Generative Probabilistic Models of Neuron Morphology

by

Stephen Rothrock Serene

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2013
[June 2013]

Author..
Department of Electrical Engineering and Computer Science
May 2013

Certified By..
Joshua Tenenbaum
Professor of Brain and Cognitive Sciences
Thesis Supervisor

Accepted By...
Professor Dennis M. Freeman
Chairman
Masters of Engineering Thesis Committee

[This page intentionally left blank]

Generative Probabilistic Models of Neuron Morphology

by

Stephen Rothrock Serene

Submitted to the Department of Electrical Engineering and Computer Science

June 2012

In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Thanks to automation in ultrathin sectioning and confocal and electron microscopy, it is now possible to image large populations of neurons at single-cell resolution. This imaging capability promises to create a new field of neural circuit microanatomy. Three goals of such a field would be to trace multi-cell neural networks, to classify neurons into morphological cell types, and to compare patterns and statistics of connectivity in large networks to meaningful null models. However, those goals raise significant computational challenges. In particular, since neural morphology spans six orders of magnitude in length (roughly 1 nm-1 mm), a spatial hierarchy of representations is needed to capture micron-scale morphological features in nanometer resolution images. For this thesis, I have built and characterized a system that learns such a representation as a Multivariate Hidden Markov Model over skeletonized neurons. I have developed and implemented a maximum likelihood method for learning an HMM over a directed, unrooted tree structure of arbitrary degree. In addition, I have developed and implemented a set of object-oriented data structures to support this HMM, and to produce a directed tree given a division of the leaf nodes into inputs and outputs. Furthermore, I have developed a set of features on which to train the HMM based only on information in the skeletonized neuron, and I have tested this system on a dataset consisting of confocal microscope images of 14 fluorescence-labeled mouse retinal ganglion cells. Additionally, I have developed a system to simulate neurons of varying difficulty for the HMM, and analyzed its performance on those neurons. Finally, I have explored whether the HMMs this system learns could successfully detect errors in simulated and, eventually, neural datasets.

Thesis Supervisor: Joshua Tenenbaum
Title: Professor of Brain and Cognitive Sciences

[This page intentionally left blank]

Acknowledgments

I thank the following people for supporting this thesis:

Josh Tenenbaum for supporting a outside his existing research.

Uygar Sumbul for providing skeletonized confocal datasets and advice on features

Jinseop Kim for providing skeletonized electron micrograph datasets

Sebastian Seung for introducing me to connectomics.

Ignacio Arganda-Carreras, and Daniel Berger for introducing me to segmentation.

Tamara Fleisher for contributing code and design ideas when this was a 9.660 class project.

Paul Hlebowitsh for providing LaTeXtemplates.

[This page intentionally left blank]

List of Figures

List of Tables

[This page intentionally left blank]

Contents

[This page intentionally left blank]

1 Introduction

Automated neuron-tracing through stacks of electron-micrographs of brain tissue would be a powerful tool for neuroscience, particularly as large-scale image acquisition becomes practical. [2] Current segmentation architectures approach this problem by passing traditional image-processing data structures (pixels or affinity edges) to filters and classifiers (convolutional networks, SVMs, etc.), weighting the data to optimize performance measures like Rand or Warping error. [2] [3] However, these algorithms too-commonly split a neuron or merge neurons together, partly because they consider only local information (1 ?m3 around a pixel). Thus, while neurons exhibit a morphological grammar on length-scales of 10s-100s of microns, [1] existing architectures neither capture nor exploit it. Since split and merge errors likely violate that grammar, an architecture that captures it might detect and even correct them. I propose to develop such an architecture by using existing tools to obtain an approximate segmentation and convert it into a skeletonized tree structure; assigning feature vectors to bins along the branches of the tree, with features like neurite width and curvature (and their derivatives), branching factor, position, and the presence of vesicles or organelles; and, finally, training a multivariate Hidden Markov Model (HMM) on those features. All the components of that pipeline have been implemented in the past, citestung with the exception of HMM and features on the skeleton structure. I have designed and implemented such a system, which I describe in this thesis.

An HMM consists of a matrix that gives the transition probabilities among a set of hidden states and a matrix specifying the chance that, from a given state, the system will emit each possible value of the observed data. The hidden state and its transitions are defined along one dimension of the data (e.g. word order in linguistics, sequence position in genomics). In this project, that dimension is position along a skeletonized neuron. [4]

In addition to feature extraction and selection, applying HMMs to neurons introduces two novel challenges: how to orient and learn on a branched structure. HMMs can learn the asymmetric transition probabilities characteristic of neurons (spines join dendrites, but dendrites dont join spines), but learning them requires systematically assigning a direction to each branch of the skeletonized tracing. In acyclic neurons where all the input-output paths pass through a single cell body, a greedy

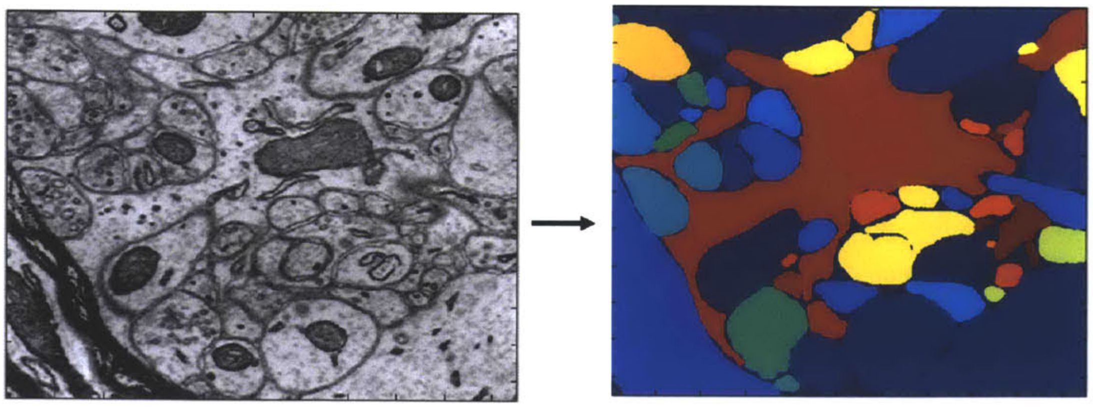


Figure 1: Electron micrographs allow segmentation of dense networks of neurons. This figure shows a section of mouse cortex hand-segmented by Daniel Berger in the Seung lab at MIT.

algorithm can orient the edges given inputs and outputs. Input and output terminals can likely be identified using morphological features (e.g. the presence of vesicles) or position relative to the cell body (e.g. in the cortex or retina).

To learn on a branched structure, this architecture will augment the standard HMM with new 3D transition arrays that specify the behavior at branches and joins. With these arrays specified, the model can be trained using a straightforward extension of an expectation- maximization method like Baum-Welch. Split and merge errors might appear as low-likelihood transitions in the HMM (e.g. an axon connecting to a dendrite). A single-class SVM could also identify errors using the HMM states as features (e.g. a neuron with two axons). Extensions of this work might include training on confocal-stack libraries rather than tracings, learning non-Markov grammars and cell types via hierarchical Bayesian methods, identifying HMM correlates of function and disease, and enabling error-correction by allowing the model to split or merge skeletons.

2 System Design

In this section, I describe my prototype system for learning Hidden Markov Models of neuron morphology. In section 2.1, I first describe the data sets I curated for the project, which consisted of an initial set of 21 confocal fluoresce micrograph stacks of single labeled neurons in mouse cortex, and their skeletonized representations. Next, I describe the object-oriented data structures I designed to represent skeletonized neurons in a probabilistic model. I then describe the process of ingesting and cleaning the raw data, and the feature set I developed based on skeleton topology.

In section 2.2, I describe the probabilistic model I use to learn Markov structures on fixed branched morphologies, and describe an extension of that model that has the flexibility to learn a joint model of morphology and topology. While the fixed-topology model suffices for the applications in image segmentation that initially inspired this work, the joint model would allow complete *de novo* simulation of neural networks according to a learned model. I concluded section 2.2 by describing the expectation-maximization routine I use to train the model, which is a straightforward generalization of the common Baum-Welch routine.

In section 2.3, I describe the methods I developed to test the performance of the model and training routines described in section 2.2. I first describe the data I track during the learning process. Simple expectation-maximization routines like the one I implement here are hill-climbing algorithms that converge to local optima, and the data I collect during training measure the speed and degree of that convergence. Next, I describe the variant of the Viterbi maximum likelihood state estimator that I implemented, and the routine I use to determine the optimal map from learned states to initial states in a ground truth model, and the data I collect to describe the match between the learned and ground-truth models, given this optimal state-mapping. Finally, I describe the methods I employed to generate simulated skeleton data based on known probabilistic models on the skeleton topologies in a given data set. These methods provide a single parameter with which the user can tune the challenge of the simulated data to the learning routines.

Finally, in section 2.4, I describe the methods I have implemented to detect segmentation errors in skeletonized neurons.

2.1 Data Sets

The ideal data set for this project would have consisted of a set of images that had been skeletonized by multiple human experts, and which had additionally been segmented by an automated neuron tracing system, with the resulting segmentation separately skeletonized. Obtaining such a data set proved difficult. Given the difficulty of obtaining and aligning large, high quality electron micrograph stacks, there are relatively few data sets available based on electron microscope data that contain full neurons. Since the goal of this project was to learn large-scale features of neuron morphology, the data sets typical of electron microscope studies were not helpful, as they typically consist of roughly 1mm on a side cubes, which contain fractions of many neurons, but no complete neurons. (In fact, it is not even possible to determine whether the ostensibly different neurons in these cubes in fact connect to each other outside the imaged volume.) I was able to obtain a single neuron skeletonized through the EyeWire online crowd-sourcing platform, but did not ultimately attempt to train a model on this single neuron. [5]

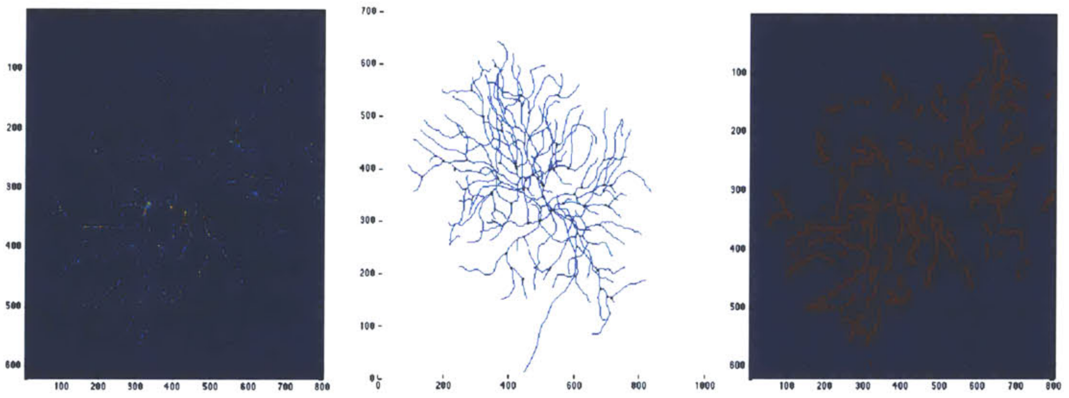


Figure 2: A confocal fluorescence microscope slice, the skeletonized neuron it is drawn from, and the predicted neuron volume around the skeleton (from right to left).

Since presently available electron micrograph stacks were poorly-suited to the needs of this project, I turned instead to data sets collected through confocal microscopy of single, fluoresce-labeled cells. To generate these images, the experimenter obtains a line of mice that express a fluorescent protein driven by a promoter activated in a random, sparse subset of cells (roughly 10% in the strain used to create the data presented here). This sparse labeling sidesteps the difficulty caused by sub-light-diffraction-limit interlacing of different neurons by simply making it unlikely that more than one of the interlacing neurons will be emitting any fluorescence. A further benefit of this technique is that the confocal microscope can obtain a stack of images from a single physical section, simplifying the challenges that arise in electron microscope studies of z-axis asymmetry and image alignment. Of course, that sparse labeling comes at a price, since it is no longer possible to reconstruct the full neural network in a region, as is (at least in theory) possible with electron microscope data. Thus, the Thyl-GFP confocal data sets are most scientifically useful for studying brains where the anatomy is stereotyped (such as insects), or regions of the mammalian brain that have relatively stereotyped micro anatomy across individuals (such as the retinal ganglion cells). citethy1

Given the greater availability of skeletonized, complete neurons in confocal data sets, I chose to use one of them. The initial data set I obtained consisted of 18 images taken in the Seung lab at MIT and the Masland lab in Harvard Medical School, from the retinas of Thyl-GFPM mice, each nominally containing a single fluorescence-labeled retinal ganglion cell. In addition, these neurons had been manually skeletonized by human experts using a graphical user interface. Finally, a variant of the convolutional network segmentation system described in citejain2010 had been used to predict the volume occupied by each neuron, based on the human-provided skeletons and the raw image data. The skeletons, the predicted borders, and the raw images were all generously provided to me by Uygar Sumbul, a post-doctoral researcher in the Seung lab at MIT.

2.1.1 Data Ingestion

I planned to develop a set of features based on both the skeleton topology and the width of the predicted neuron volume around the skeleton (and the derivatives of that width along the direction of the skeleton). Unfortunately, as described below, the raw images had been re-sized so that they could fit into the graphical processing units (GPUs) used by the convolutional network routine that produced the predicted volumes, and thus both the images and the predicted volumes did not align with the skeletons. In some cases, the raw-image and predicted-volume arrays were smaller in all dimensions than the skeletons, while in other cases the images were larger in some dimensions and smaller in others. Thus, to align the images to the skeletons, I reasoned that neither the image nor the skeleton was likely to be cropped along a given dimension on one end while overhanging on the other end of that same dimension. In other words, I was assumed that whomever did the cropping

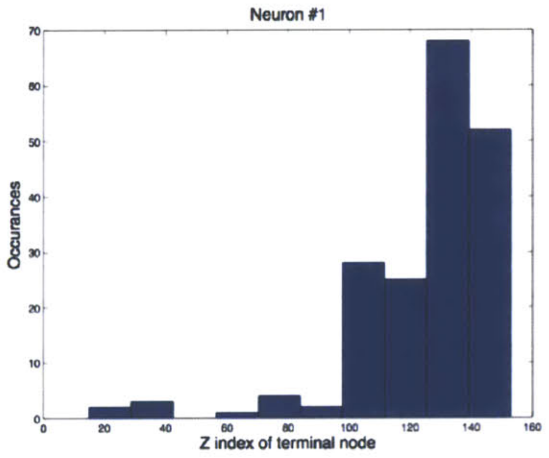
ID	Nodes	GC Nodes	Edges	GC Edges	Inputs	GC Inputs
1	1661	1411	1654	1410	180	172
2	1750	1682	1748	1681	52	52
3	1341	1341	1340	1340	97	97
4	2247	1289	2227	1288	115	88
5	669	669	668	668	71	71
6	1579	708	1565	707	101	74
7	893	893	892	892	90	90
8	1473	1473	1472	1472	57	57
9	1326	1326	1325	1325	97	97
10	1272	1063	1268	1062	71	68
11	1312	1312	1311	1311	141	141
12	893	893	892	892	70	70
13	1364	1364	1363	1363	171	171
14	2120	2120	2119	2119	113	113
mean	1421	1253	1417	1252	102	97
stdev	442	390	439	390	40	39
sum	19900	17544	19844	17530	1426	1361

Table 1: **Confocal Neuron Data Summary. Fields labeled "GC" refer to the computed giant component.**

did not needlessly throw away the true neuron data.

I enumerated all possible alignments consistent with this assumption, which required finding the (possibly negative) difference between the size of the image and the size of the skeleton, then zero-padding the image along every dimension for which that difference was negative, until the difference was positive and equal in magnitude to its original value. In terms of my assumption, that computation reflects the fact that, if the image were cropped to be smaller than the skeleton, it would not overhang the skeleton. The possible alignments to enumerate were then all combinations of x, y, and z offsets ranging from zero to the new difference between the image size and the skeleton size, corresponding to shifts from what in two dimensions would be (for instance) top-right-corner alignment. The alignment function then returned the set of offsets that caused the most nodes of the skeleton to align with points labeled as in the skeleton by users. Unfortunately, even after this step, the images were not fully aligned, so I chose to use only the skeletons themselves, and defer integrating the predicted neuron width for future development of the system.

Even the skeleton data, however, needed to be cleaned before I could use in to build Neuron objects in the data structures described below. In particular, the skeleton data entered the system as a list of node coordinates, and a list of pairs of nodes between which there was an (undirected) edge of the skeleton. However, my models (like standard HMMs) required the edges to be directed, I needed a systematic way to orient them. When I looked at the z-coordinate histograms of the skeleton nodes, I realized that for 14 of the 18 neurons, the z coordinates fell into two clusters separated by a region large relative to the size of the clusters in which there were no nodes. By visualizing the neurons, I could tell that this region with no nodes corresponded to a unbranched region of the axon, which meant that I could label every leaf node of the skeleton tree structure as an input or an output synapse depending on which z-coordinate cluster it fell into. Thus, all I needed to do was begin directing the edges away from the input synapses, and produce a conservative flow on the tree structure from the input synapses to the output synapses. While this method is certainly not universally generalizable, many neural areas of scientific interest are substantially laminar (the cortex and the retina, for example), so this simple thresholding method may work for most important situations. Furthermore, in electron microscope images, it is possible to visualize the neurotransmitter-release vesicles at the pre-synaptic terminal, so automated synapse labeling might be practical for those data sets.



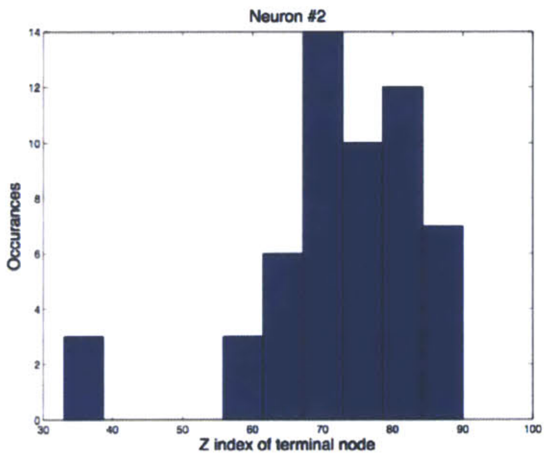


Figure 3: Terminal node z-coordinate histograms for two representative neurons, demonstrating the clustering exploited to assign input and output synapses.

Somewhat surprisingly, given that the neurons were skeletonized from a sparse fluorescent labeling, the skeletons did not all form single connected components. Thus, before I could design a method to determine that flow, however, I needed to choose which of the connected components in the neuron to keep. The obvious solution was to keep the largest component, but I rejected that idea because the input arbors in the neurons were much more branched than the output arbors (whether this reflected the choices of the microscopist and the skeletonizer or the actual neuronal anatomy is unclear), so it seemed possible that the largest component would not include any output synapses. Thus, I elected to keep the largest connected component that included at least one input and at least one output. As a practical matter, all these giant components (to adopt the term from network researchers) included exactly one output synapse, suggesting that the raw images did not in fact include the axonal arbors. This meant that I would not actually have any training data for some parameters of my HMM, it also meant that I could test the models on these neurons without using those same parameters, so I elected to stick with these giant components, but still develop the learning system to accommodate neurons with both dendritic and axonal arbors.

Since I was working with only the skeletons, and not the ill-aligned neuronal volumes predicted by the convolutional network model, I needed to develop a feature set based only on the skeleton's topology. Before I describe those features, I will describe in a bit more detail the skeleton data that constituted the input to my system. I mentioned that it consisted of a list of node coordinates and a list of pairs of nodes joined by edges, however it is important to note that, even once I'd found my input-output giant component, many nodes were connected to exactly two edges: in other words, may of the nodes were not branch points. Or, put differently, what we would think of as the branches of the skeleton's tree structure each consisted of many distinct edges in the input data. This structure arises from the fact that individual branches nonetheless need to curve in the three dimensional space in which the node coordinates sit. As mentioned above and discussed in detail below, (typical) HMMs require discrete time bins, or in the case of a morphological model discrete spatial bins along the skeleton, and it occurred to me that I could simply use the edges of the input skeleton as these discrete spatial bins. That approach would only make sense if the edges were similar in length, so I made a histogram of their lengths (after accounting for the slightly asymmetrical pixel size in the confocal data), and found that nearly all the lengths fell between 1 and 5 microns, suggesting that I could in fact use the existing edges as my spatial bins. It is important to note that the individual features may draw on aspects of the image at length scales smaller than the edge itself, though those features will be coarse-grained to the edges length scale for training (for instance, an organelle-detector in electron micrograph data might rely on nanometer-scale information, but the edge would store only the total number of organelles localized to a region of the cell 2-3 orders of magnitude larger.)

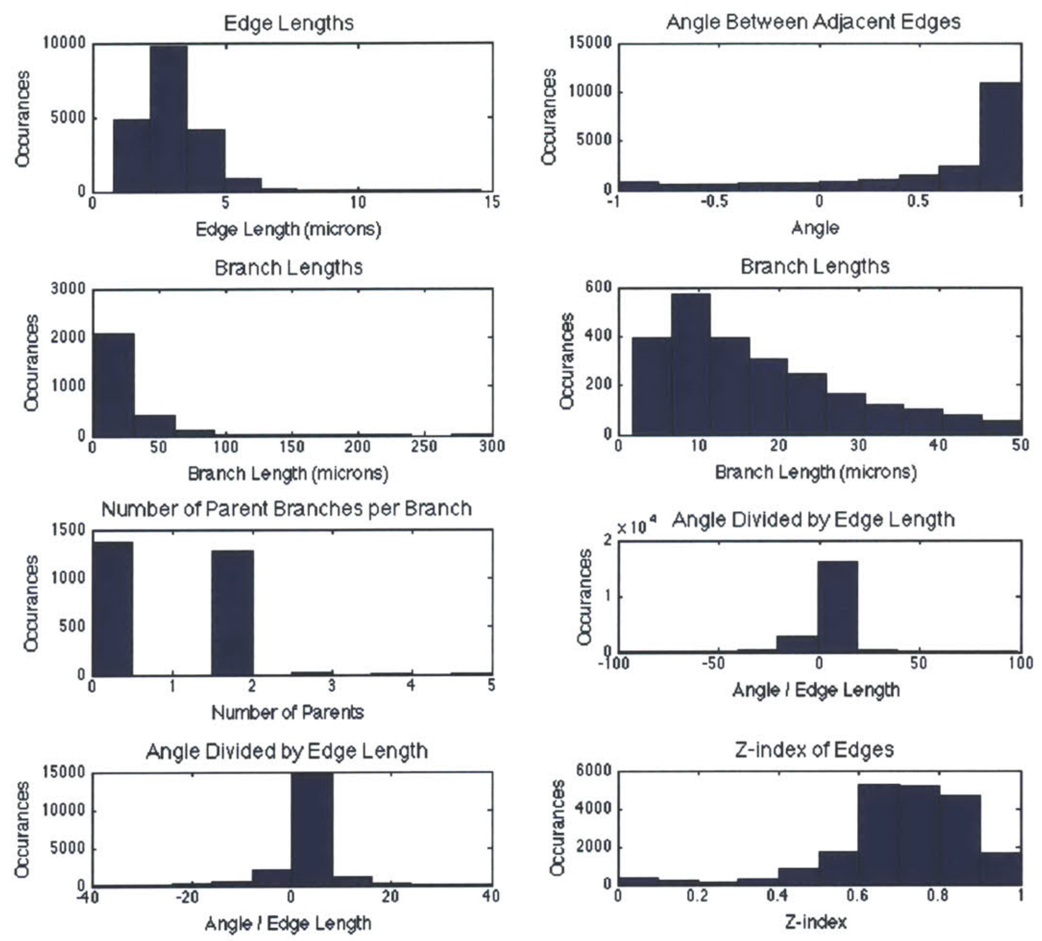


Figure 4: We designed a set of features to classify sections of skeletonized neurons based on the topology of the surrounding skeleton. The features capture both 3D spatial structure and abstract characteristics of the skeleton's tree structure.

Once I'd decided to use the input edges as bins, I needed to define features of the skeleton at or around each edge. While there are many methods for training HMMs on continuous output variables, I wanted to keep my models as simple as possible, and thus hoped to assign a set of binary variables to each edge. To do that, I simply made histograms of each continuous feature I included, then set one or more thresholds in that histogram by eye, looking (loosely) to balance maximizing the variance of the resulting features across the set of edges with matching the thresholds to apparent thresholds in the histograms themselves, with the idea that thresholds in the continuous data might in fact represent aspects of an underlying morphological model. Given only a skeleton structure, there are really two kinds of features available: those that capture the branching behavior of the skeleton around an edge, and those that describe the curvature of the skeleton around an edge. To capture the first class of features, I included the length of the branch containing the edge, as well as the number of parents of that branch in the input-output directed tree structure, in my feature set. (Since there are no axonal arbors in the giant components I considered, the number of children in this structure was always 1 for interior branches, and 0 for the axon, so that was not helpful as a feature.) To capture the curvature of the skeleton, I included the angle between adjacent edges (again accounting for the z-axis asymmetry in the confocal pixels), the edge length, and the ratio of those two quantities in my feature set. These features may in fact contain some information about the width of the neuron around a given edge, because a user tracing a neuron may have to make more fine adjustments the thinner the neuron is to stay inside it, thus leading to shorter edge lengths, and smaller angles. Finally, since I'd seen when investigating how to direct the edges that the neurons were consistently aligned along the z axis, I included the z coordinate relative to the tip of the dendritic arbor as a feature. As discussed above, including this relative coordinate is not completely unique to this particular data set, since many areas of the brain have lamina.

Deferring description of some of the details of the computation to the next section, the over-all data ingestion process consisted of loading each skeleton into MATLAB, finding the terminal nodes (those appearing only once in the edge-list of node-pairs), determining the z-cutoff separating input from output synapses for that neuron, finding the giant component with at least one input and one output based on those cutoffs. If using image data, the system then crops the skeleton coordinates to fit the giant component tightly, aligns the skeleton with the image by enumerating all possible alignments consistent with the assumptions described above. Once the image is aligned, it is then necessary to remove any nodes on a part of the skeleton that overhangs the edge of the image, and then to re-compute the input and output nodes and the corresponding giant component to reflect that removal. Finally, once that giant component is built, the system can construct the data structures that will be described in the next section, at which point the user visually determines feature cut-offs from the histograms of (continuous) feature values at each edge. The system then assigns binary features to each edge reflecting these cutoffs, completing the ingestion process.

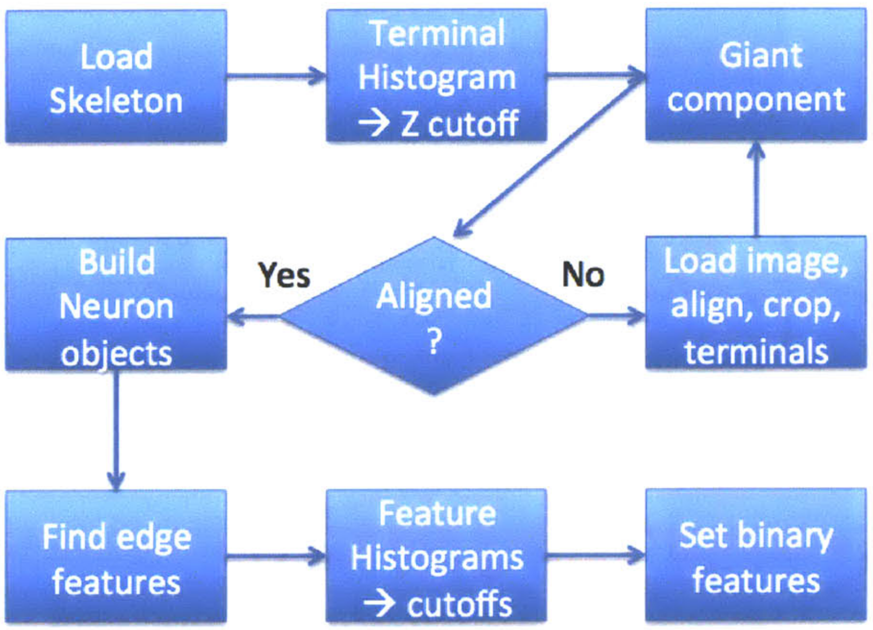


Figure 5: Data ingest flow chart.

2.1.2 Data Structures

I use a object-oriented MATLAB® system to store both the static data representing a neuron's topology and features, and the dynamic data such as the state-occupation probabilities current assigned to each edge and its maximum likelihood state assignment within the neuron. This system consists of three main classes: Neuron, Branch, and Edge. Minimally, a Neuron is a list of Branch objects, and a Branch is a list of Edge objects, plus pointers to parent and child Branch objects, if it has them. Finally, an Edge object must minimally store a binary feature vector and a vector of state occupation probabilities.

To facilitate training, evaluation, and data ingestion, these data structures implement functions beyond this minimal structure. Their fields and functions are:

Neuron fields:

- branches: a (1 x nBranches) list of Branch objects
- startIDs: a list of all branches with input synapses, by (second dimension) indices in self.branches

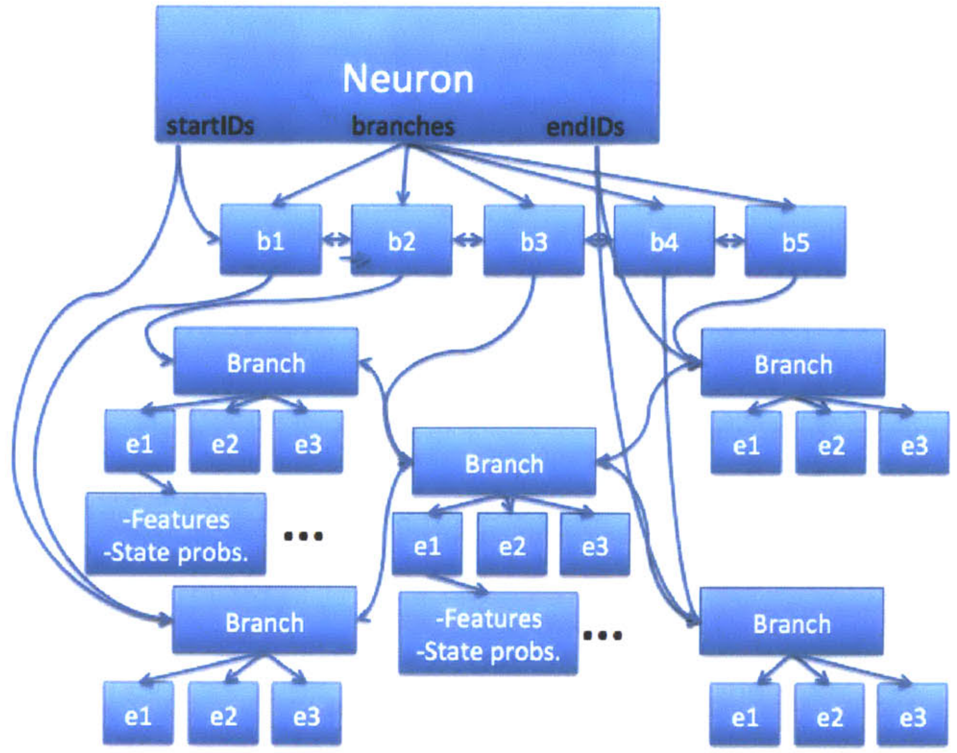


Figure 6: Core object hierarchy.

- endIDs: a list of all branches with output synapses, by (second dimension) indices in self.branches

Neuron methods:

- Neuron( ) : constructs a neuron object with empty branches, startIDs, and endIDs
- copy(self): returns a new Neuron object, newNeuron, whose startIDs and endIDs match this neuron's, and whose branches are each copied (in order according to self.branches) using Branch.copy( ). Runs newNeuron.setParentsChildren( ) before returning.
- setParentsChildren(self): updates the parents and children fields of each branch in self.branches to reflect the current assignments of the branches' startIDs and endIDs fields.
- clearUpdated(self): sets updated to false for every branch in self.branches
- checkUpdated(self): returns the fraction of branches in self.branches whose updated fields are set to true

- getModeFeatures(self, branchInds, start): returns the mode terminal-edge feature vector over the branches with indices branchInds in self.branches; can be used at 1-many, many-1, or many-many branch points in training. If start =1, the terminal edge is the first in each branch, else, it is the last.
- getAveStates(self, branchInds, start): returns the average terminal-edge feature vector over the branches with indices branchInds in self.branches; can be used at 1-many, many-1, or many-many branch points in training. If start =1, the terminal edge is the first in each branch, else, it is the last.

Branch fields:

- id: the index of this branch in it's neuron's branches list
- edges: (1 x nEdges) list of Edge objects in this branch
- parents: (1 x n) list of parent branch IDs; if fewer than 2 parents, pad with -1 until size is (1 x 2)
- children: (1 x n) list of child branch IDs; if fewer than 2 parents, pad with -1 until size is (1 x 2)
- updated: binary variable; set by forwardStep methods and used in the core loop described in this section
- length: length of the branch in 3D space; the sum of the 3D space lengths of its edges
- getLengths(self): returns a list of the length fields of the Edge objects in self.edges
- getAngles(self): returns a list of the angle fields of the Edge objects in self.edges
- setConnectionAngles(self): sets the angles fields of Edge objects in self.edges, using the direction fields of those Edge objects
- setEdgeFeatures(self, thresholds): sets the binary feature vectors of each Edge in self.edges, according to thresholds and the continuous values of the features already stored in the Edge objects

Branch methods:

- Branch( ): initializes a Branch object with an empty list of edges
- copy(self , reverse): returns a new Branch object whose edges are each copied (in order according to to self.edges) using Edge.copy(). Sets the startIDs and endIDs fields of the new edges before returning. If reverse = 1, returns a reversed version of this branch, with startIDs and endIDs in the edges set appropriately. Sets inID and outID appropriately, but does not set parents and children.
- endIDs: a list of all branches with output synapses, by (second dimension) indices in self.branches

Edge fields:

- length: length of the edge in 3D space
- direction: 3D unit vector defining the direction of the edge
- angle: angle between this edge and next in its branch
- features: (1xnFeatures) binary vector of feature values
- states: (1xnStates) vector of state probabilities, normalized
- statesDetect: (1 x n) vector of state likelihoods, not normalized, can be used for classification
- zed: the z-index of edge relative to the top of its dendritic arbor
- startCoords: pixel coordinates of the start node
- mlState: the index of the Viterbi maximum likelihood state of this edge

Edge methods:

- Edge(length, direction, zed): initialize an Edge object with the given continuous feature values
- copy(self): returns a new Edge object with the same length, direction, zed, features, and states as this one

In addition to the methods enumerated in this section, the Neuron, Branch, and Edge objects of course provide methods to get and set all their fields.

```
initialize queues B, Q;
Q <-- input terminal nodes;
while not Q.empty {
  q <-- Q.next;
  if q.parents.ready {
    Do MergingBranchAction(q.branch);
    Do MergingJunctionAction(q.branch.end,
                   q.branch.children);
    q.ready = true;
    Q <-- q.children;
    if not B.contains(q.children) {
      B <-- q.children;
    }
  else {
  Q <-- q;
  }
}
Q <-- B;
while not Q.empty {
  q <-- Q.next;
  Do SplittingBranchAction(q.branch);
  Do SplittingJunctionAction(q.branch.end,
                   q.branch.children);
  Q <-- q.children;
}
```

```
initialize queues B, Q;
Q <-- output terminal nodes;
while not Q.empty {
  q <-- Q.next;
  if q.children.ready {
    Do SplittingBranchAction(q.branch);
    Do SplittingJunctionAction(q.branch.start,
                   q.branch.parents)
    q.ready = true;
    Q <-- q.parents;
    if not B.contains(q.parents) {
      B <-- q.parents;
    }
  } else {
    Q <-- q;
}
Q <-- B;
while not Q.empty {
  q <-- Q.next;
  Do MergingBranchAction(q.branch);
  Do MergingJunctionAction(q.branch.start,
                   q.branch.parents)
  Q <-- q.parents();
}
```

Figure 7: These function skeletons at right and left are at the core of the forward and backward information flow (respectively) that is at the core of the the functions that Neuron objects, train the expectation maximization matrices, and set the maximum likelihood states.

2.1.3 BQ Routine

A significant portion of my effort during the phase of the project concerned how to efficiently building these data structures from the skeleton topology as it was stored on disk: namely, as a list of pairs of nodes connected by edges. The challenge, as described above, was to orient each edge so that they could be construed as describing a flow through the skeleton graph from input synapses to output synapses (given only positive flow rates). In fact, the algorithm we employed to direct the edges turned out to be at the core of the expectation maximization EM learning routine, as well as (reversed) at the core of the maximum likelihood (ML) state assignment routine. We call this core loop the BQ routine.

The forward version of BQ (left panel in figure 7) proceeds by initializing two queues, Q and B, and pushing all the input synapse nodes onto Q. The idea behind this routine is that, once every edge but one has been directed into a node, the remaining node must be directed out of the node. Using the parent and child pointers supplied by the Branch objects (once they have been constructed) or the raw node-pair edge list when building a neuron, the routine checks whether all the nodes but one entering a junction have been 'updated'. If they have, puts the outgoing edge on Q. If the node popped off of Q has only one non-updated edge, the algorithm expands directs that edge away from the node and adds its children to Q. Otherwise, the routine puts the node it popped back on Q, and also puts it on B if it is not already there. The key observation is that, since the neuron is an unrooted tree and is thus acyclic (even in its undirected form), Q becomes empty exactly when we get to the cell body, which is the point where branches stop merging and begin splitting. Thus, when we run out of entries in Q, we simply take whatever entries are in B and place them on Q, then repeat our previous routine (except using different update code for splits in the learning routines). If the cell body has only one axon extending from it, then Q will in fact never be empty and we will not need to swap B and Q. However, if the cell body has multiple outgoing edges, they will be exactly the edges on B when Q becomes empty.

The backwards version of this routine will be discussed briefly in the section on maximum likelihood decoding below, and is a simple generalization of the forward routine.

2.2 Morphological Models

In this section I describe the probabilistic model that is the core of this thesis.

2.2.1 Fixed-Topology Hidden Markov Model

As described above, this system represents neurons as a list of branches, which form an unrooted tree whose degree might vary across nodes. Like a traditional HMM, the model presented here is trained on a set of binary feature variables, and learns a single emission matrix E that specifies the probabilities of observing an affirmative value for each variable from each state: specifically, $E_{(i,j)}$ is the probability that the system will emit a 1 at the $i-th$ position in the feature vector if it is in the state at index j in the state probability vector. Furthermore, as in a traditional HMM, we provide a transition matrix M that describes the changes of state between adjacent edges within a single branch: $M_{(i,j)}$ is the probability that an edge purely at state index i will be followed by an edge in state index j. The rows of M are normalized; the columns are not. To build intuition, note that if our training algorithm works well and our spatial bins are small enough, we expect there to be some indexing of the states in M such that diagonal entries to dominate each row, meaning that self-self transitions are more likely than self-other transitions.

To these traditional HMM matrices, we add three more arrays: a row-vector called *inPrior* that acts as a prior-probability vector for all input nodes, a 3-dimensional array S that specifies the behavior at nodes where exactly one edge is directed into the node and two or more edges are directed out of it, and a 3-dimensional array J that specifies the behavior at all other branch points, which are those in which two or more edges are directed into a given node. $S_{(i,j,k)}$ specifies the probability that the branches leaving a node whose incoming branch terminated in an edge in state i will begin with edges having states j and k. Similarly, $J_{(i,j,k)}$ specifies the probability that a node whose incoming edges have terminal edges in states i and j will have an outgoing branch that begins with an edge in state k.

This structure reflects many design decisions. We added *inPrior* to the model as a way to coordinate training across the various input branches, an issue that does not arise on traditional linear HMM structures. In addition, *inPrior* gives us an easy way to seed the training to favor certain states near the input nodes. As we describe below, our training routine executes a forward pass before it executes its first model update, so seeding the state probabilities themselves would not affect subsequent rounds of training. We could seed the training by crafting one or more columns of the initial E matrix to match characteristics of the input node features, but *inPrior* seemed a simpler and more direct way to achieve that goal. Finally, note that the need for an input prior distribution to initialize the forward pass is not equivalent to including *inPrior* in our model, since without *inPrior* we would not learn that prior distribution during training.

In specifying the behavior at branch points, we face three main decisions: whether to enforce symmetry with respect to edge ordering, how to handle one-to-many and many-to-one junctions, and how to handle many-to-many junctions. The first decision arises because, taking the J example without losing generality, when we choose a vector along the third dimension of J that will serve as the state prior for the outgoing edge of a join junction, we must decide whether to take the (i, j) or the (j, i) position in the first dimension, assuming that we are coming from two distinct states i and j. In our system, we simply constrain these two row vectors to be identical so that the ordering does not matter, by always updating J according to both orderings at every join node each time we update our model. We similarly constrain S to be symmetric along the diagonal of its second two dimensions. In principle, we throw away information by constraining J and S to be symmetric, since the way our system orders branches in during training might happen to align with some structural characteristic in the morphology. However, it seemed unlikely that such a pattern would carry over to test data, so we believe that the symmetry constraint will act to counteract a source of over-fitting.

The second decision, how to handle one-to-many and many-to-one split and join junctions, depends more strongly on the nature of the skeletons used in the model. In particular, it depends on the prevalence of such non-ternary junctions. Given enough training data and enough many-branch junctions, it might make sense to provide specific matrices for each type of junction (1-3, 3-1, 1-4, etc.), but given the relative scarcity of such junctions in our training data, we suspected that such matrices would end up over-fitting a few examples. Thus, in one-to-many splits, we decided to make the forward pass to each outgoing branch as though there were a single other outgoing branch, whose state vector is the average of all the other branchs state vectors. When updating S, we treat these junctions as a split where both the outgoing edges have average state vectors across all outgoing edges. Similarly, when making the forward pass at a many-to-one junction, we treat that junction like a simple ternary join node in which both incoming edges are averages over all the incoming edges, and when we update J based on the transition from one of those incoming edges, we treat all the other incoming edges as a single, average edge.

The many-to-many case is special because it can occur at most once in a valid tree-structured neuron. A specific many-many transition matrix could thus specifically learn state transitions that occur at the cell body, which might improve the final model. However, since they occur at most once per neuron, many-to-many junctions may be rare even in large training sets. For this reason, we decided not to provide an explicit many-to-many transition matrix. Since we expected cell bodies to receive many more incoming branches than outgoing branches, we decided that many-to-many nodes would have their behavior specified by J. As in the case of a many-to-one join, we treat the incoming edges as two average edges when making the forward-pass update to the state probabilities of the first edges of branches that leave the node.

2.2.2 Proposed Joint Feature-Topology Model

2.2.3 Expectation MaximizationTraining Routine

We train our HMM through an expectation maximization (EM) method very similar to the Baum-Welch method for traditional HMMs on linear transition topologies. As in Baum-Welch, our model first makes a forward pass through the neuron that updates the state probability vector on each edge, then updates the model to reflect the maximum likelihood model given those state assignments. This process repeats until convergence. In the update step, we update all model matrices: E,M, J, S, and $inPrior$. To update M, we set $M_{t+1} = 0_{(nStates,nStates)}$, then loop over branches, doing:

$$M_{t+1} = M_{t+1} + \sum_{i=2}^{numEdges-1} \sum_{j=1}^{nStates} \sum_{k=1}^{nStates} (edges(1,i).getStates()(1,j)) \cdot (edges(1,i+1).getStates()(1,k))$$

Similarly, we do J, we set $J_{t+1} = 0_{(nStates,nStates)}$, then loop over branches before the cell body, doing:

$$J_{t+1} = J_{t+1} + \sum_{j=1}^{nStates} \sum_{k=1}^{nStates} \sum_{l=1}^{nStates} (edges(1,end).getStates()(1,j)) \cdot (OutBrch.edges(1,1).getStates()(1,k)) \cdot (InBrch.edges(1,end).getStates()(1,l)$$

For the update to S, we again do$S_{t+1} = 0_{(nStates,nStates)}$, then:

$$S_{t+1} = S_{t+1} + \sum_{j=1}^{nStates} \sum_{k=1}^{nStates} \sum_{l=1}^{nStates} (edges(1,end).getStates()(1,j)) \cdot (OutBrch1.edges(1,1).getStates()(1,k)) \cdot (OutBrch2.edges(1,end).getStates()(1,l)$$

The update to E is slightly more complicated, because we must keep track of the total probability mass assigned to each state for normalization purposes. Specifically, we set $E = 0_{nFeatures,nStates}$ as usual, but also initialize a row vector $nEdges = 0_{1,nStates}$, then iterate over all the edges, doing:

$$E_{(i,j)} = E_{(i,j)} + edge.get_features(i) \cdot edge.get_states()(j) \tag{2.1}$$

$$nStates = nStates + edge.get_states() \tag{2.2}$$

We now "normalize" E by dividing by the $nEdges$ vector that we have calculated.

In the forward pass of each training step, we must ensure that all branches upstream of a junction are updated before we update the J or S matrix according to that junction, and subsequently the branch after it. To facilitate this, we simply implement the BQ routine described above, but set the within-branch method to updateM and the end-of-branch methods to updateInPrior, updateJ, and updateS, as appropriate. To re-state the logic of that routine in terms of the present task, we set the updated field of each branch to 0 before each iteration of updateModel, then set it to 1 once we update the model based on that branch. We only proceed through a junction node when all its incoming edges are updated. We can obtain a list of all incoming edges from the parents field of any of the child branches. When we finish a non-terminal branch, we place all its children on the queue. Until we have finished updating the entire neuron, we pop the first element from the queue, discard it if it is already updated, place it on the back of the queue if both it and one of its parents are not updated, and update it otherwise. We initialize the queue with the list of input synapse branches at each call to updateModel. This list is a field of the neuron object.

Like Baum-Welch, this training method climbs to a local optimum and stays there. There are various ways to avoid getting stuck in globally sub-optimal maxima in these routines, including introducing noise in the updates or state assignments directly in a simulated annealing paradigm, or introducing it more obliquely by making piecewise updates to the model. We have implemented two algorithms, one that updates the state probabilities across all the neurons given a version of the HMM, then updates the HMM based on all those assignments, and a second that updates the states of a single neuron, then updates the HMM based on all the assignments (only one of which has changed). This is similar, but not equivalent, to implementing a training rate of $1/(numberofneurons)$. The more dramatic version of the piecewise updates only one feature of E at each iteration.

2.2.4 Implementing Functions

The training routine is run out of a function called BaumWelchWrap, that takes as arguments a row-vector of Neuron objects, the number of iterations of the routine to run, whether to make batch or online model updates (see below), whether to compute measures of convergence, and whether to use a uniform initial inPrior, an inPrior concentrated on some number of states, or no inPrior at all. In a batch update, the forward pass is made over all the neurons, then the model is updated based on all the neurons, while in the online (non-batch) case, the forward pass is executed over a single neuron, then the model is updated based on all the neurons, effectively implementing a 1/nNeurons training rate, which may improve the routine's ability to escape local optima.

BaumWelchWrap initializes the transition and emission matrices with uniform random numbers, normalizes appropriately, then calls BaumWelch, which executes the loop over iterations, alternatively calling forwardPass(), updateModel(), and, if it is set to produce verbose output, convergence(). Both forwardPass and updateModel call distinct methods to update the different transition matrices: forwardStep (for M), forwardStepJ, forwardStepS, and forwardStepInput for the forward pass, and updateE, updateM, updateJ, updateS, and updateInPrior for the model update.

2.3 Model Evaluation

To evaluate the performance of the learning routine described above, I needed to first develop a set of metrics to measure convergence during training. That is, since the training routine proceeds to a local optimum when it works properly, the first task was to develop a way of monitoring its progress towards that maximum. The more meaningful evaluation metric, however, is the training routine's ability to learn a model that closely matches the true underlying model of the data, and to correctly identify the underlying states. To check this match, I had to first develop a system to generate a population of neurons whose states and features are generated from a known model. Since HMMs are by construction generative models, this was relatively simple, though, as usual, it required an implementation of the BQ routine. Next, I had to choose and implement a method to determine the optimal mapping from learned states to original states. I decided that the most meaningful definition of the optimal mapping was the one that caused the most states to be correctly mapped after a round of maximum likelihood estimation using the learned states and transition matrices, so I implemented such a method by combining the classic Viterbi back-chaining method with the backward version of the BQ routine. Finally, I designed a set of tests to measure the end-state quality of a converged, optimally-state-matched learned model, in terms of its similarity to the underlying model used to fit the data, and the requisite methods to implement those tests. [4]

2.3.1 Online Convergence Testing

The simplest way to determine whether the training routines are working properly on a given data set is to observe whether or not they converge. To measure convergence, I consider each of the model matrices separately, in each case summing over indices, and finding the sum of the absolute values of the differences between the entry at a given index at the current round, and that value before the most recent update. I then convert divide that sum by the average value in the matrix before the update, and divide again by the number of entries in the matrix, to produce the final convergence metric for each matrix at each time step, which is the average magnitude change at a given position in the matrix.

2.3.2 Maximum Likelihood State Estimation

To produce maximum likelihood state estimates on a branched topology, I combined the classic Viterbi algorithm with the backward version of the BQ routine described above. Specifically, as in the Viterbi algorithm, I begin at the terminal nodes and set their maximum likelihood state to be the one assigned the highest probability by the most recent forwardPass. I then step back through the branch, multiplying each index of the state vector at each position by the transpose of the column of M indexed by the maximum likelihood assignment of the next state along the branch (that is, the one the maximum likelihood state we just set). The maximum of this multiplied state vector is then the maximum likelihood state assignment for the new branch. Note that the feature vectors do not directly enter this calculation, just as they do not enter the Viterbi backward pass algorithm. In the reverse of the update routines described so far, before it gets to the cell body (that is, when the neuron is splitting) this routine must wait for the other child of its parent to be updated before the parent can be updated. After the cell body, the parent can be updated as soon as the child is, since the neuron only merges on that side of the tree.

2.3.3 Optimal State Matching

Finding the optimal mapping from states in the original model used to generate the data to states in the learned model is a prerequisite to measuring the similarity between the emission and transition matrices, and the state assignments, produced by those two models. To determine the optimal mapping, I first build an array containing the maximum likelihood state assignment indices for all the edges in the neuron (or neurons) according to the two models, with the edges in the same order in both of the linearized topologies. Next, I enumerate all the possible maps between states, and check how well the two state assignment arrays match under that reassignment (measured by the total number of indices in the array where the maximum likelihood states match under the mapping). While this algorithm is $O(n!)$ and thus nominally highly inefficient, it is in fact practical to run because the number of states is small relative to the number of edges. As a practical matter, it runs in a few seconds on one of the neurons in this data set. Finally, I return the matching that produced the optimal alignment of the two edge-ML-state arrays. This is implemented in the function evaluate.m

2.3.4 End-point convergence tests

I test end-point convergence using the exact same convergence.m routine used to measure online convergence, except that the underlying model used to generate the data stands in for the model from before the most recent update. Thus, as before, I measure the average magnitude change at a given position in the matrix for each of E, M, S, J, and inPrior.

2.4 Error detection

The initial inspiration for this project came from the idea that split and merge errors in automated segmentations would manifest themselves as regions of low-likelihood in a probabilistic model of the neuron morphology. Thus, though implementing a full error detection routine was beyond the scope of this thesis, I did implement the core functions of such a routine. The basic structure of the error detection routine is identical to that of the forward pass routine, with a the wrapper method errorDetect.m calling forwardPassDetect.m, which in turn calls forwardStepDetect, forwardStepJoinDetect, forwardStepSplitDetect, and forwardStepInputDetect. These methods in fact make the usual forward step update, and they could in theory be used interchangeably with the standard forward pass functions. However, they also save a value in the statesDetect field of each edge equal to a constant factor times the probability of observing the features at this edge given the states at the previous edge, for each possible state at this edge (note that the factor is constant across all edges; it is used to lessen the chance of numerical underflow occurring before normalization). This row vector of likelihoods is then normalized to produce the usual state probability vector produced in the forward pass. The probability of an error is then taken to be proportional up to an offset to some large constant minus the max value of this statesDetect vector, for each edge. Note that in assuming that the likelihood given the previous edge's *probabilities* (rather than their likelihoods) determines its chance of being the site of a segmentation error, I prevent that probability from propagating down the neuron. Without testing on a real data set of mistakes made by a given automated testing routine, it is difficult to tell whether that is or is not a good design decision.

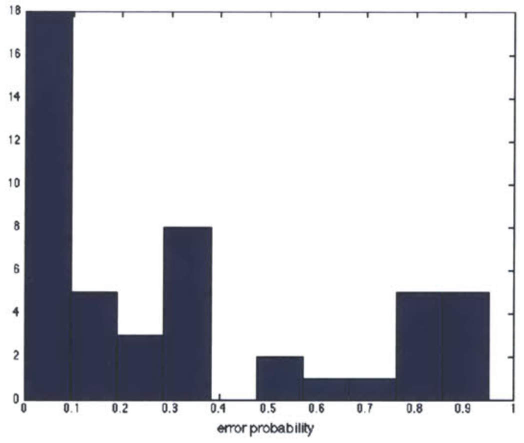


Figure 8: Error probability distribution for a neuron with simulated features.

The error probability histogram of a simulated neuron suggests that a segmentation error would need to generate an error probability roughly above .9 to be detectable. Given that a somewhat high false positive rate is likely acceptable for an error detector in this situation, the threshold could in fact likely be moved lower. The ability to generate this histogram is the core feature of an error detector based on this system, so implementing a functioning edge detector would likely be achievable.

3 Results

In this section, I report my preliminary characterizations of the models and training routines presented here. I characterize them first by evaluating the performance of the model on a variety of underlying models of varying difficulty. Next, I show that the model can converge to an optimum when trained on either single neurons of populations of neurons from the confocal micrograph skeletons described above. Finally, I show the output the output of the error detection code on a valid neuron, to demonstrate how it could be used to identify segmentation mistakes.

3.1 Simulated Features

To test the learned model against a given generating model, I wrote a method to build such generating models given one noise parameter that affects how difficult the model is to learn. To build this model, I begin by setting:

$$E = \begin{matrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{matrix} \tag{3.1}$$

Given 8 states and 10 features, this matrix uniquely satisfies the property that the matrix is full rank, which is to say, no feature is simply a linear combination of other features. That linear independence seemed like an important property for a model used to test the convergence properties of real models which are almost surely full-rank. Next, we specify the simplest ergodic within branch transition matrix, which is just a chain of states, each with a high self-transition probability and a small probability of moving to the next state in the chain. We introduce the one caveat that certain transitions in the chain are placed in the join matrix J, rather than in M. Specifically, we start with:

$$M = \begin{matrix} 0.9 & 0.1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.9 & 0.1 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.9 & 0.1 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{matrix} \tag{3.2}$$

Then, J is initialized with ones at positions$(2, 2, 3)$, $(4, 4, 5)$, and $(6, 6, 7)$, and is otherwise simply set to the average M values of the two incoming states. Since the no splits occur in the model topologies we use in testing (that is, in the topologies in our confocal data set), we simply initialize S with uniform random numbers for now. Finally, we add gaussian noise to every matrix entry, with the variance of the noise an tunable parameter of the model for each matrix, then constrain every entry to be between .001 and .999, then normalize.

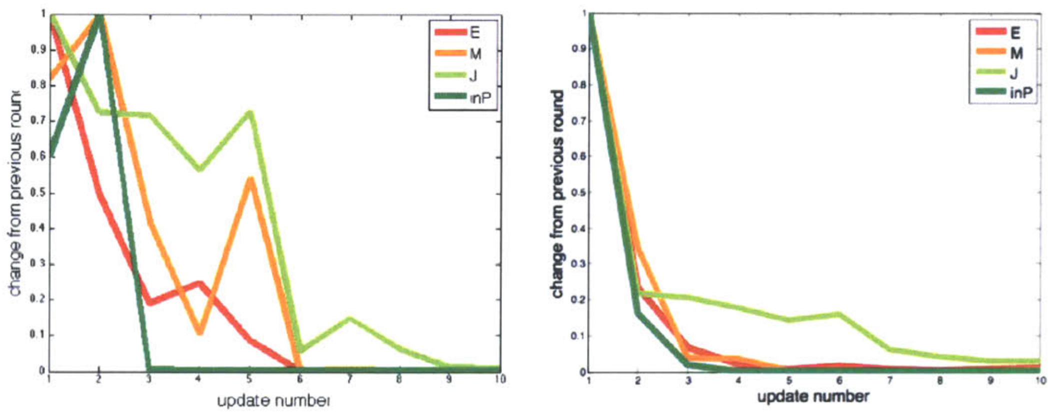


Figure 9: Convergence of the model trained on a single neuron (left) and 14 neurons (right).

3.1.1 Convergence and Response to Noise

The first tests we ran using the simulated data on real neurons sought to determine the characteristic convergence pattern of the training routine with different numbers of neurons in the training set. We first generated the model with noise standard deviation of .1 on all matrices, then generated features on a single neuron's topology (neuron 1) according to that model.

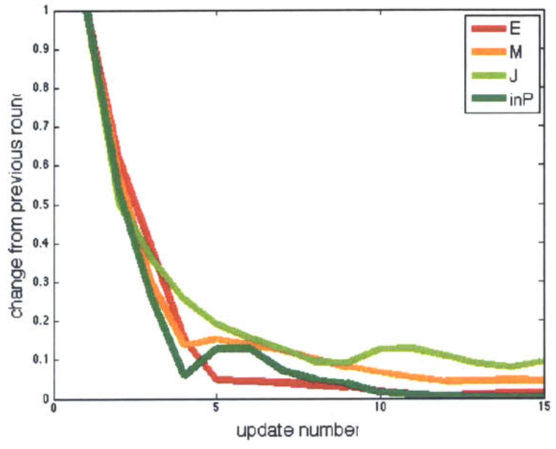


Figure 10: The model converges when trained on topological features.

Next, with a different randomly generated model with the same noise level, we generated features on the topologies of all 14 neurons for which we were able to distinguish input and output synapses. We trained both models for 10 iterations of the training routine, training inPrior, and initializing it as a uniform distribution over the eight states. For the 14-neuron trial, we used a batch update. The convergence is smoother in the 14-neuron trial, which is not surprising since each update is based on far more data. Measured in updates, the convergence is also quicker in the 14-neuron case, though in terms of computational time it is slower, since the update-units speed gain was only around a factor of 2, and each update requires an order of magnitude mode computation.

We also tested the convergence properties of the model when we trained it on the topological feature set developed in this project. The convergence was smoother than observed with one neuron in the simulated model, perhaps because there were more features in the topological set. However, we consider the speed and smoothness of this convergence to indicate that the combination of our learning routines and our feature set are in fact capturing a true pattern in the neurons. We

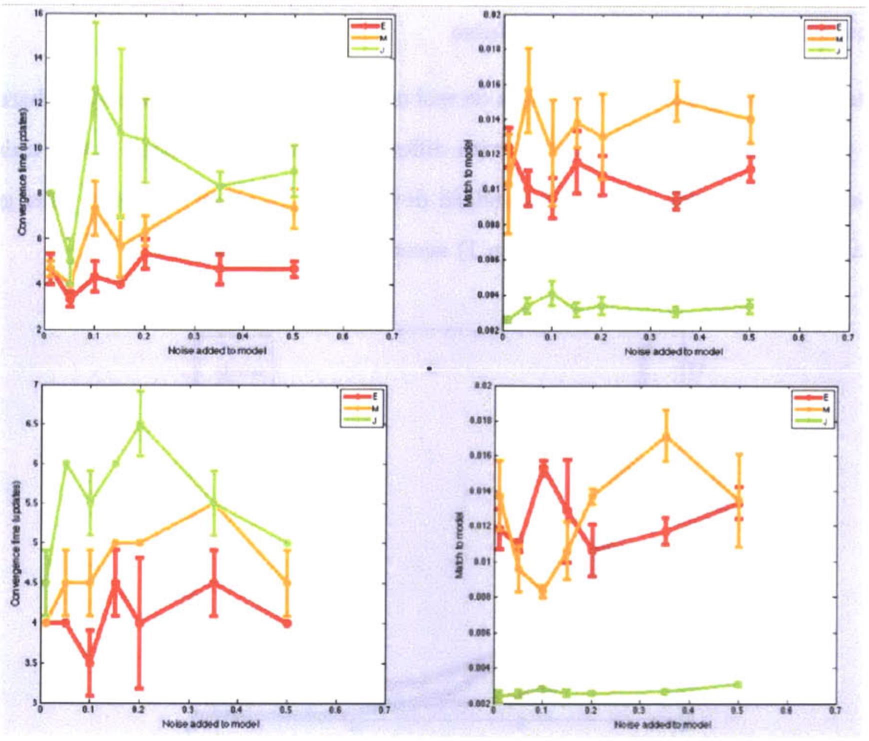


Figure 11: Model convergence time (left) and performance relative to ground truth as E (top) and M (bottom) become noisy.

next wondered whether the performance of the model would in fact suffer as we tuned the noise parameters in our randomly generated testing model. Within the range of noise conditions tested, the performance did not appear to significantly suffer as noise increased, and the absolute match between the underlying model matrices and the learned matrices was roughly 1% as we tuned the noise on both E and M. Error bars are standard error of the mean, $n = 2$.

4 Conclusions and Further Work

[?] In this project, I set out to design a system capable of learning a generative probabilistic model of neuron morphology that could subsequently be applied to detect errors in automated segmentations of electron micrographs of brain tissue. While I was unable to apply the system to electron microscope data or implement a full error detector, I have developed a flexible system able to learn simulated models accurately on real neuron topologies. Furthermore, I have developed a feature set on which that model converges with performance similar to that on simulated data for which it learns the correct ground truth model.

Possible future plans for this work include implementing an explicit training rate in updateModel; implementing a hierarchical model that simultaneously learns and classifies morphological neural cell types; using the model to simulate network given cell body positions, and comparing result to observed topology; implementing one-class SVM on the model parameters as a means of error detection; implementing error correction based on a functioning error detector; applying the model to larger confocal databases and EM databases from Seung lab; implementing new features that use image data explicitly; implementing input-output synapse detection, enforcing consistency across cells; and finally improving the performance of the system by moving computationally intensive operations to mex files, and parallelize logically parallel branch updates.

References

[1] A.P. Alivisatos et al. *Neuron*, 74:970–974, 2012.

[2] S.C. Turga et al. *Neural Computation*, 2010.

[3] V. Jain et al. *IEEE CVPR*, pages 2488–2495, 2010.

[4] L.R.Rabiner. *Proceedings of the IEEE*, 77:257–286, 1989.

[5] H.S. Seung. *Connectome: How the Brain's Wiring Makes Us Who We Are*. Houghton Mifflin Harcourt, 2012.