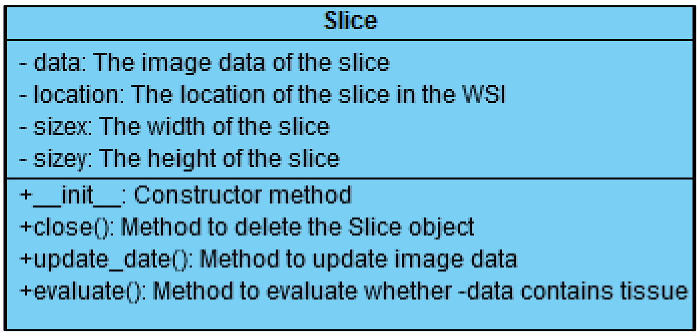
The slicing algorithm thusly employs the *read_region* function of OpenSlide which will fill an incomplete region with white pixels.

```python
1. def slice_slide(slice_positions, slide, slice_width, slice_height):
2.     slice_list = []
3.     for slice_position in slice_positions:
4.         print("Slicing " + f'{slice_position[0]}' + ':' + f'{slice_position[1]}')
5.         slice = slide.read_region(slice_position, 0, (slice_width, slice_height))
6.         temp_slice = Slice(slice, slice_position, slice_width, slice_height)
7.         slice_list.append(temp_slice)
8.
9.     return slice_list
```

*Code block 6:Slicing WSIs*

The *slice_slide* function is responsible for creating the slice objects (Line 6), these are the objects used in PIPET to represent the slice and its attributes for further segmentation. The algorithm iterates through the previously calculated *slice_position* list and utilizes *OpenSlide.read_region* to read the appropriate pixels from the WSI, returning a list of slices representing the WSI.

### 5.4 The Slice class



Slice

- data: The image data of the slice
- location: The location of the slice in the WSI
- sizex: The width of the slice
- sizey: The height of the slice

+__init__: Constructor method
+close(): Method to delete the Slice object
+update_date(): Method to update image data
+evaluate(): Method to evaluate whether -data contains tissue

*Figure 5:Class diagram for the Slice class*

The Slice class is the primary object on which PIPET operates after operations on the whole WSI are done. This is to facilitate cleaner and easier to read function calls.

**Data:** a typeless variable to store the current state of the image represented by the slice.
**Location:** a tuple containing the coordinates the slice belongs on in the WSI.
**sizex and sizey**: the actual size of the slice as it was read from the WSI.

```
def evaluate(self):
    temp_data = numpy.asarray(self.data)
    temp_data = cv2.cvtColor(temp_data, cv2.COLOR_RGB2GRAY)
    _, temp_data = cv2.threshold(temp_data, 127, 255, cv2.THRESH_BINARY)
    if cv2.countNonZero(temp_data) == temp_data.size:
        print("Slice does not contain data.")
        return False
    else:
        print("Slice does contain data")
        return True
```
*Code block 7: Evaluating slices*

The *evaluate* function within the Slice class serves to determine whether the slice contains tissue, enabling other functions to exclude empty slices from further processing (Figure 6).
In the *evaluate* function, the data contained in the slice object undergoes a binarization to enable the *countNonZero* function to determine how many black pixels are present in the current slice, black pixels in this situation representing tissue. It is then compared to the number of overall pixels present in the slice. If any pixel has been found to be black, or zero, the evaluation returns True.

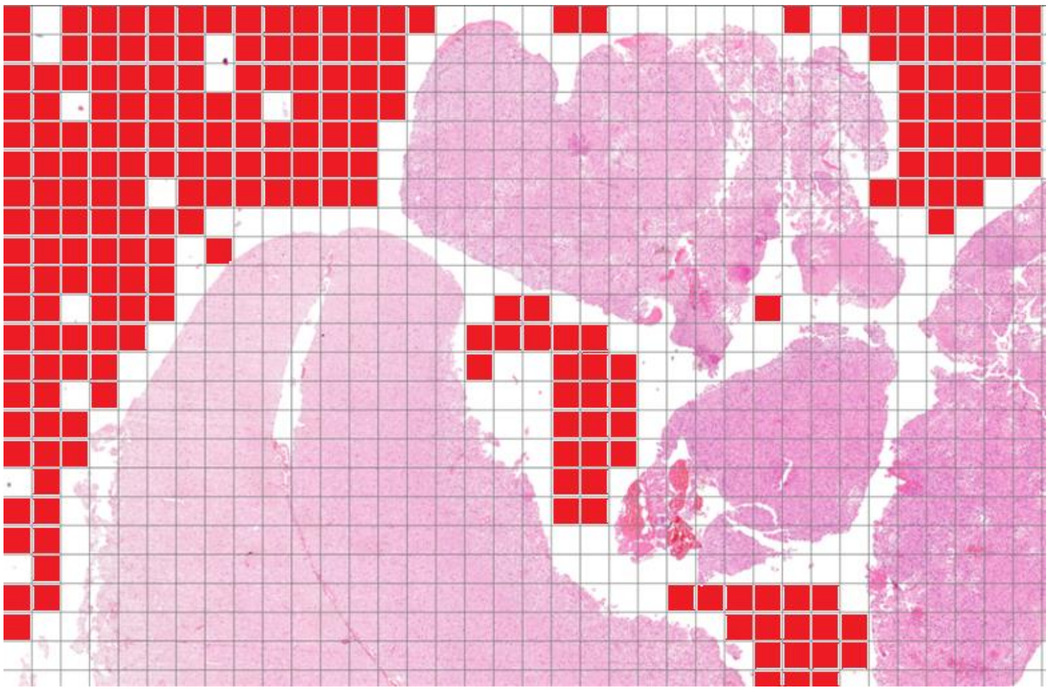


*Figure 6: Skipped slices*

## 5.5 Inference

The ML model utilized for testing and developing PIPET relies on the PyTorch segmentation models framework. It requires input slices sized 256 x 256, which should be normalized to a range between 0 and 1, spanning from 0 to 255.

```python
def run_inference(slice, pytorch_model, input_x, input_y):
    if slice.evaluate():
        print("Evaluating...")
        slice.data = slice.data.convert("RGB")
        temp_slice = resize(asarray(slice.data), (input_x, input_y))
        transformed_input = torch.from_numpy(temp_slice).type(torch.float32).permute(2, 0, 1)
        output = pytorch_model(transformed_input)
        output = output.sigmoid()
        output = output.squeeze(0).squeeze(0)
        img_array = output.detach().numpy()
        img_array = resize(img_array, (slice.sizey, slice.sizex))
        slice.data = Image.fromarray((img_array * 255).astype(numpy.uint8))
        slice.data = slice.data.point(lambda x: 1 if x > 120 else 0, mode='1')
    else:
        print("Skipped slice")
    return slice
```
*Code block 8: Inference and postprocessing using PyTorch and pillow*

The *run_inference* function expects a slice object, a ML-model compatible with PyTorch and the size of the input the model expects.
As the slices are read in the RGB colour space with alpha channel from OpenSlide, a conversion to RGB is performed to comply with the input shape of the Model.
ScikitImage is used to perform the resize and normalization step in one.
The normalized image array is then converted into a three-dimensional tensor, which is then rearranged. The numbers 2, 0, and 1 represent the new order of dimensions, such that the third dimension becomes the first, the first dimension becomes the second, and the second dimension becomes the third.
The tensor is then passed to the PyTorch model which will perform the inference on the tensor. The sigmoid function squashes the input values between 0 and 1 so the later steps can convert it back into an image with values between 0 and 255.
Squeezing the output tensor twice removes two single dimensional entries from the tensor, effectively leaving the array containing the image data.
After converting the tensor into a NumPy array, the slice is upscaled to the original slice size.
The upscaled slice is then multiplied by 255, creating a classic RGB image, following this, the lambda function (lambda x: 1 if x > 120 else 0) is applied to each pixel value x. If the pixel value is greater than 120, it is set to 1, otherwise, it is set to 0.
“Mode=1” indicates the resulting image should be a binary image.

### 5.6 Stitching and Saving

Reconstructing, or stitching the image is done by iterating over the list of slice objects returned from the inference step and inserting the slices into a blank image of the size of the original image.

```python
def stitch_slide(slice, blank):
    print("Stitching slice to:" + str(slice.location))

    vips_slice = pyvips.Image.new_from_array(asarray(slice.data))
    blank = blank.insert(vips_slice, slice.location[0], slice.location[1])
    slice.close()
    return blank
```
Code block 9: Stitching WSIs using insert().

The stitching is carried out using the insert function provided by PyVips, this function works well for this application as it automatically crops out parts of the slice that where not part of the original image.

```python
def save_slide(slide, output_path):
    print("Writing finished image.")
    os.makedirs(output_path, exist_ok=True)
    output_file_path = os.path.join(output_path, "segmented_slide.tiff")
    slide.write_to_file(output_file_path, pyramid=True, tile=True,
                        compression="jpeg")
```
Code block 10: Saving completed WSIs using pyvips.

Saving the finished WSI is done by PyVips as the other modules, like Pillow, had issues saving TIFF images above a certain size, throwing errors and not completing the image. Additionally, PIPET makes use of the python OS module to check whether the output path already exists. The output path is then combined with a standard filename for the output with which the image is then saved.

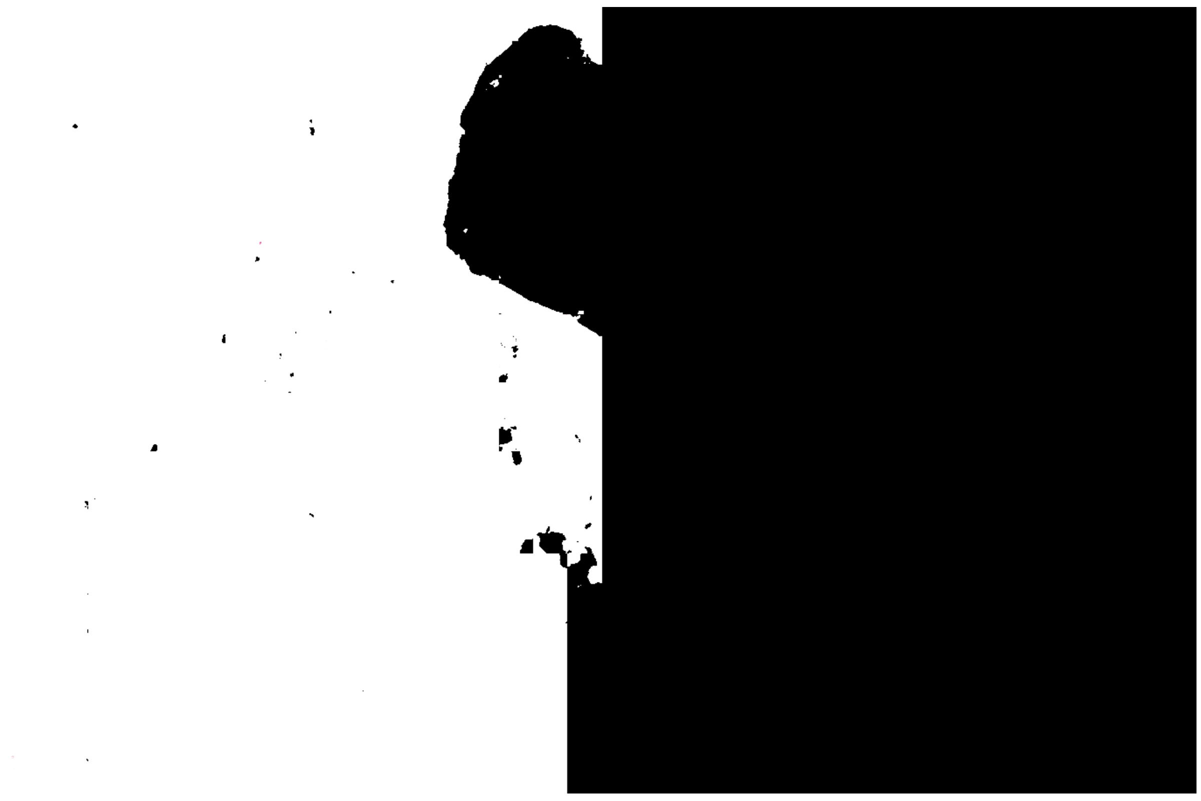


Figure 7:425 Slices stitched to finished image

## 6.0 Setup

PIPET has been developed with memory usage in mind, stemming from the fact, that WSI's can reach in excess of one gigabyte in size and 100.000 by 100.000 pixels in resolution. Special care has been put into making sure as little data as possible is in memory at a time. The machine PIPET has been developed and tested on is equipped with an AMD Ryzen 3600 six core-twelve thread CPU and 32GB of RAM.

The two WSI samples used for evaluation are TIFFs in the format needed to be compatible to OpenSlide, one has been provided for the development of PIPET and the other has been derived from the provided tiff and downsized significantly. Additionally, some samples for further testing have been obtained from the open access database of the National Cancer Institute [20]. The variety of sizes available for Testing range from 7.851 x 5.109 px(Figure 8.1) to 93.000 x 80.000 px(Figure 8.2), corresponding from 20MB to 1.288MB, most of the testing has been done on the supplied WSI with a resolution of 36.230 x 23.577 px (Figure 8.1).
The samples obtained through the National cancer Institute's portal are WSIs from breast tissue, meaning the results from PIPET will be inaccurate due to the focus of the ML-model on glioblastomas. However, they have been used to verify functionality of other file formats, svs in this case, and different WSI sizes.

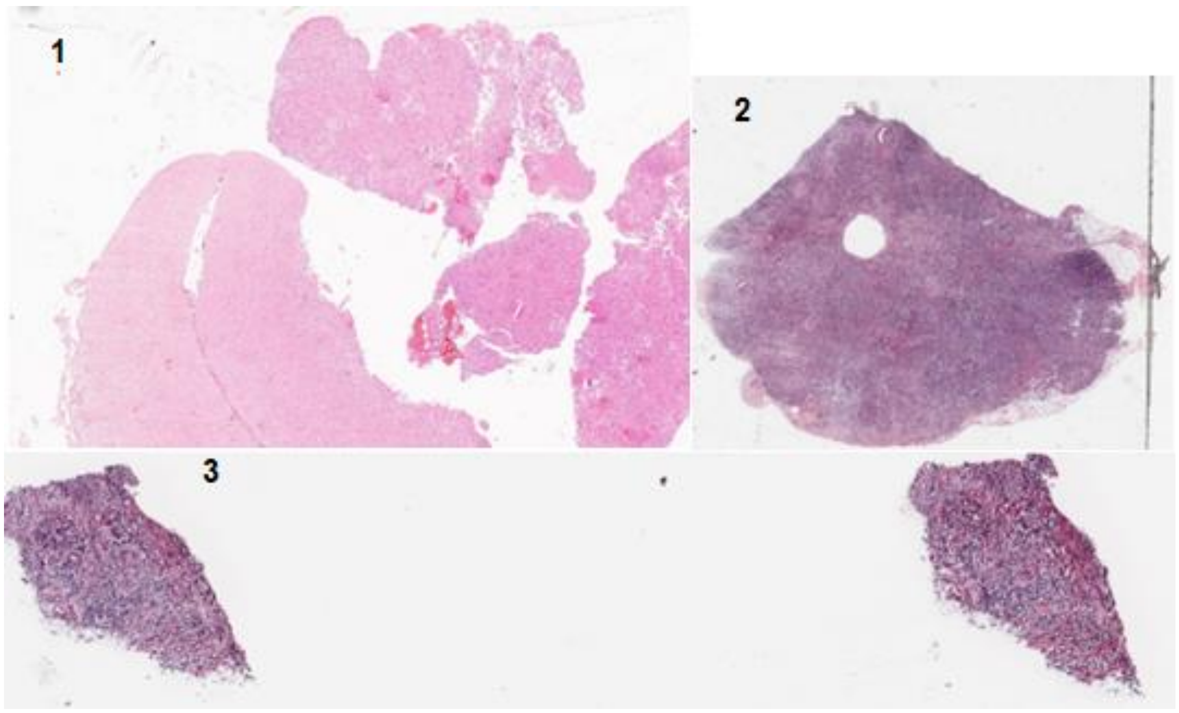


Figure 8: Samples used in development of PIPET.

To evaluate memory and CPU time usage, memory-profiler, a python package capable of monitoring and analyzing the memory usage and execution time of a python program is used. While memory-profiler is no longer actively maintained, its ease of use made it a good tool to analyze the performance of PIPET.

## 6.1 Profiling

The used memory-profile package comes with a utility called “mprof”, which can generate a logfile containing the overall memory usage of the program when it is executed as shown in table 1, afterward it can also generate a plot of the logfile. The profiler is also capable of analyzing memory profiles line-by-line when used as a decorator in the code, this functionality, while useful for optimization has limited output capabilities and was not used in this thesis.

```
CMDLINE main.exe
MEM 2.187500 1711729085.0375
MEM 21.527344 1711729085.1385
MEM 32.777344 1711729085.2395
MEM 54.277344 1711729085.3405
MEM 72.960938 1711729085.4415
MEM 89.839844 1711729085.5425
MEM 98.164062 1711729085.6440
MEM 124.828125 1711729085.7450
MEM 138.515625 1711729085.8460
MEM 147.933594 1711729085.9470
MEM 161.105469 1711729086.0475
```

*Table 1: mprof output*

The logfiles of mprof consist of two space separated values, one containing the memory usage and the other containing the Unix timestamp of the memory value. While mprof can generate a plot on its own (Figure 9), it is not great for the way PIPET operates when “compiled” to an exe.

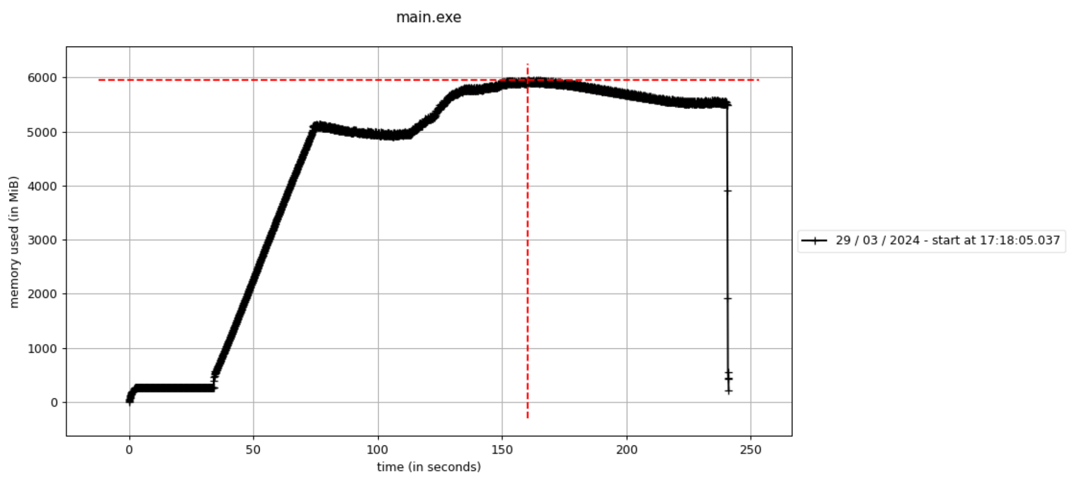


*Figure 9: mprof plot*

As it stands, PIPET needs human input when used in the command line, meaning execution time is not uniform across runs with the same parameters. Additionally, the plot itself is not very readable as the plot markers clutter visibility.

For the evaluation of the mprof logfile, matplotlib has been utilized to create a more suitable plot.

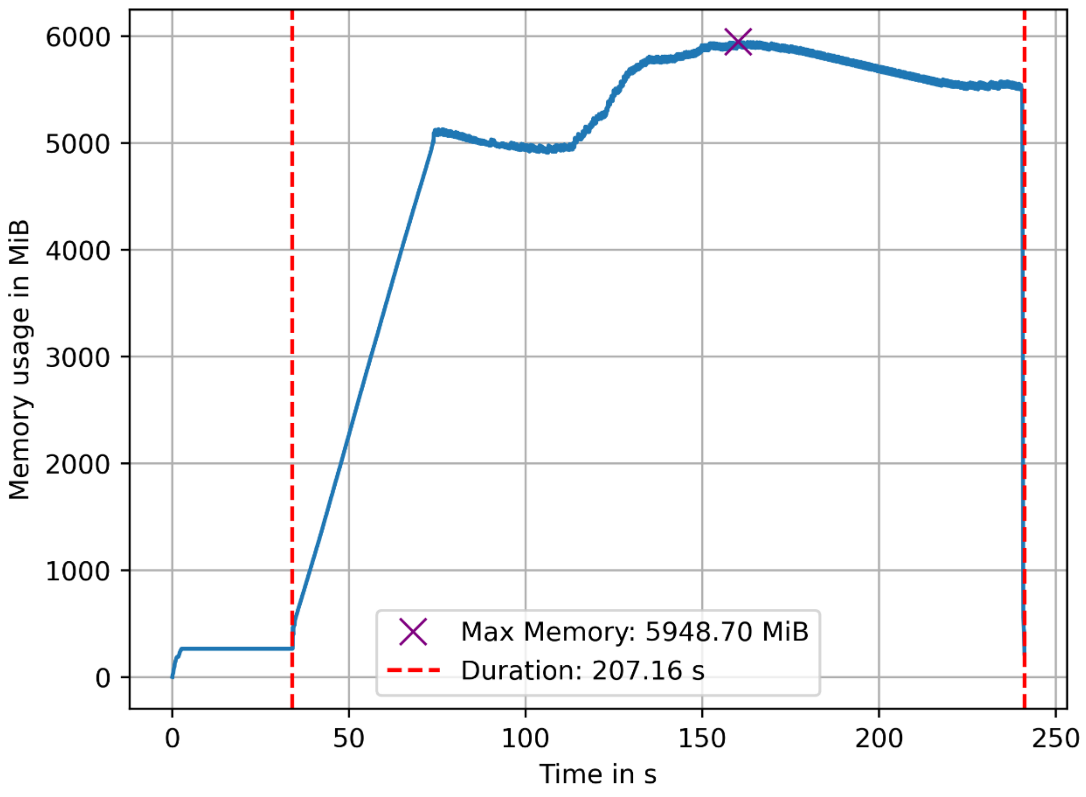


*Figure 10:matplotlib plot*

As is evident in Figure 10, the plot now shows execution time after the human input has been completed, creating comparability between runs. Furthermore, it now shows a numerical value of maximum memory usage and the run duration. As the state of the memory does not change until the input is complete, the second to last value of the memory in the input phase has been chosen as the start point of the duration evaluation.

## 7.0 Results

The following section critically evaluates the attainment of the objectives and Goals outlined in section 3. Throughout the previous chapters, the thesis comprehensively highlighted the development and functionality of PIPET.

The primary goal, as stated in section 3, was to develop a robust pipeline for processing WSIs, has been fully achieved. PIPET works with all tested resolutions and filetypes without issue. Furthermore, the goal of section 3.2, image preprocessing, has been realized as well.

### 7.1 Architecture

PIPET's structure changed from the beginning to completion of the project from being a singular function call to a collection of methods operating on a class instance to several static functions capable of being executed by themselves, while retaining a function that can execute the entire processing stack from a single call. It is also possible for PIPET to be built as an executable capable of handling WSI processing in the command line. The functions of each class, be it preprocessing, or PIPET's main code are built as static methods to encapsulate specific functionalities within the context of the class itself, enabling modularization, improved organization, and ease of access without the need for instantiation of class objects.

### 7.2 WSI Loading

In section 3 it is stated that PIPET should be efficient in loading large-scale WSIs, this goal has been achieved satisfactorily, but in review, there is still the possibility to optimize this further, in section 8, the thesis will explore this further. Currently, PIPET will keep all slices in memory at one time, this will mean an increase in memory usage with increase in file size This increase in memory utilization is not linear, for example using the provided sample WSI of 36.230 x 23.577 px (161MB), and accounting for 264,3MiB of overhead, uses 3810,62MiB of memory. While the scaled down version (7851x5109 px, 20MB) only uses 513,23MiB of ram, this is a difference in memory usage of a factor of 7.4, while the file size differs by a factor of 8,5. (Figure 11)

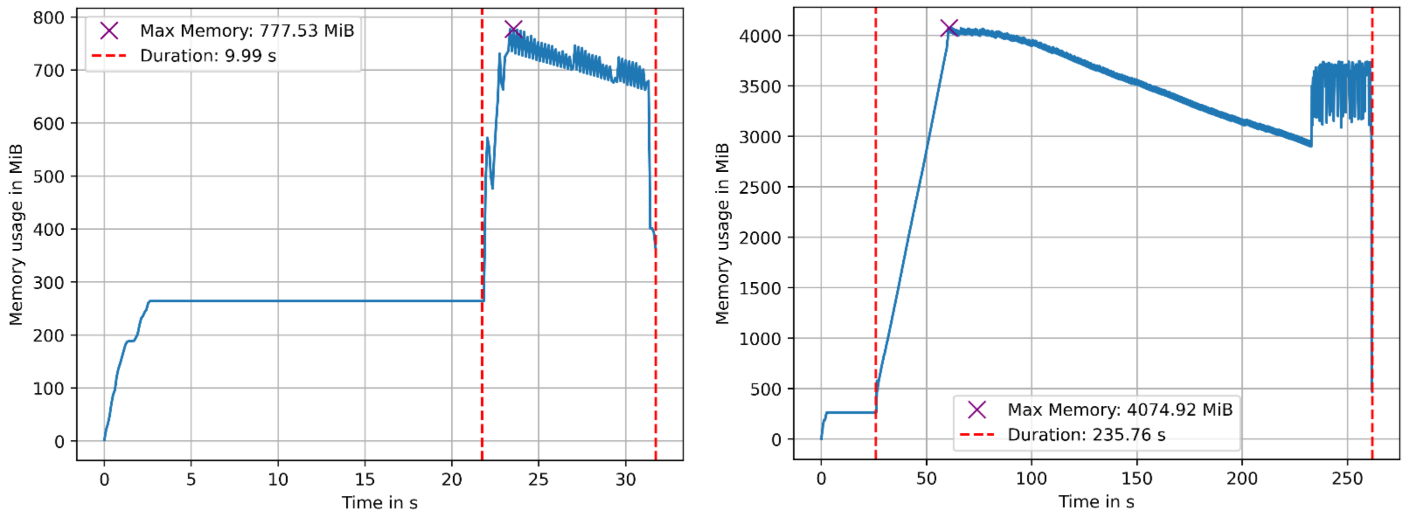


*Figure 11:Memory usage of PIPET with different WSI sizes*

This difference can be even more significant in some edge cases. For example, a sample, that has since been discarded from the test suite, was 256MB in size but exceeded 20GB of memory usage (Figure 11).

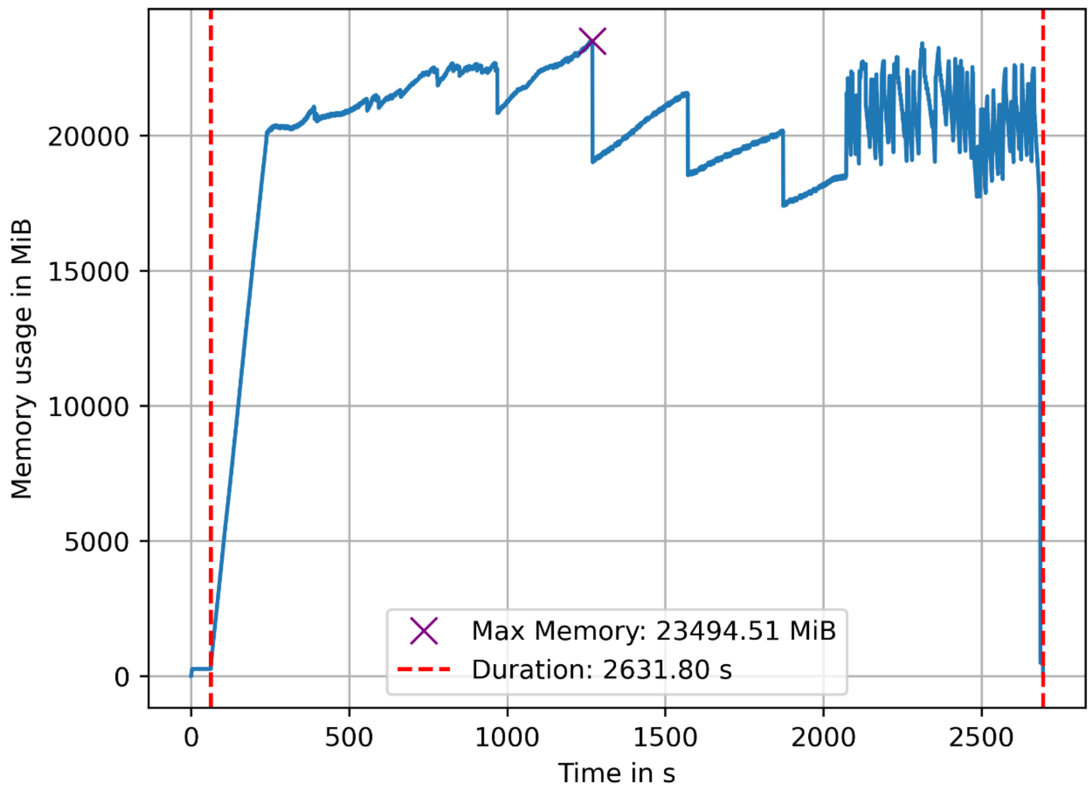


*Figure 12: Discarded sample image mprof plot.*

The difference between the samples may stem from a few factors. The samples obtained from the NHI all are in the SVS Format making them “modified” TIFF images, this could mean their compression algorithm could be the cause of this phenomenon. Furthermore, they may use a higher colour depth, causing each pixel to take up more memory than in other samples.

Overall, PIPET, in testing handled every image without issue. It should be noted, the very large resolution sample from the NHI portal (to 93.000 x 80.000 px) took up enough memory to cause the device to crash, a private server has been used to run this image once, but no mprof plot could be generated due to a lack of root access.

### 7.2.1 WSI segmentation

While memory usage was the main focus of optimization, thought has gone into reducing the runtime of PIPET’s pipeline. With the evaluation function of the slice class, it is possible to skip empty slices, this can lead to a significant reduction in CPU usage and runtime if the WSI contains a lot of empty slices (Figure 12).

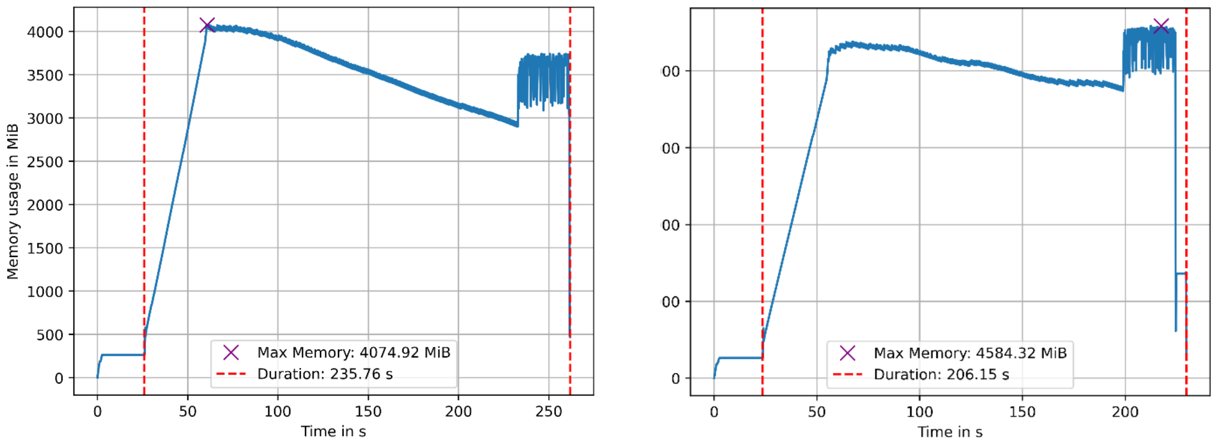


*Figure 13: Runtime of PIPET with and without skipping slices*

As is visible in Figure 12, with an appropriately clean image, the execution time in this example can be reduced by 12,6%. This reduction is dependent on the cleanliness of the WSI, for this test, the image 1 from Figure 7 has been used, once unmodified, and once after preprocessing (Figure 14).

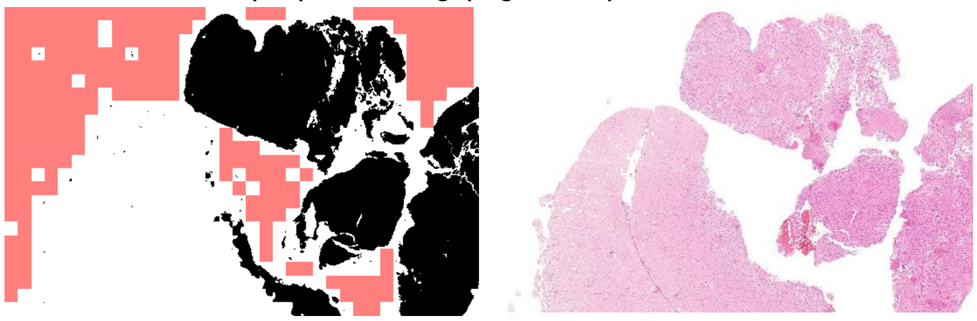


*Figure 14: Masked sample WSI. Evaluated: 638 Skipped: 226(Red)*

## 7.3 Preprocessing

The optional goal of implementing Preprocessing functionality outlined in section 3.2 has been achieved. While the preprocessing module works for most available samples, the nature of the operations done makes this functionality very memory intensive.

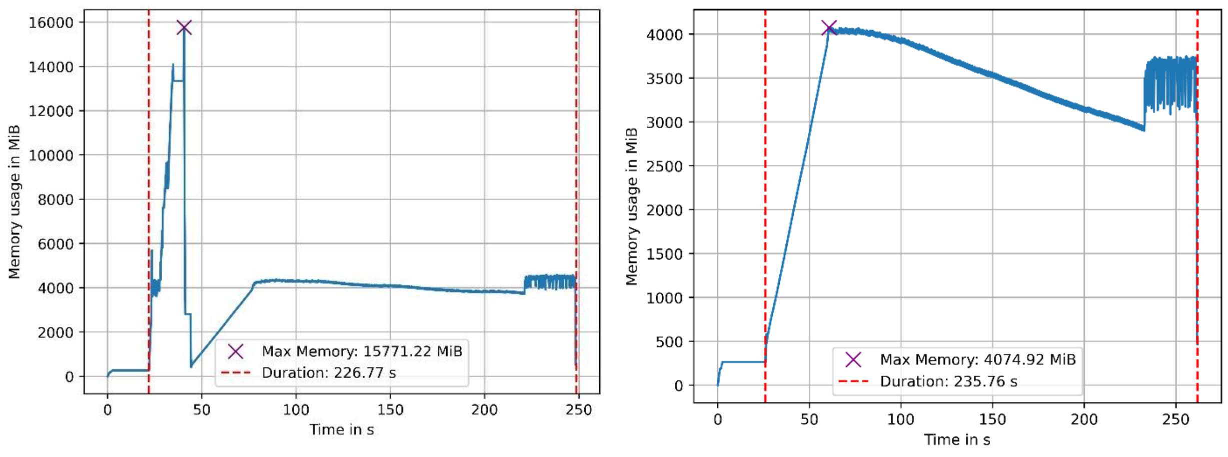


*Figure 15:Otsus binarization memory usage versus no thresholding.*

As is visible in Figure 15, the thresholding step almost quadruples the memory used with the same input. This is due to the nature of the global thresholding methods used, they operate on the whole image at once, necessitating the entire image to be stored in memory as a NumPy array with dimensions corresponding to the image size multiplied by the colour depth. It also requires another copy of the image in greyscale, out of which the mask is created. In the step merging the original image and the mask, the image is essentially in memory three times, the original, the mask and the combined image.

### 7.3.1 Otsus binarization and Simple thresholding

While PIPET has functionality for three thresholding techniques, Otsus binarization and Simple thresholding do not differ much in terms of resource usage (Figure 16).

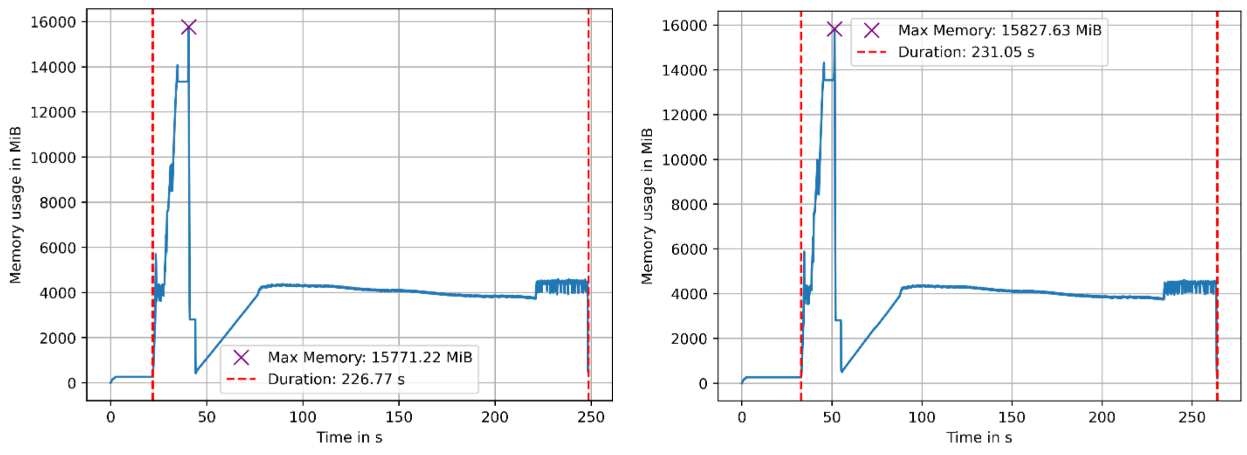


*Figure 16: Otsus binarization versus Simple thresholding.*

Both algorithms deliver good results in terms of cleaning up the WSI (Figure 17 & 18). Otsus binarization has been chosen as the default method on account of the automatic nature of the algorithm.

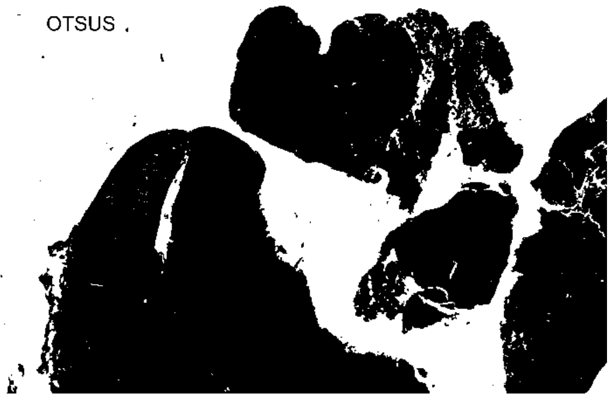


Figure 17: Mask created using Otsus binarization.

Simple thresholding requires the user to determine the optimal threshold to achieve the best results possible, if the value is chosen poorly, the result will be nigh unusable (Figure 18).

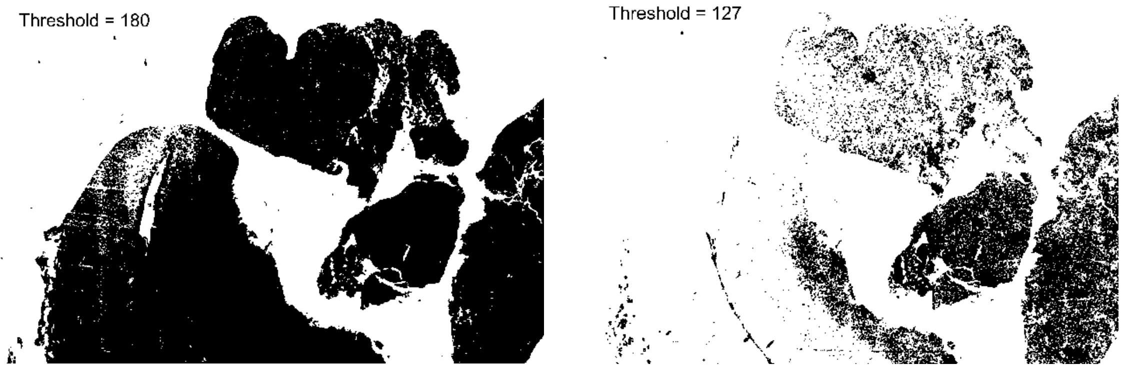


Figure 18:Simple thresholding with different values.

### 7.3.2 Adaptive thresholding

The third algorithm available in PIPET is Adaptive thresholding. Unlike global thresholding, where a single threshold value is applied to the entire image, adaptive thresholding computes different thresholds for different regions of the image. This should help in handling variations in illumination and contrast across the image. Unfortunately, while developing and testing the preprocessing module, Adaptive thresholding has remained almost unusable due to the time required to execute as well as the bad results (Figure 19).

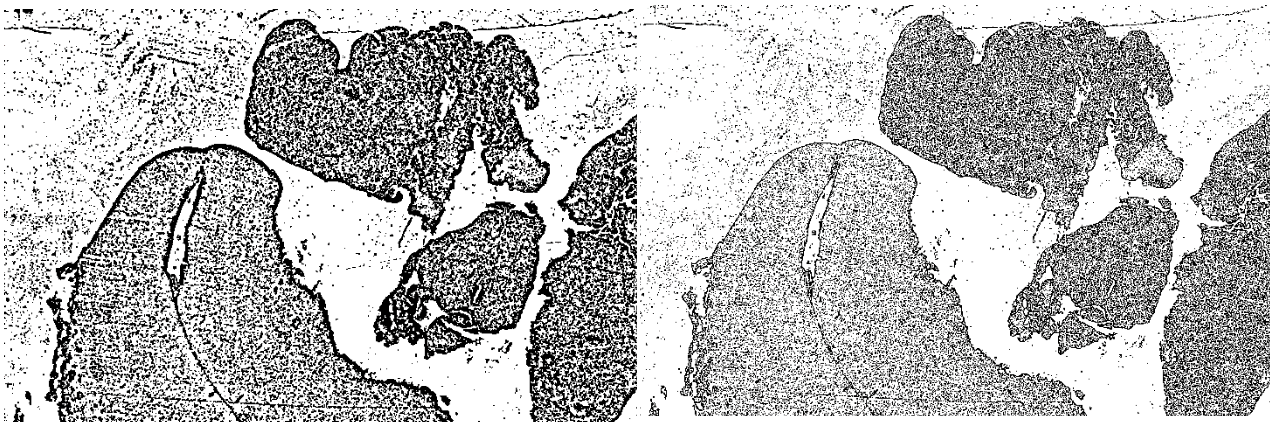


*Figure 19: Adaptive thresholding block size 1001 and 201.*

As is visible above, the results of the Adaptive thresholding are not ideal for the application while the execution time is also more than double of the other options available (Figure 20).

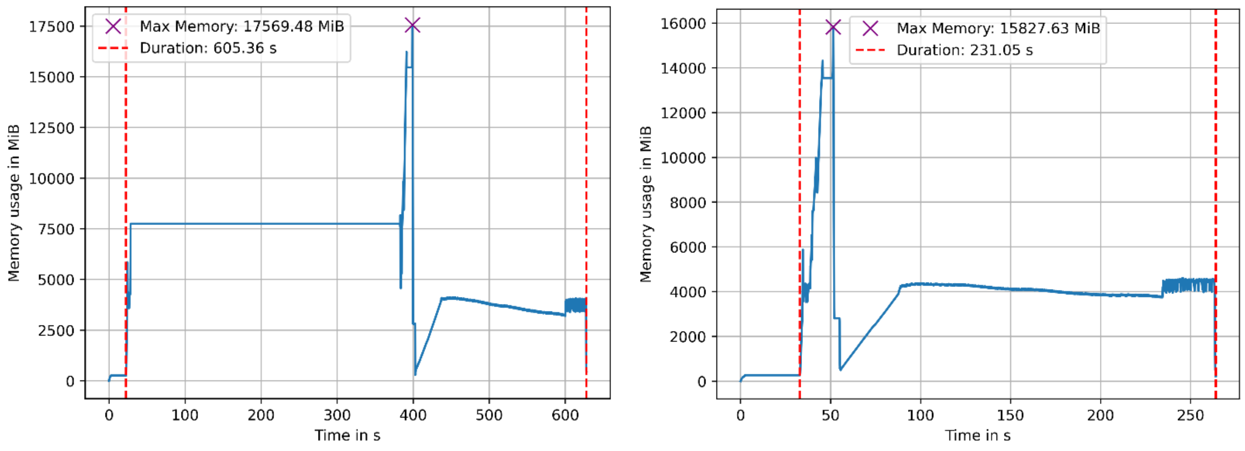


*Figure 20: Adaptive thresholding versus Simple thresholding*

Overall, Adaptive thresholding was not a good choice for PIPET's preprocessing module, the computation complexity of evaluating every pixel by calculating a local threshold from its neighbouring region is excessive for WSIs. The variability of the WSI poses another challenge to the algorithm, as is visible in Figure 19 the regions in the image that where white are being misinterpreted while regions in the tissue are falsely negated. Furthermore, choosing a block size for adaptive thresholding is harder than choosing a static threshold as with simple thresholding.

### 7.4 Interoperability

In section 3.3 it was stated that one goal is to maximize the interoperability so that PIPET can work with different ML-models and WSI file types. PIPET is currently capable of reading most image formats, in testing there was no format that threw an error. PIPET currently expects ML-models in the onnx format, as the focus of PIPET was on the development of the pipeline, no alternative models have been tested.

## 8.0 Conclusion

The Goals stated for this Thesis have been successfully completed and the optional objectives have been implemented as well. WSIs can be completely segmented with a given ML-model, tissue masks can be applied, and the resulting image will be a good representation of the input. Furthermore, a standalone version of PIPET is available for execution without setting up a python environment.

In the following, problems, and suggestions for the further development of PIPET are explored.

### 8.1 Problems

The most critical problems in PIPET's development have arisen due to data management and data structure issues. Firstly, in early stages of development, PIPET was setup in such a way that a slide class existed which needed to be instantiated with values that would be needed to execute the pipeline. This class was cluttered, and the function calls were messy. It also blocked any way to access singular functions calls from external code, which could be useful to users, for example, getting a list of slice objects and use that for training. Additionally, first iterations of PIPET had excessive memory usage due to unnecessary conversions and operations between libraries, causing issues even with scaled down images.

The aforementioned issue with the data structure proved the largest roadblock in the development of PIPET. ML-models expect input in very specific data structures called tensors, this input shape has been provided, but an issue arose when the ML-model output did not match the expected output. This caused a delay in development for debugging, which ultimately led to inspection of working example code utilizing this model and analyzing each variable in a debugger. The issue has been found to be a different function for resizing slices, PIPET used the pillow function *resize*() which resizes the image and keeps the normal RGB format. Meanwhile, the ML-model expected input generated by the *resize*() function from ScikitImage, which resizes the image but also normalizes the output to a value between 0 and 1. Once this has been identified, it was trivial to exchange the pillow function for the ScikitImage function.

A more minor issue, mostly due to less time constraints, arose due to the conversion between the supplied onnx-format model and PyTorch. In the early stages of development, it was attempted to use the onnx runtime to execute the ML-model, but this avenue of approach did not work. It was then tried to use onnx2pytorch to convert the model to PyTorch and use PyTorch to execute the model, which also resulted in many errors for which no fix could be found, in the end, it was found the appropriate package for such a conversion to be onnx2torch, which worked flawlessly.

## 8.2 Recommendation for WSI loading

As mentioned previously in section 7.2 the current WSI loading algorithm can be improved. Currently, PIPET will slice the entire WSI into slices and keep those in memory, with relatively minor modifications to the code it should be possible to reduce memory usage of PIPET without tissue masking by a large margin.

```
slice_list = PIPET.slice_slide(slice_positions, slide, slice_width, slice_height)
PIPET.close_slide(slide)
vips_output = pyvips.Image.black(slide_dimensions[0], slide_dimensions[1])
index1 = 0
index2 = 0

for slices in slice_list:
    index1, index2, evaluated_slice = PIPET.run_inference(slices, pytorch_model, ml_input_width, ml_input_height, index1, index2)
    vips_output = PIPET.stitch_slide(evaluated_slice, vips_output)

PIPET.save_slide(vips_output, output_path)
```

*Code block 11: Current pipeline*

With changing how the slicing function works, we can expect a great reduction in memory usage as visible in Figure 21.

```
slice_positions = PIPET.define_slices(slide, slice_height, slice_width)
vips_output = pyvips.Image.black(slide_dimensions[0], slide_dimensions[1])
index1 = 0
index2 = 0

for slice_position in slice_positions:
    slice = PIPET.slice_slide(slice_position, slide, slice_width, slice_height)
    index1, index2, evaluated_slice = PIPET.run_inference(slice, pytorch_model, ml_input_width, ml_input_height,index1, index2)
    vips_output = PIPET.stitch_slide(evaluated_slice, vips_output)

PIPET.save_slide(vips_output, output_path)
```

*Code block 12: Proposed pipeline.*

```python
def slice_slide(slice_position, slide, slice_width, slice_height):

        print("Slicing " + f'{slice_position[0]}' + ':' + f'{slice_position[1]}')
        slice = slide.read_region(slice_position, 0, (slice_width, slice_height))
        temp_slice = Slice(slice, slice_position, slice_width, slice_height)

    return temp_slice
```
*Code block 13: Proposed slicer*

As is visible in the two code blocks above, the change is going from slicing the entire WSI at once to one slice at a time, resulting in the memory usage reduction in Figure 21.

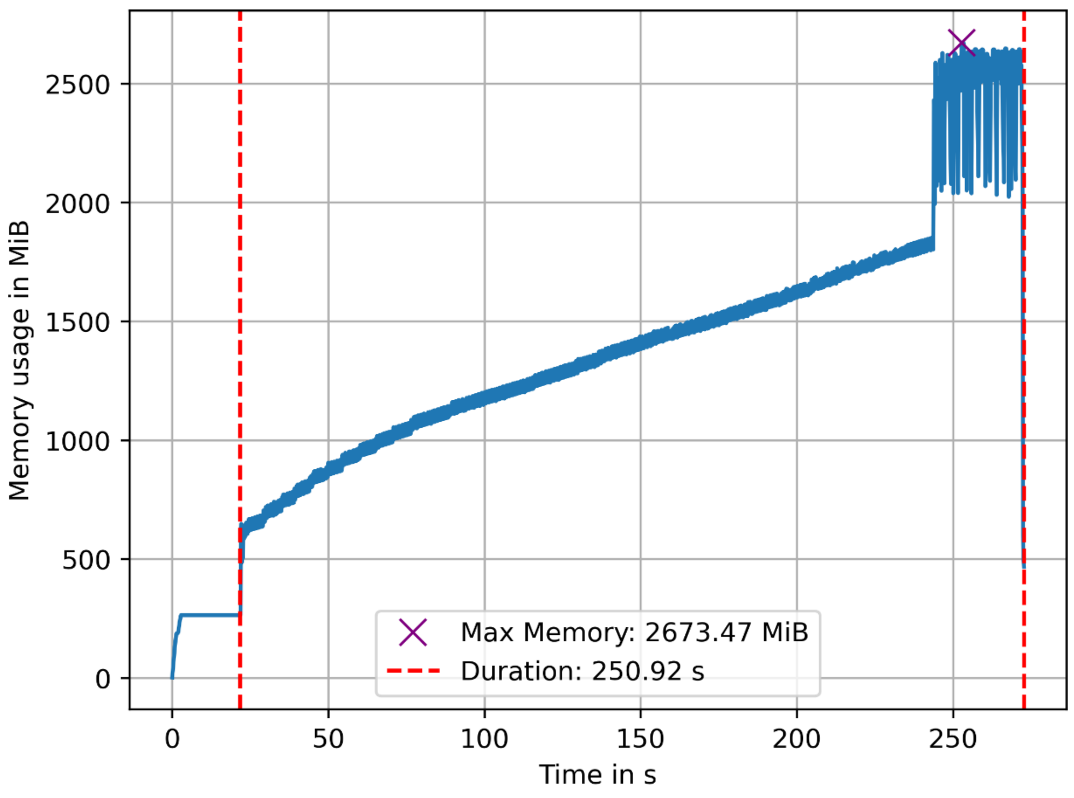


*Figure 21: Impact on memory Consumption with Proposed changes (Figure8.1 used).*

A version of PIPET containing these changes will be supplied with this thesis, regrettably this opportunity was only discovered after completion of most of the Thesis and as such it was decided to include them in this section instead of reworking section 5 and section 7 due to time constraints.

## 8.3 Recommendations for processing

To enable further compatibility with different ML-models, the *run_inference*() function should be extended with a flag to change from normalized image values to standard RGB image values, enabling the usage of ML-models trained with this data type. Additionally, an option to specify the input shape for models could be implemented as well.

### 8.3.1 Model detection

It would be useful to implement functionality to detect the filetype of ML-models, enabling dynamic conversion between models, enabling a wider range of models to be used with PIPET.

### 8.3.2 Finetuning for filtering functions

The filtering and noise reduction functions currently rely on values that have been chosen without exhaustive testing. These values could be tuned to improve results even further. Additionally, the evaluation function from the slice class could be made less restrictive, allowing a threshold to be set on how much of an image should contain data before it is evaluated.

### 8.3.3 Type Hinting

The *segment_slide*() function expects 10 arguments to be passed in its current form, if the previous recommendations are implemented this number would increase even further. To allow for an easier time calling this function, the functions should implement type hinting.

[1] N. P. Jouppi *et al.*, ‘In-Datacenter Performance Analysis of a Tensor Processing Unit’, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, in ISCA ’17. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–12. doi: 10.1145/3079856.3080246.

[2] I. Kononenko, ‘Machine learning for medical diagnosis: history, state of the art and perspective’, *Artif. Intell. Med.*, vol. 23, no. 1, pp. 89–109, Aug. 2001, doi: 10.1016/S0933-3657(01)00077-X.

[3] ‘900-world-fact-sheet.pdf’. Accessed: Feb. 27, 2024. [Online]. Available: https://gco.iarc.who.int/media/globocan/factsheets/populations/900-world-fact-sheet.pdf

[4] M. J. Iqbal *et al.*, ‘Clinical applications of artificial intelligence and machine learning in cancer diagnosis: looking into the future’, *Cancer Cell Int.*, vol. 21, no. 1, p. 270, May 2021, doi: 10.1186/s12935-021-01981-1.

[5] ‘ISO 12639:2004(en), Graphic technology — Prepress digital data exchange — Tag image file format for image technology (TIFF/IT)’. Accessed: Mar. 19, 2024. [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:iso:12639:ed-2:v1:en

[6] A. Goode, B. Gilbert, J. Harkes, D. Jukic, and M. Satyanarayanan, ‘OpenSlide: A vendor-neutral software foundation for digital pathology’, *J. Pathol. Inform.*, vol. 4, no. 1, p. 27, Jan. 2013, doi: 10.4103/2153-3539.119005.

[7] ‘3.11.8 Documentation’. Accessed: Feb. 27, 2024. [Online]. Available: https://docs.python.org/3.11/

[8] ‘Installing Python Modules’, Python documentation. Accessed: Feb. 27, 2024. [Online]. Available: https://docs.python.org/3/installing/index.html

[9] ‘OpenCV: Introduction’. Accessed: Feb. 27, 2024. [Online]. Available: https://docs.opencv.org/4.x/d1/dfb/intro.html

[10] ‘Pillow’, Pillow (PIL Fork). Accessed: Feb. 27, 2024. [Online]. Available: https://pillow.readthedocs.io/en/stable/index.html

[11] ‘Image Module’, Pillow (PIL Fork). Accessed: Feb. 27, 2024. [Online]. Available: https://pillow.readthedocs.io/en/stable/reference/Image.html

[12] C. R. Harris *et al.*, ‘Array programming with NumPy’, *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.

[13] A. Paszke *et al.*, ‘PyTorch: An Imperative Style, High-Performance Deep Learning Library’, in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019. Accessed: Mar. 19, 2024. [Online]. Available: https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[14] M. Muñoz-Aguirre, V. F. Ntasis, S. Rojas, and R. Guigó, ‘PyHIST: A Histological Image Segmentation Tool’, *PLOS Comput. Biol.*, vol. 16, no. 10, p. e1008349, Oct. 2020, doi: 10.1371/journal.pcbi.1008349.

[15] M. M. Aguirre, ‘manuel-munoz-aguirre/PyHIST’. Mar. 07, 2024. Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/manuel-munoz-aguirre/PyHIST

[16] ‘end2end-WSI-preprocessing/scripts at main · KatherLab/end2end-WSI-preprocessing’, GitHub. Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/KatherLab/end2end-WSI-preprocessing/tree/main/scripts

[17] ‘GitHub - smujiang/WSITools: Tools for whole slide image (WSI) processing. Especially for (pairwise) patch extraction, annotation parsing and data

preparation for deep learning purposes.’ Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/smujiang/WSITools

[18] S. N and V. S, ‘Image Segmentation By Using Thresholding Techniques For Medical Images’, *Comput. Sci. Eng. Int. J.*, vol. 6, no. 1, pp. 1–13, Feb. 2016, doi: 10.5121/cseij.2016.6101.

[19] N. Otsu, ‘A Threshold Selection Method from Gray-Level Histograms’, *IEEE Trans. Syst. Man Cybern.*, vol. 9, no. 1, pp. 62–66, Jan. 1979, doi: 10.1109/TSMC.1979.4310076.

[20] ‘GDC Data Portal. ‘ Accessed: Mar. 28, 2024. [Online]. Available: https://portal.gdc.cancer.gov/
UUID: f0fedcbc-9120-4494-bd1a-fd2773acc964
UUID: 83328170-e482-48b7-b972-fe82423434eb
UUID: a7d616f5-7e59-4471-a6a8-008f4eb26b4d [DISCARDED]