Preparation of Whole Slide Images for usage in Neural Networks

Master Thesis

Submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.)
in Applied Computer Science

at the

Berlin University of Applied Sciences (HTW)

First Supervisor: Prof. Dr. Peter Hufnagl

Second Supervisor: Diplom Informatiker Benjaming Voigt

Submitted by:
Sascha Nawrot (B.Sc.)

Berlin, May 24, 2016

Preface

Hello, this is the preface

Abstract

This is the abstract.

Contents

Chapter 1

Introduction

1.1 Motivation

The medical discipline of pathology is in a digital transformation. Instead of looking at tissue samples through the means of traditional light microscopes it now is possible to digitalize those samples. This digitalization is done with the help of a so called slide scanner. The result of such an operation is a *whole slide image* (WSI) [3]. The digital nature of WSIs opens the door to the realm of image processesing and analysis which yields certain benefits, such as the use of image segmentation and registration methods to support the pathologist in his/her work. A very promising area of image analysis are the so called *neural networks*[1], which follow a machine learning approach. Their use, especially in the area of image classification and object recognition, made big breakthroughs possible in the recent past, e.g. Karpathy and Fei-Fei [2], who created a neural network which is capable of describing an image or a scene with written text (see fig. 1.1 for some examples).

There is enormous potential in the use of neural networks in the digital pathology as well, but to transfer these algorithms and technologies, certain hindrances must be overcome. One of those hindrances is the need for proper training samples and an established ground truth[2]. While generally there are huge amounts of WSIs, most of them won't be usable without further preparation as a training sample. One way of preparing them are annotations. These can be added to the WSIs, stored and later used for training.

Therefore the goal of this thesis is to give tools into the hands of pathologists to annotate whole slide images and save those annotations in such a way that they will be usable later in the combination with neural networks.

[1] See chapter 2.1.3

[2] *Ground truth* refers to a training sample whose outcome is known and validated as true.

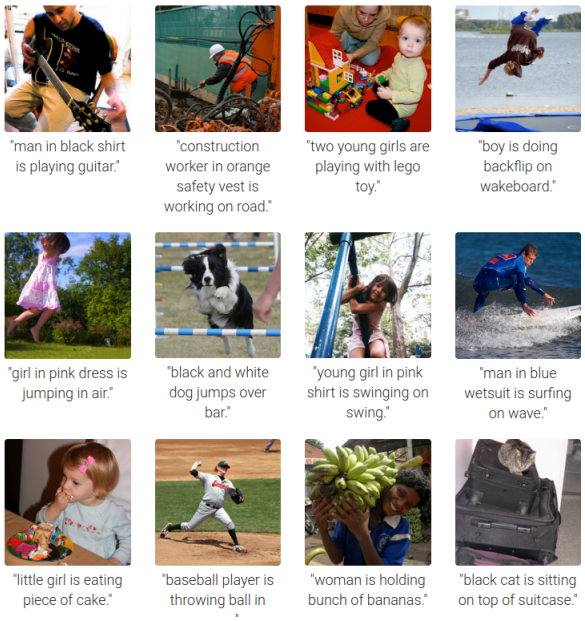

Figure 1.1: Example results of the in [2] introduced model (source: http://cs.stanford.edu/people/karpathy/deepimagesent/

1.2 Research Objective

The objective of this thesis is the conceptualization and implementation of tools for WSIs, which allow for their annotation and a further usage in neural networks. As a requirement, the tools have to be implemented in the form of microservices[3], each one with their own short documentation, including instructions for installation, usage and some examplary use cases. To achieve this goal, the implementation of 3 microservices is necessary.

The first microservice needs to be capable of converting a given set of image formats into the so called *Deep Zoom Image Format*[4]. The supported image formats are *.bif, .mrxs, .ndpi, .scn, .svs, .svslide, .tif, .tiff, .vms* and *.vmu*, in accordance with the capabilities of the Openslide framework [4].

The second microservices task is to give a tool at hand with which a pathologist will be able to annotate all those WSIs which were converted using the first microservice. Furthermore, the made annotations need to be persisted together with the highest resolution of the corresponding image.

The third microservice will be responsible for preparing the annotated WSIs for the further usage in neural networks. For that purpose the service needs to

[3]See chapter 2.1.2
[4]See chapter 2.1.1

be capable of dividing a single annotated whole slide image into multiple tiles, with the choice of either using the whole image or just the annotated areas.

1.3 About this thesis

Apart from the *Introduction*, there are 5 more chapters in this thesis.

Chapter 2 - Background defines some terminoligy and the general, required process chain which are all necessary to understand further chapters of this thesis. Furthermore, 3 microservices will be introduced in short.

Chapter 3 - Methodology gives an overview over the current state of research for each microservice, as well as best practices.

Chapter 4 - Implementation goes into further details about how each microservice is implemented and which software and frameworks were used for that.

Chapter 5 - Discussion will introduce a measurement for each microservice to measure its success. It will discuss the test setup as well as list the results.

Chapter 6 - Conclusion will interpret the Results from Chapter 5 and analyze them closer. Furthermore, it will give an idea of what steps are to be taken next in the future.

Chapter 2

Background

2.1 Definition of terms

To prevent missunderstandings and confusion, the following subsections 2.1.1 - 2.1.3 will define some terminoligy which will be mandatory for the understanding of certain areas of this thesis.

2.1.1 Deep Zoom Image Format

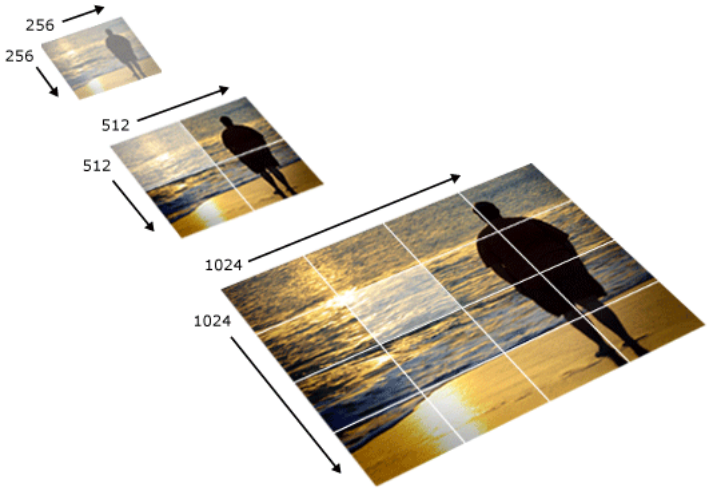

Figure 2.1: 3 consecutive levels of a dzi image (source: https://i-msdn.sec.s-msft.com/dynimg/IC141135.png)

The Deep Zoom Image Format (dzi) is an xml-based file format maintained by Microsoft to improve performance and quality in the handling of large image

files. For this purpose an image is represented in a tiled pyramid (see fig. 2.1).

As seen in fig. 2.1 there are multiple versions of a single image in different resolutions. Each resolution in the pyramid is called a *level*. At each level the image is scaled down by the factor 4 (2 in each dimension). Furthermore, the image gets tiled up into 256^2 tiles (256 in each dimension) [10].

If a viewer wants to view a certain area of the image (e.g. the highlighted tile in the last image in fig. 2.1), only the corresponding tiles need to be loaded. This saves large amounts of bandwidth and memory. The same goes for a viewer, who is zoomed out very far. In such a view the full level of detail isn't needed, so that a version from a lower level can be loaded.

A dzi file consists of two parts: a describing .xml file[1] and a folder with more subfolder. Each subfolder describes a level and as such contains all the tiles for that particular level.

2.1.2 Microservice

The concept of microservices is to seperate one monolithic software construct into several smaller, modular pieces of software. According to [12], the idea of microservices is not new, but can be found in the UNIX philosophy. Three basic ideas are stated in [12]:

- A program should fulfill only one task, and it should do it well.
- Programs should be able to work together.
- Besides, the programs should use a universal interface.

As such, microservices are a modularization concept. However, they differ from other concepts, since they are independet from each other. This is a trait, other modularization concepts usually lack. As a result, changes in one microservice don't bring up the necessity of deploying the whole product again but just the one service.

Because of their inherent traits, microservices need to be their own processes in one way or another, may it be as an actual operating system process or as e.g. a docker container[2].

One big advantage of this modularization is that each service can be written in a different programming language, using different frameworks and tools. Furthermore, each microservice can bring along its own support services and data storages, like data bases. It is imperative for the concept of modularization, however, that each microservice has its own storage of which it is in charge of.

A disadvantage of this modularization is, that inter process communication becomes a necessity. However, there are different approaches with which microservices can communicate. [12] suggests the following:

[1] Frameworks like *OpenSeaDragon* also support further formats, such as .json.

[2] Docker is a tool, which enables software to be wrapped up in so called "'containers"'. Those containers are a complete, but stripped down, filesystem containing everything the software needs to run (e.g. source code, runtime environment, system tools and libraries, ...). See `https://www.docker.com/what-docker`

- communication via protocols like REST[3]
- an HTML user interfaces with links to other microservices
- data replication

It is important to define how and with which technology to communicate with, when adressing each microservices to ensure that this particular one can actually be reached with the defined method.

2.1.3 Neural Network

Artificial neural networks (NN) are a group of models inspiried by biological neural networks[4]. In a NN, regardless if artificial or biological, many neurons are interconnected with each other. The construct of interconnected neurons can be seperated into layers, of which there are three kinds:

- input layer
- hidden layer
- output layer

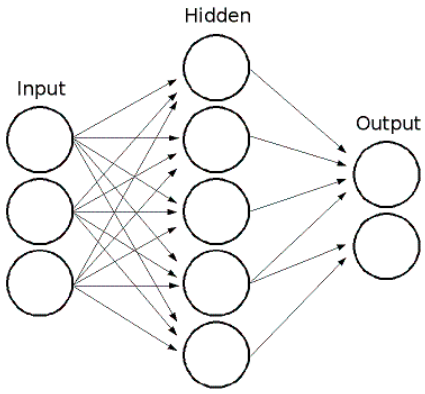

Figure 2.2: 3 layer NN (source: http://docs.opencv.org/2.4/_images/mlp.png)

Basically, the input layer, as the name suggests, is the layer where the NN gets its input data from. After that, there are a number of hidden layers[5],

[3]Representational state transfer (REST) is architectural style for distributed hypermedia systems, see [7].

[4]For the remainder of this thesis, neural network will always represent the artificial one, unless explicitly stated otherwise.

[5]A NN doesn't necessarily need to have any hidden layers. For non trivial problems however, it becomes mandatory.

which are responsible for further computation of the input values. At the end is the output layer which is responsible for communicating the results of the prior operations (compare fig. 2.2). Each single neuron has input values and an output value. Once the input reaches a certain trigger point, the cell in the neuron sends a signal as output.

A huge benefit of NN, over other software models, is their ability to learn. While certain problems are easier to solve in a sequential, algorithmic fashion (say an equation or the towers of hanoi), certain problems are so complex that new approaches are needed, while other problems can't be solved algorithmic at all. With the use of adequate training samples, a NN can train to solve a problem, not unlike a human, by learning. Since this topic alone is enough for a number of theses, the author refers to [9] for further detailed information.

2.2 Process Chain

This section and its following subsections (2.2.1 - 2.2.3) are dedicated to illustrate the process chain necessary to accomplish the research objective stated in chapter 1.2. The process chain consists of the following steps:

(a) convert WSIs of different[6] formats to dzi format

(b) annotate dzi images with the annotation tool made for this purpose

(c) persist made annotations in a file

(d) seperate annotated images into tiles of custom size

(e) keep correspondence between tiles of an image and its annotations

To fulfill those steps, 3 Microservices will be introduced in the following subsections. Those are:

- Conversion Service (see chap. 2.2.1)
  This service will be responsible for converting WSIs into the dzi format (a).
- Annotation Service (see chap. 2.2.2)
  This service will offer a tool to annotate an image (b) and persist made annotations (c).
- Tessellation Service (see chap. 2.2.3)
  This service will be responsible for seperating an image into tiles (d) and keep the correspondence between tiles and annotations (e).

[6] see chap. 2.2.1 for a listing of valid input formats.

2.2.1 Definition of Conversion Service

The devices which create an WSI, so called whole slide scanners, create images in various formats, depending on the producer. The conversion service has the goal of converting those formats into the dzi format, not only for the purpose of unification, but also to add the deep zoom feature[7] to the images (see fig. 2.3). This is of special importance, since an average WSI with 1,600 megapixels has a size of approximately 4.6 GB [6].

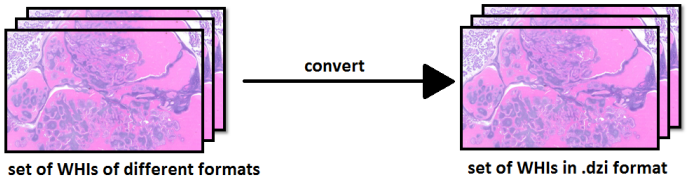

Figure 2.3: Visualization of the Conversion Service

The Conversion Service will be implemented as a python script, which upon calling takes every single WSI of a certain format inside a given folder and converts it to the dzi format. Each converted WSI will be saved in another specified folder. Valid image formats (and their corresponding producers) for conversion are:

- .bif (Ventana)
- .mrxs (Mirax)
- .ndpi (Hamamatsu)
- .scn (Leica)
- .svs (Aperio)
- .svslide (Sakura)
- .tif (Aperio, Trestle, Ventana)
- .tiff (Philips)
- .vms (Hamamatsu)
- .vmu (Hamamatsu)

[7] See chap. 2.1.1

2.2.2 Definition of Annotation Service

The first step to create a valid training sample for a NN is to annotate the WSIs which will later serve as that. To do so, a GUI must be deployed which enables a pathologist to make annotations to an WSI. Additionally, the Annotation Service also needs to be capable to persist made annotations (see fig. 2.4). This will happen by saving the annotations into a file which will be placed next to the image.

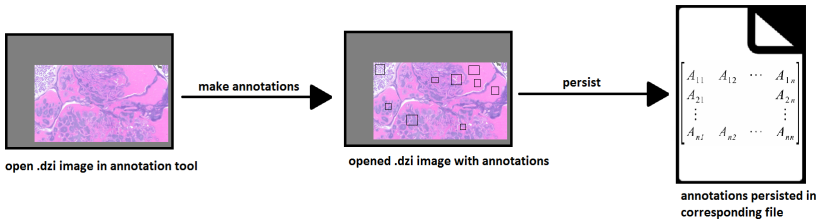

Figure 2.4: Visualization of the Annotation Service

To enable the pathologist to make annotations in the first place, a GUI needs to be offered by the service. This GUI will be developed in iterations with the help of a number of pathologists to adapt it to their wishes and grant the best possible usability. The basic concept of the first iteration will be based on the Microdraw[8] GUI (see fig. 2.5).

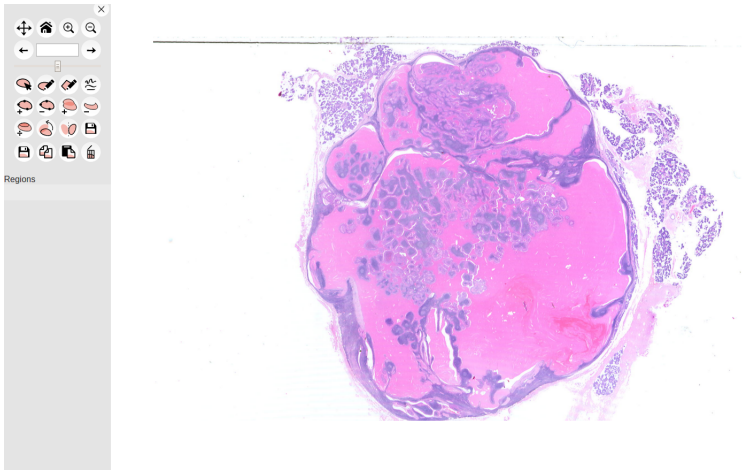

Figure 2.5: Microdraw GUI with opened WSI

[8]See `https://github.com/r03ert0/microdraw` for more information on the Microdraw project

Microdraw is a webbased annotation tool, which describes itself as

> "[...] a collaborative vectorial annotation tool for ultra high resolution data, such as that produced by high-throughput histology." [11]

Therefore, the GUI of the Annotation Service, or Annotation Service Viewer (ASV), will run as a web application in a browser. Annotations will be made by selecting a shape or annotation method from the various tools in the toolbar (see the gray bar on the left in fig. 2.5). After selecting the area to be annotated, an actual description of that area can be made via keyboard input.

When the pathologist is done or wants to save the made annotations, the Annotation Service will create a file in which they will be persisted in. Only one WSI can be opened in one ASV at a time.

2.2.3 Definition of Tessellation Service

The task of the Tessellation Service is to tessellate a given WSI into multiple tiles while keeping up the correspondence between image areas and annotations (see fig. 2.6). Tessellation describes the process of seperating a geometric space (e.g. a plane) into multiple tiles of one or more shapes. No matter if the tiles are unifrom or of different shapes, there must be no gaps or overlapping areas in the resulting tiles.

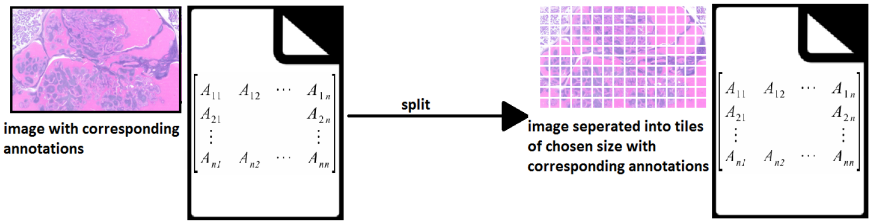

Figure 2.6: Visualization of the Tessellation Service

The tessellation of the Tessellation Service differs from the given definition in two aspects. First, all tiles will be of a unifrom shape (square). Second, the service will also offer the possibility of seperating only the annotated areas of an WSI into tiles. Therefore, the "no gaps" rule is invalid, when this option is chosen. The rule of no overlapping areas holds true in either case, however.

As mentioned before, the second task of the Tessellation Service is to ensure the correspondence between tiles and annotations of the original WSI. This means that, if the original WSI has an annotation in the area of a tile, this tile needs to have the same annotation.

Chapter 3

Conversion Service

3.1 Methodology

The objective of the Conversion Service is to convert a given set of input WHIs into dzi files. Since a dzi is layered into a pyramid scheme, it is necessary to calculate the needed number of levels, as well as the dimensions of each level (see fig. 3.1 for an example).

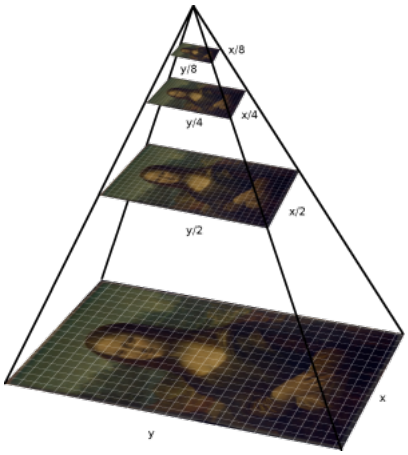

Figure 3.1: Example of a pyramid scheme in image processing (source: http://iipimage.sourceforge.net/images/pyramid.png)

Therefore, the Conversion Service must be able to open an WHI img_{input} of any of the in 2.2.1 defined formats. Based on the size of img_{input} the number of necessary levels lvl must be calculated. Once lvl has been determined, img_{input} must be resized into an appropriate scale for each lvl_i in lvl. The resized image will be called img_i, with i representing the corresponding level. In the next

step, every img_i will be tessellated into $x * y$ tiles. Each tile will be referrenced via $t^i_{c,r}$, with r being the row and c being the column of the tile in whi_i. To complete the conversion, the Conversion Service must create a describing xml file for each converted image img_{dzi}.

3.1.1 Creating a Deep Zoom Image

To create a dzi, the Conversion Service must be capable of all the afore mentioned tasks. According to [1], there is a number of frameworks, that can achieve this task (see tab. 3.1).

Since the conversion to dzi is required, two frameworks are not usable from the start. Those are MapTiler, who creates tms[1] images and Kakadu, which creates iiif[2] images. Furthermore, the various desktop applications are not usable either, since the Conversion Service should operate as a microservice.

For a python script, deepzoom.py would be the natural choice, however at a second glance it can be seen that VIPS offers "[...] for a number of languages" (see tab. 3.1). One of those languages happens to be python. Therefore, deepzoom.py and VIPS will be further investigated in the following two subsections.

[1] Tile Map Service (tms) is a tile scheme developed and maintained by the Open Source Geospatial Foundation [8]

[2] International Image Interoperability Framework (iiif), specified by the International Image Interoperability Framework group, is an image delivery API which responds to requests via HTTP and HTTPS [5]

option	**description**	**image format**
Deep Zoom Composer	dekstop app for Windows	dzi
Image Composite Editor	panoramic image stitcher from Microsoft Research for the Windows desktop	dzi
DeepZoomTools.dll	.NET-library, comes with Deep Zoom Composer	dzi
deepzoom.py	Python	dzi
deepzoom	Perl utility	dzi
PHP Deep Zoom Tools	PHP	dzi
Deepzoom	PHP	dzi
DZT	an image slicing library and tool written in Ruby	dzi
MapTiler	desktop app for Windows, Mac, Linux	tms
VIPS	command line tool and library for a number of languages	dzi (via dzsave feature)
Sharp	Node.js, uses VIPS	dzi
MagickSlicer	shell script (Linux/Max)	dzi
Gmap Uploader Tiler	C++	dzi
Node.js Deep Zoom Tools	Node.js, under construction	dzi
OpenSeaDragon DZI Online Composer	Web app (and PERL and PHP scripts)	dzi
Zoomable	service, offers embeds; no explicit API	dzi
ZoomHub	service, under construction	dzi
Kakadu	C++ library to encode or decode JPEG 2000 images	iiif
PyramidIO	Java (command line and library)	dzi

Table 3.1: Overview of conversion options for zooming image formats (source: [1])

3.1.2 Deepzoom.py

Deepzoom.py[3] is a python script and part of Open Zoom[4]. It can either be called directly over a terminal or imported as a module in another python script. The conversion procedure itself is analogous for both methods.

If run in a terminal the call looks like the following:

```
$ python deepzoom.py [options] [input file]
```

The various options and their default values can be seen in tab. 3.2. If called without a designated output destination, deepzoom.py will save the converted dzi right next to the original input file.

option	description	default
-h	show help dialog	-
-d	output destination	-
-s	size of the tiles in pixels	254
-f	image format of the tiles	jpg
-o	overlap of the tiles in pixels (0 - 10)	1
-q	quality of the output image (0.0 - 1.0)	0.8
-r	type of resize filter	antialias

Table 3.2: Options for deepzoom.py

The resize filter is applied to interpolate the pixels of the image when changing its size for the different levels. Supported filters are:

- cubic
- biliniear
- bicubic
- nearest
- antialias

When used as module in another python script, deepzoom.py can simply be imported via the usual *import* command. To actually use deepzoom.py, a Deep Zoom Image Creator needs to be created. This class will manage the conversion process:

```
# Create Deep Zoom Image Creator
creator = deepzoom.ImageCreator(tile_size=[size],
  tile_overlap=[overlap], tile_format=[format],
  image_quality=[quality], resize_filter=[filter])
```

[3] see `https://github.com/openzoom/deepzoom.py` for further details
[4] see `https://github.com/openzoom` for further details

The options are analogous with the terminal version (compare tab. 3.2). To start the conversion process, the following call must be made within the python script:

```
# Create Deep Zoom image pyramid from source
creator.create([source], [destination])
```

Upon calling, the ImageCreator opens the input image img_{input} and creates a description with all the needed information for the dzis describing xml file[5]. After that, the number of levels is calculated. For this, the bigger value of height and width of img_{input} is chosen (see eq. 3.1) and then used to determine the number of levels lvl (see eq. 3.2).

$$max\_dim = max(height, width) \tag{3.1}$$

$$lvl = \lceil log_2(max\_dim) + 1 \rceil \tag{3.2}$$

Once lvl has been determined, img_{input} will be resized in the chosen quality (-q/image_quality) for every level i, with $i \in (0, lvl - 1)$. The new resolution will be calculated for both dimensions dim with a function $scale$ (see eq. 3.3) analogously. Furthermore, the image will be interpolated with the specified filter (-r/resize_filter). The resized image will be called img_i.

$$scale = \lceil dim * 0.5^{lvl-i} \rceil \tag{3.3}$$

Once img_i has been created, it will be tessellated into as many tiles of the specified size (-s/tile_size) and with the specified overlap (-o/tile_overlap) as possible. Since not every image will be of the size $2^n, n \in$ in either dimension, it is highly likely that the set of tiles for the last column/row will be smaller then specified in either dimension.

Every tile will be saved as [column]_[row].[format] ([format] depending on -f/file_format) in a folder called according to the corresponding level i. This folder will be located inside another folder, called [filename]_files. The describing xml file will be persisted as [filename].dzi on the same level.

3.1.3 VIPS

3.2 Implementation

As stated before, the Conversion Service is implemented as a python script. The first iteration of the script used deepzoom.py for the conversion. This caused severe performance issues though. Out of all the image files in the test set[6], only one could be converted[7]. Every other file was either too big, so the process would eventually be killed by the operating system or exit with an

[5] compare chap. 2.1.1
[6] see chap. 3.3
[7] CMU-3.svs from aperio, see chap. 3.3

IOError concerning the input file from the PIL imaging library. The second iteration of the Conversion Service uses VIPS, which is not only a lot faster then deepzoom.py, but also capable of converting all the given test images.

The script has to be called inside a terminal in the following fashion:

```
$ python ConversionService.py [input dir] [output dir]
```

It is mandatory to specify the input and output directory in the call, so the script knows where to look for images up to conversion and where to save the resulting dzi files.

Upon calling, the *main()* routine will be started, which orchestrates the whole conversion process. It looks as follows:

```
def main():
  path = checkParams()
  files = os.listdir(path)
  for file in files:
    print("----------------------------------------------------")
    extLen = getFileExt(file)
    if(extLen != 0):
      print("converting " + file + "...")
      convert(path, file, extLen)
      print("done!")
```

checkParams() checks if the input parameters are valid and, if so, returns the path to the specified folder or exit otherwise. Furthermore, it will create the specified output folder, if it doesn't exist already. In the next step, the specified input folder will be checked for its content. *getFileExt(file)* looks up the extension of each contained file and will either return the length of the files extension or 0 otherwise. Each valid file will then be converted with the *convert(...)* function:

```
# convert image source into .dzi format
# param path: directory of param file
# param file: file to be converted
# param extLen: length of file extension
def convert(path, file, extLen):
  dzi = OUTPUT + file[:extLen] + "dzi"
  im = Vips.Image.new_from_file(path + file)
  im.dzsave(dzi, overlap=OVERLAP, tile_size=TILESIZE)
```

The name for the new dzi file will be created from the original file name, however, the former extension will be replaced by "dzi" (see line 6). *OUTPUT* specifies the output directory which the file will be saved to. Next, the image file will be opened with Vips' Image class. Afterwards, *dzsave(...)* will be called, which handles the actual conversion into the dzi file format. *OVERLAP* and *TILESIZE* are global variables which describe the overlap of the tiles and their respective size. Their values are 0 (*OVERLAP*) and 256 (*TILESIZE*). The output will be saved to the same folder as ConversionService.py is operating from, plus "/dzi/[*OUTPUT*]/".

3.3 Test

To test the correct functionality of the Conversion Service, an automated test will be deployed. The test will be in the form of a bash script (ConversionTest.sh), which calls ConversionService.py in a controlled evironment with a specific set of WSIs dedicated for testing purposes and verifies that the conversion succeeded.

3.3.1 Setup

The only requirement for this test is that a specified test directory (*imgTest*, the test itself (*ConversionTest.sh*) and *ConversionService.py* are located in the same directory.

If called, the conversion test will create a specific folder (*csTest*), in which the testing will be done. Furthermore, a set of images from imgTest will be copied into the same folder. Afterwards, ConversionService.py is called, with *imgTest/* as input directory. After ConversionService.py is done, the test will check if the output directory was created, as well as every image was converted successfully. The test can be considered a success, if both cases are given.

3.3.2 Result

Chapter 4

Annotation Service

4.1 Methodology

4.1.1 DICOM

4.2 Implementation

4.3 Test

4.3.1 Setup

4.3.2 Result

Chapter 5

Tessellation Service

5.1 Methodology

5.2 Implementation

5.3 Test

5.3.1 Setup

5.3.2 Result

Chapter 6

Conclusion

6.1 Results

6.2 Conclusion

6.3 Future tasks

Bibliography

[1] Openseadragon. http://openseadragon.github.io/examples/creating-zooming-images/. Accessed: 26.04.2016.

[2] L. Fei-Fei A. Karpathy. *Deep Visual-Semantic Alignments for Generating Image Descriptions*. Department of Computer Science, Standford University, 2015. http://cs.stanford.edu/people/karpathy/cvpr2015.pdf.

[3] T. Cornish. An introduction to digital whole slide imaging and whole slide image analysis. http://www.hopkinsmedicine.org/mcp/PHENOCORE/CoursePDFs/2013/13%2019%20Cornish%20Digital%20Path.pdf, July 2013. Accessed: 12.04.2016.

[4] A. Goode et al. Openslide: A vendor-neutral software foundation for digital pathology. *J pathol inform*, 4(1), September 2013. http://download.openslide.org/docs/JPatholInform_2013_4_1_27_119005.pdf.

[5] M. Appleby et al. Iiif image api 2.0. http://iiif.io/api/image/2.0/#status-of-this-document. Accessed: 26.04.2016.

[6] N. Farahanil et al. Whole slide imaging in pathology: advantages, limitations, and emerging perspectives. *Pathology and Laboratory Medicine International*, 7, June 2015. https://www.dovepress.com/whole-slide-imaging-in-pathology-advantages-limitations-and-emerging-p-peer-reviewed-fu
ref10.

[7] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of Carolina, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[8] Open Source Geospatial Foundation. Tile map service specification. http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification. Accessed: 26.04.2016.

[9] D. Kriesel. *A Brief Introduction to Neural Networks*, 2007. http://www.dkriesel.com/en/science/neural_networks.

[10] Microsoft. Deep zoom file format overview. https://msdn.microsoft.com/en-us/library/cc645077(v=vs.95).aspx. Accessed: 11.04.2016.

[11] Roberto Toro. Microdraw. https://github.com/r03ert0/microdraw/wiki. Accessed: 18.04.2016.

[12] E. Wolff. *Microservices Primer*. innoQ, January 2016. https://leanpub.com/microservices-primer/read.

List of Figures

List of Tables