Recurrent Neural Networks for Graph-Based 3D Agglomeration

Masterarbeit aus der Physik

Vorgelegt von

Thomas Kipf

18. März 2016

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuerin: Prof. Dr. Ana-Sunčana Smith

In Kooperation mit Dr. Moritz Helmstaedter,
Max-Planck-Institut für Hirnforschung, Frankfurt am Main

Acknowledgements

I would like to thank everyone, who in one way or another helped me finalize this thesis. First and foremost, I want to thank **Dr. Moritz Helmstaedter** for inviting me to spend the past year at the MPIBR in Frankfurt and for his support and constructive criticism. I would also like to thank my supervisor **Prof. Dr. Ana-Sunčana Smith** at FAU for her continuous support and for quickly organizing everything related to my studies.

This work wouldn't have been possible without the valuable advice and help from many of the members of the Connectomics department at the MPIBR. I have to thank **Manuel Berning** for the many hours he spent introducing me to the subtleties of the lab's codebase and for his valuable feedback in all phases of the project. I am also very grateful to **Benedikt Staffler** and **Emmanuel Klinger** for many enjoyable discussions and advice on my project. I have to express my thanks to **Kevin Boergens** for introducing me to his merger mode tracing tool and for providing me with a dataset of skeleton tracings. Finally, I would like to thank **Alessandro Motta** for sharing his 3D shape feature library and for proofreading the thesis.

Table of contents

Chapter 1

Introduction

The comprehensive mapping of the physical structure of neuronal circuits is an open problem at the very core of the field of neuroscience. A map of the morphology of neurons and their connections is believed to advance our knowledge and understanding of computational models of such circuits (Denk et al., 2012) and it is the goal of a field called connectomics to obtain and describe such maps.

A promising approach that is currently followed by most in the field is 3D electron microscopy (3D-EM) of fixed and stained samples of neural tissue (Briggman and Denk, 2006; Denk and Horstmann, 2004; Hayworth et al., 2006; Helmstaedter et al., 2008, 2011). 3D-EM allows for nanometer-scale resolution which is necessary to densely map the smallest structures of a neural circuit. Decades ago, this technique was used to map the complete nervous system involving all 302 neurons in *C. elegans* (White et al., 1986), an undertaking that took more than 10 years of manual labor to complete.

The greatest bottleneck for this task was—and still is today—imposed by data analysis, i.e. the reconstruction of circuits from raw 3D-EM data (Helmstaedter, 2013). While recent advances in 3D-EM (Denk and Horstmann, 2004; Hayworth et al., 2006) opened the avenue for imaging of samples larger than $1\,\mathrm{mm}^3$, dense reconstruction of such a sample would still take a minimum of roughly $1,000$ work years (Briggman, 2016) with recent semi-automated reconstruction techniques (Berning et al., 2015; Kim et al., 2014).

In this work, we address the problem of automated reconstruction by introducing a new graph-based technique based on recurrent neural networks (RNNs) (Elman, 1990; Rumelhart et al., 1986) that learns non-local information not captured by previous approaches (Andres et al., 2012; Berning and Helmstaedter, 2016; Bogovic et al., 2013; Jain et al., 2011; Nunez-Iglesias et al., 2014; Vazquez-Reina et al., 2011). Our technique builds on top of existing reconstruction pipelines and potentially allows for a more reliable and faster reconstruction of neural circuits from raw 3D-EM data.

1.1 Related work

Automated reconstruction of neural circuits from 3D-EM data has recently gained increasing attention with methods that first over-segment the raw data (Andres et al., 2008; Berning et al., 2015; Jain et al., 2007; Lee et al., 2015; Sommer et al., 2011; Turaga et al., 2010) and then agglomerate these segments to reconstruct the anatomy of individual neural processes. Most agglomeration techniques either only regard pairs of segments to decide whether they should be merged or not (Berning and Helmstaedter, 2016; Bogovic et al., 2013), or they use a more non-local or global approach (Andres et al., 2012; Jain et al., 2011; Jurrus et al., 2008; Nunez-Iglesias et al., 2014; Vazquez-Reina et al., 2011).

We here extend previous work on local interface classifiers (i.e. classifiers that regard pairs of segments only) by instead making use of the graph structure given by the over-segmentation. We propose to convert the problem of inference on graph-structured data to a sequence learning task, by sampling the graph structure with constrained random walks. We motivate the use of RNNs for this task as they have proven to be powerful sequence learning models (Graves et al., 2006; Graves and Schmidhuber, 2009). A similar neural network-based approach is introduced in Perozzi et al. (2014): They serialize the graph with unconstrained random walks to learn latent, unsupervised representations of nodes in the graph. Their approach differs from ours in that they model latent node embeddings of graph-structured data, whereas we train a supervised model to predict edge weights. To our knowledge, this is the first method that uses sequence learning on the serialized segmentation graph for automated 3D agglomeration.

As opposed to classical methods of graph-based image segmentation (Felzenszwalb and Huttenlocher, 2004; Zahn, 1971), where each node in the graph represents a single pixel or voxel, the segmentation graph we consider here is built on supervoxels (collections of voxels, i.e. individual segments) and their spatial neighbor relationships.

1.2 Main contributions

Here, we summarize our main contributions[1] that we will describe throughout the thesis in more detail. First, we develop methods for efficient training data generation for graph-based 3D agglomeration and introduce a constrained path sampling algorithm that produces practical serializations of segmentation graphs for use in sequence learning.

[1] Our work is in part based on the following published and unpublished work of others: The segmentation graph with edge weights from a Gaussian process-based interface classifier (Berning and Helmstaedter, 2016) is provided by Berning (2014), based on the 3D-EM dataset of Boergens and Helmstaedter (2012) and SegEM (Berning et al., 2015). Skeleton tracings are provided by Boergens (2015b) and 3D shape features are provided by Motta (2016). We further make use of the merger mode tracing tool developed by Boergens (2015a) for webKnossos (Helmstaedter et al., 2015).

We further provide a small yet expressive feature set that proofs to capture useful information for graph- or sequence-based agglomeration. We optimize and compare models for sequence learning and demonstrate that they can be used to improve the predictions of a classifier that solely uses local information around an interface between a pair of segments.

1.3 Thesis outline

This thesis is structured as follows. In Chapter 2, we introduce and review background information on automated reconstruction of 3D-EM data and on RNNs as sequence learning models. Chapter 3 constitutes the main part of this thesis. Here, we introduce and describe methods for training data generation, for RNN training and for testing of our proposed approach. In Chapter 4, we describe a number of experiments on sequence learning for agglomeration and their results. Finally, in Chapter 5, we summarize and discuss our findings and provide an outlook on possible future research directions.

Chapter 2

Background

In this chapter, we review both the current state of automated reconstruction of 3D-EM data and its challenges. We further introduce and review recurrent neural networks (RNNs) as powerful sequence learning models.

2.1 Automated reconstruction of 3D-EM data

The current state-of-the-art approach for automated reconstruction of neural tissue from 3D-EM data can be summarized in the following two steps:

1. Create a volumetric over-segmentation of the 3D data, i.e. split processes (cells and parts thereof) into many small segments.

2. Agglomerate segments based on similarity.

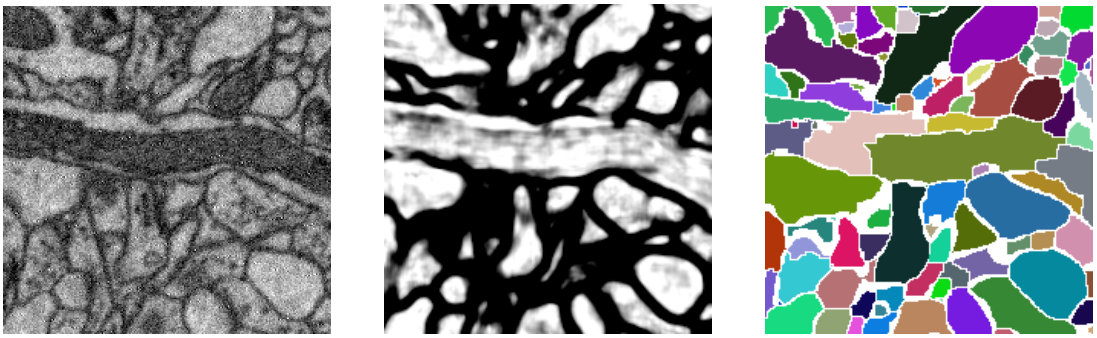
It is generally believed that reconstruction accuracy can be improved by splitting the task into these two steps, although other approaches exist (Maitin-Shepard et al., 2015). The current challenges lie in a) reducing errors in the segmentation and b) improving the automated agglomeration of segments. We begin by reviewing approaches for volume segmentation in connectomics. As of now, the error rates of these methods are still several orders of magnitude too high to use fully automated reconstruction of neural circuits from 3D-EM data in practice (Helmstaedter, 2013). Current approaches therefore still rely on manual agglomeration of segments via skeleton tracings (Berning et al., 2015) or direct agglomeration (Kim et al., 2014).

2.1.1 Volume segmentation

Volume segmentation of connectomic 3D-EM data is typically performed as a sequence of the following two steps (see, e.g. , Berning et al. (2015)). First, every voxel is classified

as being part of a membrane or of extra-/intra-cellular space (Figure 2.1b). Then, a watershed algorithm (Beucher and Meyer, 1992) is applied to create a segmentation (Figure 2.1c).

Most approaches use random forests (Andres et al., 2008; Sommer et al., 2011) or convolutional neural networks (Berning et al., 2015; Jain et al., 2007; Lee et al., 2015; Turaga et al., 2010) for step one. For a marker-based watershed procedure as in (Berning et al., 2015), connected components of local regional minima of the classification image (Figure 2.1b) are used as markers. Typically, the amount of over-segmentation is chosen via a free parameter in the local minimum procedure such that segmentation mergers, i.e. segments that span multiple processes, are very rare.



(a) Raw data (b) Membrane detection (c) Segmentation

Fig. 2.1 Illustration of a segmentation pipeline (Berning, 2014) based on a 3D-EM dataset of mouse primary somatosensory cortex layer 4 (Boergens and Helmstaedter, 2012) and SegEM (Berning et al., 2015). Images are of size $2.4 \times 2.4\,\mu m^2$. *Left*: Original 2D slice of the raw image data (Boergens and Helmstaedter, 2012). *Middle*: Membrane detection using a convolutional neural network (Berning et al., 2015). Membrane voxels are classified as 0, whereas intra-cellular space voxels are classified as 1. *Right*: Over-segmentation using a watershed-based approach (Berning et al., 2015).

2.1.2 Segmentation graph

Given the over-segmentation, we can define a segmentation graph (see Figure 2.2) that summarizes proximity information in a more efficient data structure. A node in the segmentation graph corresponds to a single segment in the original segmentation, whereas edges correspond to physical interfaces between two segments. Therefore only neighboring segments share an edge in the segmentation graph. We can furthermore introduce edge weights $p(e) \in [0, 1]$ for edges e. By limiting $p(e)$ to the interval $[0, 1]$, the weights can be interpreted as the probability of two segments to be classified as belonging to the same process.

Current approaches for edge (i.e. interface) classification and agglomeration include global optimization approaches such as GALA (Nunez-Iglesias et al., 2014), LASH (Jain

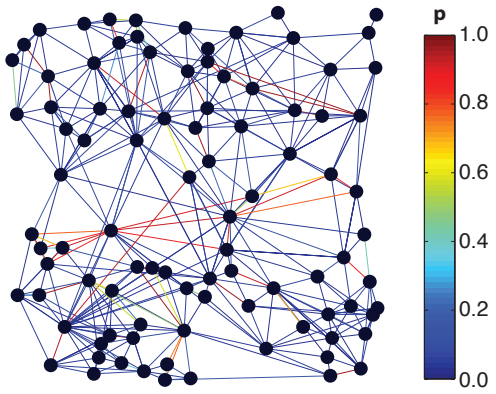


Fig. 2.2 Illustration of a segmentation graph with edge weights $p(e) \in [0, 1]$ from a Gaussian process-based interface classifier (Berning and Helmstaedter, 2016). Nodes correspond to segments and edges correspond to interfaces between two segments.

et al., 2011), segmentation fusion (Vazquez-Reina et al., 2011) and a graphical model-based approach (Andres et al., 2012), and a simple feature-based interface classifier (Bogovic et al., 2013). A variation of the latter method is also the approach currently chosen in our lab, implemented using a Gaussian process (GP) classifier (Berning and Helmstaedter, 2016) that uses local, mainly texture-based features.

2.2 Recurrent neural networks

As discussed in the last section, most approaches for agglomeration only consider local, pairwise information to predict edge weights $p(e)$. In this thesis we consider an extension to this approach, as we intend to serialize the graph structure of agglomerates and use sequence learning to improve predictions $p(e)$. By unrolling the agglomeration graph (i.e. the segmentation graph of an agglomerate) into a sequence of segments, described by feature vector, we can treat this problem as sequential inference.

RNNs (Elman, 1990; Rumelhart et al., 1986) have recently attracted considerable attention as powerful parametric frameworks for sequence prediction; for a review see (Schmidhuber, 2015). We decided to use RNNs because they have a number of desirable properties that are especially useful for our task:

- RNNs are known to perform well on large amounts of data (Sak et al., 2014).
- Trained networks can be re-trained on new datasets and therefore require only little new labelled data for transfer learning.
- RNNs can be used on sequences of arbitrary length.

While most of these properties are also shared by hidden Markov models (HMMs), recent experiments have shown that HMMs are outperformed by RNNs on many

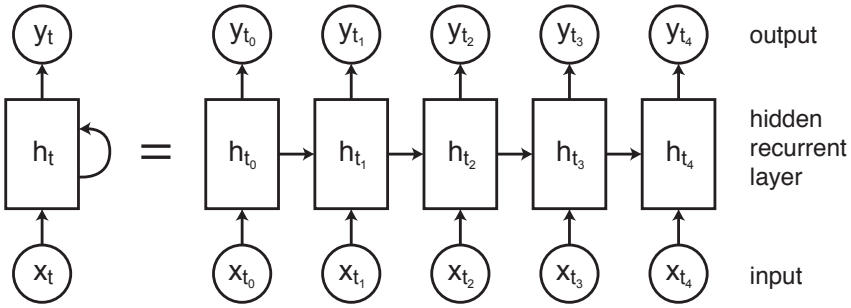


Fig. 2.3 Illustration of an RNN in compact form (left) and with recurrency unfolded along the time dimension (right). At every time step t, the network reads an input vector $\mathbf{x}^{(t)}$ and produces an output vector $\mathbf{y}^{(t)}$, while updating its hidden state $\mathbf{h}^{(t)}$.

sequence learning tasks such as handwriting recognition (Graves and Schmidhuber, 2009) or voice recognition (Graves et al., 2006).

In the following sections, we introduce and review RNNs for sequence learning and discuss their use in practice.

2.2.1 Overview

The main idea behind RNNs, as opposed to standard feed-forward neural networks, is the concept of (temporal) weight sharing. RNNs share parameters across the length of a sequence and allow generalization to sequences of arbitrary length.

A particular use case for RNNs is to model the probability of an output sequence $\mathbf{y} = (\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)})$ given an input sequence $\mathbf{x} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)})$:

$$p(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)}) = \prod_{t=1}^{T} p(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t-1)}) \,, \tag{2.1}$$

or simply the conditional probability of the last example given all previous inputs

$$p(\mathbf{y}^{(T)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T-1)}) \,. \tag{2.2}$$

We will mainly consider the latter case in this thesis. In general, this leaves us with a network architecture as shown in Figure 2.3: An input layer feeds into a hidden recurrent layer that is read out by an output layer. The recurrency can be unrolled in time to provide a feed-forward respresentation of the model.

A forward pass through the RNN is typically defined as follows:

$$\mathbf{h}^{(t)} = \theta(\mathbf{W}_i \mathbf{x}^{(t)} + \mathbf{W}_h \mathbf{h}^{(t-1)}) \,, \tag{2.3a}$$

$$\mathbf{y}^{(t)} = f(\mathbf{W}_o \mathbf{h}^{(t)}) \,, \tag{2.3b}$$

with an activation function $\theta(x)$ and weights summarized in weight matrices $\mathbf{W}_i$ (input to hidden), $\mathbf{W}_h$ (hidden to hidden) and $\mathbf{W}_o$ (hidden to output). In most cases, the hyperbolic tangent $\tanh(x)$ is used as an activation function. Optionally, an activation function can also be applied for the output layer. Typical activation functions $f(x)$ here are the logistic sigmoid for two-class classification tasks and the softmax function for multiple classes (Bishop, 2006). Most implementations further add so called bias terms to each layer transformation. These can be incorporated, e.g. , by extending the state vectors with a unit term $\mathbf{h}'^{(t)} = (1, \mathbf{h}^{(t)})^\intercal$ (and $\mathbf{x}^{(t)}$ accordingly). The hidden layer activation $\mathbf{h}^{(0)}$ is typically initialized as zero.

2.2.2 Backpropagation through time

In the supervised setting, where for each example $\mathbf{x}_n$ labels $\mathbf{t}_n = (\mathbf{t}^{(1)}, \ldots, \mathbf{t}^{(T)})$ are available, training of the RNN can be performed by iteratively updating the network parameters $\mathbf{w}$ (weights and biases) in small steps in the direction of the negative slope (i.e. the gradient) of an error function. Error functions, such as the mean square error

$$E(\mathbf{w}, \mathbf{X}) = \frac{1}{2} \sum_{n=1}^{N} (\mathbf{y}_n - \mathbf{t}_n)^2 \tag{2.4}$$

can be motivated via maximum likelihood (Bishop, 2006). Here, we defined the error in terms of the full dataset $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. If single training examples are used instead for the calculation of the error, we arrive at stochastic gradient descent (Bishop, 2006). In practice, the error is typically calculated for mini-batches of multiple training examples to make more efficient use of hardware and to reduce variance in the parameter updates.

The question remains, of how to evaluate the gradient of the error function in Equation 2.4. It turns out that this problem can be easily solved for feed-forward networks, by repeatedly applying the chain rule for partial derivatives, which gives us a technique commonly known as backpropagation (Rumelhart et al., 1986; Werbos, 1988; Williams and Zipser, 1995). For RNNs, the computational graph defined by Equations 2.3 can be unfolded into a feed-forward like structure (see Figure 2.3). This allows us to apply backpropagation to RNNs as well, a technique known as backpropagation through time (Werbos, 1990). For reasons of brevity, we omit lengthy derivations of these derivates here, as these can be found in most textbooks on neural network-based machine learning (Bishop, 2006; Goodfellow et al., 2016).

2.2.3 Decaying gradients

As each update step of the hidden unit activations $\mathbf{h}^{(t)}$ involves a matrix multiplication with $\mathbf{W}_h$, one can easily convince oneself that this repeated multiplication with the same matrix will either lead to vanishing or exploding activations. Let us assume that $\mathbf{W}_h$ can be decomposed as follows:

$$\mathbf{W}_h = \mathbf{V}\,\mathrm{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1}\,, \tag{2.5}$$

with a vector of eigenvalues $\boldsymbol{\lambda}$. Then the calculation of the hidden layer activation after τ time steps will involve multiplication by $\mathrm{diag}(\boldsymbol{\lambda})^\tau$, if we ignore activation functions and biases for now. It is therefore easy to see that for eigenvalues $\lambda_i < 1$, activations will exponentially go to zero and otherwise explode. The latter case can be avoided by careful initialization and gradient clipping (Pascanu et al., 2013). Vanishing activations, and by symmetry also vanishing gradients during training, will significantly impede the ability to train an RNN (Hochreiter, 1991). Experiments in Bengio et al. (1994) show that the probability of successfully training an RNN with stochastic gradient descent will quickly reach 0 for sequences as short as 10 time steps.

2.2.4 Alternative unit architectures

To alleviate the problem of vanishing gradients, alternative unit architectures have been suggested. This involves changing the update function from Equation 2.3a in such a way, as to allow better flow of activations or gradients through multiple time steps. The first and most widely used variant is called the long short-term memory (LSTM) unit (Hochreiter and Schmidhuber, 1997) and involves a separate hidden activation state that is gated from the rest of the network and passed on in a constant manner through multiple time steps. Recently, a powerful alternative, the gated recurrent unit (GRU), was introduced by Chung et al. (2014). It introduces fewer additional parameters than LSTM while similarly alleviating the problem of vanishing gradients. The GRU approach changes the hidden layer update equation (see Equation 2.3a) to

$$\mathbf{h}^{(t)} = (1 - \mathbf{z}^{(t)}) \odot \mathbf{h}^{(t-1)} + \mathbf{z}^{(t)} \odot \tanh\left[\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{W}_h(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t-1)})\right], \tag{2.6}$$

with an update gate activation $\mathbf{z}^{(t)}$ and a reset gate activation $\mathbf{r}^{(t)}$ defined as

$$\mathbf{z}^{(t)} = \sigma(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{U}_z\mathbf{h}^{(t-1)})\,, \tag{2.7a}$$

$$\mathbf{r}^{(t)} = \sigma(\mathbf{W}_r\mathbf{x}^{(t)} + \mathbf{U}_r\mathbf{h}^{(t-1)})\,, \tag{2.7b}$$

where $\sigma(x)$ is the logistic sigmoid function and $\tanh(x)$ the hyperbolic tangent. The symbol $\odot$ denotes element-wise multiplication. Note that we have introduced four additional gating weight matrices $\mathbf{U}_z$ and $\mathbf{U}_r$ (hidden-to-hidden) and $\mathbf{W}_z$ and $\mathbf{W}_r$ (input-to-hidden).

Gradients can, again, be computed in a straightforward manner with the help of backpropagation. As the GRU architecture alleviates the problem of vanishing gradients, we will make use of it for all hidden recurrent layers described in this thesis.

2.2.5 Computation graph and symbolic differentiation

The implementation of backpropagation for the aforementioned neural network architectures can be simplified by working with a computational framework that represents expressions internally in the form of a computation graph. See Figure 2.4 for an example. A popular deep learning framework that operates in this way is Theano (Bastien et al., 2012; Bergstra et al., 2010) which will be our framework of choice for the rest of this thesis.

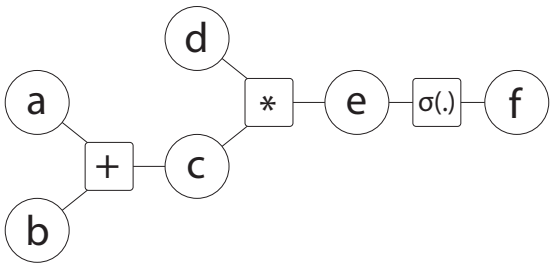


Fig. 2.4 Example of a computation graph for the expression $f = \sigma\left[(a + b) * d\right]$. We have introduced additional intermediate variables c and e, so that every operation has a separate output variable.

By representing expressions symbolically in a graph structure, exact analytical differentiation can be performed automatically. Theano makes use of a highly-optimized version of symbolic differentiation, which essentially relies on the application of the chain rule to compute derivatives directly on the computation graph. In Theano, symbolic differentiation comes at the expense of increased compilation time, which can render exploring large neural network architectures a tediously long process. As opposed to numerical differentiation, however, symbolic differentiation does not suffer from inaccuracies due to numerical rounding issues (Jerrell, 1997) and is therefore well worth the trade-off with respect to compilation time.

We note that there exists an alternative method, namely automatic differentiation (AD), that was recently introduced to the field of machine learning. We shall refer to a recent review article by Baydin et al. (2015) for more details on this method, as a discussion of AD would be beyond the scope this thesis.

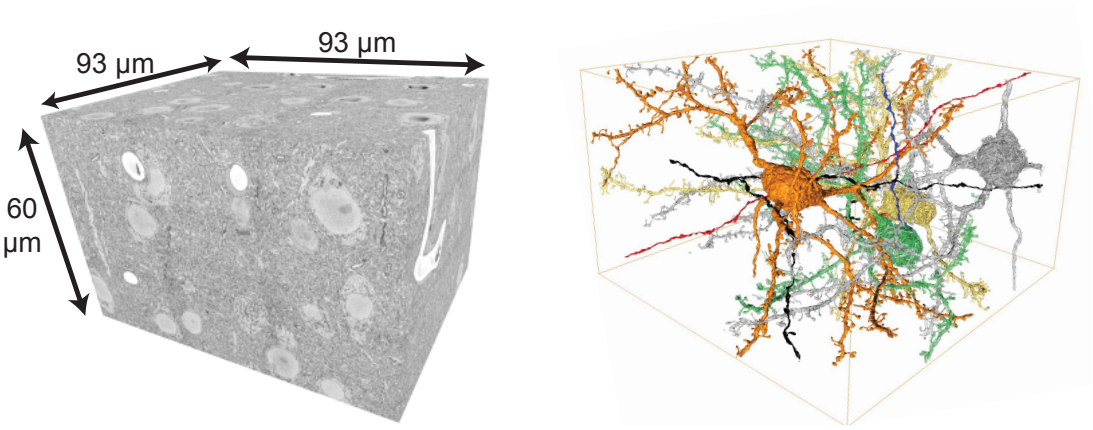
Chapter 3

Methods

This chapter constitutes the main part of this thesis. We introduce and discuss methods both for training and testing of our RNN-based sequence learning approach for agglomeration. A major part of this chapter deals with the mining of training data from skeleton tracings and with methods to make agglomeration graphs derived from these skeletons usable for sequence learning. This includes the introduction of a constrained path sampling algorithm and a mapping of both shape- and orientation-based features onto such paths. In the last part of this chapter, we discuss architecture- and optimization-related choices for RNNs in more detail.

All the methods described in the following sections were developed and tested on a single $93 \times 60 \times 93\,\mu m^3$ 3D-EM dataset from layer 4 mouse primary somatosensory cortex (Boergens and Helmstaedter, 2012) (Figure 3.1) that was obtained via the serial block-face electron microscopy (SBEM) technique (Denk and Horstmann, 2004). The segmentation (Berning, 2014) of the dataset was performed using SegEM (Berning et al., 2015), where the parameters were tuned towards a higher degree of over-segmentation than in the original paper. The segmentation graph (Berning, 2014) was generated with edge weights given by predictions of a GP-based interface classifier (Berning and Helmstaedter, 2016) as described in Section 2.1.2.

Unless otherwise noted, all methods outlined here were implemented in MATLAB® and are available to the research group via a private Github repository. The RNNs were implemented in Python, while making heavy use of the open source deep learning frameworks Theano (Bastien et al., 2012; Bergstra et al., 2010) and keras (Chollet, 2015). We would also like to note that as part of this project, a contribution to the sequence masking functionality in keras (Chollet, 2015) was made. Sequence masking allows for mini-batch training on arbitrary sequence lengths (see Section 3.7.3).



(a) 3D-EM dataset (b) Volume reconstruction

Fig. 3.1 *Left:* Overview of the 3D-EM mouse primary somatosensory cortex layer 4 dataset (Boergens and Helmstaedter, 2012). *Right:* Volume reconstruction of four neurons obtained by combining skeleton tracings with segmentation objects (Berning et al., 2015). Figures adapted from (Berning et al., 2015, reprinted with permission from Elsevier).

3.1 Training data

In this section we describe the steps undertaken to derive a training dataset from a collection of human-traced axons. We briefly describe the method of skeleton tracing and discuss potential issues that might arise during tracing or when trying to match skeletons with the segmentation graph of the dataset. We suggest a number of heuristics that can alleviate these issues to some degree, such as the identification of annotation errors or the automated completion of gaps in skeletons.

We will later see that the quality of a training dataset derived this way will be sufficient to successfully train a well-performing classification model on the segmentation graph. Model testing, however, will require a lesser degree of noise or annotation errors. We will therefore later introduce a different annotation method, namely merger mode tracing, for this purpose that aims to collect all the segments belonging to a particular process (i.e. a cell or a part thereof).

3.1.1 Skeleton tracings

In order to train our models for improved automated agglomeration of segments, we decided to make use of the vast amount of human-traced processes that are currently available for the mouse cortex dataset. We train and evaluate our models on a set of tracings [1] which we refer to as *Iris axons*, named in honor of the student who carried

[1] Skeleton tracings provided by Boergens (2015b)

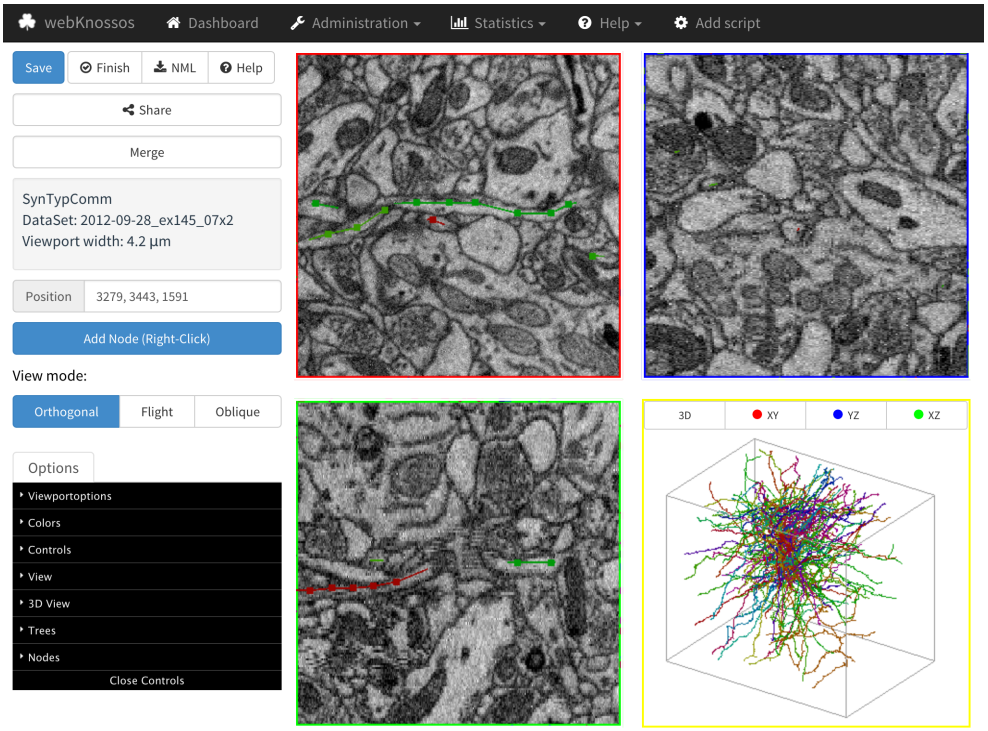


Fig. 3.2 Screenshot of the webKnossos user interface. webKnossos is a browser-based tool for visualization and annotation of 3D image data (Helmstaedter et al., 2015). The dataset is rendered in three movable orthogonal slice planes. Skeletons can be created by iteratively placing nodes in a respective slice plane. The panel on the lower right shows a 3D overview of all skeletons in the currently active session.

out this tracing task. The Iris axons dataset is a collection of high-quality tracings of 91 axons that innervate (i.e. share synapses with) a single dendrite, which is also part of the dataset. An overview of the Iris axons dataset is shown in the lower right panel of Figure 3.2.

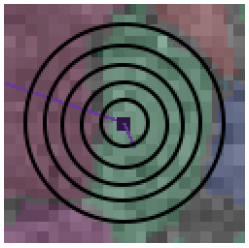
In total there are 92 traced processes which we will refer to as *trees*. Every synapse of the 91 axons was labelled using a technique called three-point annotation: For every synapse, single nodes are placed firmly inside the pre- and postsynaptic structures and on the synaptic interface between the two. For our purposes, it will be important to detect and remove these three-point annotations before we proceed to any further post-processing steps of the skeletons, since a naïve agglomeration of all segments touched by the skeletons would otherwise result in false training data wherever a synapse is present.

3.1.2 Skeleton post-processing

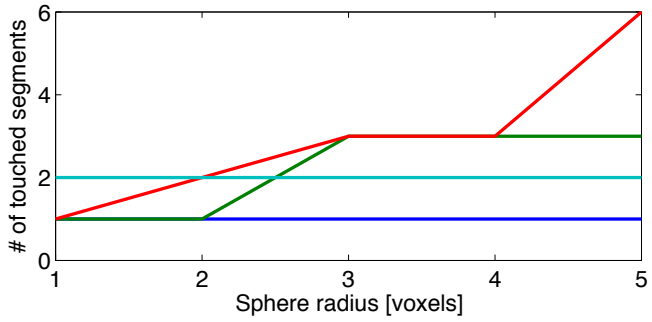
After an initial manual inspection of the tracings for general tracing quality (e.g. node placement frequency and skeleton completeness, i.e. no missed branches) and for other peculiarities (e.g. three-point annotations or nodes placed for meta information, such as dataset alignment issues or debris) we automatically identified and removed three trees that were located to a large part outside of the segmentation bounding box and therefore couldn't be used for the following steps. We also removed the dendrite. This leaves us with a total of 88 trees for the rest of this project.

Node placement noise

The initial inspection revealed an issue that we term *node placement noise*, i.e. noisy placement of skeleton nodes which occasionally resulted in nodes being placed on or very close to cell membranes or even into other neural or glial processes. Our main goal of this project is to model the structure of neural processes and to make continuation predictions that should avoid merge errors at any cost, i.e. errors that lead to continuation into a wrong process. Therefore errors of this kind in the training dataset should be kept at a minimum.



(a) Spheres in 2D



(b) Distribution of touched segments per sphere radius

Fig. 3.3 Illustration of the sphere growing process (left) and placement noise statistics (right). *Left:* Spheres of different radii (in black) around a skeleton node (in violet) shown on a 2D $0.25 \times 0.25\,\mu m^2$ slice of the raw data with overlaid segmentation (colored tiles). *Right:* Number of touched segments for different sphere radii shown for four randomly selected nodes of the Iris axons dataset.

We developed a simple heuristic to detect and remove erroneously placed nodes. First, we grow a sphere around every single node with radius $r \in \{1, \ldots, 5\}$ voxels (Figure 3.3a). Then, for each integer valued radius, we count the number of segments that are touched by the respective sphere (Figure 3.3b). By defining a joint threshold (θ_r, θ_n) on a maximally allowed number of touched segments θ_n for a given radial threshold θ_r, we can exclude nodes with certain characteristics. A threshold of $(2, 1)$,

for example, would exclude both the nodes described by the red and cyan curves in Figure 3.3b.

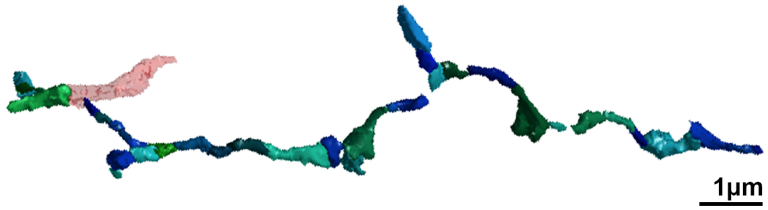


Fig. 3.4 3D rendering of the collected segments of part of an axon. An erroneously added segment (in red, half-transparent) was automatically detected and removed. Note the gaps between some segments, where parts of the axon were missed by the annotator.

Finding an ideal threshold would typically require access to a ground truth set of axons against which the accuracy of the procedure could be compared. Since the generation of such a dataset is expensive, we instead chose a simpler, heuristic optimization procedure which can be described as follows: We randomly sample a subset (e.g. $N = 10$) of the axons from the dataset and manually check the 3D rendering of the collected segments (Figure 3.4) for merge errors. We then mark segments automatically for deletion using the (θ_r, θ_n) threshold from above. We start with $\theta_r = 1$ and $\theta_n = 1$ and increase the radial threshold until all erroneously added segments are removed. By subsequently increasing θ_n in steps, we can recover some of the segments that have been erroneously marked for deletion. We increase θ_n while keeping $\theta_r = 1$ fixed until the first previously deleted merger segment is no longer marked for deletion and then pick the previous value of θ_n as the final threshold. Thereby, we mark all of the merge-error segments that have been accumulated due to node placement noise for deletion while keeping as many of the true axonal segments as possible.

We should note that segmentation mergers, i.e. single segments that span multiple processes, typically cannot be excluded in that way and should be kept out of the procedure. For the Iris axons dataset, we found that the threshold $(\theta_r = 2, \theta_n = 1)$ produced desirable results. We should point out that the choice of this threshold is highly dependent on dataset and annotation style and should therefore be re-evaluated for new datasets.

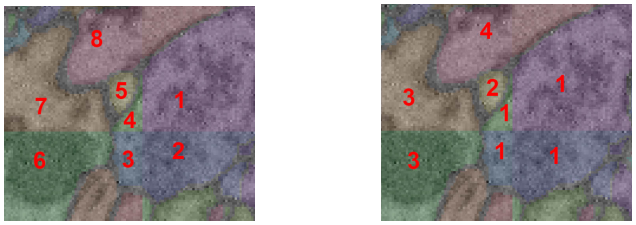
Agglomeration graph

Here, we briefly describe the procedure of matching a skeleton to the segmentation objects of the segmentation graph. For every skeleton node with coordinates (x, y, z), we find and store the respective segment ID at this location in the dataset. Note that the segmentation is constructed in such a way that there will always be a border of at

least one voxel between two adjacent segments. These are assigned the ID 0. When a node is placed on a border voxel, we remove it from the further analysis. In this way we collect a number of unique segment IDs from which the constrained segmentation graph of the skeleton can now be calculated. In this graph—which we will refer to as *agglomeration graph*—the nodes now represent segment IDs and edges are assigned based on connectivity of the segments. We assign an edge if and only if two segments share a border.

Globalization and segment ID mapping

At this point, let us note one peculiarity about segment ID conventions in the dataset. The segmentation is initially constructed only for local cubes of size $1024 \times 1024 \times 512$ (all units in voxels). The total sub-volume of the dataset for which segmentation information is available is parcellated into $15 \times 9 \times 12$ of these cubes with overlapping regions, since every local cube shares its outer padding with the neighboring cubes for technical reasons. The padding-free volume of a cube measures $512 \times 512 \times 256$. Excluding the outer padding, segmentation information is therefore available for a total region of size $86.3 \times 51.8 \times 86.0\,\mu\text{m}^3$ with an anisotropic voxel size of $11.24 \times 11.24 \times 28\,\text{nm}^3$. Segment IDs were initially defined only for local cubes. The current globalization procedure of these IDs reduces cubes to their padding-free volume and then, while iterating through all cubes, simply adds an offset (the maximum ID of the previous cube) to the ID of every segment in the current cube.



(a) Unique IDs (b) Re-mapped IDs

Fig. 3.5 Segment ID re-mapping at cube borders. *Left*: Original segment enumeration convention. Here, unique segment IDs are given to parts of segments that are located in different local cubes (distinguishable by color). *Right*: Segment ID re-mapping applied. Segments share the same ID across cube borders.

Segments that span multiple cubes are therefore assigned multiple IDs—one for each part that touches one of the inner volumes of a local cube (see Figure 3.5). To avoid ambiguities that might arise due to this redundancy, we here decided to collect all these IDs of a single segment and map them to a single ID. What we gain is the following: We can calculate features for the whole segment, even if it extends into a neighboring

cube. That is, if it does not extend beyond the padding of a local cube. In this rare case, artifacts in the feature calculation might arise. We shall later see, however, that this is not the limiting factor for our model performance and can therefore be tolerated.

Gap completion

An important issue that arises when matching the skeleton tracings to the segmentation of the dataset becomes apparent in Figure 3.4. The naïve collection of all segments touched by a skeleton node typically results in a disconnected segmentation graph that has gaps. These gaps generally arise due to an inter-node distance in the skeletons that is larger than the smallest segment size in the dataset. This suggests that simple interpolation along edges between skeleton nodes would solve the problem. Simple interpolation could, however, introduce additional merge errors in curved structures.

Instead, we therefore post-process the segmentation graph as follows. First, we identify gaps using pairwise matching of loose ends in the graph. Then, we try to close the gaps using a heuristic rule that favors likely paths given the GP edge probabilities (see Section 2.1.2) while avoiding pathological configurations, such as paths that take a shortcut through a different process, e.g. a dendrite.

Let us first focus on gap identification. Given the fragmented segmentation graph of an axon (see Figure 3.6), we start by detecting all connected components in the graph. Then, for each connected component we calculate the euclidean distance from every node in this component to every node in every other component. For each pair of connected components we then identify the pair of nodes with minimum distance. We discard pairs that are more than $\theta_d = 2.5\,\mu\text{m}$ apart, since we observed that the gap completion typically failed for gaps larger than θ_d. This leaves us with a set of node pairs that will serve as the start and end point, respectively, for the automated gap completion.

Now, every edge e is assigned a probability score $p_e \in [0, 1]$, where p_e can be interpreted as the confidence of the GP classifier that two segments are connected, i.e. belong to the same neural process. Analogously, $1 - p_e$ describes the confidence of the classifier that the two corresponding segments belong to different neural processes each.

In a probabilistic framework, the most likely continuation between two nodes is therefore given by the shortest path where each edge e is assigned a weight

$$w(e) = -\log(p_e)\,, \tag{3.1}$$

i.e. the negative log-likelihood of the edge. Since $w(e) > 0$ for all edges, we can use Dijkstra's algorithm (Dijkstra, 1959) to find the most likely path in practice.

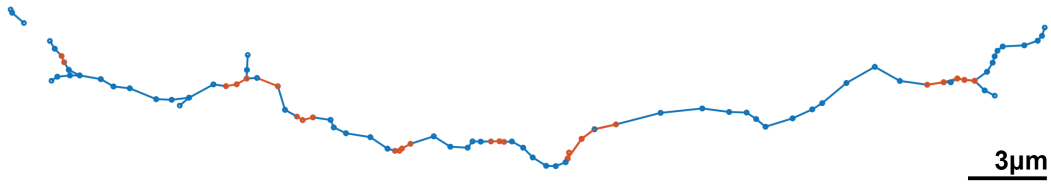


Fig. 3.6 Example of a segmentation graph skeleton where most of the gaps were automatically completed. Segmentation graph skeletons are obtained by matching traced skeletons to the segmentation of the dataset where nodes are plotted in the center of mass of segments and connected only if the two segments share an interface. The original skeleton is shown in blue, whereas gap auto-completions are shown in orange.

We selected a random subset of the 88 Iris axons and visualized the candidate gap completion paths as in Figure 3.6. In this way, correct completions could readily be verified. False completions could be identified by their shape and cross-checked within the webKnossos tool in uncertain cases.

An initial investigation revealed that a substantial fraction of the gap completions were not following the correct neural process and instead followed shortcuts through nearby dendrites or other structures. This effect can be understood in terms of the class distribution of the edge classifier probabilities, which has outliers both in the low-probability and high-probability regime. Outliers with $p_e \approx 0$ (false negatives) are edges within a single neural process that tend to get classified as a negative edge, mostly due to local noise in the dataset or organelles that look like cell membranes to the classifier. On the other hand, outliers with $p_e \approx 1$ (false positives) are typically assigned to edges where segmentation mergers are present. Here, the path search often chooses to follow such an edge instead of the correct continuation inside the actual process. Especially the false negative outliers are difficult to exclude, since the cost assigned to such an edge is in principle unbounded due to the logarithm. Therefore simple thresholding of the total path cost, i.e. excluding paths that are too expensive, does generally not produce favorable results.

We approached this problem with a heuristic as follows. Instead of using negative log-likelihood weights, we replaced the edge weights with

$$w(e) = (1 - p_e) + w_0 \,, \tag{3.2}$$

where we further introduced a weight offset $w_0 > 0$. The cost per edge is now bounded by w_0 and $w_0 + 1$, respectively. This allows us to find an upper threshold of the total path cost θ_c that excludes paths that take unwanted detours through other processes. We manually tuned the parameters and found that $w_0 = 0.45$ and $\theta_c = 3$ produced

the best results. Additionally, we introduced a threshold for the maximum mean cost per edge in a path θ_m. With $\theta_m = 0.95$ we were able to exclude all erroneous gap completions in the Iris axons dataset. For new datasets, these parameters will most likely have to be re-evaluated. Figure 3.7 shows the fraction of successfully closed gaps for different gap sizes.

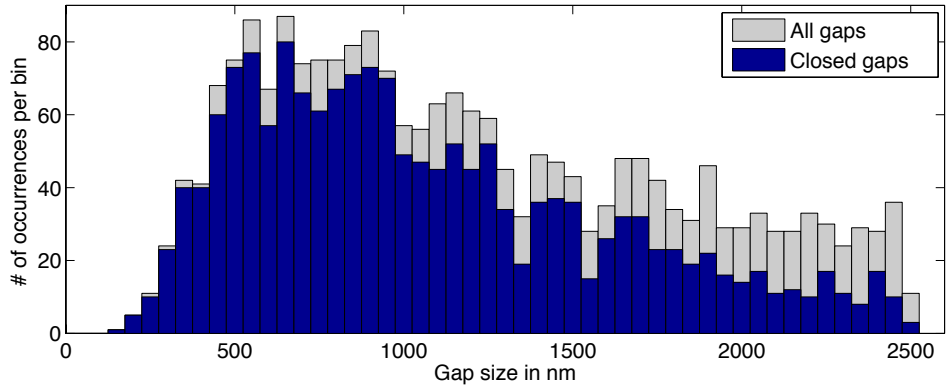


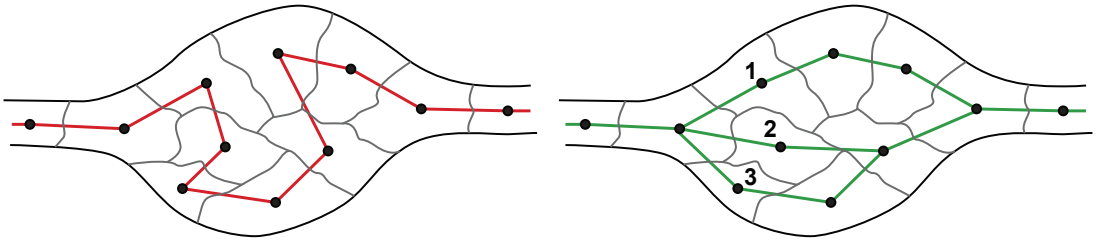
Fig. 3.7 Histogram of successfully closed gaps vs. total number of gaps for different gap sizes. Gaps larger than 2.5 µm were excluded from the analysis.

3.2 Constrained path sampling

Having completed the steps described in the last section, this now leaves us with a dataset of 88 human-traced axons that are matched to the segmentation graph and further post-processed to remove erroneously placed nodes and to automatically fill in gaps. The skeletons that were matched to the segmentation graph are now in general represented by a complex graph structure with edges that represent the connectivity of segments, i.e. whether two segments share a common interface or not. This can result in situations where cycles are present in this subgraph (the total graph being the overall segmentation graph of the dataset). Predictive modeling of the complete, potentially cyclic graph using deep learning models is still a topic of ongoing research (Duvenaud et al., 2015; Henaff et al., 2015) and generally does not scale well to large amounts of data that we are dealing with in our case. An alternative method is presented by Baldi and Pollastri (2003) that uses a variant of an RNN that operates directly on a directed acyclic graph. This would, in our case, require the transformation of our present cyclic graph structure to an acyclic structure. This could be done by finding the minimal spanning tree of our subgraph which is by definition acyclic. This conversion step, however, could introduce unwanted artifacts by discarding information that could otherwise be useful for the predictive modeling.

We instead follow a different approach here that allows the direct application of state-of-the-art sequence learning models, such as RNNs, while also allowing the direct comparison to models that operate on static data, such as RFs. Instead of considering the whole graph structure, we randomly sample paths—including several domain-specific constraints—that carry what we think to be useful information for the predictive modeling. We resort to sampling methods since an exhaustive path search would be computationally intractable for highly over-segmented areas such as axonal boutons.

Let us first briefly recall the goal we wish to address by modeling the graph structure. Given a subgraph that describes part of a neural process (e.g. an axon), we would like to make a prediction on where the neural process will continue. For every node in the subgraph, we can restrict the decision to edges with neighboring segments that are not currently part of the graph. The goal is then, given such a candidate edge, to read the most important parts of the subgraph up to this edge that are relevant to make a decision whether the new edge should be added to the agglomerate or not.



(a) Zig-zag path in a bouton (b) Three constrained random walks

Fig. 3.8 Examples of random walks inside a heavily over-segmented axonal bouton. *Left*: Pathological zig-zag path that can occur in unconstrained path sampling. *Right*: By introducing a suitable constraint, random walks will follow paths that a human annotator would typically follow during skeleton tracing.

3.2.1 Constrained path sampling algorithm

Our simplified approach can then be described as follows. Instead of considering the whole graph or an approximation thereof (e.g. minimal spanning tree graph), we only consider simple paths originating in the decision edge and traveling along the graph until a certain criterion is reached. By putting constraints on this path sampling procedure, we can limit the vast search space of available paths. Furthermore, the constraint leaves us with paths that mimic the skeletonization procedure that a human annotator would follow. That is, paths that follow relatively straight through a process, without collecting every single segment in an over-segmented region (Figure 3.8) in a zig-zag path. For axons, this is especially important for structures called axonal

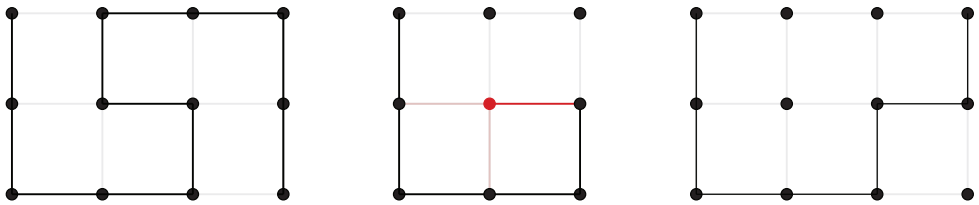


Fig. 3.9 *Left:* Self-avoiding random walk (SARW) on a lattice. The walk terminates successfully after no more candidate nodes are available. *Middle:* Unsuccessful second-order SARW. The walk is discarded, because a node was chosen that shares connections to previously visited nodes (violation of second-order self-avoidance criterion). *Right:* Successful second-order SARW. Although shown here on a regular two-dimensional grid, the concept of self-avoiding walks can directly be applied to arbitrary undirected graphs.

boutons (bulges in an axon that constitute presynaptic regions), as these are typically highly over-segmented.

For the path sampling procedure, we follow the well-studied framework of a self-avoiding random walk (SARW) (Madras and Slade, 1996). We extend this framework with a second-order self-avoidance criterion, so that paths are not only restricted to visit nodes only that were not visited before, but to also avoid nodes that are neighbors to previously visited nodes (Figure 3.9). We term the resulting algorithm *second-order self-avoiding random walk*; see Algorithm 3.1. Note that we are encoding nodes $\mathbf{n}$ in form of a 1-of-K vector of length K, where K is the total number of nodes. If, for example, our graph has a total number of $K = 5$ nodes, then the third node in the graph will be represented as $\mathbf{n} = (0, 0, 1, 0, 0)^\mathsf{T}$. This allows for simplified equations and more efficient implementations when dealing with random walks in practice. In analogy, we represent a self-avoiding walk of length N in the form of a N-of-K vector. Note that this notation cannot be applied to simple paths, where each node is visited only once. We can keep track of the order of the visited nodes by representing a walk of length $N = 3$, for example, by the vector $\mathbf{w} = (0, 2, 1, 0, 3)^\mathsf{T}$ for the walk $w = n_3 n_2 n_5$ in canonical notation.

An interesting question that arises in connection with Algorithm 3.1 is whether, given two nodes in a connected component of a graph, it is possible to find a path between the pair that does not violate the second-order self-avoidance criterion.

Definition 3.1 (Second-order self-avoidance criterion)**.** *Let P be a path in an undirected finite graph $G(V, E)$. Further let $V_P \subset V$ be the set of vertices that constitute P and $E_P \subset E$ be the subset of edges that contain at least one node of V_P. P is said to fulfill the second-order self-avoidance criterion if and only if both the source (first) node $s \in V_P$ and the target (last) node $t \in V_P$ of P are of degree 1 in the subgraph $K(V_P, E_P) \subset G(V, E)$ and every other node $v \in V_P$ of P is of degree 2 in K.*

Algorithm 3.1: Second-order self-avoiding random walk on an undirected graph. Stopping criterion is reached if either no more new neighbors are available or if one of the new neighbors has already been visited by the walk.

Data: Adjacency matrix $\mathbf{A}$ (binary, no self-loops), source node $\mathbf{s}$ (1-of-K coding).
Result: Walk $\mathbf{w}$ of length N (N-of-K coding).

```
init i ← 1;
init walk w ← s;
init current node c ← s;
repeat
    get neighbors n ← Ac;
    if n^T n > 0 ∧ w^T n = 0 then
        select single non-zero entry of n uniformly at random and set all other
        entries to zero: n_next;
        update current node: c ← n_next;
        add node to walk: w ← w + ic;
        i ← i + 1;
    else
        stop;
    end
until stopping criterion reached;
```

Theorem 3.1. *Let $C(V, E)$ be a connected component of an undirected finite graph G, then there exists a path from every node $u \in V$ to every other node $v \in V$ that satisfies the second-order self-avoidance criterion.*

Proof. We proof the theorem by showing that every path that does not fulfill the second-order self-avoidance criterion can—in a finite number of steps—be reduced to a path that satisfies the criterion, while keeping the source vertex $s \in V_P$ and the target vertex $t \in V_P$ unchanged. P, V_P, E_P and K are defined as in Definition 3.1. Let us further define $d_0(s) = 1$, $d_0(t) = 1$ and $d_0(n) = 2$ for every other node n in P. We do not put any further constraints on the path P so that one of the following two statements is true:

1. For all nodes $v \in V_P$: $\deg(v) = d_0(v)$ in $K \iff P$ satisfies the second-order self-avoidance criterion.

2. For at least two nodes $v \in V_P$: $\deg(v) > d_0(v)$ in $K \iff P$ does not satisfy the second-order self-avoidance criterion.

Due to symmetry, when one node $v \in V_P$ has $\deg(v) > d_0(v)$, then there is always a second node in V_P with the same property that is not the direct neighbor of v in P. For case 2, we can therefore always identify a pair of nodes v_1, v_2 with $\deg(v_{1/2}) > d_0(v_{1/2})$ that are not direct neighbors in P. Without loss of generality, we can declare a direction

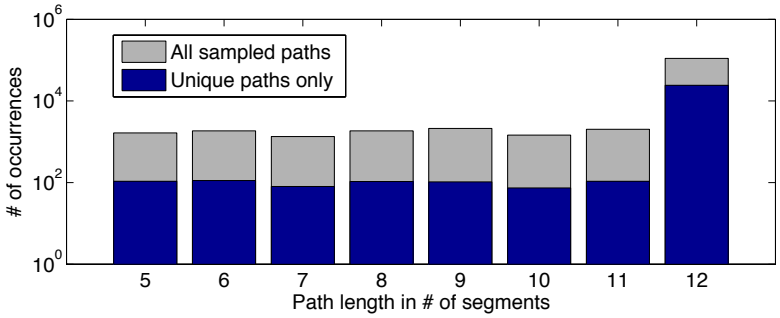


Fig. 3.10 Histogram of path lengths for sampled paths using the second-order self-avoidance criterion and a maximum path length of 12 nodes. Paths shorter than 5 nodes were discarded as well. The y-axis is shown in logarithmic scale.

for P and write $P = s...v_1...v_2...t$ (or $P = v_1...v_2...t$ if v_1 is either source or target node). Then, by removing all intermediate nodes between v_1 and v_2 from P and V_P, i.e. $P = s...v_1v_2...t$ (or $P = v_1v_2...t$), the degree of both nodes will be decreased by at least 1. Now either case 1 will be true and we are finished, or case 2 will be true and we can again identify a pair of nodes and repeat the process. As for every iteration the node degree of both nodes is decreased by at least 1 and the degree of every node is finite, the procedure will finish in a finite number of steps after which case 1 will be true. □

It is important to note that the implementation of our second-order self-avoidance criterion as in Algorithm 3.1 involves discarding paths that end up violating the criterion. Therefore only paths are kept that either reach a maximum path length or that terminate early on, when no more neighbors are available while the second-order self-avoidance criterion still remains fulfilled. This explains the distribution of path lengths of successfully sampled paths in Figure 3.10.

The choice of discarding paths that do not meet the criterion will introduce a further bias. Specifically, paths that traverse a cycle-free part of the graph will be overrepresented. This can be understood as follows: In a cycle-free part of the graph, the second-order self-avoidance criterion will always be met. These paths will therefore only be discarded if they don't meet a minimum-length criterion, which we shall introduce later. Walks that traverse a cyclic part of the graph will, however, always have a non-zero probability of violating the criterion. Given only a limited number of samples, we will therefore always bias the successful walks towards cycle-free parts of the graph (if there are any) or, in an equivalent picture, towards walks along nodes of with low a degree.

3.2.2 Path sampling procedure

Having described the path sampling algorithm, we will continue by discussing the design and parameter choices we made for the application of the path sampling procedure in practice.

For reasons of computational complexity, we limit the maximum length of paths to $l_{\max}$. The random walk is stopped when $l_{\max}$ is reached. We chose $l_{\max} = 12$ for an initial experiment, which turned out to work well in practice. The overall path sampling procedure for a maximum of 10 paths per node in all of the 88 axons took approximately 20 minutes on a single CPU. With no further assumptions for the graph structure, the number of possible paths scales—in the worst case—exponentially with the path length. We don't expect to gain substantial additional information from longer paths, as the initial segments of a path will most likely be redundant and over-represented. As the average probability of failure per node of the path sampling procedure is independent of the path length and larger than zero for cyclic paths, the probability of successfully sampling a path that meets the second-order self-avoidance criterion decreases exponentially with path length. Therefore, a length limit where the sampling procedure can still be completed in several minutes wall-clock time seemed reasonable.

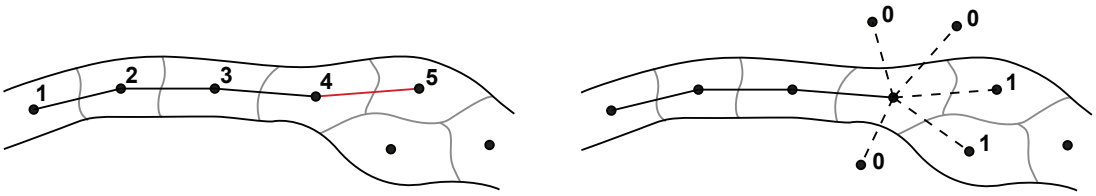
In practice, we repeat the path sampling procedure for each node until either a minimum number of paths $N_{\min}$ is sampled or a pre-defined number of failed iterations $N_{\max}^{\text{fail}}$ is reached. For all further experiments, we set $N_{\min} = 10$ and $N_{\max}^{\text{fail}} = 100$.

3.3 Path augmentation and label generation

Having the set of sampled paths, we introduce a simple augmentation as follows. Every path is first copied and then the copy is added to the set of paths with nodes in reversed order, i.e. the source node is made the new target node and so forth. In the next step, we remove all duplicate paths from the set. Figure 3.10 shows the fraction of paths that are left after the removal of duplicates.

We generate a class-labelled data set as follows. In a pruning step, we identify and remove the target node of every path (see Figure 3.11a). We then create copies of this pruned path and add one edge per path to all neighboring segments of the new target node, as shown in Figure 3.11b. We call this additional edge the decision edge. For each path, we assign label 1 (positive) if the decision edge connects two segments of the same process and label 0 (negative) otherwise. We should note that the second-order self-avoidance criterion is not considered for this step.

With the introduction of the pruning step, we avoid border effects in the training set, e.g. where gaps are still present. Label information is in general not available for



(a) Original path (b) New paths with labels

Fig. 3.11 Label generation procedure. *Left*: Original path with node labels in ascending order and with the source node labelled as 1. In the pruning step, the edge 4-5 (in red) is removed and node 4 is made the new target node. *Right*: New paths are generated by copying the pruned path $N-1$ times, where N is the number of neighbors of the new target node. Then, for each path an edge (dashed lines) is added to one of the neighbor nodes and the path is labelled 0 or 1 according to whether an edge belongs to the process or not.

decision edges at open gaps or borders of the dataset. We should note that for the application of the trained agglomeration model in practice the pruning step will not be necessary.

Split	**Number of paths**	**Mean length**	**Label ratio (pos : neg)**
Training set	582,127	11.8	1 : 17.6
Validation set	30,988	11.9	1 : 17.7
Test set	63,108	10.9	1 : 23.7

Table 3.1 Summary statistics of training, validation and test set data derived from the Iris axons dataset. Mean length reflects the average number of nodes per path. Splits were chosen at random to roughly reflect a 85 : 5 : 10 ratio in number of paths. The test set seems to have a particularly challenging label ratio.

During all steps we keep track of the axon ID from which every path was sampled. We use this information to split the dataset into three non-overlapping parts, namely a training, a validation and a test set. We group paths by axon ID, randomly shuffle the group order and count the number of paths per group. We then split the dataset in three parts such that every split contains a certain number of groups, to arrive at a specific number of paths in each split. We chose a 85 : 5 : 10 (training : validation : test) ratio of the path number to generate the splits. So that, e.g. , the test set contains roughly 10% of the total number of paths and is completely disjunct from the other two sets, i.e. there are no overlapping edges between the sets. A summary of the respective splits is given in Table 3.1.

3.4 Feature calculation

This section describes the methods we used to calculate features for both single segments and pairs of segments to arrive at a feature description of the paths sampled in the previous step of the pipeline. Edge features are calculated on a segment-to-segment basis and describe the relation of a pair of segments for a given directionality, e.g. relative location of segment A with respect to segment B (see Figure 3.12). Node features are calculated on the basis of a single segment alone.

We limit our feature calculation to segmentation data, i.e. features are calculated only on the basis of shape, location and orientation of supervoxels (segments) in the segmentation. We omit any features derived from the underlying raw 3D-EM data in this approach. Texture information of this kind is typically extensively used by interface classifiers, as, e.g. , in (Berning and Helmstaedter, 2016) (see Section 2.1.2). We therefore only expect marginal gain from including these types of features into our approach and will refrain from doing so here.

We should note that prior to calculating the features, we have applied the global segment ID mapping as described in Section 3.1.2. Segment features calculated in this way will however occasionally still suffer from border artifacts. These artifacts arise when a single segment spans a larger volume than what is covered by the local cube (including its padding) in which it is calculated. Axonal segments, however, are typically constrained to smaller length scales and are not to be expected to extent beyond these borders. But even for larger segments we expect the additional noise in the feature set from this type of artifact to be negligible.

3.4.1 Edge-based features

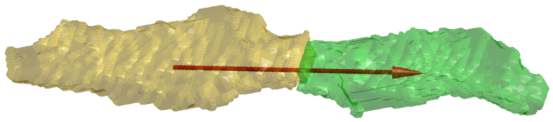


Fig. 3.12 Edge-based features are calculated for pairs of segments—illustrated here as yellow and green—given in a specified order. The red arrow denotes the relative position of the green segment with respect to the yellow segment.

We describe every edge in our path dataset (see Table 3.1) with a feature vector that is calculated on the basis of the pair of segments described by the respective edge. For reasons of efficiency, we calculate this feature vector for every edge in the segmentation graph, of which there are approximately 71.9×10^6 in total for the segmentation used here. A considerable amount of computational resources has to be devoted to this feature extraction task. Here, the feature calculation occupied a CPU cluster with

Feature name	Description	d
borderSurfaceArea*	Area of interface between two segments	1
differenceObjSize**	Volume difference	1
distanceCentroids*	Centroid distance	1
differenceDirection	Difference of major axis directions	3
relativePosition	Displacement vector (difference of centroid vectors)	3
differencePcFracs	Difference of normalized covariance matrix eigenvalues	3

Table 3.2 List of edge-based features, i.e. features describing the relation of a pair of segments. Features include relative direction, position and distance of objects. Features marked with (*) were logarithmized using $x \to \log(1+x)$; (**) indicates the following transformation: $x \to \mathrm{sgn}(x)\log(1+x)$, where $\mathrm{sgn}(x)$ is the signum function.

over 200 nodes for several days. We expect, however, that a substantial fraction of the feature calculation routine can still be optimized to reduce computational load, making the method feasible for larger datasets.

Some of the edge features used here are direction-sensitive, i.e. sensitive to the order in which the segments are presented. We calculate the features for the directionality implied by the order of the edge list, by which the segmentation graph is defined. Edge features are then later mapped onto paths of arbitrary direction by keeping track of direction-based features and by inverting them where necessary. With the exception of the feature borderSurfaceArea, all edge features either describe shape differences, relative orientation or relative position of segments. A complete list of the edge features used here is given in Table 3.2. We should emphasize that, of this set, only the feature borderSurfaceArea is currently captured by the GP-based interface classifier (Berning and Helmstaedter, 2016) that we are comparing our model against.

The choice of features was motivated by the intuition that orientation and relative position of a sequence of segments form powerful predictors for potential path continuations. Later in this thesis, we will use a mechanism for feature importance ranking to show that this intuition is indeed correct.

3.4.2 Node-based shape features

In addition to edge-based features, we derive a set of features for every path node from their respective segments. We make use of a library of shape descriptors that has been put together for the purpose of glia cell detection[2] in our lab. We select and post-process a subset of these features (see Table 3.3) that we expect to capture the

[2]Segment shape features for segmentation CNN $20141007T094904_{8,3}$ provided by Motta (2016)

most important shape characteristics for our task. Shape features are based either on the isosurface, the convex hull (see Figure 3.13) or the mask of a segment. The segment mask is a three-dimensional binary matrix with non-zero entries only for voxels that constitute the shape of the respective segment. Again, this task is very resource intensive as there are approximately 10×10^6 segments in the dataset used here.

Feature name	**Description**	**d**
convFeatPacking	1 - (segment mask volume / convex hull volume)	1
convHullArea*	Area of convex hull	1
convHullCompactness*	$(\text{Convex hull area})^3$ / $(\text{convex hull volume})^2$	1
convHullInertiaMat*	Upper triangular part of convex hull inertia tensor	6
convHullVolume*	Convex hull volume	1
convHullVolumeToArea*	Convex hull volume / convex hull area	1
crumpliness	Segment mask area / convex hull area	1
isoSurfArea*	Isosurface area	1
isoSurfCompactness*	$(\text{Isosurface area})^3$ / $(\text{isosurface volume})^2$	1
isoSurfVolume*	Isosurface volume	1
isoSurfVolumeToArea*	Isosurface volume / isosurface area	1
maskPcaCoefs	Normalized direction of major axis	3
maskPcaVariance*	Eigenvalues of segment mask covariance matrix	3
maskVolume*	Segment mask volume	1
pcaVarFracs	Normalized eigenvalues of segment mask cov. matrix	3
pcaVarMax*	Largest eigenvalue of segment mask cov. matrix	1
pcaVarRatios	Segment mask cov. matrix eigenvalue ratios	3

Table 3.3 List of segment-based shape features, based on work by Motta (2016) (see text for more details). Features marked with (*) were logarithmized using $x \to \log(1 + x)$.

We combine edge and node features into a single feature sequence of length $N - 1$ for each path, where N is the path length in number of nodes/segments. We decide to drop the feature vector of the source node of each path to make this representation possible. The source node is the farthest away from the decision edge and is therefore expected to carry the least amount of information for an agglomeration decision. Since we calculated both edge and node features for the full dataset, this feature mapping can be readily applied to new paths on the fly and is not limited to the pre-sampled paths described above.

3.4.3 Feature histograms and optimization of distributions

As indicated in Table 3.2 and Table 3.3, several features with very broad and long-tailed distributions were logarithmized. This step is motivated by the structure of the

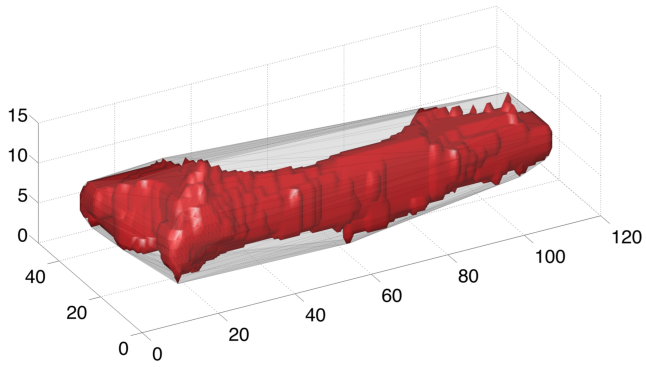


Fig. 3.13 3D rendering of the isosurface (red) of an axonal segment enclosed by its convex hull (grey). The axis labels depict voxel number in each dimension.

activation functions used in the RNN for training and prediction. Here, we typically use the hyperbolic tangent $\tanh(x)$ as an activation function which is most sensitive to inputs in the interval $x \in [-2, 2]$. It is therefore considered good practice to normalize the input distributions, so that the vast majority of values lies within this interval.

We use the following normalization procedure. For each feature, we calculate the 10th and 90th percentile of their distribution. We summarize these values in vectors $\mathbf{x}_{10}$ and $\mathbf{x}_{90}$, respectively. The normalization of a single data vector $\mathbf{x}$ is then defined as follows:

$$\tilde{\mathbf{x}} = \mathrm{diag}^{-1}\left(\frac{\mathbf{x}_{90} - \mathbf{x}_{10}}{2}\right) \cdot (\mathbf{x} - \mathbf{x}_{10}) - 1\,, \tag{3.3}$$

where $\tilde{\mathbf{x}}$ is the normalized data vector and $\mathrm{diag}^{-1}(\mathbf{a})$ the inverse of a diagonal matrix with the elements of $\mathbf{a}$ along its diagonal. The expression $\mathrm{diag}^{-1}(\mathbf{a}) \cdot \mathbf{b}$ then corresponds to the element-wise division of $\mathbf{b}$ by $\mathbf{a}$. The 10th and the 90th percentile are thus mapped to -1 and 1, respectively, whereas the width of the overall normalized distribution in principle remains unbounded. Figure 3.14 shows the normalized distributions for edge features of the decision edge only for both negative and positive examples.

We should note that the percentiles for the normalization were computed from the feature vectors of the training data alone. These values then remain fixed and are later used to normalize all further feature vectors, e.g. from the validation or the test set. Re-calculation of these percentiles for the validation or the test set could lead to poor generalization performance.

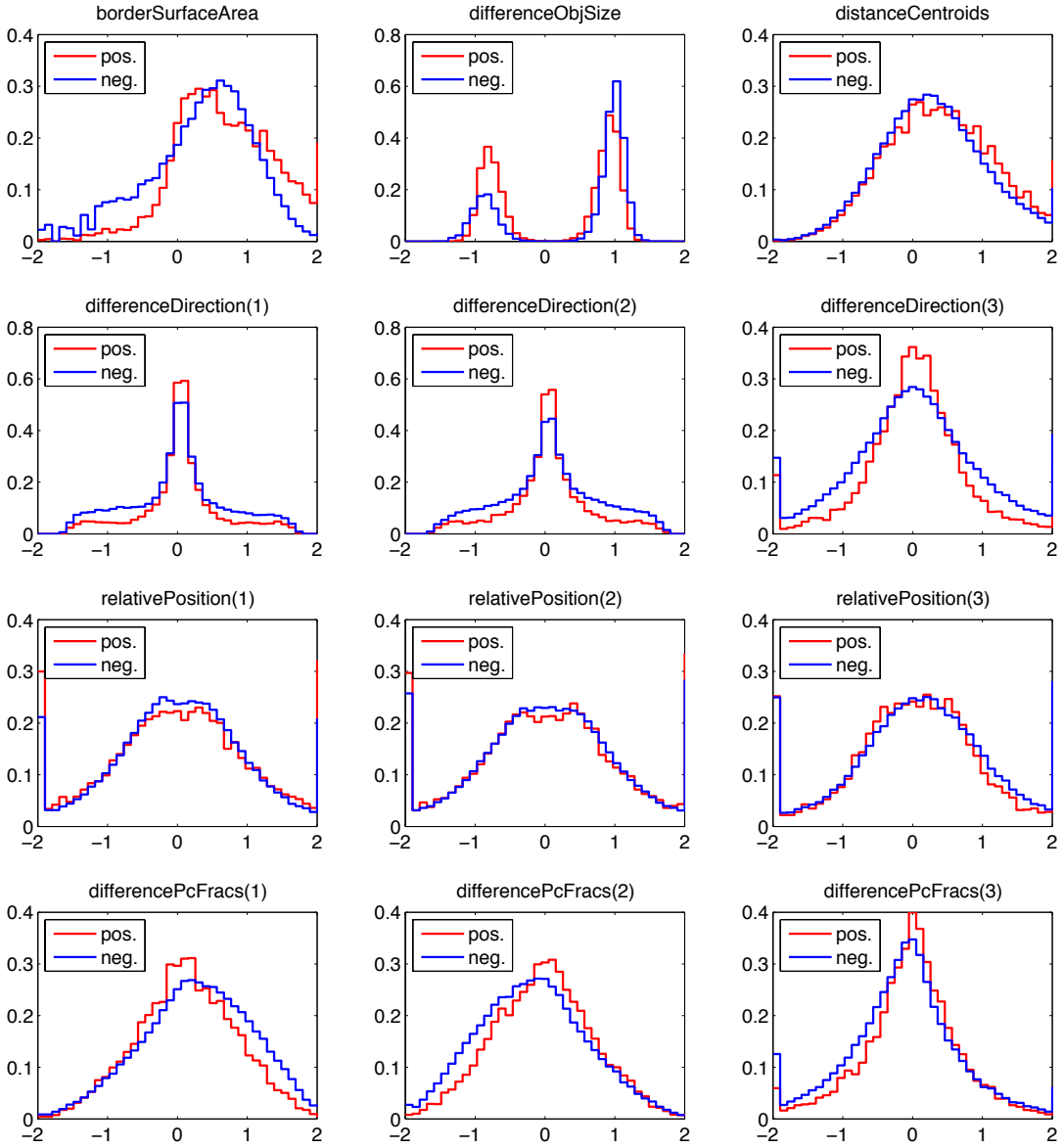


Fig. 3.14 Histograms of normalized edge features for both positive and negative decision edges from the training set. There is no evident class separation based on single features alone. Due to the anisotropic voxel size in the dataset, the third feature dimension, e.g. differenceDirection(3), shows different characteristics than feature dimensions in the imaging plane (first two dimensions).

3.5 Model evaluation

In this section, we introduce and discuss the following two approaches for model evaluation on the segmentation graph, namely:

1. Transform model prediction back to a skeleton representation and compare to ground truth skeleton, or
2. transform ground truth skeleton to segmentation graph representation and compare to model prediction directly on segmentation graph.

3.5.1 Skeleton-based evaluation

Let us first discuss approach 1. The prediction of our agglomeration model comes in the form of a decision edge that is classified as a correct or as a false continuation of a path within an agglomerate of segments. This path, including the decision edge, can be transformed back into a skeleton representation by placing skeleton nodes in the center of mass of segments and by connecting the skeleton nodes in accordance with the path edges.

A method to compare and quantify the similarity of skeletons was introduced by Helmstaedter et al. (2011) in the form of the redundant-skeleton consensus procedure (RESCOP). The situation we encounter in our case, however, differs from the situation described in the paper as follows. We aim to compare a candidate skeleton fragment (the model prediction) to a ground truth skeleton, whereas RESCOP was developed to form a consensus skeleton from multiple equally relevant redundant tracings. Nonetheless, RESCOP introduces a voting scheme as part of the consensus procedure that can be adapted to our case.

The voting scheme, in short, distributes positive and negative votes for skeleton edges based on the vicinity of nodes from other skeletons. A detailed description of the algorithm is given in the paper (Helmstaedter et al., 2011). We can use this voting procedure solely on the candidate skeleton and distribute votes according to the vicinity of nodes of the ground truth skeleton. Let us define S_c to be the candidate skeleton and S_g to be the ground truth skeleton. Then, $T_{\alpha ij}$ is the number of positive votes for the edge from node i to node j in skeleton S_α. In contrast to the original RESCOP method, we do not count self-votes. Positively voted edges in the candidate skeleton (i.e. where $T_{cij} > 0$) will then count as true positive edges. False positives are edges with $T_{cij} = 0$ and false negatives are edges in the ground truth skeleton with $T_{gij} = 0$.

While this framework in principle allows to calculate model scores such as precision at a specific recall value, we found that it is difficult to optimize the hyperparameters

involved in the RESCOP voting procedure. The voting scheme is sensitive to inter-node distance variabilities that arise due to significant variations in size and shape of segments. Depending on the choice of hyperparameters, a RESCOP-based approach will either be insensitive to merge errors of small segments very close to the original process, or it will be sensitive to classifying the lateral extension of boutons as a merge error. Furthermore, the computational load of the voting procedure makes it infeasible to use a RESCOP-based score as an error function under which we optimize our models (e.g. via gradient descent), as it would require the evaluation of the score for every single optimization step.

3.5.2 Segmentation graph-based evaluation

Let us now discuss approach 2. Here, we transform ground truth skeletons to a representation on the segmentation graph. Limitations and problems that arise in this conversion step were addressed in Section 3.1.2. On the basis of the segmentation graph representation, true positive edges can then be defined as positive candidate edges that connect two segments in the ground truth set. With false positives being candidate edges classified as positive that connect two segments that are not both part of the same process in the ground truth set and false negatives being edges that connect two segments of the same process but are classified as negative. This evaluation is computationally efficient and can be used as basis for an error function under which we optimize our model.

Artifacts introduced in the conversion step from skeletons to segmentation graph objects can be divided into systematic and random errors. Node placement noise and skeleton gaps (see Section 3.1.2) are a source of random errors in the conversion, as segment sizes and shapes undergo statistical variation. A bias (or systematic error) is introduced, as the tracing task (follow the main line of a process) and the testing task (agglomerate all segments of a process) differ systematically. In the following sections, we will discuss how this bias can be alleviated by using a special tracing mode to create the test set skeletons. This makes approach 2 favorable for model evaluation.

At this point we should note that typically multiple paths in the test set converge onto a single decision edge. Our sequence classifier model will predict a score for every single one of these paths. For the evaluation of the model predictions we therefore first have to combine the predictions in some meaningful way. We found that simple averaging of scores worked better than only keeping the minimum or maximum score for a single decision edge.

3.5.3 Error types and performance metrics

Two error types arise on the segmentation graph, namely split and merge errors. In our terminology here, split errors correspond to false negative and merge errors to false positive edges. Merge errors are typically much harder to resolve than split errors, which is why the former should be avoided at any cost. We use a precision/recall (p/r) metric with precision and recall defined as:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad \text{and} \quad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}\,, \tag{3.4}$$

where TP is the total number of true positive edges. FP and FN are defined as the total number of false positive edges and false negative edges, respectively. In terms of the p/r metric, we should therefore aim for high precision (i.e. low incidence of merge errors) at the expense of low recall (i.e. high incidence of split errors).

Typically, a low number of merge errors per axon can be tolerated, as these can later be resolved when a densely labelled dataset is available. The reasoning here is the following: Once every single process is labelled (as axonal, dendritical, glial, somatic or vascular), ambiguities that arise when two processes are merged can be resolved. Merge errors among different types of processes, e.g. axon-dendrite or axon-glia, can readily be identified. Axon-axon merge errors can be resolved combinatorically to some extent. A single merge error between two axonal processes is easily identified, as it differs from axonal branching behavior.

Choosing a meaningful performance metric is still topic of ongoing discussion in the field of connectomics. We here suggest to evaluate models based on the maximum recall value at precision = 0.98. This choice can be motivated as follows. For the GP interface classifier, we found that for a probability threshold lower than $p_\theta = 0.95$ (i.e. classifying every edge as positive if $p \geq p_\theta$), the rate of merge errors results in percolation: The formation of a super-cluster that spans the whole dataset. This threshold p_θ corresponds to a precision of 98%, which is the minimum precision that we can therefore safely tolerate. The recall value is then a strong indicator for the reconstructed path length, which is the intuitive score that one typically aims to optimize.

We should note that we are using a different metric to derive the error signal required to train our model, namely the cross-entropy metric. We will discuss this choice in more detail in Section 3.7.1. There is no direct correspondence between the cross-entropy metric and the p/r metric, i.e. a model can both have a lower cross-entropy error and a lower recall (at a given precision) than another model. We found, however, that in most cases the cross-entropy error is a good indicator for the performance under p/r-based metrics.

3.5.4 Label noise

Due to the process in which we obtained our training data, labels for decision edges can be quite unreliable, as discussed, e.g. , in Section 3.5.2. In order to understand the implications of training and testing a model with label noise, we here provide a framework to quantify different types of noise and to calculate an upper bound on the precision/recall metric for the presence of noisy labels.

Let us first introduce a ground truth dataset GT with labels "+" for positive examples and "−" for negative examples. We further introduce a corrupted ground truth dataset CT with label noise: Some labels in CT are flipped compared to the correct labels in GT. Let us assume that the event of flipping a label is class-conditional and independent with probabilities $\rho_+ := p(-_{\mathrm{CT}}|+_{\mathrm{GT}})$ for positive and $\rho_- := p(+_{\mathrm{CT}}|-_{\mathrm{GT}})$ for negative labels in GT, with $\rho_+ + \rho_- < 1$. We split both datasets into training set $\mathrm{GT}_{\mathrm{train}}$ ($\mathrm{CT}_{\mathrm{train}}$) and test set $\mathrm{GT}_{\mathrm{test}}$ ($\mathrm{CT}_{\mathrm{test}}$).

In the scenario we encounter in this thesis, we only have access to CT, i.e. a noisy version of the ground truth, as an actual ground truth dataset GT is very expensive and labor-intensive to obtain. We now hypothetically train a classifier on $\mathrm{CT}_{\mathrm{train}}$ that makes predictions $+_{\mathrm{P}}$ and $-_{\mathrm{P}}$. By evaluating predictions on the $\mathrm{CT}_{\mathrm{test}}$ set, we can derive an upper bound for the precision/recall metric that we can expect to achieve under a noisy test set. This will motivate, whether it is worth investing in obtaining a small noise-free ground truth test set or not.

Let us assume that our classifier is well-regularized and is able to discover the underlying true structure of the data even though noise is present in the labels (Natarajan et al., 2013; Scott et al., 2013). In this case, the predictions on the test set will satisfy

$$p(+_{\mathrm{P}}|+_{\mathrm{CT}_{\mathrm{train}}}) < 1 \quad \text{and} \quad p(-_{\mathrm{P}}|-_{\mathrm{CT}_{\mathrm{train}}}) < 1\,, \tag{3.5}$$

i.e. the classifier will choose to deliberately mistrust labels that appear to be flipped due to label noise. As the classifier learns the underlying distribution, its performance on a noise-free GT test set will be optimal, i.e.

$$p(+_{\mathrm{P}}|+_{\mathrm{GT}_{\mathrm{test}}}) = 1 \quad \text{and} \quad p(-_{\mathrm{P}}|-_{\mathrm{GT}_{\mathrm{test}}}) = 1\,, \tag{3.6}$$

even though the classifier was trained on a corrupted training set. The interesting case, however, arises when considering the maximum performance on a CT test set with label noise. We will omit the *test* subscript in the following and further use $\tilde{+} := +_{\mathrm{CT}}$

and $+ := +_{\mathrm{GT}}$ (and accordingly for negative examples). Using Bayes' rule, we get:

$$p(+_{\mathrm{P}}|\tilde{+}) = (1-\rho_+) \cdot \frac{p(+)}{p(\tilde{+})}\,, \tag{3.7a}$$

$$p(-_{\mathrm{P}}|\tilde{-}) = (1-\rho_-) \cdot \frac{p(-)}{p(\tilde{-})}\,. \tag{3.7b}$$

Where $p(+)$ and $p(-)$ are the label ratios in the respective test sets. We can now calculate the (normalized) confusion matrix entries as follows:

$$\mathrm{tp} = \frac{\mathrm{TP}}{\mathrm{N}} = p(+_{\mathrm{P}}, \tilde{+}) = (1-\rho_+) \cdot p(+)\,, \tag{3.8a}$$

$$\mathrm{tn} = \frac{\mathrm{TN}}{\mathrm{N}} = p(-_{\mathrm{P}}, \tilde{-}) = (1-\rho_-) \cdot p(-)\,, \tag{3.8b}$$

$$\mathrm{fp} = \frac{\mathrm{FP}}{\mathrm{N}} = p(+_{\mathrm{P}}, \tilde{-}) = p(\tilde{-}) - \mathrm{tn}\,, \tag{3.8c}$$

$$\mathrm{fn} = \frac{\mathrm{FN}}{\mathrm{N}} = p(-_{\mathrm{P}}, \tilde{+}) = p(\tilde{+}) - \mathrm{tp}\,, \tag{3.8d}$$

where N denotes the total number of data points in the test set.

Let us now use the numbers from our training data (Table 3.1) and calculate the maximum precision and recall that is possible under the given constraints in a CT test set. From the table, we roughly have $p(\tilde{+}) = 0.05$ and $p(\tilde{-}) = 0.95$. We further evaluated a random sample of 20 decision edges from the training data and found the following label flipping probabilities: $p(-|\tilde{+}) = 0$ and $p(+|\tilde{-}) = 0.15$. Using Bayes' rule and the sum rule for probabilities we get $p(+) = 0.193$, $p(-) = 0.807$, $\rho_+ = p(\tilde{-}|+) = 0.74$ and $\rho_- = p(\tilde{+}|-) = 0$.

We can now use the confusion matrix definitions above to calculate precision and recall for our perfectly regularized classifier:

$$\text{precision} = \frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fp}} = 0.25\,, \tag{3.9a}$$

$$\text{recall} = \frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fn}} = 1\,. \tag{3.9b}$$

With $\rho_+ > 0.5$ it seems unlikely that a classifier could still learn the underlying distribution of the data. However, the assumption of unbiased random label noise is most likely not valid in our case, as most false labels arise due to tracing of only the center line of a process. Therefore lateral extensions are easily missed. We expect that a classifier will learn to replicate this behavior which could prove to be a useful feature as it makes the classification more robust to merge errors.

We can, however, conclude that the original, noisy test set $\mathrm{CT}_{\mathrm{test}}$ will not suffice to evaluate the performance of our model, as the biases in the label noise together with

the bound on the precision make the result very difficult to interpret. It is therefore necessary to create a well-controlled noise-free version of the test set in order to reliably evaluate classifier predictions.

3.6 Merger mode test set

Our aim is to obtain a test set that reflects a gold standard ground truth segmentation of single processes. For such a purpose, an add-on suite for webKnossos (Helmstaedter et al., 2015) was developed by Boergens (2015a) that we shall make use of for this project. The add-on suite introduces merger mode tracing, a method to visually guide the task of collecting every single segment that belongs to a process, such as an axon or a dendrite.

In merger mode, the user is shown the typical webKnossos tracing interface with overlaid segmentation. Whenever a node is placed inside a segment, the color of the segment changes to some pre-defined common color for all collected segments. Therefore it is easy to spot missing segments or erroneously added segments. It is not possible to place nodes on border regions between segments, thereby avoiding a typical source of error.

We identified the axons constituting the original test set (six in total) and re-traced them in merger mode, collecting segment by segment. We followed the policy that a segment is only collected if at least 50% of its volume is inside the axon that we are trying to reconstruct. This policy, however, is typically difficult to follow, as it is difficult to estimate the volume coverage from sequences of 2D images in a reasonable amount of time. Also, attention-related mistakes, such as left out segments, missed branches or accidental mergers are quite common and difficult to avoid.

In order to resolve annotation errors, we iterated the following steps after an initial candidate test set was created:

- Evaluate predictions of both the candidate (RNN) and the baseline model (GP interface classifier) on the merger mode test set and inspect all false positive predictions of the respective model for the decision threshold at 20% recall.
- Classify false positives as a) actual false positives, b) false positives due to missed segment in annotation, c) false positives due to mistakenly added segments in merger mode.
- Correct all respective locations for case b) and c) in the merger mode test set.

We iterated these steps until no more annotation errors were found. Note, however, that this procedure might leave errors that only appear for lower decision thresholds

Test set	Number of paths	Mean length	Label ratio (pos : neg)
Original	63,108	10.9	1 : 23.7
Merger mode	154,420	10.5	1 : 4.14

Table 3.4 Comparison of original test set obtained from the Iris axons dataset and the re-traced merger mode test set. Both test sets represent the same six axons. The difference in the class ratios is due to the way the original test set was traced: Here, only the center line of an axon was traced and therefore edges to segments of lateral extensions of a process are falsely labelled as negative. In the merger mode test set, more than twice as many paths were found due to the larger number of segments compared to the original test set.

untouched. We then estimated the overall annotation error rate in the merger mode test set by manually re-investigating a number of randomly chosen edges and their according labels. Doing so, we found that we can estimate the error rate for both negative to positive flips and vice versa to be less than 1 in 100. This is a major improvement in comparison to skeletons traced in standard mode, as discussed in the previous section.

We applied the same path sampling and augmentation procedures as in the original dataset. Table 3.4 shows a comparison of the original test set and the re-traced merger mode test set.

We should note that we have excluded a small fraction of one of the axons due to a misaligned region in the dataset. Automated tracing in this region becomes intractable due to significantly reduced dataset quality. Also, we later noted that in one of the test set axons, a branch was completely missed. This error is present both in the original test set from the Iris axons dataset and in the re-traced merger mode test set.

3.7 Training procedure

In this section we specify details about the architecture, implementation and training of the RNN sequence classifier model in practice. We based our implementation on the open source deep learning framework keras (Chollet, 2015): keras provides an additional abstraction layer to computation graph frameworks, such as Theano (Bastien et al., 2012; Bergstra et al., 2010), and allows a variety of combinations of different neural network layers. Keras also provides implementations for a selection of optimizers, such as stochastic gradient descent and AdaGrad (Duchi et al., 2011), and allows training of both recurrent and feedforward network architectures, or combinations thereof.

We make use of the computation graph framework provided by Theano to automatically compute gradients for backpropagation. Training was performed on a NVIDIA® Tesla M2090, which provided a significant speed-up in comparison to training on modern CPU architectures. Training a single model took up to 100 passes through the full training dataset with 300-800s per pass, i.e. roughly 8-20hrs of training time until convergence.

We also trained a fixed-length (i.e. all sequences cut to the minimum length) random forest (RF) model based on MATLAB®'s TreeBagger class to serve as a baseline model. RFs are insensitive to feature scaling and generally perform very well with little to no hyperparameter tuning. Furthermore, RFs provide a form of interpretability, as the importance of individual features for the classification decision can be evaluated.

3.7.1 Network architecture

The network architecture of the RNN (see Figure 3.15) can be separated into three components: An input layer, a hidden recurrent layer and an output layer. The hidden recurrent layer can itself again be composed of multiple layers or separate modules. We only consider single- or two-layered recurrent networks, as more than two hidden layers are typically harder to train and therefore advised against (Karpathy et al., 2015).

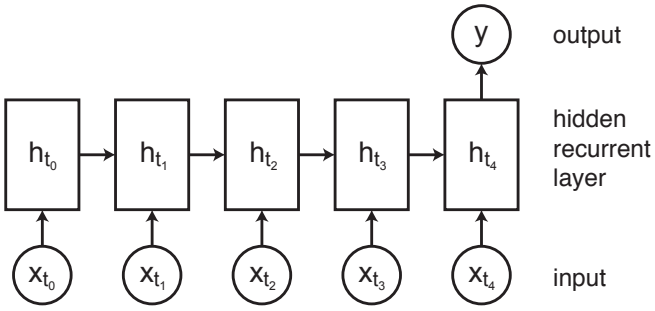


Fig. 3.15 Illustration of the network architecture for an RNN with a single hidden recurrent layer used for sequence classification. At each time step, the input layer reads an N-dimensional feature vector and passes a transformation thereof on to the hidden layer. Only at the last time step, the learned representation of the hidden layer is read out by a single output unit that provides the class probability for a given input sequence.

Input layer

The input layer is composed of N units, where N is the dimension of the feature vector of a single sequence step. At each time step t, a feature vector $\mathbf{x}^{(t)}$ is fed into the input layer of the network. It passes through a layer transformation $\mathbf{h}_{\text{cand}}^{(t)} = \tanh(\mathbf{W}\mathbf{x}^{(t)} + \mathbf{b})$

with a weight matrix $\mathbf{W} \in \mathbb{R}^{\mathrm{D}\times\mathrm{N}}$, bias vector $\mathbf{b}$ and $\tanh(x)$ activation function, where D is the number of hidden units. The result is then passed forward to the hidden layer.

Hidden recurrent layer

The hidden recurrent layer updates its current state at every time step depending on both the network input $\mathbf{x}^{(t)}$ (or its transformation, i.e. $\mathbf{h}^{(t)}_{\text{cand}}$) and the state of the hidden layer in the previous time step $\mathbf{h}^{(t-1)}$. The exact transformation depends on the type of hidden unit used (see Section 2.2.4). As training with a standard neural network layer transform (as in the input layer) is not feasible due to the problem of decaying gradients (see Section 2.2.3), we instead employ gated recurrent units (GRU) (Chung et al., 2014) for this purpose. The number D of hidden units will range from 64 to 1024 in our experiments.

Output layer

As we are facing a two-class classification task, the class label of a single example $\mathbf{x}_n \in \mathbf{X}$ from the dataset $\mathbf{X}$ is easiest encoded with a single binary variable $t \in \{0, 1\}$. We therefore only require a single output unit y_n at the last timestep with an activation function $\sigma(x)$ that maps the real-valued output $f(\mathbf{w}, \mathbf{x}_n)$ to the interval $[0, 1]$. The neural network is here defined as $f(\mathbf{w}, \mathbf{x}_n)$, with the parameter vector $\mathbf{w}$. With

$$y_n(\mathbf{w}, \mathbf{x}_n) = \sigma(f(\mathbf{w}, \mathbf{x}_n)) = p(\mathcal{C}_1|\mathbf{w}, \mathbf{x}_n)\,, \tag{3.10}$$

we can interpret the network output as a class-conditional probability with $p(\mathcal{C}_1|\mathbf{w}, \mathbf{x}_n)$ for the label class $t = 1$ and $p(\mathcal{C}_2|\mathbf{w}, \mathbf{x}_n) = 1 - p(\mathcal{C}_1|\mathbf{w}, \mathbf{x}_n)$ for $t = 0$.

The natural choice for the activation function is the logistic sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}\,, \tag{3.11}$$

as it represents the canonical link function for Bernoulli-distributed target variables t (Bishop, 2006) with the likelihood function

$$p(t_n|\mathbf{w}, \mathbf{x}_n) = y_n(\mathbf{w}, \mathbf{x}_n)^{t_n} \left[1 - y_n(\mathbf{w}, \mathbf{x}_n)\right]^{1-t_n}\,. \tag{3.12}$$

For a set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ of N independent data samples, we then aim to minimize the negative log-likelihood, which is often referred to as the cross-entropy error function

$$E(\mathbf{w}, \mathbf{X}) = -\sum_{n=1}^{N} \left[t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\right]\,. \tag{3.13}$$

In combination with the sigmoid activation function, the gradient of this error function then has the desirable property that it is proportional to the linear error $y_n - t_n$:

$$\nabla_{\mathbf{w}} E(\mathbf{w}, \mathbf{X}) = \sum_{n=1}^{N} (y_n - t_n) \nabla_{\mathbf{w}} f(\mathbf{w}, \mathbf{x}_n) \,, \tag{3.14}$$

where $\nabla_{\mathbf{w}} f(\mathbf{w}, \mathbf{x}_n)$ can be evaluated via backpropagation (see Section 2.2.2). During training, the parameter vector $\mathbf{w}$ is then iteratively updated in small steps along the direction of the negative gradient.

3.7.2 Weight initialization

We initialize all weight matrices $\mathbf{W} \in \mathbb{R}^{\mathrm{M} \times \mathrm{N}}$ of feed-forward layers uniformly at random with the normalization introduced by Glorot and Bengio (2010)

$$(\mathbf{W})_{ij} \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{M+N}}, \frac{\sqrt{6}}{\sqrt{M+N}} \right) , \tag{3.15}$$

where $\mathcal{U}(-a, a)$ is the uniform distribution for the interval $(-a, a)$. Recurrent layer weight matrices are initialized as random orthogonal matrices, as suggested in Saxe et al. (2013).

At this point we should note, that we initialize the activity of all recurrent units to zero for the first time step during the forward pass.

3.7.3 Training objective and optimization

For the classification task, we aim to minimize the binary cross-entropy error function (Equation 3.13). The optimization task is non-convex due to the highly non-linear network function $f(\mathbf{w}, \mathbf{x}_n)$. Instead of evaluating Equation 3.13 for all training samples $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ at every iteration, we instead iterate over random mini-batches $\mathbf{I}_k \subset \mathbf{X}$ of size $L < N$ and update weights w_i according to the backpropagated gradient

$$w_i^{(t)} = w_i^{(t-1)} - \eta_i^{(t)} \frac{\partial E(w, \mathbf{I}^{(t)})}{\partial w} \bigg|_{w = w_i^{(t-1)}} . \tag{3.16}$$

Mini-batch stochastic gradient descent typically offers shorter convergence times than full batch gradient descent (Li et al., 2014) and makes more efficient use of hardware than stochastic gradient descent with single data samples per iteration. We set the mini-batch size to $L = 64$ for all experiments described here.

Following the adaptive subgradient method as introduced in Duchi et al. (2011), also known as AdaGrad, we choose an adaptive learning rate for each weight

$$\eta_i^{(t)} = \frac{\eta_0}{\sqrt{a_i^{(t)}}}, \tag{3.17}$$

with an initial learning rate $\eta_0 = 0.01$ and an accumulator

$$a_i^{(t)} = a_i^{(t-1)} + \left(\frac{\partial E(w, \mathbf{I}^{(t)})}{\partial w} \Big|_{w=w_i^{(t-1)}} \right)^2 . \tag{3.18}$$

AdaGrad is less sensitive to the choice of an initial learning rate than, e.g. , standard stochastic gradient descent, and therefore saves time as we need to evaluate less hyperparameters. On the other hand, however, better performance could potentially be achieved by fine-tuning the learning rate using a different scheduling method. Due to time constraints we could not explore other approaches in much detail.

We should note that in practice we choose random mini-batches by randomly shuffling the full dataset and then splitting it into batches according to the mini-batch size L. The last mini-batch is typically of size smaller than L and therefore left out. We refer to one full pass over the dataset as an *epoch*. The dataset is re-shuffled for every epoch. Training typically takes several epochs until convergence.

In summary, a single training iteration consists of four steps. A forward pass of a mini-batch through the network, the evaluation of the error function, a backward pass of the error gradients and the weight update according to Equation 3.16.

As sequence length varies between single data samples, additional steps have to be taken to allow efficient forward and backward passes through the network with mini-batches of data. We combine every sample from a mini-batch in a single $L \times M \times l_{\max}$ matrix, where L is the number of samples in the mini-batch, M is the number of features per sequence step and $l_{\max}$ is the maximum sequence length that appears in the full dataset. We pad sequences that are shorter than $l_{\max}$ with a special mask token. The mask token is then, both for training and prediction, passed through the network and sets all activities and error gradients to zero until the first non-mask token value arrives.

We further use dropout (Srivastava et al., 2014; Zaremba et al., 2014) for every recurrent layer as an effective means for regularization. Dropout randomly sets a certain fraction p of the weights to zero during training. This prevents overfitting and improves generalization performance.

3.7.4 Validation score monitoring and early stopping

To further regularize our classifier, we follow the standard technique of monitoring the generalization performance on a separate validation set during training. After each training epoch, we perform a forward pass on the validation set and evaluate the cross-entropy error (Equation 3.13). We keep track of this result for every epoch and stop training if the validation set error has not decreased for $n_{\text{tol}} = 5$ consecutive epochs. Stopping training after convergence on the validation set prevents the model from overfitting on the training set and typically improves generalization performance.

To prevent overfitting due to hyperparameter tuning (e.g. number of parameters or dropout rate), we set aside another part of the dataset, namely the test set. We evaluate the performance on the test set only once after having optimized for hyperparameters using the validation set.

3.7.5 Model averaging

After training, we finally combine predictions from our RNN model with predictions from the GP-based interface classifier (Berning and Helmstaedter, 2016) by simple averaging of both predictions. This typically improves predictive performance, as both models are trained on complementary sets of features (with the exception of the borderSurfaceArea feature, see Section 3.4).

Chapter 4

Experiments and results

In this chapter we describe the main experiments and results. We train a selection of RNNs on the sequence classification task for 3D agglomeration. Finally, we compare the performance of our best model with a benchmark random forest (RF) (Breiman, 2001) classifier and a GP-based interface classifier (Berning and Helmstaedter, 2016).

4.1 Hyperparameter optimization

Hyperparameters can be categorized in training data-related hyperparameters (e.g. path length and class label ratio), architecture-related hyperparameters (e.g. network size, number of layers and unit types) and optimization-related hyperparameters (e.g. learning rate, dropout rate and other means of regularization).

4.1.1 Model-related hyperparameters

Firstly, let us investigate model-related hyperparameters, i.e. settings related to architecture and optimization of our RNN model.

Unit types

To alleviate the problem of vanishing gradients (see Section 2.2.3), several special recurrent unit types have been introduced (see Section 2.2.4).

We choose gated recurrent units (GRU) as introduced by Chung et al. (2014) for all further experiments. They deal very well with the problem of vanishing gradients while introducing fewer additional parameters as, say, the popular long short-term memory (LSTM) unit, first introduced by Hochreiter and Schmidhuber (1997).

We leave evaluation of other unit types and variations thereof open for further study, as we only had limited computational resources and time available for the experiments described here.

Dropout

We explored several different values for the free parameter in dropout (Srivastava et al., 2014; Zaremba et al., 2014) (see Section 3.7.3). We found that a dropout rate of $p = 0.5$ generally worked well. Similar as described in (Srivastava et al., 2014), results were not very sensitive on the exact choice of the dropout rate, as long as it was roughly in the interval $[0.4, 0.8]$.

Optimizers

There exist a number of powerful optimizers for the problem of mini-batch gradient descent. Standard (mini-batch) stochastic gradient descent is known for inferior performance on RNNs (Pascanu et al., 2013). Due to time constraints, we only evaluated the AdaGrad optimizer as introduced by Duchi et al. (2011), as it adaptively sets the learning rate for every parameter and produces reliable results with little to no hyperparameter tuning. We found that the recommended learning rate of $\eta_0 = 0.01$ works well in practice.

Further improvements can probably be expected from using a different optimizer, such as RMSProp (Tieleman and Hinton, 2012), and searching for both optimal learning rate and learning rate decay parameters.

Number of units

The number of hidden recurrent units h_{dim} is one of the most important hyperparameters for RNNs. The higher h_{dim}, the more expressive the model will be. If h_{dim} is chosen too low, the model will most likely underfit the data. On the other hand, if h_{dim} is chosen too high, the model will overfit and generalize poorly to other datasets. While the latter point can be alleviated by regularization, it is still not advisable to choose h_{dim} as large as possible. A large value for h_{dim} will significantly slow down training, as the number of parameters in the network grows quadratically with h_{dim}.

We performed a grid search over $h_{\text{dim}} \in \{64, 128, 256, 512, 1024\}$, with the other parameters chosen as: $\eta_0 = 0.01$ (AdaGrad learning rate), $p = 0.5$ (dropout rate) and $L = 64$ (mini-batch size). We trained the network by optimizing the cross-entropy error (see Section 3.7.3). We stopped training after the validation error plateaued (i.e. no improvement for 5 consecutive epochs). Results are summarized in Table 4.1.

While the validation error remains relatively constant for all architectures, clear signs of overfitting can be seen with regard to the training error for higher numbers of hidden units. The models with $h_{\text{dim}} = 256$ generalized best to the validation set, as indicated by the low validation error.

$h_{\mathbf{dim}}$	**Layers**	**Training error (%)**	**Validation error (%)**
64	1	12.23 ± 0.14	13.69 ± 0.11
128	1	11.44 ± 0.18	13.36 ± 0.23
256	**1**	$\mathbf{10.89 \pm 0.06}$	$\mathbf{13.22 \pm 0.05}$
512	1	10.34 ± 0.34	13.31 ± 0.11
1024	1	9.83 ± 0.46	13.53 ± 0.13
128	2	11.75 ± 0.10	13.40 ± 0.08
256	2	11.04 ± 0.12	13.30 ± 0.17

Table 4.1 Final training and validation error (cross-entropy) after convergence for different numbers of units h_{dim} in the hidden recurrent layer. Mean and standard deviation of the error are calculated from three experiments with random initial conditions each. Experiments with two hidden layers were performed with identical (hidden recurrent) layers of size h_{dim}. The network with $h_{\text{dim}} = 256$ and a single hidden recurrent layer (highlighted) seems to find the right balance between over- and underfitting.

Number of layers

Most applications of RNNs use one or two hidden recurrent layers in practice. More than two hidden layers are typically advised against (Karpathy et al., 2015), although specific scenarios might benefit from a deeper representation. We here compare architectures with one and two hidden recurrent layers. Results are shown in Table 4.1.

While the difference in validation error might not be significant between the single- and the two-layered options with same number of hidden units per layer, we choose in favor of the simpler model (i.e. single hidden layer, $h_{\text{dim}} = 256$).

4.1.2 Data-related hyperparameters

Here, we evaluate input data-related hyperparameters, namely the influence of sequence length and class ratios in the dataset on the generalization ability of the classifier.

Influence of sequence length

We use the parameters of the best model so far, i.e. a GRU-RNN with a single hidden recurrent layer with $h_{\text{dim}} = 256$ and parameters otherwise as described in the previous section. We investigate the influence of sequence length on the generalization performance of our classifier in terms of the mean validation error for several different settings.

Firstly, we investigated how a model that was trained on the full sequence length (i.e. the original training dataset) performs on a validation set with shorter sequences.

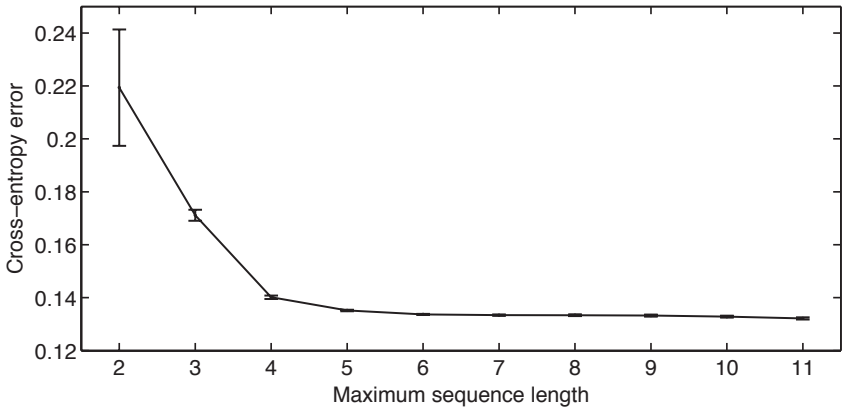


Fig. 4.1 Mean validation error (cross-entropy) on validation sets with sequences cut to a maximum sequence length $l_{\text{max}} \in [2, 11]$. Average predictions of three experiments. The RNN classifier was trained on the full training set with a single GRU hidden layer of $h_{\text{dim}} = 256$. Error bars show standard deviation.

For the validation set, we cut sequences to a maximum length $l_{\text{max}} \in [2, 11]$. Figure 4.1 summarizes the results. Apparently, generalization ability of our model severely suffers for datasets with sequences shorter than $l_{\text{max}} = 4$, as these were not part of the training set. The minimum sequence length that appears in our original dataset is $l_{\text{min}} = 4$.

We further evaluated, how our model generalizes when we both train and predict on datasets with a constraint for a maximum sequence length l_{max}. Therefore we also cut all sequences in the training data to a maximum length l_{max}. Due to time constraints (as the model needs to be re-trained for every experiment) we only investigated generalization performance for $l_{\text{max}} \in \{4, 7, 11\}$ in terms of the mean validation error. Results are shown in Table 4.2.

l_{max}	**Training error (%)**	**Validation error (%)**
11	$\mathbf{10.89 \pm 0.06}$	$\mathbf{13.22 \pm 0.05}$
7	11.16 ± 0.11	13.46 ± 0.12
4	11.07 ± 0.04	13.27 ± 0.04

Table 4.2 Final training and validation error (cross-entropy) after convergence for different sequence lengths. All training and validation set sequences were cut to a maximum length l_{max}, with $l_{\text{max}} = 11$ being the original dataset. Mean and standard deviation of the error are calculated from three experiments with random initial conditions each. While the experiments using the full length dataset scored best (highlighted), the difference in validation error is not very significant.

Here, generalization ability is very similar for the model trained on $l_{\max} = 4$ and the original model $l_{\max} = 11$. Training on shorter sequence lengths significantly speeds up training and puts fewer constraints on possible applications for 3D agglomeration of our sequence learning model. We therefore choose the model with $l_{\max} = 4$ for all further experiments and recommend its use in practice.

Class equalization

The original dataset as summarized in Table 3.1 exhibits a particular imbalanced class ratio of approximately $1:18$ (positive : negative) for both the training and the validation set. We here investigate whether generalization ability of our RNN classifier can be improved by training on a class-balanced version of the dataset. We compare two approaches to achieve balanced classes.

Oversampling of positive class In this approach, we perform copies of members of the positive class (i.e. examples in the training dataset with class label $t = 1$) until we arrive at a balanced class ratio. We uniformly at random (with replacement) select positive examples from the original training dataset and duplicate them. We stop this procedure as soon as we arrive at a class ratio of $1:1$.

Undersampling of negative class Here, we uniformly at random select examples from the negative class (i.e. class label $t = 0$) and remove them from the training dataset until a balanced class ratio is achieved. This significantly decreases the size of the dataset and potentially removes informative examples that might benefit the training of the classifier.

Type	**Validation error (%)**
Original class ratio	$\mathbf{13.22 \pm 0.05}$
Oversampling (positive class)	34.26 ± 2.75
Undersampling (negative class)	48.31 ± 2.19

Table 4.3 Final validation error (cross-entropy) after convergence for different class ratios in the training dataset. The validation set was left unchanged in order to allow direct comparison of the validation error. Mean and standard deviation of the error are calculated from three experiments with random initial conditions each. Equalizing the class ratios in the dataset lead to significantly impeded generalization ability of the classifier in comparison to training on the original class ratio (highlighted).

We again compare the different approaches in terms of the mean validation error with three random initializations per approach. We use the model with optimized hyperparameters as described in the previous sections with a single hidden layer of $h_{\text{dim}} = 256$ and a maximum sequence length of $l_{\text{max}} = 4$ both for the training and the validation set. Results are summarized in Table 4.3.

Apparently the model fails to generalize to the validation set when it was trained on an augmented training set with a balanced class ratio. The approach using undersampling of negative examples performs particularly poorly, as the size of the training set is significantly reduced (to roughly $60,000$ examples).

4.2 Evaluation of best model

In this section, we compare the best model[1] from the previous section (i.e. lowest validation error) with a simple benchmark model, namely a random forest (RF) (Breiman, 2001) classifier that acts on fixed-length inputs. We compare both models for sequences with a length of $l_{\text{min}} = l_{\text{max}} = 4$. Both models are tested on the merger mode test set as described in Section 3.6.

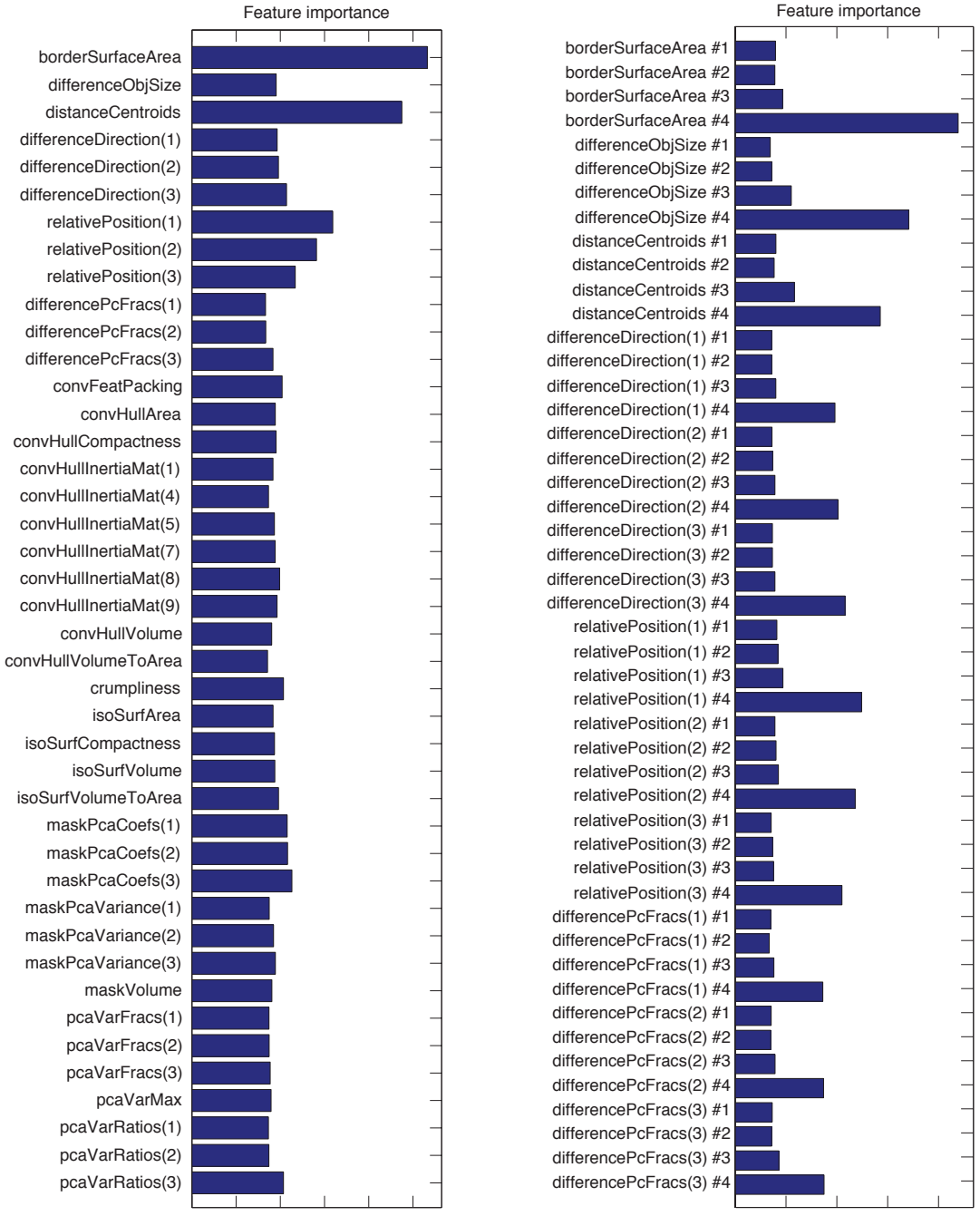
The RNN has a single GRU hidden layer with $h_{\text{dim}} = 256$ and is trained on the training dataset with paths cut to $l_{\text{max}} = 4$ (the same restriction applies to the RF classifier). We further use a dropout rate of $p = 0.5$ and an AdaGrad learning rate of $\eta_0 = 0.01$. The RF classifier is implemented using MATLAB®'s TreeBagger class with $N = 100$ trees.

4.2.1 Feature importance

Training an RF classifier on our dataset provides us with some insight on the importance of single features on the classification decision. We can estimate this importance by keeping track of the out-of-bag error during training as suggested in (Breiman, 2001). MATLAB® provides an implementation for this method. Results are summarized in Figure 4.2.

It is apparent that edge features are on average more important than segment-based features. Furthermore, as one would expect, features of edges closer to the decision edge are assigned a higher importance than those of distant edges. Note that results may vary from trial to trial, as the decision trees in an RF are randomly grown. This result is to be understood as a very rough estimate of the actual predictive power of individual features.

[1] GRU-RNN 20160302-141311



(a) Averaged over path length

(b) Individual; trained on edge features only

Fig. 4.2 Feature importance of an RF classifier (in arbitrary units) for the training set with paths cut to a maximum length of $l_{\max} = 4$. *Left*: Feature importance averaged over the whole sequence. *Right*: Feature importance for an RF classifier trained on edge features only, shown for all edges individually. #1 denotes the edge from the source node; #4 denotes the decision edge.

4.2.2 Precision and recall

With both the RNN and the RF trained on the same dataset, we can compare their performance on the merger mode test set that we have previously created for this purpose (see Section 3.6). As the feature set (see Section 3.4) for our models was chosen complementary (with the exception of the borderSurfaceArea feature) to the feature set that the GP-based interface classifier (Berning and Helmstaedter, 2016) was trained on, we combine our new models with the GP predictions for better predictive performance. We create ensemble models as follows. An RNN+GP model is built by averaging the RNN prediction for a decision edge with the respective GP prediction (the latter is available for every edge in the dataset). Analogously, we create an RF+GP benchmark model using the RF predictions in the same way.

We then use the trained RNN and RF models to make predictions on every decision edge (see Section 3.3) in the test set. We average predictions in case of multiple paths converging onto a single decision edge.

As described in Section 3.5.3, we use the precision/recall metric to compare models. Specifically, we are interested in the maximum recall achieved at a precision of 98%. Figure 4.3 summarizes this result for all models.

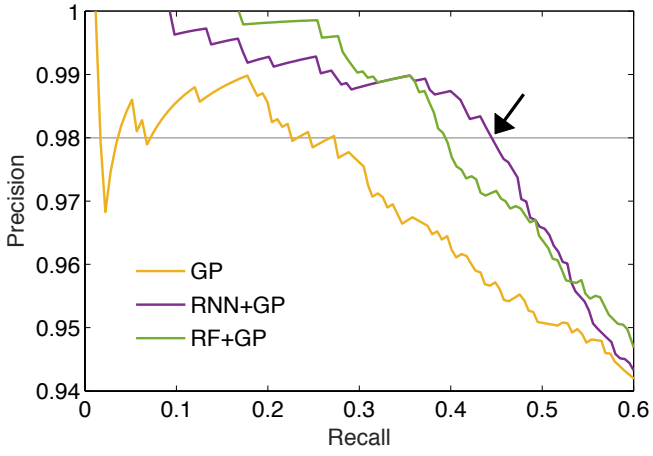


Fig. 4.3 Comparison of our RNN-based model (RNN+GP) with the original baseline GP-based model (GP) (Berning and Helmstaedter, 2016) and a benchmark RF-based model (RF+GP). Up to a recall of 60%, both the RF+GP and the RNN+GP model consistently perform better than the baseline GP model. Compared at 98% precision (gray line), the RNN+GP model wins by a small margin (arrow).

At the 98% precision level, our proposed RNN+GP model has a slight edge compared to the other models. Both the RNN+GP and the RF+GP model consistently outperform the GP interface classifier (Berning and Helmstaedter, 2016) alone for recall values smaller than 60%. For higher recall values, the situation reverses (not shown here).

4.2.3 Mean reconstructed path length

We further compare models on a more practically motivated metric, namely the mean reconstructed path length. We measure this quantity again at 98% precision, i.e. with a certain tolerated merge-error rate. This defines the decision threshold under which we shall accept edges (add to agglomerate) or decline (keep processes split). We then find connected components (i.e. agglomerates) on the merger mode test set given our model predictions. We discard connected components consisting only of a single edge. Then, we calculate the reconstructed path length as the diameter (in a graph-theoretic sense) of the connected component, with edge lengths given by their physical length (center of mass distance) in 3D space. Table 4.4 shows the results.

Model	**Reconstructed length**
GP (Berning and Helmstaedter, 2016)	$2.9 \pm 2.5\,\mu m$
RNN+GP	$\mathbf{7.3 \pm 1.2\,\mu m}$
RF+GP	$\mathbf{7.6 \pm 1.5\,\mu m}$

Table 4.4 A comparison of mean reconstructed path lengths on the merger mode test set at 98% precision. See text for more details. The error denotes the standard deviation of reconstructed path lengths on the test set. Both new models (highlighted) show a significant improvement over the GP interface classifier alone.

We should note that for both the RNN- and the RF-based models, the classifiers were seeded with 4 segments of the ground truth dataset to allow for sequence classification. These segments do not count towards the reconstructed length to allow for a fair comparison with the GP-based interface classifier (Berning and Helmstaedter, 2016) that acts on single edges (pairs of segments) only. Under this metric, both ensemble models (RNN+GP and RF+GP) improve upon the baseline GP-based interface classifier (Berning and Helmstaedter, 2016) alone.

Chapter 5

Discussion and outlook

In this chapter, we summarize and discuss our method and the results described in the previous chapters. We conclude by providing an outlook on possible future work.

5.1 Summary and discussion

In this thesis, we introduced a new method for 3D agglomeration of over-segmented connectomics image data. Our method integrates information of the full agglomeration graph and thereby extends previous work on interface-based classifiers for 3D agglomeration (Berning and Helmstaedter, 2016; Bogovic et al., 2013). We framed the problem of agglomeration as a sequence learning task and developed methods to efficiently derive training data from human-traced skeletons. We demonstrated this capability on a dataset of traced axons annotated by a single student and trained an RNN sequence classifier on this dataset.

Comparing the performance of the best RNN-based classifier with a GP-based interface classifier (Berning and Helmstaedter, 2016) and a separate RF-based classifier, provided us with valuable insights on both the benefits and the restrictions of our proposed model. We have seen that our proposed method does not necessarily benefit from long sequences (i.e. long paths sampled from an agglomerate) and that the restriction of sequences to a common fixed length (see Section 4.1.2) does not significantly alter the performance of our model.

The strength of RNNs in comparison to fully connected neural networks or other classifiers that operate on fixed-size input (such as an RF) lies in weight sharing along the time domain. For stationary time series inputs this property allows for an effective reduction of the number of model parameters without compromising predictive performance. On the other hand, RNNs, being dynamical systems, typically require a finite number of time steps during a forward pass to "settle in" and adapt to the input

sequence. As sequences get shorter, the benefit of temporal weight sharing vanishes and models that operate on fixed-size input might become viable alternatives. This could explain the similar performance of the RF- and the RNN-based models on the dataset of fixed-length sequences of length $l_{\max} = 4$ (see Sections 4.2.2 and 4.2.3).

Although not fully investigated here, we expect that it should be possible to restrict both training and testing datasets to even shorter sequences of length two to three edges without significantly reducing predictive performance. This suggestion is motivated by the rapid drop in feature importance (see Section 4.2.1) over the length of the sequence.

An interesting idea related to our approach was introduced in Jurrus et al. (2008). They frame the problem of agglomeration as finding an optimal path through consecutive 2D slices of the segmentation and they allow this path to skip slices of bad quality, thereby improving reconstruction accuracy. This skip-prediction approach could be transferred to our method as follows: Instead of predicting on candidate edges to direct neighboring segments, we can allow to predict on the second-order neighbor of a segment, thereby skipping a potentially difficult segment. Segments that are difficult for the classifier to ambiguate typically arise due to imaging- or alignment-related artifacts in the dataset. This approach, however, could be very challenging for a classifier to be trained on, as the number of higher-order neighbors increases exponentially with the number of skipped segments. As opposed to the approach in Jurrus et al. (2008), we are considering the full 3D segmentation and not just 2D slices thereof. We thereby have to deal with a considerably larger number of neighbors: On the order of 10 first-order and 100 second-order neighbors for the given segmentation (Berning, 2014). One would have to come up with a smart way of restricting this set, as otherwise even a very small false positive rate of a classifier would result in a significant drop in precision due to a strongly imbalanced label ratio.

5.2 Future work

Our work here is to be understood as a first proof of principle that opens up a variety of avenues for future research. We propose a couple of ideas in the following.

The obvious next step for our method is to implement and test it in a full reconstruction pipeline. For that, one can make use of the fully trained models described here, but the efficient combination with existing methods and heuristics for manually seeded agglomeration of, say, axonal segments still remains an open problem.

Another potential application of our method lies in the classification of full paths, instead of merely classifying what we call the decision edge (see Section 3.3). This could be useful for the problem of determining whether two spatially separate segments (e.g. two axonal boutons) are connected, via classification of potential connecting paths

as being plausible or not. For this problem, RNNs are the models of choice, as these paths will generally vary in length.

As noted previously, we have only explored a limited set of hyperparameters and features for our models. Additional performance improvements can be expected from extending these. Furthermore, increasing the size of the training set might be beneficial and worth exploring.

A particularly interesting extension to the work presented here would be to replace hand-designed features (see Section 3.4) altogether via a so-called end-to-end approach as in Karpathy and Fei-Fei (2015). Here, an RNN is combined with a convolutional neural network as a trainable feature extractor in a single model that is fully (end-to-end) trainable via backpropagation.

With or without these proposed extensions, we expect that the proof of principle method introduced in this thesis will be a useful tool for the challenging problem of (semi-)automated reconstruction of 3D-EM data.

References

Andres, B., Koethe, U., Kroeger, T., Helmstaedter, M., Briggman, K. L., Denk, W., and Hamprecht, F. A. (2012). 3d segmentation of sbfsem images of neuropil by a graphical model over supervoxel boundaries. *Medical image analysis*, 16(4):796–805.

Andres, B., Köthe, U., Helmstaedter, M., Denk, W., and Hamprecht, F. A. (2008). Segmentation of sbfsem volume data of neural tissue by hierarchical classification. In *Pattern recognition*, pages 142–152. Springer.

Baldi, P. and Pollastri, G. (2003). The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *J. Mach. Learn. Res.*, 4:575–602.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Baydin, A. G., Pearlmutter, B. A., and Radul, A. A. (2015). Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral presentation.

Berning, M. (2014). Segmentation pipeline CNN 20141007T094904$_{8,3}$. Personal communication.

Berning, M., Boergens, K. M., and Helmstaedter, M. (2015). SegEM: Efficient image analysis for high-resolution connectomics. *Neuron*, 87(6):1193–1206.

Berning, M. and Helmstaedter, M. (2016). Gaussian process interface classifier. Unpublished.

Beucher, S. and Meyer, F. (1992). The morphological approach to segmentation: the watershed transformation. *Optical Engineering*, 34:433–481.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Boergens, K. M. (2015a). Merger mode tracing tool. Personal communication.

Boergens, K. M. (2015b). SynTypComm.nml. Personal communication.

Boergens, K. M. and Helmstaedter, M. (2012). 3D-EM dataset: Mouse primary somatosensory cortex layer 4, 2012-09-28_ex145_07x2. Unpublished data.

Bogovic, J. A., Huang, G. B., and Jain, V. (2013). Learned versus hand-designed feature representations for 3d agglomeration. *arXiv preprint arXiv:1312.6159*.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Briggman, K. L. (2016). EM techniques for connectomics. IMPRS Master Class. Oral presentation.

Briggman, K. L. and Denk, W. (2006). Towards neural circuit reconstruction with volume electron microscopy techniques. *Current opinion in neurobiology*, 16(5):562–570.

Chollet, F. (2015). keras. https://github.com/fchollet/keras.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*.

Denk, W., Briggman, K. L., and Helmstaedter, M. (2012). Structural neurobiology: missing link to a mechanistic understanding of neural computation. *Nature Reviews Neuroscience*, 13(5):351–358.

Denk, W. and Horstmann, H. (2004). Serial block-face scanning electron microscopy to reconstruct three-dimensional tissue nanostructure. *PLoS Biol*, 2(11):e329.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Duvenaud, D., Maclaurin, D., Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems (NIPS)*.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Felzenszwalb, P. F. and Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. *Int. J. Comput. Vision*, 59(2):167–181.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10)*.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376.

Graves, A. and Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552.

Hayworth, K. J., Kasthuri, N., Schalek, R., and Lichtman, J. W. (2006). Automating the collection of ultrathin serial sections for large volume tem reconstructions. *Microscopy and Microanalysis*, 12(S02):86–87.

Helmstaedter, M. (2013). Cellular-resolution connectomics: challenges of dense neural circuit reconstruction. *Nature methods*, 10(6):501–507.

Helmstaedter, M., Berning, M., Boergens, K. M., Wissler, H., et al. (2015). webKnossos. https://webknossos.brain.mpg.de. Unpublished.

Helmstaedter, M., Briggman, K. L., and Denk, W. (2008). 3d structural imaging of the brain with photons and electrons. *Current opinion in neurobiology*, 18(6):633–641.

Helmstaedter, M., Briggman, K. L., and Denk, W. (2011). High-accuracy neurite reconstruction for high-throughput neuroanatomy. *Nat Neurosci*, 14(8):1081–1088.

Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen*. Diploma thesis, TU Munich.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Jain, V., Murray, J. F., Roth, F., Turaga, S., Zhigulin, V., Briggman, K. L., Helmstaedter, M. N., Denk, W., and Seung, H. S. (2007). Supervised learning of image restoration with convolutional networks. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE.

Jain, V., Turaga, S. C., Briggman, K., Helmstaedter, M. N., Denk, W., and Seung, H. S. (2011). Learning to agglomerate superpixel hierarchies. In *Advances in Neural Information Processing Systems*, pages 648–656.

Jerrell, M. E. (1997). Automatic differentiation and interval arithmetic for estimation of disequilibrium models. *Computational Economics*, 10(3):295–316.

Jurrus, E., Whitaker, R., Jones, B. W., Marc, R., and Tasdizen, T. (2008). An optimal-path approach for neural circuit reconstruction. *Proc IEEE Int Symp Biomed Imaging*, 2008(4541320):1609–1612.

Karpathy, A. and Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137.

Karpathy, A., Johnson, J., and Li, F. (2015). Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078.

Kim, J. S., Greene, M. J., Zlateski, A., Lee, K., Richardson, M., Turaga, S. C., Purcaro, M., Balkam, M., Robinson, A., Behabadi, B. F., et al. (2014). Space-time wiring specificity supports direction selectivity in the retina. *Nature*, 509(7500):331.

Lee, K., Zlateski, A., Vishwanathan, A., and Seung, H. S. (2015). Recursive training of 2d-3d convolutional networks for neuronal boundary detection. *arXiv preprint arXiv:1508.04843*.

Li, M., Zhang, T., Chen, Y., and Smola, A. J. (2014). Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Madras, N. and Slade, G. (1996). *The Self-Avoiding Walk*. Birkhäuser Boston.

Maitin-Shepard, J., Jain, V., Januszewski, M., Li, P., Kornfeld, J., Buhmann, J., and Abbeel, P. (2015). Combinatorial energy learning for image segmentation. *arXiv preprint arXiv:1506.04304*.

Motta, A. (2016). 3D shape features. Personal communication.

Natarajan, N., Dhillon, I. S., Ravikumar, P. K., and Tewari, A. (2013). Learning with noisy labels. In *Advances in Neural Information Processing Systems (NIPS)*.

Nunez-Iglesias, J., Kennedy, R., Plaza, S. M., Chakraborty, A., and Katz, W. T. (2014). Graph-based active learning of agglomeration (gala): a python library to segment 2d and 3d neuroimages. *Frontiers in neuroinformatics*, 8.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1310–1318.

Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and Mcclelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA.

Sak, H., Senior, A. W., and Beaufays, F. (2014). Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Interspeech*, pages 338–342.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Scott, C., Blanchard, G., and Handy, G. (2013). Classification with asymmetric label noise: Consistency and maximal denoising. *J. Mach. Learn. Res.*, 30:489–511.

Sommer, C., Straehle, C., Koethe, U., and Hamprecht, F. A. (2011). ilastik: Interactive learning and segmentation toolkit. In *IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 230–233. IEEE.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5 - RMSProp. Coursera: Neural Networks for Machine Learning.

Turaga, S. C., Murray, J. F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., Denk, W., and Seung, H. S. (2010). Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural computation*, 22(2):511–538.

Vazquez-Reina, A., Gelbart, M., Huang, D., Lichtman, J., Miller, E., and Pfister, H. (2011). Segmentation fusion for connectomics. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 177–184. IEEE.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

White, J. G., Southgate, E., Thomson, J. N., and Brenner, S. (1986). The structure of the nervous system of the nematode caenorhabditis elegans: the mind of a worm. *Phil. Trans. R. Soc. Lond*, 314:1–340.

Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. *Back-propagation: Theory, architectures and applications*, pages 433–486.

Zahn, C. T. (1971). Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, C-20(1):68–86.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Erklärung zur Masterarbeit

Hiermit erkläre ich an Eides statt, die vorliegende Masterarbeit selbstständig und lediglich unter Benutzung der im Literaturverzeichnis und anderweitig angegebenen Quellen und Hilfsmittel verfasst zu haben.

Erlangen, den 18. März 2016

.....
Unterschrift