

High Performance Data Processing Pipeline for Connectome Segmentation

by

Wiktor Jakubiuk

Submitted to the Department of Electrical Engineering and Computer
Science

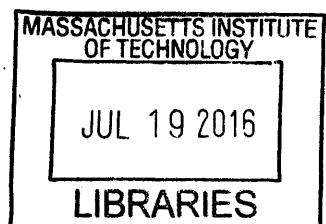
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2015 [February 2016]



© Massachusetts Institute of Technology 2015. All rights reserved. **ARCHIVES**

Signature redacted

Author

Department of Electrical Engineering and Computer Science

Dec 18, 2015

Signature redacted

Certified by

Nir Shavit

Professor

Thesis Supervisor

Signature redacted

Accepted by

Dr. Christopher J. Terman

Chairman, Masters of Engineering Thesis Committee

High Performance Data Processing Pipeline for Connectome Segmentation

by

Wiktor Jakubiuk

Submitted to the Department of Electrical Engineering and Computer Science
on Dec 18, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

By investigating neural connections, neuroscientists try to understand the brain and reconstruct its connectome. Automated connectome reconstruction from high resolution electron microscopy is a challenging problem, as *all* neurons and synapses in a volume have to be detected. A mm³ of a high-resolution brain tissue takes roughly a petabyte of space that the state-of-the-art pipelines are unable to process to date.

A high-performance, fully automated image processing pipeline is proposed. Using a combination of image processing and machine learning algorithms (convolutional neural networks and random forests), the pipeline constructs a 3-dimensional connectome from 2-dimensional cross-sections of a mammal's brain. The proposed system achieves a low error rate (comparable with the state-of-the-art) and is capable of processing volumes of 100's of gigabytes in size.

The main contributions of this thesis are multiple algorithmic techniques for 2-dimensional pixel classification of varying accuracy and speed trade-off, as well as a fast object segmentation algorithm. The majority of the system is parallelized for multi-core machines, and with minor additional modification is expected to work in a distributed setting.

Thesis Supervisor: Nir Shavit
Title: Professor

Acknowledgments

First, I would like to thank my advisor and project supervisor Nir Shavit. I began working with Nir in 2011 in the Multiprocessor Algorithmics Group on a concurrent data structure and then transitioned to the Connectomics project. It was not only an honor to work with such an accomplished professor, but also a great inspiration and a lot of fun. The doors to his office were always open, and regardless of the research problem difficulty, he asked the right questions and pointed me towards a better direction. I always left his office feeling inspired and motivated.

The project would not progress without my co-supervisor Yaron Meirovitch. Even though we only worked together for nine months, it felt like I completed a PhD-long program in natural sciences and mathematics. I am humbled by his relentless desire to understand and advance neuroscience. Our pair programming sessions together felt like the most productive hours of my life, and never before had I experienced anything near this level working with anyone else. Had we had more opportunity to work together, I have no doubt we would move many mountains. I am forever grateful for his direct and constructive feedback and if I do continue working in natural sciences, it will be solely due to him. Working with Yaron was always enjoyable, and while I might not have won in ping-pong too many times, the introduction to the Israeli culture and language more than compensated for this. Toda and mazel tov!

My code would not be nearly as fast if a day did not go by without Alex Matveev asking me what the speed up was. Alex is a world-class expert in multi-core performance engineering, and during our slow coffee breaks in the zula taught me tremendous amount about designing fast data structures. It was an extremely humbling experience to see Alex go from an idea to a super-fast prototype in a heartbeat. Truly, the mythical 10x developer.

I would also like to thank Adi Peleg who first gave me a hands-on introduction to the Connectomics project and provided guidance on the MapRecurse framework. I probably would not be writing this thesis, if Charles Leiserson did not first inspire me with multi core processors through his class 6.172 at MIT. My undergraduate adviser

Costantinos Daskalakis was always there for me, always cheerful and happy.

Last but not least, I would like to thank my life-long teachers, my parents. Thank you for the inspiration in the early years, and for the total freedom to pursue my dreams in the later years. I hope I have not disappointed. Lastly, thank you for my grandmother who flashed the first spark in the ways of algebra.

Contents

1	Introduction	13
1.1	Early Neuroscience	14
1.2	Brain Structure	15
1.3	Connectome	16
1.4	Imaging at High Resolution	17
1.4.1	Scanning Techniques	18
1.4.2	Our Microscope	18
1.5	Data Analysis	19
1.6	Contributions	20
2	Related work	23
2.1	Ilastik	24
2.2	GALA	24
2.3	Neuroproof	25
2.4	RhoANA	25
3	Our Segmentation Pipeline	27
3.1	Initial Pipeline	29
3.2	Modified Pipeline	30
4	Ground Truth Transformation	33
5	Probability Map Generation	37
5.1	Random Forest	37

5.1.1	Ilastik	38
5.1.2	Structured Decision Forest	39
5.2	Convolutional Neural Networks	39
5.2.1	Overview of Deep Neural Networks	40
5.2.2	Frameworks	43
5.2.3	Networks	44
5.2.4	Sliding Window Classification	46
5.2.5	Fully Convolutional Networks	47
5.3	Combined RF and CNN	50
5.4	Training Sets	51
6	Oversegmentation	53
6.1	Watershed	53
6.1.1	GALA's Implementation	55
6.1.2	OpenCV's Implementation	56
6.1.3	Serial Implementation	56
6.1.4	Parallel Watershed	58
6.2	Alternative Approaches	58
6.3	Other Heuristics	59
7	Agglomeration	61
8	Skeletonization	63
9	MapRecurse	65
9.1	Other Frameworks	66
10	Evaluation	69
11	Summary and Outlook	71
A	Nervous System of <i>Caenorhabditis Elegans</i>	73
B	Visualization of Convolutional Neural Networks	75

List of Figures

1-1	Left: an example of a phrenological map. Right: Phineas Gage's skull.	14
1-2	Ramon y Cajal's drawings. Left: a cell from pigeon cerebellum. Right: a chick cerebellum.	15
1-3	Left: Flatau's atlas. Right: Brodmann's areas	15
1-4	Neuron cell	16
1-5	Scanning techniques (source: [45])	19
1-6	Beams of our microscope (source: [23])	20
1-7	Output grid of our microscope (source: [23])	21
3-1	Simplified pipeline	28
3-2	Pipeline stages. Left to right: ram EM image, probability map, over-segmentation, segmentation.	29
3-3	Initial pipeline	30
3-4	Modified pipeline	31
4-1	1: Labels, 2: thick boundary ground truth, 3: thin boundary ground truth 4: double boundary ground truth, 5: double narrow boundary ground truth	34
5-1	Decision tree, decision forest	38
5-2	Deep neural networks	40
5-3	CNN	41
5-4	The sliding window approach	47

5-5 Overlapping kernels (green - max-pooling, blue - convolutional) in neighboring sliding windows.	48
6-1 Watershed flooding	54
6-2 Left: input probability map. Right: Generated watershed oversegmentation	55
6-3 Structuring element for z-closing	59
7-1 Left: Oversegmentation. Right: Segmentation	61
8-1 An example of a skeletonized object	64
9-1 MapRecurse framework. Top: map phase. Bottom: reduce phase . . .	66
9-2 Neuroproof's Spark	67
9-3 Tensorflow	68
A-1 C. Elegan's nervous system	74
B-1 Left: N3, right: AlexNet	80
B-2 Left to right: ShortNet1, ShortNet2, ShortNet3	81
B-3 Left: BN1, Right: BN2	82

List of Tables

5.1	Network Evaluation	46
5.2	Theoretical speedup	50
5.3	Training Data Sets	52
10.1	Variation of Information	70



Chapter 1

Introduction

The study of the nervous system, in particular of the central nervous system, has been an endeavor for millennia. Yet, despite significant progress in understanding the anatomy of other body organs, the structure and dynamics of the mammalian brain remain largely a mystery. Since the time the Ancient Greeks first located sensations in the brain, philosophers and scientists have been trying to dissect it to find the soul, understand the human's behavior, and cure disease.

Through dualism, Descartes suggested that sensations cause the "animal spirit" to flow from the brain towards the muscles, while an internal pineal gland interacts with the non-material soul. Later, in the 19th century, phrenology associated mental faculties with specific areas in the brain, which resulted in unacceptable racial theories. Misunderstanding of the brain's anatomy produced a multitude of biased theories.

Nowadays, while we still do not have the full picture of the brain, we actively investigate the organ. On the highest level, psychology describes mental functions and behaviors, treating the brain pretty much as a black box. On the lower level, neuroscience attempts to describe the internals of this blackbox by analyzing, for example, blood and oxygen flow through magnetic resonance imaging. Somewhere in between, neurology attempts to treat the diseases of this misunderstood organ.

We believe that the only method for these three disciplines to succeed is to fully understand the brain's first principles. To do this, we attempt to reconstruct a mammal brain's structure by digital means. Thus, in this work, we provide software tools

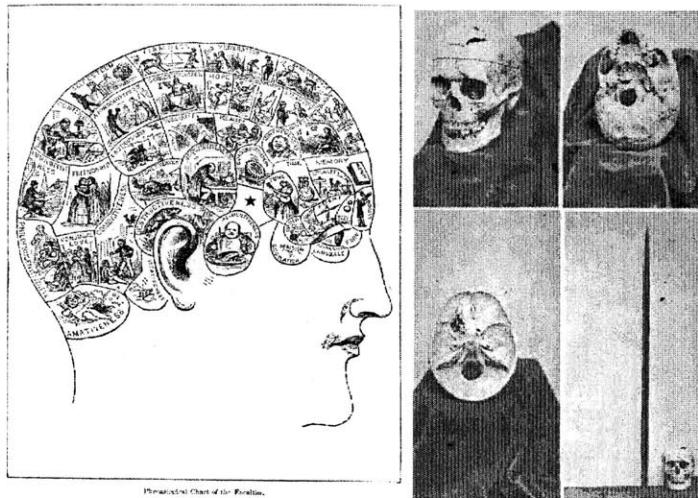


Figure 1-1: Left: an example of a phrenological map. Right: Phineas Gage's skull.

for fully automated brain analysis, with the ultimate goal of mapping the architecture of the brain for future digital simulation.

1.1 Early Neuroscience

Modern neuroscience dates back to 1873, when Camillo Golgi invented a silver staining technique that could be used to visualize nervous tissues under light microscopy at 300 nanometer resolution. Using this technique, Santiago Ramon y Cajal managed to entirely visualize a number of individual cells and proposed the *neuron doctrine* that led then to share the 1906 joint Nobel Prize. Instead of a previously accepted diffuse network, the doctrine states that the nervous system consists of a large number of individual cells.

In 1936, Otto Loewi and Henry Dale (shared Nobel Prize 1936) discovered that information is transmitted between neurons by chemical molecules - neurotransmitters. Acetylcholine is released in one cell and sensed by the other, and the places of contact are called synapses [38].

On the macro level, in 1894 Edward Flatau created an influential brain Atlas. In 1909, Korbinian Brodmann created the first brain map based on neuronal organization, which divides the cerebral cortex into 52 regions, each responsible for certain

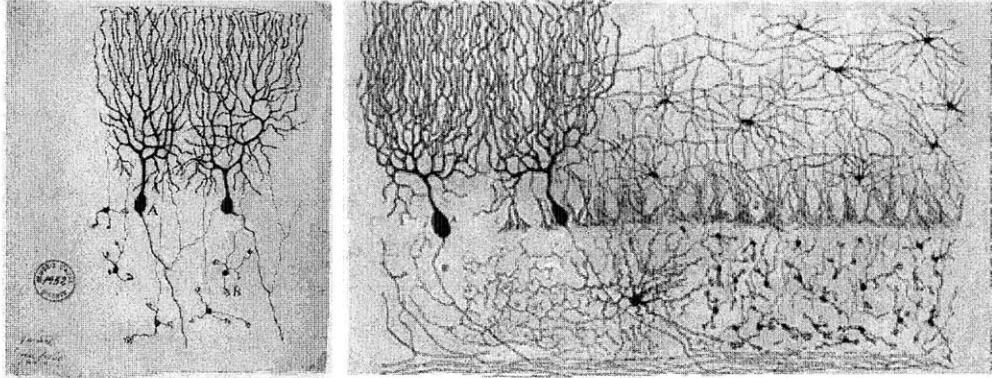


Figure 1-2: Ramon y Cajal’s drawings. Left: a cell from pigeon cerebellum. Right: a chick cerebellum.

functions. For example, Brodmann’s areas 44 and 45 are commonly nowadays known as Broca’s speech and languages areas [22].

The famous case of Phineas Gage, whose skull was pierced by an iron rod leaving much of the left frontal lobe destroyed, sparked further discussion in neurology and psychology [46].

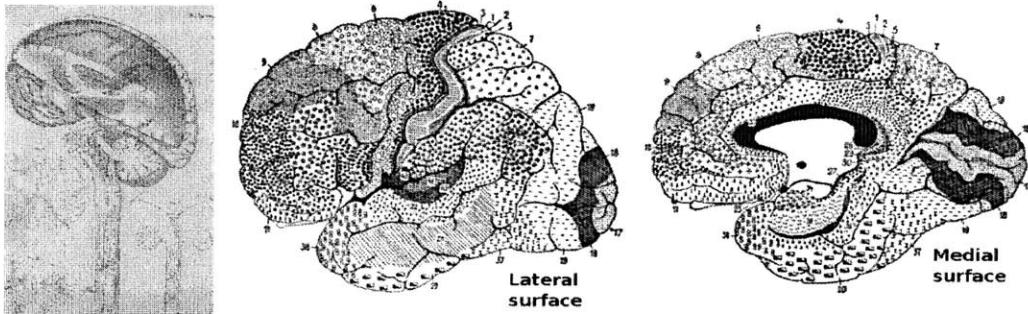


Figure 1-3: Left: Flatau’s atlas. Right: Brodmann’s areas

1.2 Brain Structure

While the brain’s connectivity as a whole is still a mystery, individual biological cell composition is well understood. By the neuron doctrine, the brain is made of individual cells (*neurons*) transmitting information to neighboring neurons through synapses. The neuron’s body contains of the *soma* cell, from which *neurites* (*den-*

drites and axons) emanate. Typically, multiple dendrites emanate from the soma, forming a tree-like structure. The dendrites form the receiving end of the neuron - each dendrite ends with a *synapse* called the *postsynaptic terminal*. A single, long and cylindrical *axon* extends from the soma to its final destination, where it branches to form multiple *presynaptic terminals*. When a neuron *spikes*, a directional electric impulse is elicited from the soma to the presynaptic terminals, which in turn release neurotransmitter molecules from *vesicles*, their containers. These molecules then drift through the *synaptic cleft* towards the postsynaptic terminal of the receiving neurons. Some scientific theories argue that a neuron spikes if the sum of the electric potentials generated on postsynaptic terminals is above some threshold. There exist two broad categories of synapses: *excitatory synapses* contribute positively to the generated electric potential, whereas *inhibitory synapses* contribute negatively, ultimately attenuating the overall magnitude of excitatory signals.

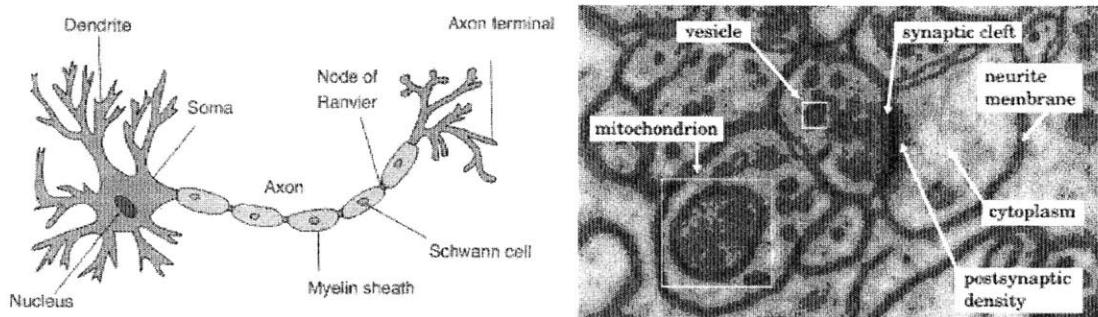


Figure 1-4: Neuron cell

Lastly, other cell organelles such as mitochondria or cytoplasm appear in microscopy images. While important for the cell's functionality, they are of lesser concern in this work.

1.3 Connectome

Just like the term genome represents an organism's DNA material, the term *connectome* represents the set of all neuronal connections in the brain [30].

As of 2015, the only fully reconstructed nervous system of an animal is that of *Caenorhabditis elegans*, a 302-neuron roundworm of 1 mm length [76]. Its manual reconstruction took 12 years and resulted in a full connectivity map published in 1986. Efforts all around the world are under way to provide other full or partial connectomes, such as those of the mouse retina [12], or the mouse primary visual cortex [8]. Janelia Farm Research [35] focuses on flies' connectomes. The Open Connectome Project [58] provides public sets of brain tissue data. These electron microscopy projects should not be confused with the Human Connectome Project by Massachusetts General Hospital, which reconstructs the brain *in vivo* via other means [32].

The term connectomics has been further popularized by Sebastian Seung in [68] and [69], where he argues that the intellect, personality, or memories are all stored in the brain's connections. Seung suggests that by careful full post-mortem deconstruction and digital reconstruction, we might be able to extend our lives posthumously. In contrast to Descartes' dualism, the connectome, by its reweighing, reconnecting, rewiring and regeneration would become the "soul's place".

In order to test this thesis, we need to progress from the 302-neuron worm to the 80 billion-neuron human's brain [62].

1.4 Imaging at High Resolution

Multiple investigative techniques allow us to see into the brain at various resolutions. Techniques similar to the Golgi method, allow us to stain an individual neuron and trace it throughout the tissues, while optogenetics [9, 10] enables us to readily control a neuron. Unfortunately, such techniques are currently not applicable to *all* neurons at once in the connectome. On the other hand, *magnetic resonance imaging* provides a very high-level non-invasive measurement of structural connectivity (*dMRI*) and functional connectivity (*fMRI*) at millimeter level resolution.

Since the somata have diameters of 50 μm and the axons 30 nm, and there are billions of them, neither of these techniques work. However, with the latest advances

in volume electron microscopy, we can achieve both the high-resolution and a sufficient throughput.

1.4.1 Scanning Techniques

In the base scenario, a single, flat, 2-dimensional tissue needs to be scanned to obtain a cross section of the brain. The traditional use of an optical microscope limits the maximum resolution to the wavelength of visible light (300nm to 700nm), and is insufficient for deciphering a connectome. Techniques such as *brainbow* imaging [75] and *expansion* microscopy [16], [24] significantly improve the resolution, but have not yet been proven to be applicable for connectomics, because it is not clear how to view synapses in these techniques. On the other hand, electron microscopy achieves much higher resolution, since De-Broglie wavelength is much shorter. *Transmission Electron Microscopy (TEM)* beams electrons through the tissue, creating accurate contrast on the other end, but requires a very thin tissue. *Scanning Electron Microscopy (SEM)* scans the surface of the tissue, collects back-scattered electrons, and can thus work with thicker tissues.

To process a volume image and reconstruct a 3-dimensional structure we repeatedly scrap thin slices of the tissue block's surface, performing *serial sectioning*. Popular techniques use a combination of scraping or cutting and scanning the surface (with SEM), or scanning the shaved off slice (with TEM). The resolution achieved varies, usually the x, y -axis' (4nm to 20nm) is more accurate than the z -axis' (27nm to 100nm) [45].

1.4.2 Our Microscope

My work is part of the joint effort between the MIT's Computational Connectomics Group [14] and Harvard's Lichtman Lab [47], data comes from the latter's high-resolution, high-throughput multibeam scanning electron microscope [23]. By using 61 beams in a single column in parallel, the speed increases by two orders of magnitude over a single beam, resulting in a detector bandwidth of 1.22 GPixel/s.

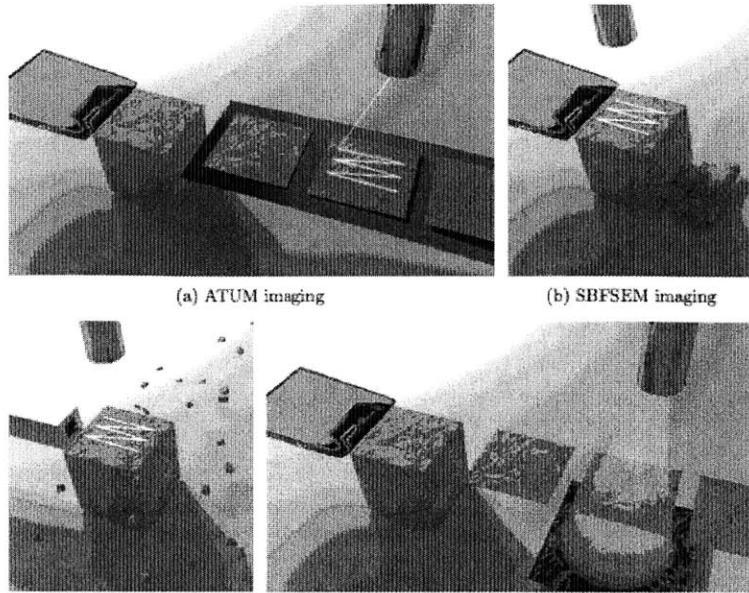


Figure 1-5: Scanning techniques (source: [45])

1.5 Data Analysis

Because the structure of the brain is quite complicated, and a single neuron can interconnect with thousands of other cells, we need to provide extremely accurate imaging. At our microscope's resolution of $4\text{nm} \times 4\text{nm}$ per pixel, even a 1-mm^2 section requires a 62.5GB image [36]. As the microscope automatically cuts 27-nm thin slices, a 1-mm^3 volume will generate roughly 2PB of data. A complete rat cortex (500mm^3) would produce 1 exabyte, while a complete human cortex (1,000 times that of a rat) would require 1 zetabyte, an amount of data on par with that of all the information recorded globally today. This is a lot of data to store, not to mention to process in our lifetimes.

We can avoid the problem of digitally storing such extraordinary amounts of data, by keeping the tissue slices on a tape, and reconstructing in-real time by re-scanning the tissues, as desired. However, this increases the demand for high processing rate. The microscope provides data at a rate of several terabytes per hour, so if the rest of the digital processing pipeline can match this rate, we would be able to process a cubic millimeter of brain in about 800 hours.

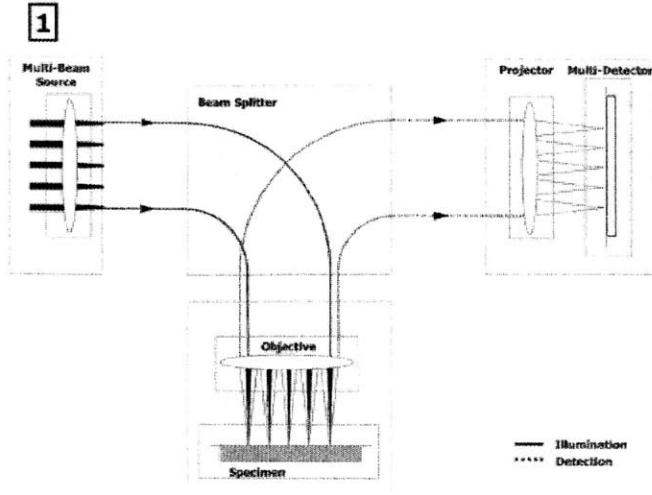


Figure 1-6: Beams of our microscope (source: [23])

Thus, our goal is to provide an efficient, fully automated data processing pipeline, that takes as an input the sets of images scanned by the microscope, and outputs the connectivity graph (potentially with auxiliary data) in real time. There are multiple challenges throughout this pipeline: images need to be correctly aligned, objects segmented, features detected, and, it is estimated, in all of this we should not exceed about 10,000 instructions per pixel of raw data [36].

1.6 Contributions

The primary objective of my work is to provide an efficient computational process for 3 sections of the pipeline: probability map generation, oversegmentation and segmentation. While we briefly cover the entire pipeline (Chapter 3), our work was not focused on steps immediately following the microscope (ie. image alignment), nor on steps after agglomeration (Chapter 7). Our main contributions to the project are:

1. Probability map generation using convolutional neural networks (CNNs), as well as ensembles of CNNs and random forests (Chapter 5).
2. Fast oversegmentation algorithm, using watershed (Chapter 6).

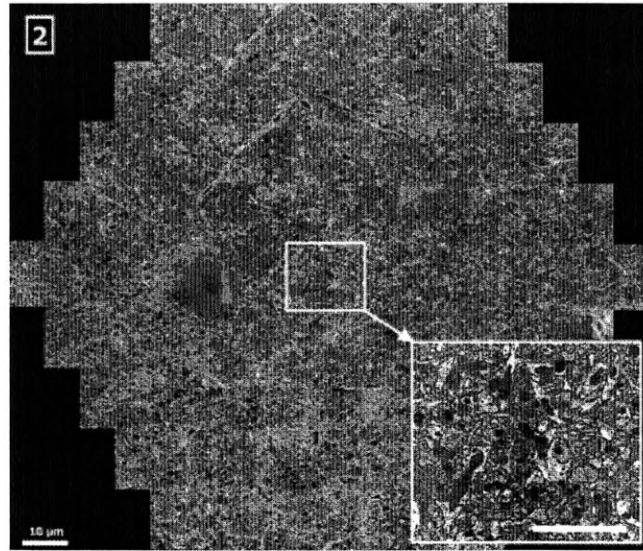


Figure 1-7: Output grid of our microscope (source: [23])

3. MapRecurse, a recursive computational framework (Chapter 9).

The rest of this thesis is constructed as follows: Chapter 2 provides an overview of related work, Chapter 4 covers the ground truth transformation process, Chapter 7 covers the segmentation step, Chapter 10 provides quantitative and performance analysis, Chapter 11 discusses further work and challenges ahead, and Chapter 12 concludes.

Chapter 2

Related work

With a number of international teams working on data processing systems with various types of microscopy, the software packages are abundant. Some of the packages help manual human annotators label data sets, some provide fully automated machine-learning-based functionality, while other combine both approaches. Data types conversion, machine learning, image visualization are all available in some capacity, but due to this variety, they do not work well together, are slow (for our data size), or are designed for a slightly different purpose. In this section, we attempt to cover the most relevant software packages.

In 2013, the *International Symposium on Biomedical Imaging* (ISBI) held a competition [33] for 3D segmentation of neurites in electron microscopy images. They released a training set of 100 grayscale 1024 pixel by 1024 pixel images, with corresponding manually annotated labels (object identifiers). After the submission period, numerous teams were tested on a similar (but different) testing set. As of 2015, this data set remains the "gold standard" to measure error rates of processing packages.

Two separate processing pipelines, Gala [57] and Neuroproof [60], both from the Janelia Farm Research group, ranked highly in this challenge, and thus became the benchmark for our comparisons. Gala uses Ilastik ([72], [45]) for the initial probability map generation. RhoANA [41], a newer tool, uses Fusion for segmentation. All four packages are briefly described below.

2.1 Ilastik

Ilastik is an open-source desktop software that provide a nice user interface for data annotation and combines interactive machine learning with active learning. Released in 2011, it promises a user, even without expertise in image processing, to quickly perform segmentation and classification. Based on user's manual segmentation, it interactively trains a random forest classifier and provides real time feedback. The classifier can be later exported and used in other data processing pipelines.

While its ability to annotate out-of-memory data sets and the ease of use are appealing, Ilastik is insufficient for our needs. It is written in Python, and despite the promise of multi-threading, is a couple orders of magnitude too slow. Its basic random forest classifier also does not allow us to achieve the desired accuracy.

2.2 GALA

GALA, Graph-based Active Learning of Agglomeration, is a Python library for image segmentation that makes extensive use of the Python's scientific stack. GALA works using a two-step approach: first it trains the classifier on annotated data, and then uses it to classify new (not annotated) data. As a first step it takes the output of Ilastik (a pixel probability map) and oversegments the data using the watershed algorithm from the Python's SciKit package. Based on the oversegmentation, it builds a *regional adjacency graph* (RAG), where supervoxels are represented as nodes, and the edges have probabilities assigned based on the boundary values. Then, it trains a random forest classifier on this graph and agglomerates supervoxels based on edges' probabilities to, ultimately, minimize the variation of information (see Chapter 10) from the provided ground truth.

The classification step is analogous to the training step: first Ilastik is used to generate probability maps, then the watershed generates oversegmentation and lastly the classifier merges adjacent supervoxels.

Similarly to Ilastik, GALA suffers from unacceptable (for our purpose) perfor-

mance issues and Python’s Global Interpreter Lock prevents it from multi-threading. Additionally, it suffers from exaggerated memory consumption. It is not uncommon for the SciKit’s watershed (the middle step) to produce a working set of over 100 times the (uncompressed) input data size.

2.3 Neuroproof

Neuroproof, also developed by the Janelia Farm Research group in parallel to GALA, offers very similar functionality, but it is written in C++ and thus promises much better performance. Specifically, Neuroproof implements the last (agglomeration) step of GALA using *regional adjacency graphs (RAGs)*, and depending on the options specified, the OpenCV or Vigna [74] random forest classifiers are used. Using small sample learning [61] the need for the annotated training size decreased 5 times, while preserving accuracy. Quality is improved by using context-aware delayed agglomeration [60], where some merge decisions are postponed in order to generate a more confident boundary prediction.

Overall, the single-threaded performance of Neuroproof is adequate. By the efforts of my colleague [54] the classification step of this package was parallelized, resulting in a near-linear multi-core speedup. While the agglomeration performance provides us with a good starting point to benchmark against, Neuroproof still relies on GALA’s watershed algorithm, and requires as input probability maps generated by Ilastik. Thus, as a whole pipeline, it is still not fast enough.

2.4 RhoANA

RhoANA is a segmentation pipeline, that as of 2013, claimed state-of-the art reconstruction performance for data set in the GB-TB range. Similarly to GALA, first a random forest classifier is trained on interactive sparse user annotations, then anisotropic smoothing is applied and a Conditional Random Field framework generates multiple segmentation hypothesis per image slice. By the process of fusion [73],

these segmentations are then combined into consistent 3D objects. The following six steps best present the approach:

1. Membrane classification
2. Segmentation
3. Block dicing
4. Window fusion
5. Pairwise matching
6. Local and global remapping

Using RhoANA, a $27,000 \mu\text{m}^3$ volume of brain tissue has been classified. The package also contains Mojo, an influential proofreading tool for semi-automated correction of merge errors. For visualization in our work we use Dojo [29], a web-based version of Mojo. While Dojo currently does not offer as many capabilities as Mojo, it is convenient to locate it on our data servers and it minimizes the amount of data transfer required.

Chapter 3

Our Segmentation Pipeline

In this chapter we provide a high-level overview of the processing pipeline. As described in Chapter 1, the fully automated microscope outputs slices of raw images at the rate of about a gigapixel per second. In order for data not to accumulate at an intermediate stage, it is important that each stage not only achieves comparable throughput, but can also autonomously receive the input from the preceding stage, and can pass its output to the following stage.

Since we utilize numerous machine learning algorithms, it is also crucial to distinguish between the training and the classifying phases at some of the stages. During the training phase, the machine learning algorithm takes as input a stack of raw EM images and its corresponding labels (of the same dimension) and outputs a classifier that minimizes a loss function. In general, we are not particularly concerned about the total training time (hours or days is acceptable), nor its complexity, as long as the resulting classifier produces high quality results. However, the classification performance is crucial - this stage needs to eventually match the throughput of the microscope (a gigapixel per second).

Due to the knife's cutting off slices of the block's surface and the digitization process of the microscope, the raw images are misaligned, contain noise and various physical distortions. Thus, the first stage of the pipeline is the image alignment, both hexagonally on the x, y -axis, as well as in the z -direction. This stage is not the focus of our work, and we assume that the input EM images are already aligned, but this

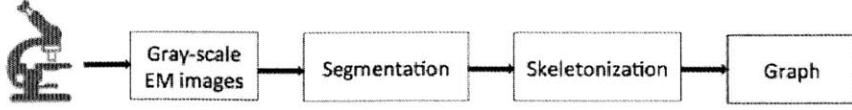


Figure 3-1: Simplified pipeline

is an area of ongoing research (based on [13]) in our lab.

With the images aligned, we generate the *pixel probability* map for each slice on the z-index, that is, we binary classify each pixel as a membrane or non-membrane, and assign probabilities to each class. Thus, a pixel classified as 1 with probability 1 means that we believe it is a cell membrane (boundary), and a pixel classified as 0 with probability 1 is a non-membrane. This is the core functionality of previously described Ilastik package. In Chapter 5 we proposed multiple solutions to the probability map generation.

Once each pixel is assigned a probability in the probability map stack, we proceed to generate an *oversegmentation* - the first attempt at merging adjacent pixels into 3D objects called supervoxels. An efficient watershed-based oversegmentation, as well as alternative techniques, are described in Chapter 6.

With the raw EM images, probability maps and oversegmentation, we proceed to generate full 3D segmentation - we further merge adjacent supervoxels in the oversegmentation to produce biologically consistent objects, corresponding to neurons. As of this writing, we still use a parallelized version of Neuroproof.

We care less about the physical shape of resulting 3D neurons - our core goal is to obtain a (directed) graph connectivity between the neurons. In this step, called *skeletonization*, we extract the core skeleton of each neuron, and try to connect them together. Since the pixel accuracy is irrelevant here, the output data set tends to be much sparser, thus appropriate for permanent storage and further analysis. Skeletonization is briefly covered in Chapter 8.

With the results of segmentation or skeletonization, we might notice biological, or physical discrepancies between the pipeline's results and our prior expectations,

suggesting the pipeline produced an error at some point. For example, if a neuron connects to itself, we have likely made a mistake at some stage, since (most) neurons do not form loops. Thus, in that particular spatial region, the pipeline should revert and re-run the calculations using a different algorithm. Such a recursive framework, called MapRecurse, is introduced in Chapter 9. At the time of this writing, it is still not integrated with the whole pipeline, but nonetheless, its functionality is demonstrated on small problems.

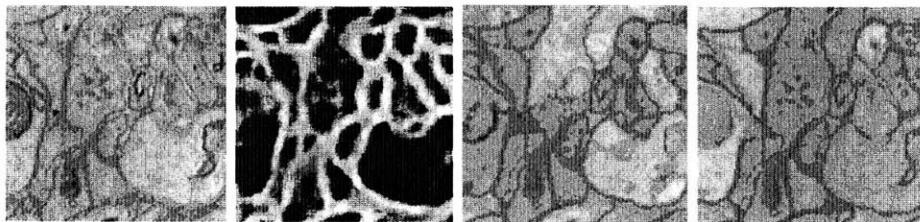


Figure 3-2: Pipeline stages. Left to right: ram EM image, probability map, oversegmentation, segmentation.

3.1 Initial Pipeline

When we first started working on the project, we inherited a pipeline heavily-dependent on GALA, NeuroProof and Ilastik (implicitly). While the intentions to borrow from the state-of-the-art packages guaranteed matching the correctness metrics, the individual stages did not communicate well with each other, were cumbersome to use or modify, and had a complicated build process. Let it suffice to say that due to a complicated build process and many dependencies, we had exactly one machine where the whole pipeline could be executed from the start to the end.

Initially, the pipeline relied on ground truth generated by Ilastik. Since we did not have the man-hours to manually annotate data sets to generate a new classifier, for most of the time we relied on 3rd parties to provide us the probability maps. This soon became a bottle-neck to run various experiments.

Once we had the probability map, we could invoke GALA’s oversegmentation algorithm, which runs the Python SciKit’s watershed algorithm. Due to its slow

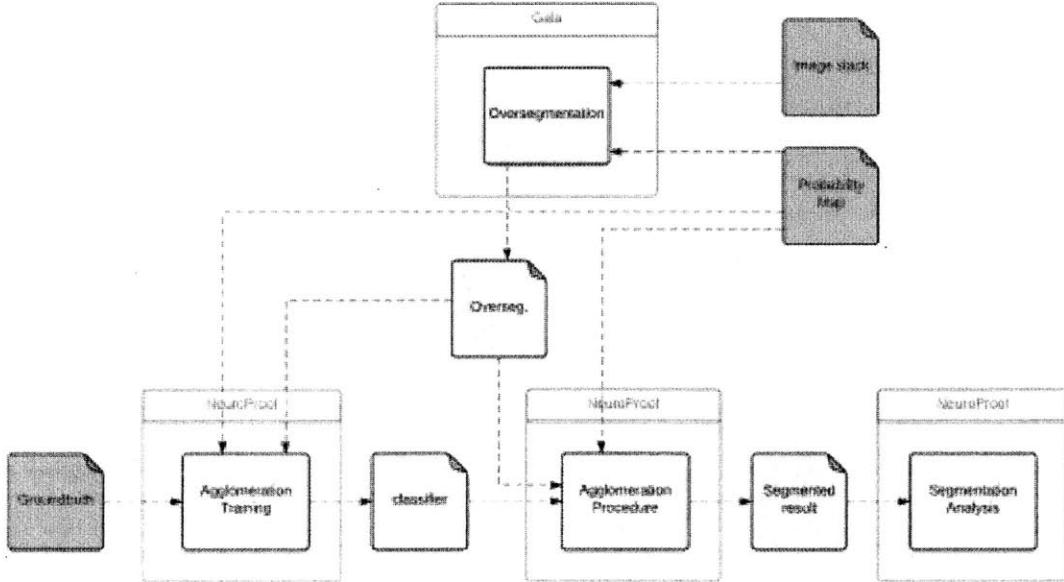


Figure 3-3: Initial pipeline

implementation, it took on the order of minutes to process a 100 megapixel data stack. In 8-bit images 1 megapixel takes 1MB of data, but even with this 100MB set the memory working set was engrieving, over two orders of magnitude bigger than the input.

With the oversegmentation and the ground truth, we can train NeuroProof’s classifier, and then run agglomeration. The particular strength of this pipeline was the parallelized version of Neuroproof, which could take advantage of our multi-core machines for the agglomeration stage (only classification).

3.2 Modified Pipeline

In the modified pipeline, we first provide new probability map generation algorithms - variations of convolutional neural networks (CNN), random forests (RF), their ensembles, as well as a combinations of RF and CNN algorithms (described in detail in Chapter 5). Before we can train these machine learning algorithms, we need to

slightly transform the object ground truth into the boundary ground truth (covered in Chapter 4).

Once appropriate probability maps are generated, we invoke a C++ implementation of the watershed algorithm. Its output is (almost) identical to the GALA's output, but due to a more problem-specific implementation it performs much better. Because we are now capable of processing much bigger data sets, these algorithms operate on flat image (PNG) files, instead of previously used huge individual blobs (HDF).

As the last step that my work is focused on, we pass the oversegmentation and the probability map to the parallelized Neuroproof for agglomeration (Chapter 7).

There are other steps that follow agglomeration that are mostly omitted here, such as biological and statistical analysis of neurons, or skeletonization (briefly covered in Chapter 8).

A new framework, MapRecurse, is introduced in Chapter 9 as an extension to the pipeline. Currently, the pipeline is run as a single forward pass from aligned images to skeletonized output. While the framework is not integrated yet, it would allow recursive computations and error correction in subvolumes of data in all stages.

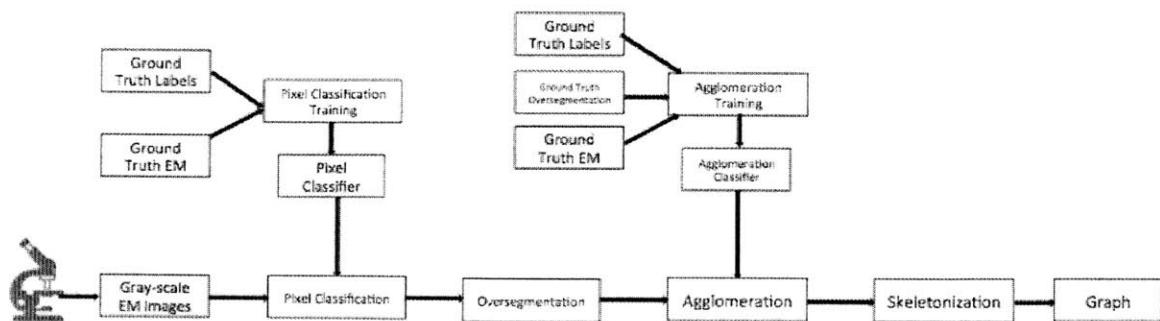


Figure 3-4: Modified pipeline

Chapter 4

Ground Truth Transformation

As the first pre-processing step for the probability map generation training, we need to transform the ground truth representation. Each pixel in the EM image stack is classified either as an object with its own label (a positive integer), or as the extracellular space (with a zero label), if available. However, in order to train the probability map classifier, we need to provide it with the boundary ground truth - that is, each pixel in the EM must be classified either as a membrane (value of 1) or a non-membrane (value of 0).

In one of our approaches, we do this by performing a set of morphological operations on the ground truth labels. First, we take a 2-dimensional gradient [65] of the labels, then we apply a quadratic polynomial and a threshold. Then we remove connected components that are too small (*area opening*) to decrease the amount of noise (which we assign to annotation errors) and produce the *double* membrane ground truth. *Thick* ground truth is obtained from the double ground truth by dilation with a 13 pixel by 13 pixel structuring element. This has the effect of covering gaps in between membranes (extracellular space) with membrane markers. *Thin* ground truth is created by morphologically thinning thick ground truth. Lastly, the *double narrow* ground truth is obtained by removing interior pixels from the thin ground truth. While these operations and choosing the right parameters may seem more like art than science, we later benchmark these four ground truth transformations to minimize the error rate.

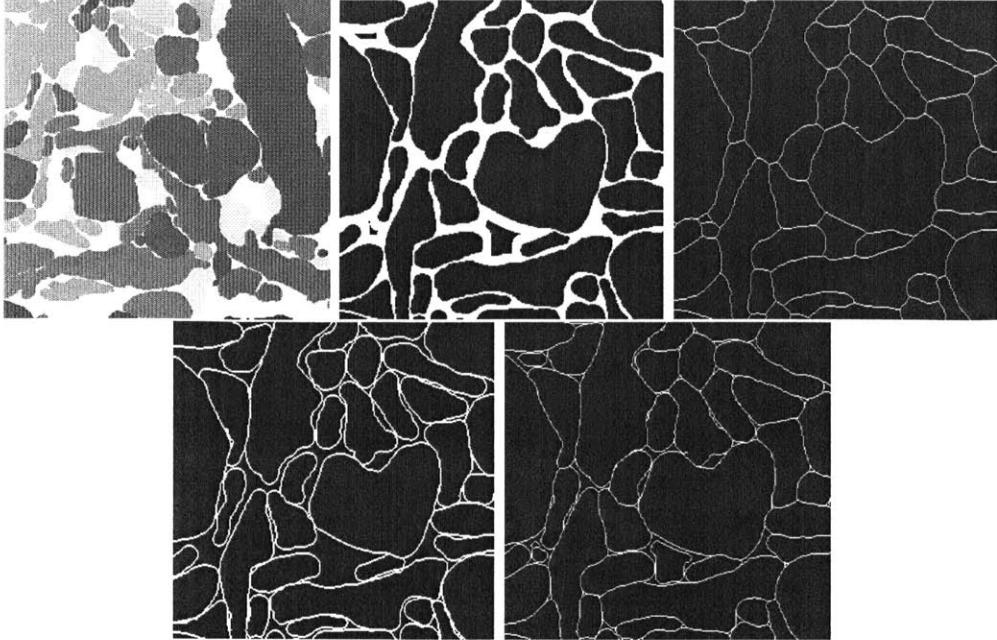


Figure 4-1: 1: Labels, 2: thick boundary ground truth, 3: thin boundary ground truth 4: double boundary ground truth, 5: double narrow boundary ground truth

The primary objective of the project is to execute the pipeline in real-time on data coming from our high-throughput multi-beam microscope. However, for some of the experimental work and benchmarking purposes other data sets are used as well. The *ISBI* data set consists of two 100MB train and test sets, each as a volume of 1024px by 1024px by 100 slices. The *kasthuri11* data set from OpenConnectome is a 10752px by 13312px by 1850 slices volume, of total compressed size of 36GB, or 250GB uncompressed. The *P7* data set contains 2514 slices, where each slice is

Algorithm 1 Ground Truth Transformation [MATLAB]

```

1: procedure TRANSFORMGT(labels)
2:   excell ← labels == 0
3:   [fx,] ← gradient(double(labels));
4:   membraneSkeleton ← fx.2+fy.2 > 1/1000;
5:   doubles ← bwareaopen(membraneSkeleton, 10);
6:   thick ← imdilate(doubles,ones(13,13)) & excell | membrane;
7:   thin ← bwmorph(padarray(thick,[1 1 ],true),'thin',inf);
8:   doublesNarrow ← bwmorph(thin,'remove');
9: end procedure
```

created by stitching together 42 (6×7) patches, each of size 10000px by 10000px, for the total volume size of 2.7TB (compressed).

Chapter 5

Probability Map Generation

In this chapter we cover the various efforts we have taken to generate probability maps. Roughly, the probability map is a pixel map of the EM stack indicating the probability that a given pixel is a membrane or non-membrane. Since we operate on 8-bit grayscale EM images, we normalize and discretize the real values from $[0, 1]$ to integers in $[0, 256]$.

It may seem like a straightforward task to perform a number of simple operations (thresholding, contrasting etc.), or even use the Canny edge detector [5] to transform a raw EM image into the probability map, but this is unfortunately not the case. The amount of noise prevents these naive techniques from producing accurate results. In this chapter we cover two machine learning algorithms, random forests and convolutional neural networks, that perform well on the task.

5.1 Random Forest

A decision classification tree [63] is a function $f_t(x)$ that classifies a sample x (in our case a pixel and its neighborhood) into one of the classes (membrane or non-membrane). Each node in the tree represents a decision to be made on a particular feature, edges represent learned probabilities and leaves produce the final classification labels. To classify a sample, it is enough to traverse the tree from the root to one of the leaves, following the most locally probable edge at each node.

Random forests are a combination of decision trees [11], such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. In essence, it is an ensemble of decision trees initialized with random features, where the output class is the mode of classes produced by all trees (for classification) or their mean (for regression). Random forests tend to decrease decision trees' overfitting on the training set, as they tend to sacrifice optimality of an individual tree for an increase in diversity of the ensemble.

Due to its relative simplicity, random forest is a highly efficient machine learning model. While training still requires many labeled samples and may take time (in our experience, on the order of hours for a forest of tens of thousands of trees trained on a 16-core CPU), the classification latency is small, because traversing the tree from the root to a leaf by making (binary) decisions does not require many instructions.

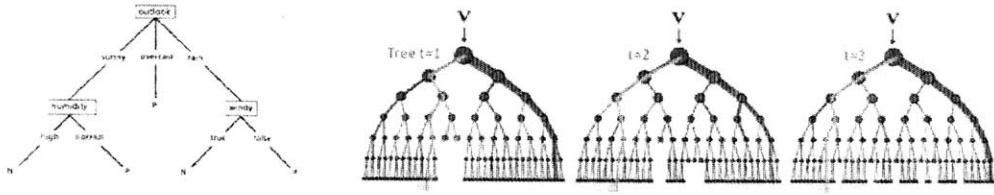


Figure 5-1: Decision tree, decision forest

5.1.1 Ilastik

Ilastik's core feature - its pixel classification module - uses a random forest of binary decision trees. Its main advantage is that it works well with *sparse* training labels (probability map ground truths), and there exists a dynamic feedback loop. After examining the current classification, the user can correct erroneous voxel predictions by providing more training samples. Thus, by focusing on problematic areas, good accuracy can be obtained faster than labeling all pixels [42]. Ilastik uses the scikit-learn's RandomForestClassifier [67] with $n_{estimators} = 100$ trees. Unfortunately this method does not produce nearly good enough results, and because scikit-learn is a high-level, general purpose library written in Python, its performance is inadequate.

5.1.2 Structured Decision Forest

To work around the problems with Ilastik, we implemented a structured decision forest based on [21]. The original implementation obtains performance that is orders of magnitude faster than many other competing state-of-the-art approaches (in particular, convolutional neural networks), while also achieving state-of-the-art results on the NYU Depth and BSD500 Segmentation datasets.

Instead of directly mapping the high-dimensional input space X onto the high-dimensional output space Y , a structured decision forest introduces a low-dimensional mid-representation C , that is sufficient to approximate Y well. Using randomization and *principal component analysis* (PCA), this reduction is fast. In our edge detection application, the input consists of 32x32 image patches, and the output is 1 to 4 trees per patch, with 16x16 segmentation masks of 2-pixels wide strides.

We have trained each tree with $0.5M$ to $1.5M$ sampled patches, where training of each tree takes about 30 ms and is embarrassingly parallelizable. More importantly, even with the combination of Matlab and C++ implementations, we achieved prediction time of about 0.15 seconds per megapixel.

5.2 Convolutional Neural Networks

Artificial neural networks are inspired by the biological structure of the primary visual cortex, which cascades simple and complex cells. Of particular interests to us are *convolutional neural networks* (CNN), where one, or more, intermediate layers are convolutions. Combining multiple layers, each with relatively simple computations, results in a *deep network* that, with a sufficiently large training set, can approximate any function. While the theoretical concepts for such networks were proposed in 1980s and 1990s, it was not until late 2000s that the availability of high-performance GPUs enabled us to train deep enough networks on millions of training samples. As of 2015, CNNs are the state-of-the-art choice for nearly all computer vision problems. Their inspiration by the visual cortex and high accuracy on many data sets, make them a natural choice for our EM images segmentation problem. While this is not the first

attempt ever to use the CNN [34], we achieved promising results. Our methodology is described below.

5.2.1 Overview of Deep Neural Networks

A *deep neural network* (DNN) consists of the input layer, the output layer, and multiple "hidden" layers of units in between [6]. Each layer takes as the input the output of the preceding layer, executes its own specific function (usually a combination of linear and non-linear function), and passes its output to the following layer. The advantage of a deep network over a *shallow* network is its ability to compose features from lower layers, and thus model more complex data with fewer units than would be required for a similarly performing shallow network. Most DNNs are feedforward networks - data passes from the input towards the output without cycles, forming a directed acyclic graph.

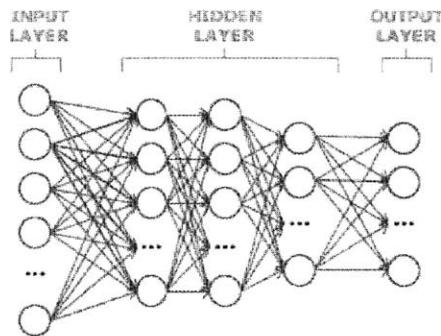


Figure 5-2: Deep neural networks

The *convolutional layer* is the core of the CNN. Each layer consists of a set of filters (kernels) of small spatial dimensions (usually between 3x3 and 19x19). During the prediction phase (forward pass), each kernel convolves (slides) across the width and height of the input image, producing a 2D activation map. The convolution is, essentially, a discrete dot product of the kernel parameters and the input (in our case, the "central" pixel and its neighborhood).

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

Since multiple kernels are convolved with the input layer, the output consists of a stack of activation maps. For example, if the input layer is a volume $1024\text{px} \times 1024\text{px} \times 3$ (such as a 3-channel RGB image), and the square *kernel size* is 7, then the kernel will need to learn $7 \times 7 \times 3 = 147$ weights. Additionally, sometimes we specify the *stride* of the layer, which correspond to the number of pixels skipped during the convolution. If the stride is 1, then we convolve every input pixel, but for strides $S > 1$, we convolve every S^{th} input pixel. Since the kernel is of size > 1 , we may need to pad the input layer to preserve the same dimensions in the output layer. If the input layer is of size $N \times N$, the kernel size is K , padding P and the stride S , then the output layer will have dimensions:

$$N_{output} = \frac{N - K + 2P}{S}$$

Thus, by stacking convolutional layers together, we can reduce the input layer size, but the number of convolutions (multiplications) is high, and its usually the most computationally expensive layer.

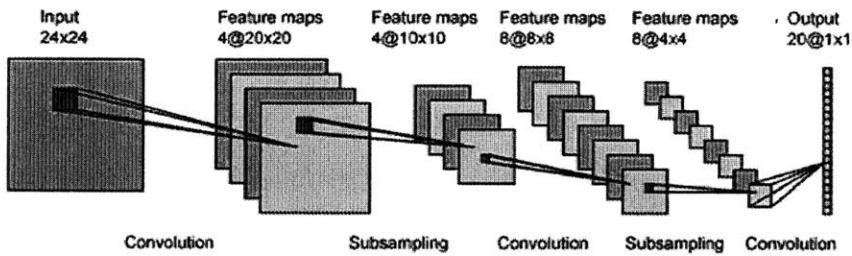


Figure 5-3: CNN

The *pooling layer* progressively reduces the dimensions of the input layer, which is helpful in minimizing the number of parameters and multiplications required throughout the network, as well as in decreasing random error of the model (*overfitting*). The most common pooling function is the max kernel, but averaging or l^2 -norm are also

used. The kernels are usually small and, similarly to the convolutional layer, the stride is also applied. The most popular variations are of $K = 2, S = 2$ and $K = 3, S = 2$. Thus, for the input layer of dimensions $N_{in} \times N_{in} \times D_{in}$, kernel size K and stride S , the output has dimensions:

$$N_{out} = \frac{N_{in} - K}{S}$$

$$D_{out} = D_{in}$$

The *Rectified Linear Units* (ReLU) layer uses a kernel with an activation function $f(x) = \max(0, x)$ to increase the nonlinearity of the convolutional layers. The advantage of this layer is that it is fast (faster than any exponential or sigmoid function) and does not suffer from the vanishing gradient problem. The alternatives to the max function are the more computationally intensive sigmoid function $f(x) = (1 + e^{-x})^{-1}$ and the hyperbolic tangent $f(x) = \tanh(x)$.

The *normalization* layer is useful when other layers use unbounded activation functions (such as ReLU), as it allows for detection of high-frequency features with a large response, and it attenuates responses that are uniformly large in a local neighborhood. Intuitively, it encourages competition for large activations among nearby group of neurons, thus provides regularization. The output dimensions are always the same as the input dimensions.

The *dropout* layer's main goal is to prevent overfitting [55]. At the training stage, each kernel is kept with probability p , or dropped out of the layer with probability $1 - p$. Usually $p = 0.5$, thus the network size is reduced by half and not all kernels are trained on the training data, thus avoiding overfitting. It is inspired by random forests and resembles taking the average from an ensemble of networks, at a much lower cost. During the test phase, it would be ideal to test all possible dropout configuration to minimize the loss function, but since there could be exponentially many of them (2^n , where n is the number of kernels), we usually approximate the output by taking the expected value of the kernel, that is, multiplying its output by p . Thus, although (implicitly) 2^n network configurations are trained, we only classify on one of them.

The *fully connected* layer (also known as the *inner product*) has full connections to all activation maps in the previous layer. It simply multiplies the input by a weight matrix and introduces a bias offset. In some way, the fully connected layer is equivalent to the convolutional layer. Whereas the convolutional layer is connected to a local region in the input, the fully connected layer is connected to all inputs. One can be easily converted into the other.

The *loss* layer is the last computation used to compute the error of the network. The most commonly used function is the softmax, defined as:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

where x_i is an input value. Other loss functions encountered are the sigmoid cross-entropy and euclidean loss.

The *training* of artificial neural networks is done with backward propagation of errors, *backpropagation*, along with an optimization method, the *gradient descent*. For an input x , upon a full forward propagation through the network, the loss function L is computed on the network's output y . We would like to learn the weights in each of the hidden layers, such that they minimize the value of $L(y)$. By recursively applying the chain rule, we obtain the $\nabla f(x)$ for each of the hidden layers and can correct the weights associated with each kernel. The initial weights' values are usually assigned at random. Depending on the depth and width of the network, we might need a large number of labeled samples to learn all weights accurately. This process usually takes a lot of time (on the order of hours, or days) even on powerful GPUs.

5.2.2 Frameworks

As of this writing, there exist three widely popular, open-source, general-purpose deep neural network frameworks: Theano, Caffe and Torch7 [3]. We evaluated each of them to determine the best fit for our needs.

Theano is a Python library developed at the University of Montreal - it is a linear algebra compiler that optimizes a user's symbolically-specified mathematical com-

putations to produce low-level implementations [4]. It is straightforward to specify custom networks and the high-level execution code, but the symbolic logic compiler is rather complicated, and it is hard to modify its functionality. It is convenient to be able to run the networks both on the CPU or the GPU, but despite using the NVIDIA CUDA compiler (`nvcc`) to optimize the code for the GPU, the classification phase is too slow.

Caffe is a state-of-the-art C++ library developed by the Berkley Vision and Learning Center [37]. It is specifically designed for practitioners of machine learning and computer vision, and provides convenient Python and MATLAB bindings, both for training and classification. It supports execution both on the CPU and the GPU and comes with a public repository of networks, called Model Zoo. Additionally it has a strong support of NVIDIA, which provided its own training package with a convenient user interface, call DIGITS [20]. Many other libraries, such as Julia’s Mocha, borrow heavily from Caffe.

Torch7 is a versatile numeric computing framework that extends Lua [17]. By combining OpenMP, SSE and CUDA implementations, it claims to be the fastest machine learning library. While it does have the support of Facebook and Google’s DeepMind, can be parallelized, and in theory can be integrated with C++, it lacks documentation and support, and because of (relatively) unfamiliar reliance on Lua, seems the least popular choice.

Upon evaluating these frameworks, as well as considering writing our own from scratch, we decided to use Caffe as the base for our work.

5.2.3 Networks

Throughout our experiments, we evaluated multiple convolutional neural networks. While choosing the network layers, their ordering, number of features, or kernel sizes may sometime look more like an art than science, the following networks are a representative sub-sample. See appendix B for a graphical visualization.

3-layer Network

The first network we experimented with consists of three max-pooling layers (stride of 2, kernel sizes 4×4 , 4×4 and 5×5 respectively) and the softmax loss layer. The network was implemented in Theano and its author claimed a high accuracy on the ISBI data set, but we failed to obtain satisfactory results, even after re-implementing it in Caffe. Thus, the network was abandoned in our further experiments.

N3 Network

The *N3* network consists of 14 layers and was first proposed in [66], and later used for vesicle detection in [59]. It achieved a state-of-the-art error rate on the MNIST digits data set, and it performed reasonably well (accuracy of 85%) on the ISBI data set. N3 consists of three sets of convolution, max-pooling and normalization layers, followed by two fully connected layers. See B-1 for a detailed visualization.

AlexNet Network

The AlexNet network is based on [44] and consists of 18 layers, requiring 650,000 neurons and 60 million parameters. First, there are two sets of convolutional, normalization and max-pooling layers, followed by three convolutional layers, followed by a max-pooling layer, followed by three fully connected layers. AlexNet won the ImageNet LSVRC-2010 contest and achieved a pixel accuracy of 86% on the ISBI data set. See B-3 for a detailed visualization.

Shallower Networks

We also implemented a number of shallower networks to better understand the importance of the depth of network on its performance. We started with the AlexNet network and begun removing layers, down to only two layers (convolutional and fully-connected) in BN2, where the accuracy dropped to 50% (equivalent to a random guess). Surprisingly, shallower networks did not necessarily mean lower accuracy, or faster processing time, suggesting that even small changes to the network's architec-

Table 5.1: Network Evaluation

Name	Training time	Size (MB)	Pixel Accuracy	GPU Latency (s)	CPU Latency (s)
N3	3h 2min	45	81%	3.27	41.53
AlexNet	14 min	80	81%	3.34	37.54
ShortNet1	11 min	225	81%	3.10	83.55
ShortNet2	22 min	150	78%	2.73	58.43
ShortNet3	38 min	630	77%	5.69	222.85
BN1	2 min	0.12	68%	1.79	1.98
BN2	2 min	0.12	50%	1.46	1.36

ture can change its behavior. Of particular interests may be ShortNet1, ShortNet2 and ShortNet3, which achieve pixel accuracy similar to AlexNet, as well as BN1, consisting of just one layer more than BN2, yet achieving 18% higher accuracy. See Appendix B for their visualization.

Evaluation

All networks were trained using Caffe with the DIGITS package for visualization. Our primary concern is the classification time (forward pass latency) on the CPU. We evaluated both the GPU- and CPU-based implementations of these networks, but the training phase was executed only on the GPU. See Table 5.1 for an analysis. Notice that the CPU latency is correlated with the model size, while the GPU latency is not. Similarly, increasing the classifier size (more neurons, or parameters) does not imply higher accuracy.

5.2.4 Sliding Window Classification

All of our networks were trained on small, labeled patches. Each 2-dimensional patch classifies (assigns probability to) exactly one pixel from the training set either as a membrane or non-membrane. The patch defines a small neighborhood of odd dimensions, between 19px by 19px and 255px by 255px, to ensure that the middle pixel is unambiguously discriminated. In a single forward pass through the network, we provide such a neighborhood (most commonly 49px by 49px) and classify the central

pixel in the output.

However, our EM image slices have dimensions between 1024 px by 1024 px to 16,000 px by 16,000 px, so a single pass of such a network cannot classify each pixel in the input. Instead, we use the sliding window technique, where a square window of our network's input size (ie. 49x49) slides across the entire image. If the network's input size is $K \times K$ pixels, and the image size is $N_{input} \times N_{input}$, the output image would have dimensions of $N_{output} = N_{input} - K + 1$. To avoid this decrease in size, we pad the image with a border of width $\lfloor K/2 \rfloor$, filled with 0 values, or a mirror of the image values.

This results in a significant increase in the number of computations: instead of being forward propagated through the network once, each pixel now contributes to K^2 forward computations.

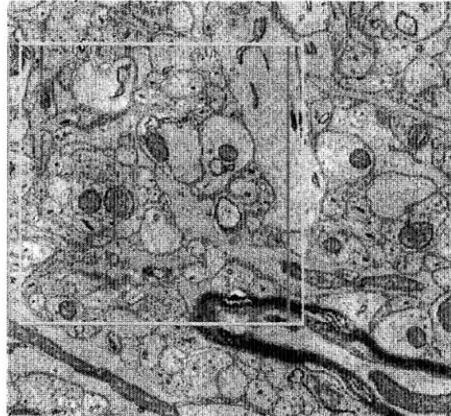


Figure 5-4: The sliding window approach

5.2.5 Fully Convolutional Networks

This computational blowup quells even the most powerful GPUs. For example, on the Nvidia Quadro K6000 with 12GB memory, processing a 1024px by 1024px grayscale image (1MB) with the sliding window of size $K = 49$ and the AlexNet network takes a staggering 30 minutes (forward propagation only), and clearly is not a viable approach.

Looking closer into the computations performed across all $(1024 - K)^2$ window positions, we observe that most of them are repeated, or follow a very similar pattern. Using the dynamic programming technique, we can significantly speed it up, preserve exactly the same output ([26], [50]) and re-use the same weights from the original (patch-based) classifiers. Specifically, here is how we optimize the convolutional and max-pooling layers.

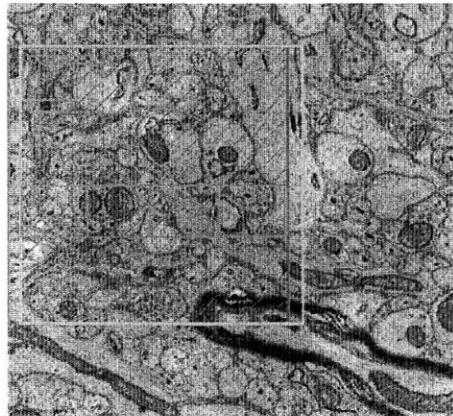


Figure 5-5: Overlapping kernels (green - max-pooling, blue - convolutional) in neighboring sliding windows.

Let $L + 1$ by the number of layers in a network. $l = 0$ is the input map, and the max-pooling and convolutional layers are indexed $1..L$. Let P_0 represent the input image with one or more input maps (for example, equal to the number of input image's channels) of width and height w_0 (for simplicity, we assume they are all square).

If the l^{th} layer is a convolution with kernel size k , then its output P_l will be a set of square maps, each of size $w_l = w_{l-1} - k$. In general, $|P_l| \neq |P_{l-1}|$, unless $k = 1$. If the l^{th} layer is a max-pooling layer with kernel size k , and since max-pooling processes every input map, we get that $|P_l| = |P_{l-1}|$, and $w_l = w_{l-1}/k$, assuming that $w_l \equiv 0 \pmod{k}$.

With a patch-based (window-sliding) approach, we assume that $w_0 = s$, the size of the input patch. Instead, let's take an input image $s > w_0$. Define F_l as a set of *fragments*, where each fragment f (indexed $1..F_l$) is associated with a set of I_l^f *extended* maps, each of the same size. Define $s_{x,l}^f$ and $s_{y,l}^f$ as the width and height of the extended map in I_l^f . Thus, for $l = 0$, we get $|F_0| = 1$ and $s_{x,0}^1 = s_{y,0}^1 = s$.

In the convolution layer l , the number of fragments is the same as in the input, $F_l = F_{l-1}$ and each extended map shrinks by the convolutional kernel size, that is $s_{x,l}^f = s_{x,l-1}^f - k + 1$ and $s_{y,l}^f = s_{y,l-1}^f - k + 1$. I_l^f is obtained by applying convolutional kernel to preceding map I_{l-1}^f in same way as in the window-sliding method.

The max-pooling layer computes the kernel at k^2 offsets for each fragment, thus $F_l = k^2 F_{l-1}$. Specifically, let I_{l-1}^f be the input extended map, and $O = \{0, 1, \dots, k-1\} \times \{0, 1, \dots, k-1\}$ the set of offsets. Then for each offset $(o_x, o_y) \in O$, an output extended map I_l^f is created, such that each of its pixels (x_o, y_o) corresponds to the maximum value of all pixels (x, y) in I_{l-1}^f :

$$o_x + kx_o \leq x \leq o_x + kx_o + k - 1$$

$$o_y + ky_o \leq y \leq o_y + ky_o + k - 1$$

While the number of output maps is k^2 times the number of input maps, each output map's size decreases by a factor of k : $s_{x,l}^f = \lfloor \frac{s_{x,l-1}^{f'} - o_x}{k} \rfloor$ and $s_{y,l}^f = \lfloor \frac{s_{y,l-1}^{f'} - o_y}{k} \rfloor$.

Lets compute the theoretical speed up in convolutional layers. Let C_l be the number of floating point calculation required per layer l . In the sliding-window approach, let $|P_l|$ be the total number of maps, s the size of the image, w_l the size of the map, then:

$$C_l = s^2 \cdot |P_{l-1}| \cdot |P_l| \cdot w_l^2 \cdot k_l^2 \cdot 2$$

The extra factor of 2 comes from performing one addition and one multiplication per weight.

In the fully convolutional approach, the number of operations is given by:

$$C_l = s_{x,l} \cdot s_{y,l} \cdot |P_{l-1}| \cdot |P_l| \cdot F_l \cdot k_l^2 \cdot 2$$

The theoretical speedup over the sliding window can be now calculated. Assuming a 10-layer architecture similar to N3 (see Appendix B, we obtain significant speedup per layer, see Table 5.2.

Table 5.2: Theoretical speedup

Layer	s	s_{l-1}	$ P_{l-1} $	$ P_l $	w_l	k_l	F_l	window FLOPS $\times 10^9$	fully FLOPS $\times 10^9$	speedup
1	512	559	1	48	92	4	1	3408	0.5	7114.8
3	512	279	48	48	42	5	4	53271	35.9	1485.1
5	512	139	48	48	18	4	16	6262	22.8	274.7
7	512	69	48	48	6	4	64	695	22.5	30.9

Experimental speedup

The Caffe library has an implementation of the fully convolutional approach in one of the experimental branches [49], but we unfortunately did not manage to adopt it to our production code. Instead, we provide an implementation Julia - a high-level dynamic, scientific programming language [39]. The implementation is based on Mocha.jl [52], so the same Caffe models (parameters) can be re-used. Benchmarked against a pure Julia sliding window approach, the fully convolutional (also pure Julia) approach achieves almost two-orders of magnitude speedup.

A colleague in our lab implemented this approach in highly-optimized C++ and managed to compute an entire 1024×1024 image in roughly 2 seconds on a multicore machine (CPU only). That's an improvement of 900x over the initial (GPU-based) 30-minute running time.

5.3 Combined RF and CNN

Random forest and deep neural network classifiers are not mutually exclusive. In a recent Microsoft Research publication [43], the random forest was put on top of the deep neural network obtaining state-of-the-art results on MNIST and ImageNet data sets. Similarly, due to a significant speed advantage of the random forest approach (5.1) over the sliding-window convolutional neural network (5.2.4) we also implemented different combined approaches to take advantage of the best of both worlds.

The first approach is based on sparse sub-sampling of the CNN. Because the RF models tend to be orders of magnitude faster, but less accurate, first, the entire

EM volume is classified with a RF to generate probability maps, effectively creating 2-dimensional supervoxels. Then a step similar to agglomeration (Chapter 7) is performed - we iterate over all membranes in the probability map, sample $K = \frac{1}{100}$ pixels, and re-classify them with a (slow, but more accurate) CNN. After thresholding such sub-sampled membranes, it is decided whether they should stay classified as membranes, or not (effectively, merging two adjacent supervoxels). This gives us the effect of correcting the RF with the CNN at the cost of $\frac{1}{100}$ of the running time of the CNN.

The second approach is based on *ensembles* of probability map predictions (similar to [48]). Instead of using only one RF model, an ensemble of models is trained with different starting hyper-parameters. The ensemble's output can be either averaged, or further optimized with a (sparse) linear regression. Although running an ensemble slows down prediction time proportionally to the number of models in the ensemble, we observed higher accuracy, even for a small ensembles (2-10 models). While our CNN implementation is too slow for our purposes, in theory, ensembles of CNNs (or both CNNs and RFs) could also benefit from this approach.

The third approach uses two consecutive RFs, where the second RF takes as its input the first RF's output. This resembles a two-layer neural network and has the effect of correcting the errors generated by the first RF. Interestingly, we observed a higher pixel-accuracy than by only using the previous two approaches. Furthermore, all three methods can be combined together, effectively generating a *directed acyclic graph (DAG)* of classifiers (see Chapter 9).

5.4 Training Sets

We experimented with multiple training data sets, using different ground truth transformations and pre-processing steps. Table 10.1 shows six data sets generated and the models that achieved the highest pixel accuracy on each of them, as well as their training times.

All data sets are generated from the ISBI training set, as its EM and ground

Table 5.3: Training Data Sets

Data Set Name	Best Model	Pixel Accuracy	Training time
ISBI49	AlexNet	85%	14 min
ISBI256	GoogLeNet	88%	5 hr 48 min
Doubles	AlexNet	81%	14 min
ISBI Skewed 60	AlexNet	88%	43 min
ISBI Skewed 70	AlexNet	89%	58 min
ISBI65	N3	85%	15 hr

truth have the same resolution as our microscope. The *ISBI49* data set consists of 90,000 patches, each of size 49px by 49px. There are 60,000 training patches, 20,000 validation patches and 10,000 test patches. Similarly, the *ISBI256* set uses patches of size 256px by 256px, with the same split in between the sets. The *Doubles* data set uses 49px by 49px patches with the double membrane ground truth transformation.

The *ISBI Skewed 60* uses 200,000 patches of size 49px by 49px, but they are biased towards membranes. 130,000 train patches are split with proportions 60% of membranes and 40% of non-membranes (80,000 : 50,000). Similarly 50,000 of the validation patches and 20,000 of the test patches are biased in the same proportion. The *ISBI Skewed 70* uses 70% to 30% ratio instead. Our intuition to use skewed data sets was to predict membranes with higher accuracy, at the cost of (potentially) misclassifying extracellular space or cell's body as membranes. This has an effect of creating thicker membranes, thus increasing the split error and decreasing the merge error, which is desirable.

Lastly, the *ISBI65* set is an un-skewed set of 200,000 patches of dimensions 65px by 65px (a bigger neighborhood).

Chapter 6

Oversegmentation

In order to efficiently segment a volume, we first need to perform oversegmentation, that is, group individual pixels in the volume into supervoxels. The intuition behind the supervoxel primitive (in 2D, superpixel) is to group similar and meaningful regions together, replace the rigid pixel grid representation and capture image redundancy, such that the complexity of following pipeline stages is reduced [2]. While it is much easier to segment (or, merge) features computed from irregular supervoxels, as they are more expressive than a rigid pixel neighborhood, we need to ensure that no supervoxel straddles more than one atomic region. In particular, it is better in our case to heavily oversegment an image, thus introduce many split errors, than to under-segment an image, and merge regions that should not be merged together.

Oversegmentation is a well-studied problem and there exist multiple algorithms to generate it, such as *simple linear iterative clustering* (SLIC), *watershed* or *random walker* [27]. In this chapter, we concentrate on the watershed method, briefly cover popular implementations and provide our much faster sequential and parallel implementations.

6.1 Watershed

Watershed is a popular image segmentation technique. We begin with a 3-dimensional stack of probability maps, where each pixel represents a probability of being a mem-

brane, scaled and discretized to 8-bit integers, in the range [0, 256]. The goal is to label spatially neighboring pixels into supervoxels of the same (arbitrary) label. On the high level, watershed considers the input image as a *topographic surface* and places water (fluid) sources in regional minima of its relief. Once the entire relief is flooded from the sources, the borders are put in places (pixels) where different sources meet. At that time, all points at the surface of a given minimum constitute the *catchment basin*. Alternatively, one can visualize the watershedding process as continuously and uniformly increase in fluid level from the minimum value (0 in our case) to the maximum value (256), until the entire image is covered [51].

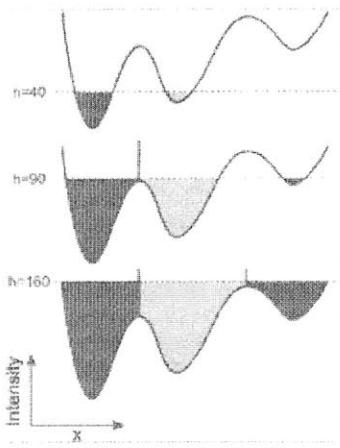


Figure 6-1: Watershed flooding

There are other, equivalent, interpretations of the watershed. Intuitively, the catchment basins define the supervoxels where the fluid "dropping" on a pixel follows the path to the "nearest" minimum, that is, follows the path of the steepest descent, thus ultimately reaching a minimum. This was first algorithmically implemented in [7], and does not produce separating borders (pixel-wide lines between objects). The alternative approach, with borders, was suggested by [18].

There exists a surprising number of incongruent watershed implementations that vary by parametric settings. Since we further pass the outcome of our watershed to NeuroProof, we have to make sure that our implementations are acceptable by that package. The following section covers two open-source implementations and provides our own, much faster, implementation.

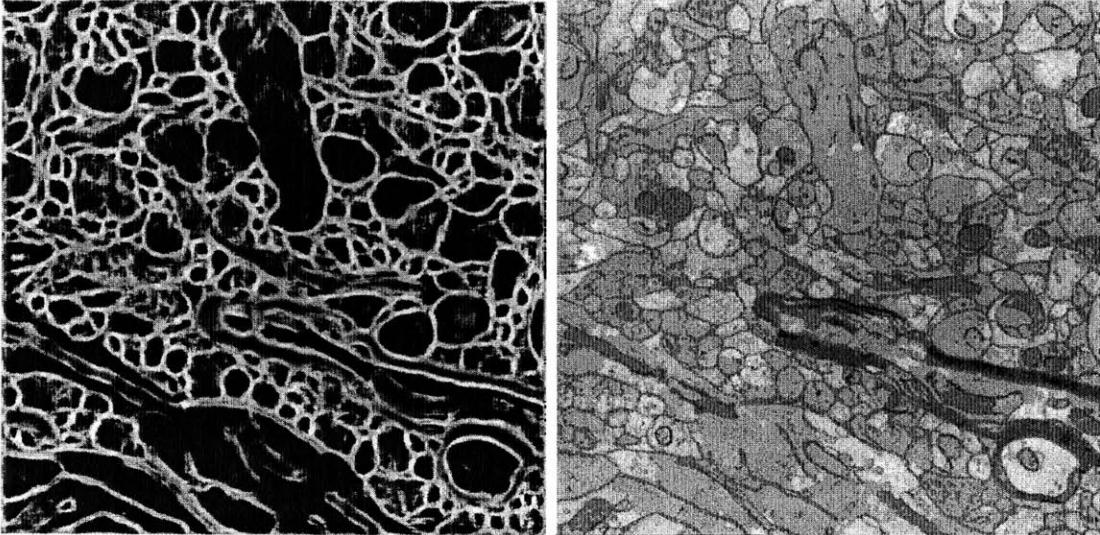


Figure 6-2: Left: input probability map. Right: Generated watershed oversegmentation

6.1.1 GALA’s Implementation

GALA directly uses the Python SciKit’s watershed implementation [40], which in turns implements [71]. By default, GALA initiates the watershed seeds ("minima") by choosing all connected components with pixel value of 0 (from the range 0..255) of size ≥ 5 . In 2D it uses 4-connectivity (a cross), while in 3D uses 6-connectivity. It also performs a number of general-purpose transformations (ranking the values, padding the image etc.) for the convenience of implementation and the theoretical a speed-up. In essence, the core algorithm implements the following:

While this is a widely-used implementation, it is too general-purpose for oversegmentation of 8-bit probability maps, and suffers from significant memory overhead and poor performance (despite being written in Cython and being compiled). While processing a 100 megapixel volume (100x1024x1024 8-bit pixels, 100MB uncompressed in memory), we observed a working memory set of about 140x that size (14GB), which is unacceptable, not only due to the sheer size, but also because it overfills CPU caches and decreases performance. Additionally, the processing time was over 2 minutes (single-threaded implementation), which would not scale for our data sets.

Algorithm 2 GALA’s Watershed

```
1: procedure GALAWATERSHED(seeds)
2:   Mark each seed with a unique label (one per connected component)
3:   Initiate queue with these seeds
4:   while not queue is empty do
5:     s  $\leftarrow$  queue.pop()
6:     for pixel q in s’s neighborhood do
7:       if q not processed yet then
8:         marker[q]  $\leftarrow$  marker[s]
9:         enqueue(q, height[q])
10:        mark q processed
11:       end if
12:     end for
13:   end while
14: end procedure
```

6.1.2 OpenCV’s Implementation

OpenCV implements a similar queue-based approach to GALA’s, and additionally marks the output with a pixel-wide boundary around objects, which is not good for our needs. Despite being written natively in C++, it suffers from similar memory over-consumption and inefficient cache access patterns. The running time for the 100 megapixel volume is about 110 sec (single threaded).

6.1.3 Serial Implementation

To avoid the memory and speed limitations of GALA, we re-implemented the SciKit algorithm and introduced four main enhancements: smaller memory footprint, an efficient processing queue, a decreased number of pre-processing steps, and sequential file processing.

First, we realized that since our probability maps store 8-bits of information per pixel, it does not make sense to rank and re-label the entire volume. This saves us one pass on the entire volume, and removes the need to cast values to 32-bit integers.

Then, we decided to use only three buffers: one 8-bit buffer for the input probability map, one 32-bit buffer for the output labels and one 64-bit buffer for reverse indexing into the output buffer, all of the same dimensions as the input volume. Ad-

ditionally, we re-used the third buffer for the watershed's processing queue. Each pixel in the first two buffers is accessed exactly three times (initialization, seed selection, watershed's queue), while the reverse index is accessed exactly twice (during initialization and in the queue).

Instead of using a queue or heap of elements with three keys (index, value and timestamp), which in SciKit takes 128-bits total, we use a single queue built on the reverse-index buffer. We noticed that the timestamps enqueued per each value are monotonically increasing, eliminating the need for a general-purpose queue, allowing us to just use an array. Here's the queue implementation:

Algorithm 3 Watershed Queue Push

```

1: procedure WATERSHEDQUEUEPUSH(index, value)
2:   if value < queue.min_value then
3:     queue.min_value ← value
4:     indexBuffer[tail[value]] ← index
5:     tail[value] += 1
6:   end if
7: end procedure

```

Algorithm 4 Watershed Queue Pop

```

1: procedure WATERSHEDQUEUEPOP
2:   while queue.min_value < 256 do
3:     if head[queue.min_value] < tail[queue.min_value] then
4:       head[queue.min_value] += 1
5:       return indexBuffer[head[queue.min_value] - 1]
6:     end if
7:   end while
8:   return -1                                ▷ not found - the queue is empty
9: end procedure

```

Using this monotonically increasing queue, the memory working set decreased from 140x to 13x, and the single-threaded running time is about 0.1s per megapixel of probability maps (including 0.03s spent in I/O), which is on par with our fastest probability map generate algorithm.

6.1.4 Parallel Watershed

While the single-core serial implementation's speed is adequate, there is space for multi-core parallelization for further speed up. There are two parallelization approaches.

First, we can split the input volume into sub-volumes of (roughly) the same size and execute the watershed algorithm as an independent process for each of the sub-volumes (for example, one process per core). This gives us a theoretical linear speed up, but due to CPU cache contention does not scale linearly with the number of cores available. Additionally, this approach increases over-segmentation, as the fluid does not spill in-between the volumes, thus more split-errors are introduced (but, generally, no merge-errors should be introduced). We hope that the agglomeration step will be able to correct for this issue.

The second approach is to actually parallelize the watershed algorithm internally. In essence, this means we have to introduce a thread-safe queue in place of 6.1.3 and 6.1.3.

6.2 Alternative Approaches

There exist other approaches to generate oversegmentation that we did not evaluate, but we briefly survey here.

The *random walker* algorithm ([28], [15]) also begins with a set of seeds, but instead of increasing fluid elevation, calculates probabilities for a random walk. Specifically, the input probability map is modeled as a graph, where each pixel corresponds to a node connected to other nodes in its (pixel) neighborhood, and each edge is assigned a probability (as a function of intensity, color etc.). Then, for each pixel, the probability of a random walk reaching all seeds is calculated, and the seed with highest probability is chosen as this pixel's label.

The *simple linear iterative clustering (SLIC)* [2] is an adaption of k -means clustering with two modifications. Firstly, the number of distance calculations is reduced by limiting the search space. It becomes proportional to the superpixel size, linear

in the total number of pixels in the input image and independent of the number of supervoxels. Secondly, the weighted distance function combines both the spatial proximity and color, giving control over size and compactness of the supervoxels.

6.3 Other Heuristics

We noticed that the 3-dimensional watershed sometimes spills in between the z-index slices and generates too few objects (*under-segments*). This is caused by the input volume anisotropy: pixels on the x, y -plane are 4 nm apart, while the z -index is separated by 27 nm, resulting in bigger and less-smooth translation of a membrane between two z -index slices. This has turned out to be a major problem: the thinner and more accurate the probability maps are, the more spilling and (ultimately) merge errors become.

The issue is temporarily fixed with a single post-processing step. The morphological operation of closing on the z -axis is performed. The closing of A by a structuring element B is defined as dilation of A by B , followed by erosion of the resulting structure by B . Our structuring element is a 3-pixel long segment in the z -direction.

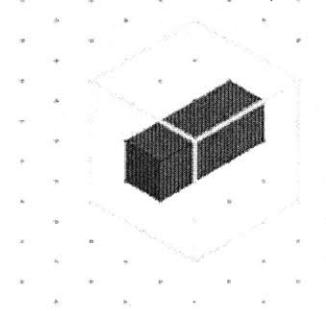


Figure 6-3: Structuring element for z-closing

Chapter 7

Agglomeration

While the middle step of the pipeline (oversegmentation) provides us with some spatial information, by itself it is not enough to render the connectivity of neurons. The goal of agglomeration is to merge adjacent supervoxels generated in oversegmentation, in a way that resembles neurons' physical shape. Similarly as in the previous step, we would like to avoid merge errors at all cost, since splitting an object is hard, while we find it acceptable to have a small number of split errors.

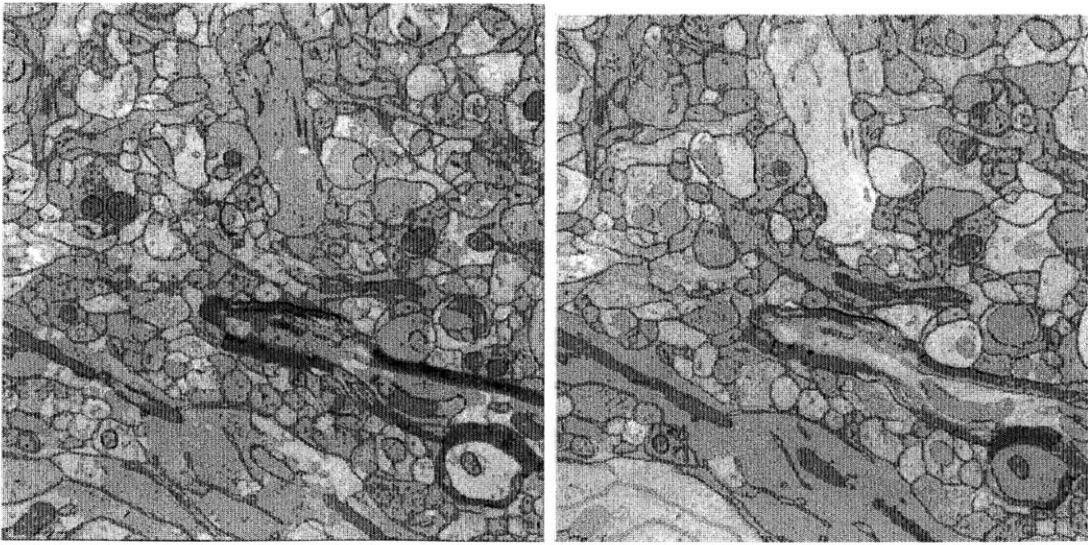


Figure 7-1: Left: Oversegmentation. Right: Segmentation

Both GALA and Neuroproof implement essentially the same random forest-based *agglomeration (hierarchical clustering)* algorithm [56]. First, they construct a *regional*

adjacency graph (RAG) by iterating over all pixels in oversegmentation and creating a graph node for each unique label, and an edge between different adjacent labels. Then each edge is assigned a probability of being a valid boundary in the segmentation (a function of probability map pixel values).

The merging process works in a loop by extracting the edge with lowest probability from the priority queue and merging the edge’s nodes. Once two nodes are merged, the values of neighboring edges might have to be re-calculated and the queue updated. The process repeats until a parametric threshold value is reached. Both linear procedures have been parallelized by our colleague in [54].

Similarly to using the ensemble of random forest classifiers to generate a probability map (section 5.3), multiple probability maps can be provided to NeuroProof to increase the accuracy of agglomeration. By default, the package accepts multiple channels of class predictions, with channel 0 always treated as the boundary channel and channel 2 treated as the mitochondria channel (if mitochondria detection is enabled). Currently there are two probability maps passed to Neuroproof and mitochondria detection is turned off. However, our colleagues are implementing other biological features that are expected to further improve agglomeration.

As the last phase of the pipeline that still requires high-density input data, agglomeration must be implemented as a distributed process. Watershed-based oversegmentation can be easily distributed by splitting the input volume into smaller sub-volumes that fit into memory. For physical objects split on the sub-volume boundary, this process assigns (at random) different labels to the oversegmentation, effectively deferring the label-merging process to agglomeration. We do not have a straightforward solution to this problem. By re-generating some of the original RAGs in two neighboring sub-volumes, Neuroproof might be able to correctly merge labels. But this approach has not been evaluated yet.

Chapter 8

Skeletonization

For the study of neuronal connectivity, we are not directly interested in a pixel-accurate physical shape of neurons, but rather just their connectivity with synapses. Ideally, we would like to trace the center lines of each object, an operation called *skeleton tracing (skeletonization)*, and thus deduce the connectome. Skeletonization is a popular research problem, and some solutions for high-accuracy and high-throughput skeletonizations had already been suggested [31].

A colleague of ours implemented a voxel-based thinning approach based on surface points [70]. The algorithm satisfies centeredness (skeleton is geometrically centered within the boundary), preserves connections and topology, stays invariant to shift, zoom and rotation and aims be as thin as possible (1-voxel). The algorithm defines 38 *simple points* in a 3-dimensional lattice and proceeds to thin the objects, starting with the boundaries, such at the simple points (their connectivities) are always preserved. It runs in linear time (in the number of pixels) and with additional coarsening is sufficiently fast.

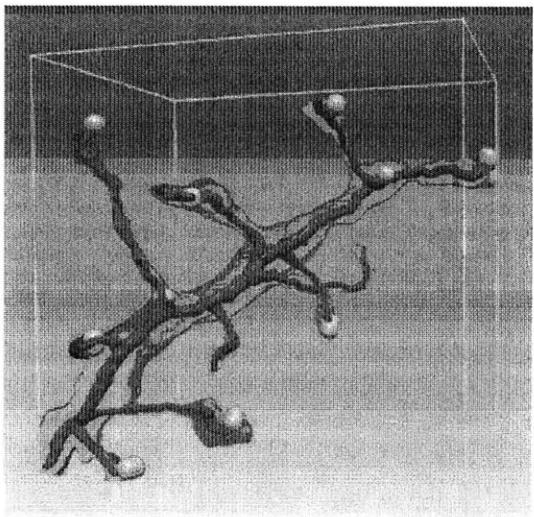


Figure 8-1: An example of a skeletonized object

Chapter 9

MapRecurse

Coming into the Connectomics project, the initial goal of our research was to propose a new, efficient and concurrent computational framework for multicore machines. However, throughout the challenges encountered, the research focus has significantly diverted. In this chapter we provide an overview of the developed framework.

Due to the size of the connectome data, computations need to be performed in a distributed manner, with majority of data residing out of memory. A volume of brain tissue as well as image segmentation have highly spatial properties, and, because the brain's regions differ in their complexity, it is desirable to be able to run different algorithms on different brain regions.

The framework should support a generic mechanism to split the input volume into smaller sub-volumes of dynamic size (or complexity), run computations on each sub-volume, merge back the results and verify that the processes succeeded. If the verification fails, the framework needs to provide a re-try mechanism.

MapRecurse is heavily based on Google's MapReduce framework [19]. Since MapReduce was released in 2004, it has been well tested in practice, with an entire eco-system of frameworks built on top of it [77].

The framework is implemented in C++11 using variadic templates and all its logic is generated at compile-time with little run-time overhead. It enables recursion in the map step, in the reduce step, as well as across the entire job. It is currently non-distributed, but it is easily parallelizable with Cilk [25], assuming that the underlying

data structures (problem-specific) are thread-safe.

The user can cascade multiple steps of map and reduce functions, and after each step can also (optionally) specify a verifier. The verifier checks correctness of the previous step (map or reduce) and directs data forward to the next step if the output is correct, or sends it back to the previous step, where it is recalculated with a different algorithm (specified in the variadic template). Additionally, there is the global verifier that can re-do calculations starting with the first step. This enables the user to create (limited) computational graphs.

MapRecurse has not yet been adapted to the connectomics pipeline. It was evaluated on small example problems: a prime number generator and a BTree creation and merging logic. It is hoped that it can generalize beyond just the application to connectome.

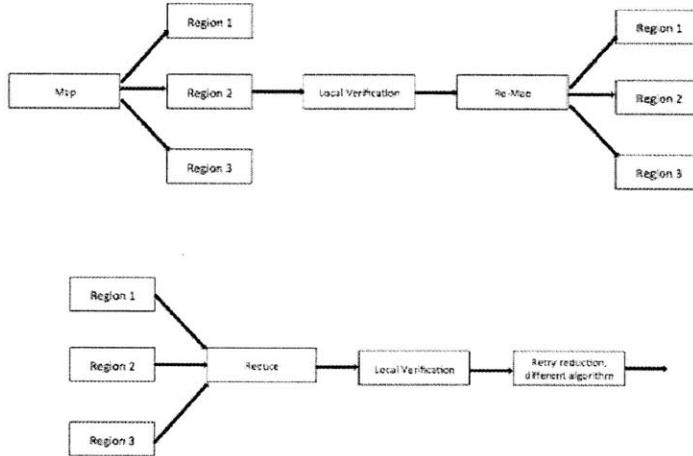


Figure 9-1: MapRecurse framework. Top: map phase. Bottom: reduce phase

9.1 Other Frameworks

Janelia Farm Research has recently released Spark utilities for Neuroproof [53]. Spark is a in-memory computing framework well-suited for distributed machine learning algorithms. The newer version of the packages comes with multiple workflows, such as

evaluating the similarity between two image segmentations, RAG building and large-scale image segmentation. Internally it uses Janelia’s *DVID* (*distributed, versioned, image-oriented dataservice*). While the package looks promising, we have not evaluated it in practice yet.

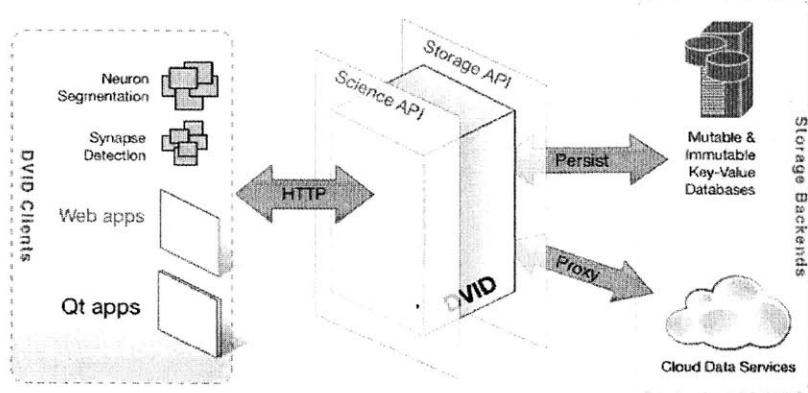


Figure 9-2: Neuroproof’s Spark

The Google Brain team has released Tensorflow as a library for numerical computations using data flow graphs, with a special focus on machine learning. Nodes in the graph represent mathematical operation, while the graph edges represent the multidimensional data arrays (tensors) communicated between them [1]. With a multi-architecture implementation (CPUs and GPUs) and an unrestricted computational graph, it seems more flexible than MapReduce.

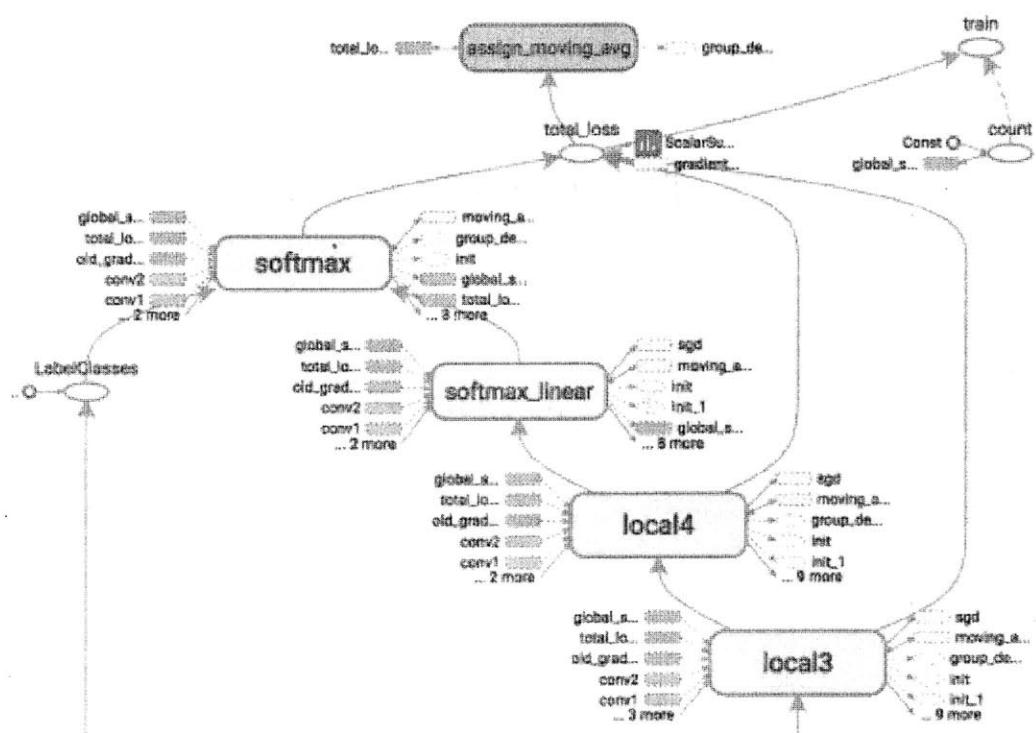


Figure 9-3: Tensorflow

Chapter 10

Evaluation

The performance of our algorithms should be evaluated with the application in mind. On the lowest *pixel accuracy* can be used for each pixel classification, and this is the measure used in training probability map classifiers (RFs and CNNs). However, in image segmentation we are more concerned with accurate labeling of the entire image (as compared with the ground truth) rather than individual pixels. Thus, a higher level measure is more appropriate.

The *Rand Index (RI)* is a measure of the similarity of two different clusterings of a given volume [64]. Consider a lattice defined by the merge-split operation on clusterings. Let C and C' be the two clusterings, with a function f defined on them. Let $C = \{C_1, C_2, \dots, C_k\}$ and $n_i = |C_i|$. The distance function between two clusterings is defined as:

$$d(C, C') = f(C) + f(C') - 2f(C \wedge C')$$

where $C \wedge C'$ is the join of the two clusterings in the lattice. Then $f(C) = \sum_i^k n_i^2$ becomes the RI.

The *Variation of Information (VI)* is another measure closely related to mutual information between two clusterings. By setting $f(C) = \sum_i^k n_i \log n_i$ we get the VI. This is our primary measure used compare against other state-of-the-art connectome pipelines. Because we are mostly concerned with the merge error, in table ?? the

Table 10.1: Variation of Information

Pipeline Name	Split Error	Merge Error	Total VI
NIPS	1.4164	0.6035	2.0199
Random Forest	1.7870	0.7069	2.4938
Ax-MO	1.5941	0.7915	2.3856
CNN (AlexNet)	1.6032	0.7889	2.3921

total VI is separated into split and merge errors.

Aside from evaluating the pipeline on the ISBI data sets (for which we have 100MB of the ground truth), we also run the pipeline on the 80GB kasthuri11 data set. For the evaluation of probability map generation see Chapter 5, and for the performance evaluation of oversegmentation see Chapter 6.

Chapter 11

Summary and Outlook

The research in this thesis focused on the problem of automated connectome reconstruction from electron microscopy images of a mammalian brain. Towards solving this problem, we proposed a multi-stage pipeline based on machine learning algorithms, starting with aligned images and ending with a skeleton. Our particular contributions are in the earliest stages of the pipeline: probability map and oversegmentation generation.

In Chapter 3 we provided a high-level architecture of the pipeline, and in Chapter 9 we proposed an extension for more advanced computation flows.

Machine learning-based image segmentation, in particular the probability map generation, is the core focus of this thesis. In Chapter 5 we proposed two approaches to image segmentation: the random-forest and the convolutional neural network. We implemented both of them and provided a quantitative analysis. Because the sliding-window CNN achieved much lower error rate, at a very high computational cost, we proposed two performance improvements. The fully convolutional network offers a two-orders of magnitude theoretical speedup and the (early) experimental results are promising. The combined RF and CNN sampling approach is both more accurate and fast. Our classifiers were trained on multiple data sets, based on different ground truth transformations and labels non-uniformity.

In Chapter 6 we described image oversegmentation based on the watershed algorithm. We performance engineered our watershed implementation to offer two-orders

of magnitude improvement in speed and one-order of magnitude decrease of memory working set.

We concluded the pipeline with a brief description of the agglomeration and skeletonization stages. As of this writing, the pipeline can process in-memory data sets in range 100 GB - 1 TB, and both the probability map generation and the watershed algorithms can process much bigger data sets by splitting the input into smaller sub-volumes (sequentially or in parallel).

There are multiple potential improvements to the pipeline. On the performance side, we can generate probability maps and oversegment at the order of 1 MB per second. This is fast, but the microscope is about two orders of magnitude faster, thus we are still the blocking stage. Furthermore, multi-core performance engineering could bring the pipeline closer to the desired real-time processing speed.

With improved splitting and merging techniques, we might be able to provide high-accuracy agglomeration. Having all steps of the pipeline distributed across many multi-core systems, we should be able to gain another order of magnitude in throughput.

Oversegmentation with watershed is not necessarily the most accurate technique, as the fluid tends to spill in between the z -index slices. To mitigate this problem, it seems intuitive to use both 2D and 3D deep neural networks to further decrease pixel error rate and the variation of information.

Incorporating biological prior knowledge should also improve the agglomeration step. Currently, only the membrane and synapses probability maps are consider in Neuroproof. Efforts are being undertaken to include other biological features, such as the axons' skeleton probability map.

Lastly, current visualization software is adequate only for data sets on the order of hundreds to thousands MB. With the processing volumes soon exceeding the TB range, we will need to address the issues visualization and biological information extraction.

We are optimistic about the outlook of the connectome project. We believe we will soon see interesting, novel neuroscientific results.

Appendix A

Nervous System of *Caenorhabditis elegans*

Consists of 302 neurons.

THE MIND OF A WORM

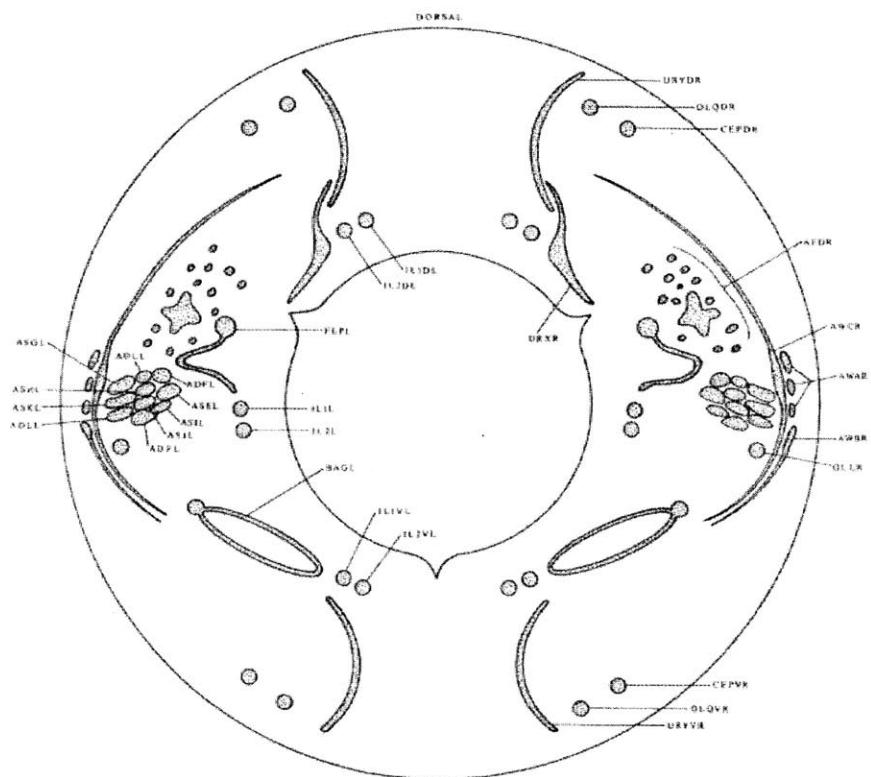


Figure A-1: C. Elegans's nervous system

Appendix B

Visualization of Convolutional Neural Networks

An example of the network specification file.

```
1: procedure NETWORKSPECIFICATION
2:     input: "data"
3:     input_dim: 1
4:     input_dim: 1
5:     input_dim: 49
6:     input_dim: 49
7:     layer {
8:         name: "conv1"
9:         type: "Convolution"
10:        bottom: "data"
11:        top: "conv1"
12:        param {
13:            lr_mult: 1.0
14:            decay_mult: 1.0
15:        }
16:        param {
17:            lr_mult: 2.0
18:            decay_mult: 0.0
19:        }
20:        convolution_param {
21:            num_output: 96
22:            kernel_size: 11
23:            stride: 4
24:            weight_filler {
25:                type: "gaussian"
26:                std: 0.01
27:            }
28:            bias_filler {
29:                type: "constant"
30:                value: 0.0
31:            }
32:        }
33:    }
34:    layer {
35:        name: "relu1"
36:        type: "ReLU"
37:        bottom: "conv1"
38:        top: "conv1"
39:    }
```

```
40: layer {
41:     name: "norm1"
42:     type: "LRN"
43:     bottom: "conv1"
44:     top: "norm1"
45:     lrn_param {
46:         local_size: 5
47:         alpha: 0.0001
48:         beta: 0.75
49:     }
50: }
51: layer {
52:     name: "conv4"
53:     type: "Convolution"
54:     bottom: "norm1"
55:     top: "conv4"
56:     param {
57:         lr_mult: 1.0
58:         decay_mult: 1.0
59:     }
60:     param {
61:         lr_mult: 2.0
62:         decay_mult: 0.0
63:     }
64:     convolution_param {
65:         num_output: 384
66:         pad: 1
67:         kernel_size: 3
68:         group: 2
69:         weight_filler {
70:             type: "gaussian"
71:             std: 0.01
72:         }
73:         bias_filler {
74:             type: "constant"
75:             value: 0.1
76:         }
77:     }
78: }
79: layer {
80:     name: "relu4"
81:     type: "ReLU"
82:     bottom: "conv4"
83:     top: "conv4"
84: }
```

```
85:    layer {
86:        name: "fc6"
87:        type: "InnerProduct"
88:        bottom: "conv4"
89:        top: "fc6"
90:        param {
91:            lr_mult: 1.0
92:            decay_mult: 1.0
93:        }
94:        param {
95:            lr_mult: 2.0
96:            decay_mult: 0.0
97:        }
98:        inner_product_param {
99:            num_output: 4096
100:            weight_filler {
101:                type: "gaussian"
102:                std: 0.005
103:            }
104:            bias_filler {
105:                type: "constant"
106:                value: 0.1
107:            }
108:        }
109:    }
110:    layer {
111:        name: "relu6"
112:        type: "ReLU"
113:        bottom: "fc6"
114:        top: "fc6"
115:    }
116:    layer {
117:        name: "drop6"
118:        type: "Dropout"
119:        bottom: "fc6"
120:        top: "fc6"
121:        dropout_param {
122:            dropout_ratio: 0.5
123:        }
124:    }
```

Algorithm 5 An Example of the Network Specification File

```
125:   layer {
126:     name: "fc8"
127:     type: "InnerProduct"
128:     bottom: "fc6"
129:     top: "fc8"
130:     param {
131:       lr_mult: 1.0
132:       decay_mult: 1.0
133:     }
134:     param {
135:       lr_mult: 2.0
136:       decay_mult: 0.0
137:     }
138:     inner_product_param {
139:       num_output: 2
140:       weight_filler {
141:         type: "gaussian"
142:         std: 0.01
143:       }
144:       bias_filler {
145:         type: "constant"
146:         value: 0.0
147:       }
148:     }
149:   }
150:   layer {
151:     name: "prob"
152:     type: "Softmax"
153:     bottom: "fc8"
154:     top: "prob"
155:   }
156: end procedure
```

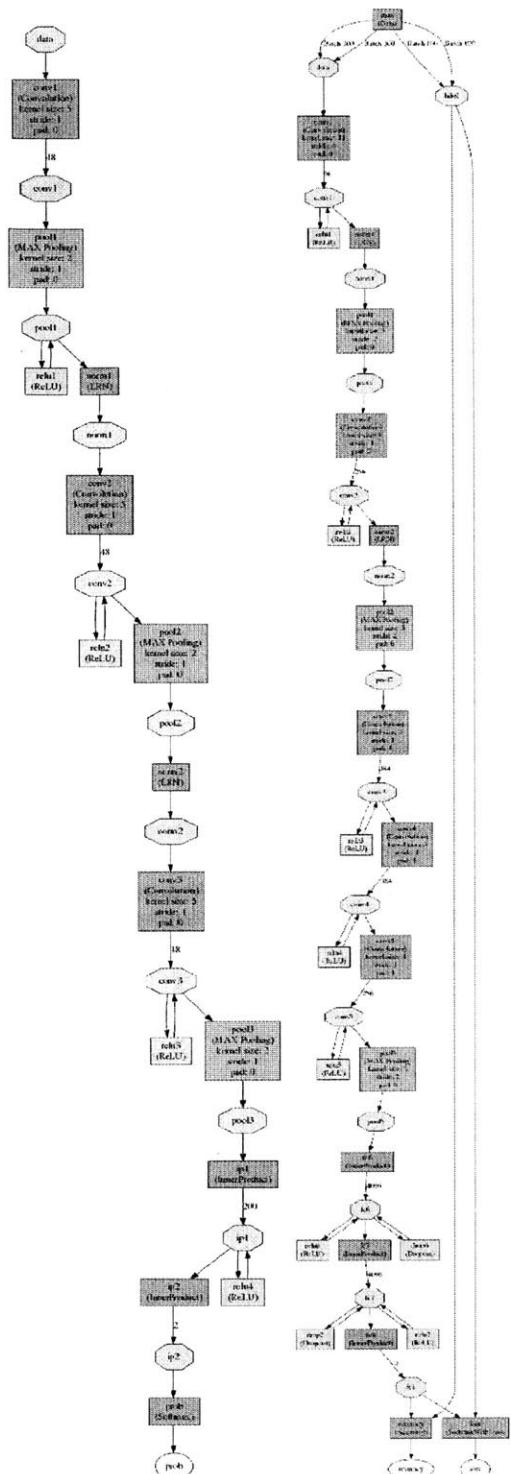


Figure B-1: Left: N3, right: AlexNet

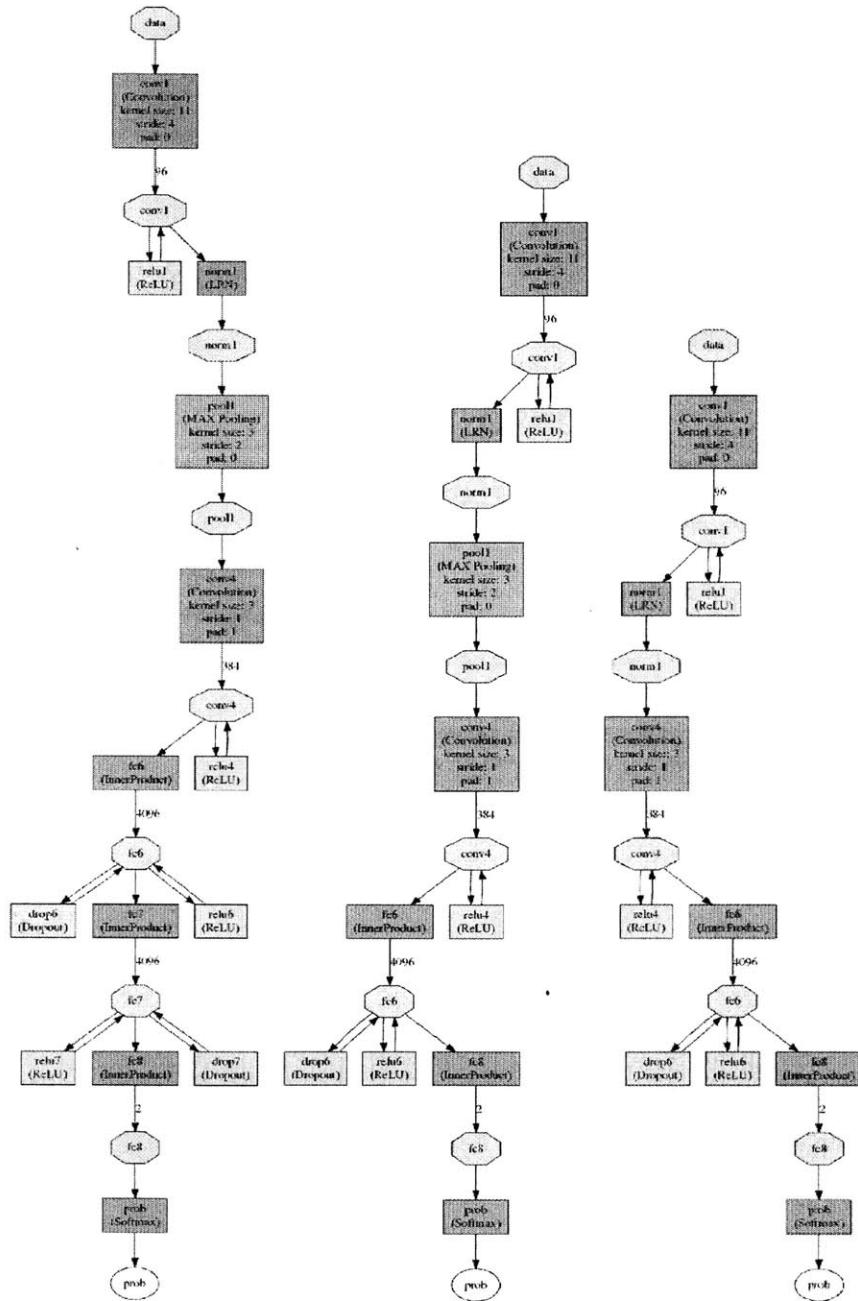


Figure B-2: Left to right: ShortNet1, ShortNet2, ShortNet3

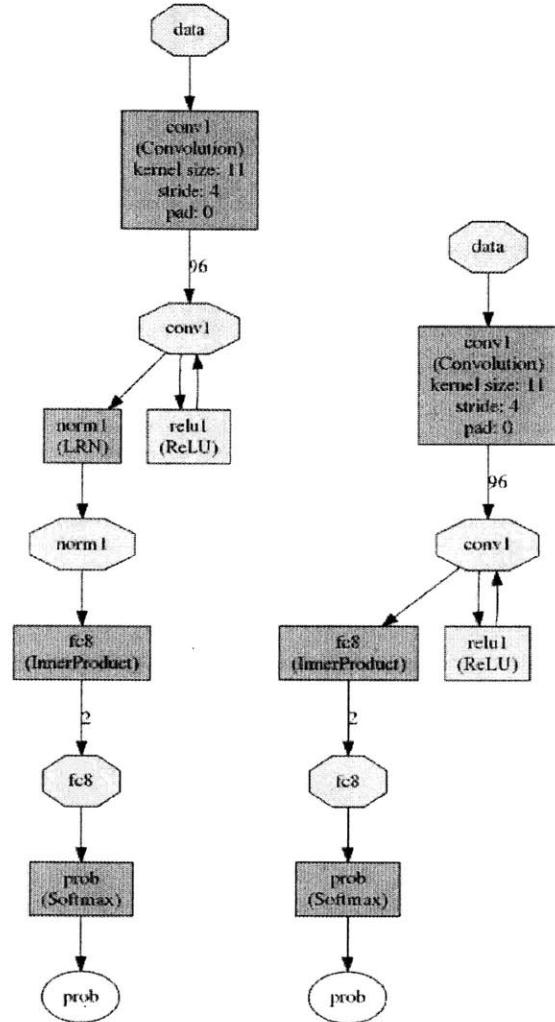


Figure B-3: Left: BN1, Right: BN2

Bibliography

- [1] Martín Abadi and Ashish Agarwal et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(11):2274–2282, November 2012.
- [3] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. *CoRR*, abs/1511.06435, 2015.
- [4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.
- [5] George Bebis. University of nevada, reno, cs791e, computer vision, lecture notes. <http://www.cse.unr.edu/~bebis/CS791E/Notes/EdgeDetection.pdf>. Accessed: 2015-12-06.
- [6] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(9):1–127, 2009.
- [7] S. Beucher and F. Meyer. The morphological approach to segmentation: the watershed transformation. Mathematical morphology in image processing. *Optical Engineering*, 34:433–481, 1993.
- [8] Davi D. Bock, Wei-Chung Allen Lee, Aaron M. Kerlin, Mark L. Andermann, Greg Hood, Arthur W. Wetzel, Sergey Yurgenson, Edward R. Soucy, Hyon Suk Kim, and R. Clay Reid. Network anatomy and in vivo physiology of visual cortical neurons. *Nature*, 471(7337):172–182, 2011.
- [9] E. S. Boyden, F. Zhang, E. Bamberg, G. Nagel, and K. Deisseroth. Millisecond-timescale, genetically targeted optical control of neural activity. *Nat. Neurosci*, 8(9):1263–1278, 2005.
- [10] Ed Boyden. A light switch for neurons. https://www.ted.com/talks/ed_boyden, 2011. Accessed: 2015-12-03.

- [11] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [12] K. L. Briggman, M. Helmstaedter, and W. Denk. Wiring specificity in the direction-selectivity circuit of the retina. *Nature*, 471(7337):183–8, 2011.
- [13] Albert Cardona, Stephan Saalfeld, Johannes Schindelin, Ignacio Arganda-Carreras, Stephan Preibisch, Mark Longair, Pavel Tomancak, Volker Hartenstein, and Rodney J. Douglas. TrakEM2 Software for Neural Circuit Reconstruction. *PLoS ONE*, 7(6):e38011–, June 2012.
- [14] Computational connectomics group. <http://ccg.csail.mit.edu/>.
- [15] Christophe Chefd’Hotel and Alexis Sebbane. Random walk and front propagation on watershed adjacency graphs for multilabel image segmentation. In *ICCV*, pages 1–7. IEEE, 2007.
- [16] F. Chen, P.W. Tillberg, and E.S. Boyden. Expansion microscopy. *Science*, 347(6221):543–548, 2015.
- [17] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.
- [18] Michel Couprie and Gilles Bertrand. Topological grayscale watershed transformation. In *IN SPIE VISION GEOMETRY V PROCEEDINGS, 3168*, pages 136–146, 1997.
- [19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [20] Nvidia digits library. <http://devblogs.nvidia.com/parallelforall/easy-multi-gpu-deep-learning-digits-2>.
- [21] Piotr Dollár and C. Lawrence Zitnick. Structured forests for fast edge detection. In *ICCV*, 2013.
- [22] N. F. Dronkers, O. Plaisant, M. T. Iba-Zizen, and E. A. Cabanis. Paul broca’s historic cases: high resolution mr imaging of the brains of leborgne and lelong. *Brain*, 130(5):1432–1441, 2007.
- [23] A.L. Eberle, S. Mikula, R. Schalek, J.W. Lichtman, M.L. Tate, and D. Zeidler. High-resolution, high-throughput imaging with a multibeam scanning electron microscope. *J Microsc.*, Jan, 2015.
- [24] Expansion microscopy. <http://expansionmicroscopy.org/>.
- [25] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA ’09*, pages 79–90, New York, NY, USA, 2009. ACM.

- [26] Alessandro Giusti, Dan C. Ciresan, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. *CoRR*, abs/1302.1700, 2013.
- [27] Leo Grady. Random walks for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(11):1768–1783, November 2006.
- [28] Leo Grady. Random walks for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(11):1768–1783, November 2006.
- [29] Daniel Haehn, Seymour Knowles-Barley, Mike Roberts, Johanna Beyer, Narayanan Kasthuri, Jeff W Lichtman, and Hanspeter Pfister. Design and evaluation of interactive proofreading tools for connectomics. *IEEE Transactions on Visualization and Computer Graphics*, 20:2466–2475, 2014.
- [30] Patric Hagmann. *From diffusion MRI to brain connectomics*. PhD thesis, Lausanne: EPFL, 2014.
- [31] Moritz Helmstaedter, Kevin L. Briggman, and Winfried Denk. High-accuracy neurite reconstruction for high-throughput neuroanatomy. *Nature Neuroscience*, 14(8):1081–1088, July 2011.
- [32] Human connectome project. <http://www.humanconnectomeproject.org/>.
- [33] Isbi: 3d segmentation of neurites in em images. <http://brainiac2.mit.edu/SNEMI3D/>.
- [34] V. Jain, J.F. Murray, F. Roth, S. Turaga, V. Zhigulin, K.L. Briggman, M.N. Helmstaedter, W. Denk, and H.S. Seung. Supervised learning of image restoration with convolutional networks. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, Oct 2007.
- [35] Janelia research lab. <https://www.janelia.org/>.
- [36] Nir Shavit Jeff W. Lichtman, Hanspeter Pfister. The big data challenges of connectomics. *Nat Neurosci*, 17(11):1448–1454, 2014.
- [37] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM ’14, pages 675–678, New York, NY, USA, 2014. ACM.
- [38] Barbara E. Jones. From waking to sleeping: neuronal and chemical substrates. *Trends in Pharmacological Sciences*, 26(11):578 – 586, 2005.
- [39] Julia programming language. <http://julialang.org/>. Accessed: 2015-12-06.

- [40] Lee Kamentsky. Scikit watershed implementation. <https://github.com/scikit-image/scikit-image/blob/master/skimage/morphology/watershed.py>. Accessed: 2015-12-06.
- [41] Verena Kaynig, Amelio Vazquez-Reina, Seymour Knowles-Barley, Mike Roberts, Thouis R. Jones, Narayanan Kasthuri, Eric Miller, Jeff Lichtman, and Hanspeter Pfister. Large-scale automatic reconstruction of neuronal processes from electron microscopy images. *arxiv*, 2013.
- [42] Verena Kaynig, Amelio Vazquez-Reina, Seymour Knowles-Barley, Mike Roberts, Thouis R Jones, Narayanan Kasthuri, Eric Miller, Jeff Lichtman, and Hanspeter Pfister. Large-scale automatic reconstruction of neuronal processes from electron microscopy images. *Medical Image Analysis*, 22:77–78, 2015.
- [43] P. Kontschieder, M. Fiterau, A. Criminisi, and S. Rota Bulo'. Deep neural decision forests. [winner of the david marr prize 2015]. In *Intl. Conf. on Computer Vision (ICCV), Santiago, Chile*, December 2015.
- [44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [45] Thorben Kroeger. *Learning-based Segmentation for Connectomics*. PhD thesis, Ruperto-Carola University of Heidelberg, 2014.
- [46] Andrew Larner and John Paul Leach. Phineas gage and the beginnings of neuropsychology. *Advances in Clinical Neuroscience and Rehabilitation*, 2(3):26, 2002.
- [47] Lichtman lab. <http://lichtmanlab.fas.harvard.edu/>.
- [48] Toyota Technological Institute at Chicago Liran Chen. Learning ensembles of convolutional neural networks.
- [49] Jonathan Long. Fully convolutional branch of caffe. <https://github.com/BVLC/caffe/wiki/Model-Zoo#fcn>. Accessed: 2015-12-06.
- [50] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CVPR (to appear)*, November 2015.
- [51] Fernand Meyer. The watershed concept and its use in segmentation : a brief history. *CoRR*, abs/1202.0216, 2012.
- [52] Mocha.jl library. <https://github.com/pluskid/Mocha.jl>. Accessed: 2015-12-06.
- [53] Neuroproof spark. <https://github.com/janelia-flyem/DVIDSparkServices>. Accessed: 2015-12-06.

- [54] Quan Nguyen. Parallel and scalable neural image segmentation for connectome graph extraction. Master's thesis, Massachusetts Institute of Technology, 2015.
- [55] Srivastava Nitish, C. Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [56] Juan Nunez-Iglesias, Ryan Kennedy, Toufiq Parag, Jianbo Shi, and Dmitri B. Chklovskii. Machine learning of hierarchical clustering to segment 2d and 3d images. *PLoS ONE*, 8(8):e71715, 08 2013.
- [57] Juan Nunez-Iglesias, Ryan Kennedy, Stephen M. Plaza, Anirban Chakraborty, and William T. Katz. Graph-based active learning of agglomeration (gala): a python library to segment 2d and 3d neuroimages. *Frontiers in Neuroinformatics*, 8(34), 2014.
- [58] Open connectome project. <http://www.openconnectomeproject.org/>.
- [59] Openconnectome: Vesicle. <https://github.com/openconnectome/vesicle>.
- [60] Toufiq Parag, Anirban Chakraborty, Stephen Plaza, and Louis Scheffer. A context-aware delayed agglomeration framework for electron microscopy segmentation. *PLOS ONE*, 10(5), 2015.
- [61] Toufiq Parag, Stephen Plaza, and Louis Scheffer. Small sample learning of superpixel classifiers for em segmentation. *MICCAI*, 2014.
- [62] D.P. Pelvig, H. Pakkenberg, A. K. Stark, and B. Pakkenberg. Neocortical glial cell numbers in human brains. *Neurobiology of Aging*, 29(11):1754–1762, 2008.
- [63] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
- [64] William M. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, December 1971.
- [65] Jean-F. Rivest, Pierre Soille, and Serge Beucher. Morphological gradients. *Journal of Electronic Imaging*, 1993.
- [66] Jurgen Schmidhuber. Multi-column deep neural networks for image classification. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, pages 3642–3649, Washington, DC, USA, 2012. IEEE Computer Society.
- [67] Scikit learn python library. <http://scikit-learn.org/>.
- [68] Sebastian Seung. I am my connectome. <https://www.youtube.com/watch?v=HA7GwKXfJB0>, 2010. Accessed: 2015-12-03.

- [69] Sebastian Seung. *Connectome: How the Brain's Wiring Makes Us Who We Are*. Mariner Books, 2013.
- [70] F. H. She, R. H. Chen, W. M. Gao, P. H. Hodgson, L. X. Kong, and H. Y. Hong. Improved 3d thinning algorithms for skeleton extraction. In *DICTA*, pages 14–18. IEEE Computer Society, 2009.
- [71] P. J. Soille and M. M. Ansoult. Automated basin delineation from digital elevation models using mathematical morphology. *Signal Process.*, 20(2):171–182, May 1990.
- [72] C. Sommer, C. Straehle, and U. Koethe andF. A. Hamprecht. ilastik: Interactive learning and segmentation toolkit. *IEEE International Symposium on Biomedical Imaging*, 8:230–233, 2011.
- [73] Amelio Vazquez-Reina, Daniel Huang, Michael Gelbart, Jeff Lichtman, Eric Miller, and Hanspeter Pfister. Segmentation fusion for connectomics. Barcelona, Spain, 06/11/2011 2011. IEEE.
- [74] Vigra library. <http://ukoethe.github.io/vigra>.
- [75] T.A. Weissman, J.R. Sanes, J.W. Lichtman, and J. Livet. Generating and imaging multicolor brainbow mice. *Cold Spring Harb Protoc.*, pages 763–9, 2011.
- [76] J.G. White, E. Southgate, J.N. Thomson, and S. Brenner. The structure of the nervous system of the nematode *caenorhabditis elegans*. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 314(1165):1–340, 1986.
- [77] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.