TreeMoCo: Contrastive Neuron Morphology Representation Learning

Hanbo Chen[*,†]
Tencent AI Lab
hanbochen@tencent.com

Jiawei Yang[*,†]
University of California, Los Angeles
jiawei118@ucla.edu

Daniel Maxim Iascone
University of Pennsylvania, Philadelphia
daniel.iascone@pennmedicine.upenn.edu

Lijuan Liu
Southeast University, Nanjing
juan-liu@seu.edu.cn

Lei He
University of California, Los Angeles
lhe@ee.ucla.edu

Hanchuan Peng
Southeast University, Nanjing
h@braintell.org

Jianhua Yao[‡]
Tencent AI Lab
jianhuayao@tencent.com

Abstract

Morphology of neuron trees is a key indicator to delineate neuronal cell-types, analyze brain development process, and evaluate pathological changes in neurological diseases. Traditional analysis mostly relies on heuristic features and visual inspections. A quantitative, informative, and comprehensive representation of neuron morphology is largely absent but desired. To fill this gap, in this work, we adopt a Tree-LSTM network to encode neuron morphology and introduce a self-supervised learning framework named TreeMoCo to learn features without the need for labels. We test TreeMoCo on 2403 high-quality 3D neuron reconstructions of mouse brains from three different public resources. Our results show that TreeMoCo is effective in both classifying major brain cell-types and identifying sub-types. To our best knowledge, TreeMoCo is the very first to explore learning the representation of neuron tree morphology with contrastive learning. It has a great potential to shed new light on quantitative neuron morphology analysis. Code is available at https://github.com/TencentAILabHealthcare/NeuronRepresentation.

1 Introduction

The central nervous system is composed of hundreds to thousands of neuronal subtypes. Subtype is the critical determinant of neuronal function within local and long-range circuits and is identified by a combination of gene expression patterns, dendrite/axon morphology, synaptic connectivity, and electrophysiological properties. In particular, dendritic morphology is a central node of neuronal subtype identity, as dendritic branches receive the vast majority of the presynaptic input onto each neuron and directly shape the electrophysiological properties of the neuron that governs its firing output. Delineating neuron cell-types based on their dendritic morphology is one of the core problems of neuroscience for the past century [55].

[*]: Equal contribution. [†]: Work done when authors were at Tencent AI Lab. [‡]: Corresponding author.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

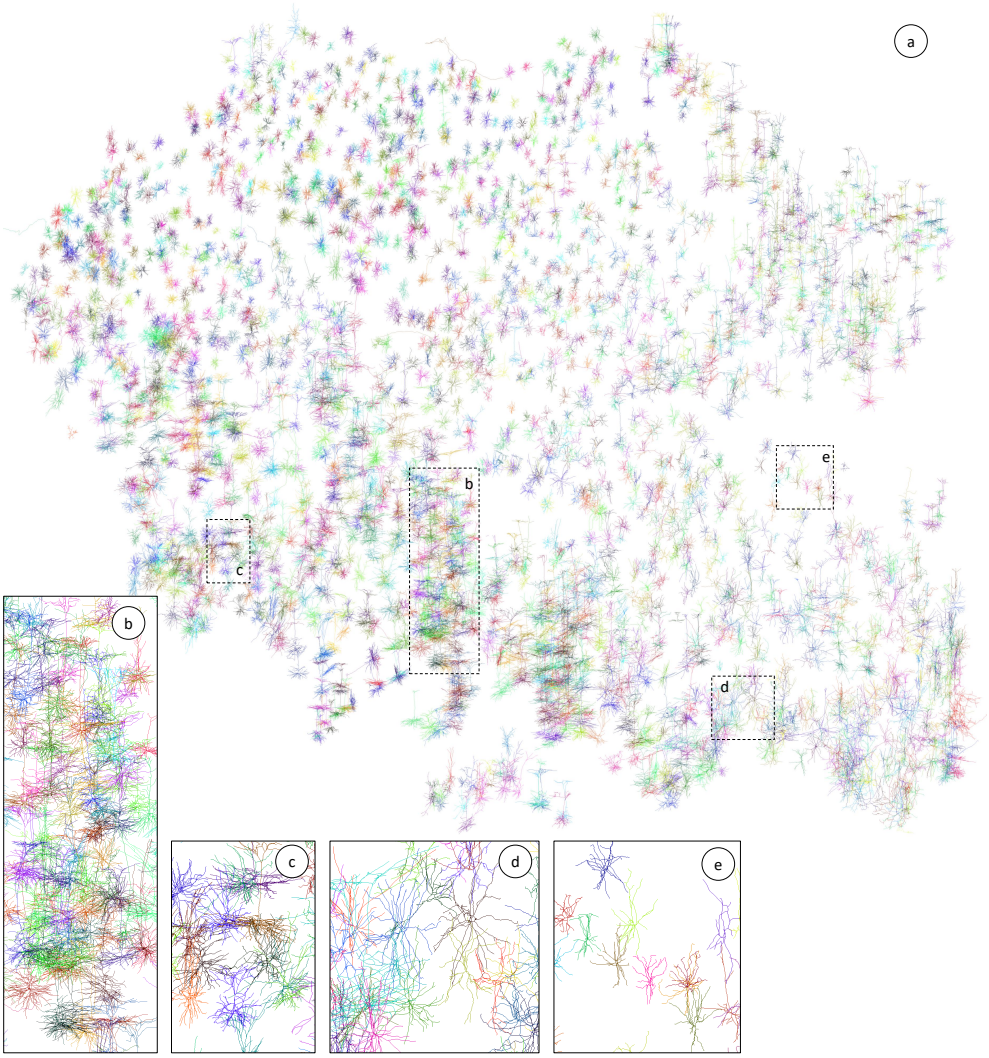


Figure 1: Distribution of neurons in the embedding space of TreeMoCo visualized by t-SNE [42]. Thumbnails of dendrites are randomly colored. (b-e) Zoom-in view of regions in (a) showing clustering of isocortex layer 5 cells (b), layer 2/3 cells (c), basket cells (d), and Martinotti cells (e).

Neuron dendritic morphology can be digitally represented by a 3D neuron reconstruction (Fig. 2(a)), a tree graph with 3D coordinates of each node; the root is placed at soma. With recent advances in imaging [10, 12, 51] and computational methods [21, 30, 46], 3D reconstruction of whole neuron trees with high-throughput is now accessible for mouse brain [31, 48]. However, due to the lack of effective measurement, most analyses of these 3D reconstructions are still largely based on biologists' visual inspection. Considering the enormous number of neurons in the brain and the 3D formation of neuron morphology, such visual inspection is subjective, biased, and has low throughput.

The past decade has witnessed the success of deep learning in learning representations of almost all kinds of data such as image, language, sound, and graph [2]. However, as a special type of graph or graphic data, most of the quantitative analysis of neuron trees is still based on simple heuristic features [36]. Only a few attempts have been made in learning deep representations of neuron morphology. In [57], the authors proposed to encode neuron morphology with tree recurrent neural network (TRNN). However, this method relies on manual annotations, limiting its application to supervised training only. In [24], the authors proposed a generative self-supervised learning (SSL) method for neuron morphology representation learning, named MorphVAE, to reconstruct original neuron trees via a variational autoencoder and random walk sampling. Since this method

cannot encode tree topology structure, its classification performance is limited and the method still relies on label information to generate reasonable predictions. In [35], the authors adopt multi-view based SSL to learn representations of neuron morphology. Though the work mainly focuses on the local components of neuron branch which is not available in whole neuron reconstruction data, its promising results advocate the possibility of learning morphology representation without labels.

Inspired by the contrastive learning successes in visual representation, we experiment with adapting MoCo for neuron morphology representation learning, namely TreeMoCo. It can learn embedding of 3D neuron reconstructions without needing prior knowledge such as cell type annotation. It first maps neuron trees input to a deep embedding space by our proposed neuron encoder, and then incorporates an SSL pipeline composed of neuron tree augmentations and a contrastive loss to learn representations. By testing the framework on 2403 neuron reconstructions from 3 different data resources, we are excited to find that TreeMoCo can clearly delineate neuron morphology in the embedding space (Fig. 1). Comparison with prior arts shows that TreeMoCo can achieve premium results than supervised methods even *without annotations*.

2 Background and Related Works

Heuristic features of neuron morphology. To quantitatively describe the neuron morphology, a set of heuristic features and related tools namely L-measure [36] has been proposed and widely used. These features can be summarized as 4 categories including (1) distance-based measurements (*e.g.*, branch length, distance to soma); (2) angle-based measurements (*e.g.*, bifurcation angle); (3) topology-based measurements (*e.g.*, number of bifurcations, branch order); and (4) size-based measurements (*e.g.*, branch radius, surface area). Statistics of these features such as mean or standard deviation are usually adopted to describe neuron morphology. Such representation is easy to compute and explain. However, they cannot differentiate complex morphologies (examples in Appx. A.1).

Cell-type and neuron morphology classification. As the primary function of dendritic branches is to compartmentalize functionally related synaptic inputs, dendritic morphology can be used to categorize neuronal subtypes based on total dendritic length, amount of branching across the dendritic trees, and localization of dendritic branch fields to different brain regions (such as layers of neocortex). Most existing cell-type classification algorithms rely on hand-crafted features [36] to predict or cluster cell types [43, 26]. In some recent works, researchers started to adopt tree-based model [57, 25] and sequence-based model [24] to classify neuron morphology. Despite competing performance than classic methods, training of these models requires manual annotations of cell types.

Contrastive learning. Contrastive learning (CL) is a SSL method that aims at bringing positive sample pairs closer and spreading away the negative sample pairs in the feature embedding space. This method dates back to [14] or earlier, and has recently drawn significant interest across different communities [49, 28, 18, 52, 17, 7, 6]. Some research efforts have been made in graph contrastive learning (GCL) [40, 33, 16, 53, 50]. Particularly, GraphCL [53] studies the graph data augmentations systematically for GCL, and InfoGCL [50] provides a theoretical analysis for it. However, to our best knowledge, no work has studied the tree graph, a special instance of graph, within the context of CL.

3 Methods

3.1 Tree graph representation of neuron

We define a neuron reconstruction as $T \triangleq (\mathbf{V}, \mathbf{E})$, where $\mathbf{V} = \{v_k\}; v_k = (x_k, y_k, z_k)$ are nodes with 3D coordinates and $E = \{e_k\}; e_k = (v_k, v_k^{parent})$ are edges connecting each node with its parent. As shown in Figure 2-(e), the number of 1-degree nodes could vary by reconstruction and image resolutions, while the bifurcation/tip nodes preserve major topology information. For computational efficiency, we remove 1-degree nodes and directly connect the root node, the bifurcation nodes, and the tip nodes in T (Fig. 2(b)). To preserve the shape information of 1-degree node sequences (branch in Fig. 2(e)), we calculate the related features and concatenate them with the coordinate property of its downstream node, as illustrated in Figure 2-(e,f). The processed tree graph is defined as $\Upsilon \triangleq \mathrm{Process}(T) \triangleq (\mathbf{X}, \mathcal{G})$, where $\mathcal{G}$ represents the tree topology and $\mathbf{X} = \{\mathbf{x}_k\}_{k=1}^{N}$ is an $N \times (3 + Q)$ matrix that is from N nodes with 3D coordinates and Q-dim *branch* features.

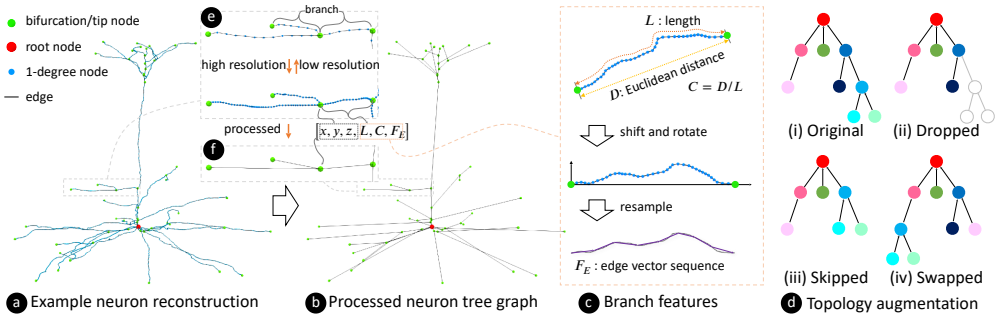


Figure 2: Illustration of (a) neuron reconstruction, (b) processed tree graph, (c) branch feature computation, (d) topology augmentations. (ii) to (iv) correspond to (9) to (11) in Table 1. (e) and (f) illustrate neuron tree processing for computation efficiency and resolution normalization purposes.

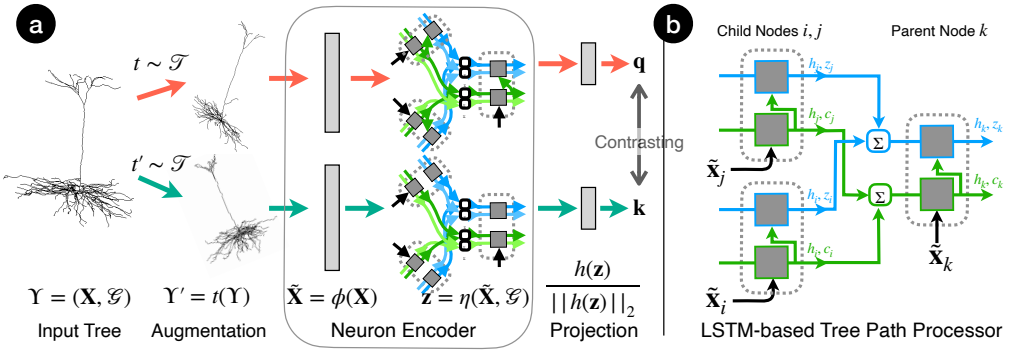


Figure 3: The overview of TreeMoCo: (a) TreeMoCo pipeline. (b) The zoom-in view of how the LSTM-based tree path processor η propagates information from child nodes i and j to a parent node k. Blue and green colors denote the corresponding information flows for two LSTM cells, respectively. The symbol "Σ" denotes summation. The dashed rectangles mark our double-cell LSTM processor.

Branch feature is a Q-dim vector, of which two dimensions denote length and contraction. Length L is the accumulated distance between adjacent nodes along a branch: $L = \sum_i^{start \leq i < end} ||v_i - v_{i+1}||_2$, which measures size. Contraction is the ratio between the Euclidean distance $D = ||v_{start} - v_{end}||_2$ and the length L of a branch: $C = D/L$, which measures the curviness. Notably, both measurements are invariant to the order of edges and thus cannot differentiate certain patterns (details in Appx. A.1). To allow a comprehensive encoding of branch pattern, we resample each branch with a fixed number of vertices (8 in this paper) and take the sequence of the edge vectors $\vec{E}_i = v_i - v_{i+1}$ as the its shape feature: $F_E \in \mathbb{R}^{24}$. In addition, the branch orientation is unified before extracting feature vectors such that the start and end points are on the x-axis and the principle direction is aligned with y-axis (Fig. 2(c)). Together, there are 29 features for each neuron node: $\mathbf{x} = [x, y, z; L; C; F_E]$.

3.2 Neuron Encoder

We next propose a neuron encoder f to project the neuron tree graph into a feature space. Our model has two components: a trainable MLP (multi-layer perceptron) embedding layer (ϕ) that maps point coordinates and other node-related features into the embedding space, and a LSTM-based tree-path processor (η) that exploits tree topology information. Formally, we define the neuron encoder as $f \triangleq \eta \circ \phi$. Figure 3 (a) shows an overview of our neuron encoder.

Embedding Layer (ϕ). The embedding layer maps input features vector $\mathbf{x}$ into d-dim embedding vectors $\tilde{\mathbf{x}}$. It is implemented by a three-layer perceptron with `BatchNorm` and `ReLU` attached to the input and hidden layers. We refer to the outputs of this embedding layer as node embeddings. Formally, we compute $\tilde{\mathbf{x}}_k = \phi(\mathbf{x}_k)$, where $\mathbf{x}_k$ is the k-th node of a neuron, or jointly as $\tilde{\mathbf{X}} = \phi(\mathbf{X})$.

LSTM-based tree-path processor (η). The original LSTM [19] architecture applies for strictly sequential data. However, a neuron tree structure is more complicated than a simple sequence. Therefore, we adopt a child-sum Tree-LSTM architecture, similar to [41], to process such data. Recall the hidden state and the cell state at time t and layer i are usually denoted as $\mathbf{h}_{t,i}$ and $\mathbf{c}_{t,i}$ in LSTMs [19]. Here we denote them similarly, but indexed by tree node, as $\mathbf{h}_{k,i}$ and $\mathbf{c}_{k,i}$ at node k. To aggregate information from multiple child nodes, we simply sum over the hidden states and cell states. The transition equations can be formulated as followings:

$$\mathbf{h}'_{k,1} = \sum_{j \in \mathrm{C}(k)} \mathbf{h}_{j,1}, \quad \mathbf{c}'_{k,1} = \sum_{j \in \mathrm{C}(k)} \mathbf{c}_{j,1} \tag{1}$$

$$\mathbf{h}_{k,1}, \mathbf{c}_{k,1} = \mathrm{LSTM}_1(\tilde{\mathbf{x}}_k, (\mathbf{h}'_{k,1}, \mathbf{c}'_{k,1}))) \tag{2}$$

$$\mathbf{h}'_{k,2} = \sum_{j \in \mathrm{C}(k)} \mathbf{h}_{j,2}, \quad \mathbf{z}'_{k} = \sum_{j \in \mathrm{C}(k)} \mathbf{z}_{j} \tag{3}$$

$$\mathbf{h}_{k,2}, \mathbf{z}_k = \mathrm{LSTM}_2(\mathbf{c}_{k,1}, (\mathbf{h}'_{k,2}, \mathbf{z}'_k)) \tag{4}$$

where $\mathrm{C}(k)$ denotes the set of child nodes of node k, and $\mathbf{z}_k$ is the final cell state. We use two layers of LSTM cells (non-shared weights) to better capture the topology information (Eqns. 2, 4). Our tree-path processor is designed in a bottom-up manner where the information is aggregated from each individual leaf nodes and flowed to the root node. We then use the final cell state of the root node $\mathbf{z}$, referred to as *neuron representation*, for downstream tasks, *e.g.*, predicting class labels by an additional single linear layer classifier or projected by a projection head [6, 17] for contrastive learning. Formally, we compute the representation of a neuron tree by $\mathbf{z} = \eta(\tilde{\mathbf{X}}, \mathcal{G})$ (the propagation path is determined by $\mathcal{G}$). Figure 3 (b) demonstrates how information is propagated along the path.

3.3 Contrastive learning of tree graph

Recent progress in self-supervised contrastive learning, stemmed from visual representation learning [49, 17, 7, 6, 13], has been seen in many other fields, including graph contrastive learning [50, 53] — an overall under-explored direction. Attempts have been made to study self-supervised graph representation learning in terms of data augmentation [53], network architecture, and overall pipelines [50]. Although the neuron tree is a special case of graph structure, we argue that it is too special to share the same setups as traditional graphs in terms of contrastive learning [53]. Below we first introduce our TreeMoCo pipeline, then elaborate on the subtle but important difference between a neuron tree and a general graph, subsequently introducing our data augmentations for tree structure.

Tree Contrastive Learning. Figure 3 illustrates the overall pipeline of TreeMoCo. The core idea of contrastive learning is to maximize the similarity between positive sample pairs and minimize it among negative sample pairs by optimizing a contrastive loss function [49, 6, 14]. We follow a successful exemplar, MoCo [17, 7], in visual contrastive learning to build our framework.

Conceptually, MoCo [17] abstracts contrastive learning as a dictionary look-up problem. For each encoded query $\mathbf{q}$, there is a set of encoded keys $\{\mathbf{k}_0, \mathbf{k}_1, \mathbf{k}_2, ...\}$ in a dictionary. The task for a model is to pull closer $\mathbf{q}$ and its matched positive key $\mathbf{k}^+$ in the dictionary while to spread $\mathbf{q}$ away from all other negative keys $\mathbf{k}^-$. To that end, we follow the successful experience in [6, 17] to use the dot-product as similarity measurement, and the InfoNCE [49] contrastive loss:

$$L_q = -\log \frac{\exp(\mathbf{q} \cdot \mathbf{k}^+/\tau)}{\exp(\mathbf{q} \cdot \mathbf{k}^+/\tau) + \sum_{\mathbf{k}^-} \exp(\mathbf{q} \cdot \mathbf{k}^-/\tau)} \tag{5}$$

where τ is a temperature hyper-parameter [49]. This loss is defined for one input instance, and the total loss is averaged over a batch.

To construct a pair of query and key, given a set of trees $\mathcal{D}$ and a set of possible augmentations $\mathcal{T}$, we first uniformly sample a tree $\Upsilon \sim \mathcal{D}$ and transform it into two augmented views $\Upsilon_q = t(\Upsilon)$ and $\Upsilon_k = t'(\Upsilon)$, referred to as the query view and the key view, by applying two sampled augmentations $t \sim \mathcal{T}, t' \sim \mathcal{T}$. Then, we compute their neuron representations using a neuron encoder defined in Sec. 3.2, *i.e.*, $\mathbf{z}_q = f(\Upsilon_q), \mathbf{z}_k = f'(\Upsilon_k)$. Following the improvement in SimCLR [7], we further map the neuron representations to another latent space using a non-linear projection head $h(\cdot)$ and normalize them by $\mathbf{q} = h(\mathbf{z}_q)/||h(\mathbf{z}_q)||_2$ and $\mathbf{k} = h'(\mathbf{z}_k)/||h'(\mathbf{z}_k)||_2$. Here, f' and h' stand for the momentum copies of f and h, named momentum encoder [17]. Therefore, $\mathbf{q}$ and $\mathbf{k}^+$ in Equation 5 are naturally generated from a single sample, and $\mathbf{k}^-$'s are from a queue [17] that stores features from previous batches.

Discussion: data augmentation on graphs, trees, and neuron morphology. Data augmentation is essential for self-supervised contrastive learning [6]. Current successful data augmentations for graphs [53] mainly include (1) "node dropping" that randomly drops graph nodes and their associated edges, (2) "edge perturbation" that randomly adds or drops edges, (3) "attribute masking" that acts like dropout [39] to node features, and (4) "subgraph sampling" that uses random walk to sample a partial graph. These augmentations can apply to general graph data without loss of generality. However, some of them are unsuitable for tree-structure graphs without special consideration. For example, the node dropping at a rate of 0.2 (default in [53]) would produce a pruned tree since all the subtrees would be removed once their parent node was dropped. Besides, edge perturbation is unlikely to be feasible in a tree structure if no strategical design is considered. Two nodes in a neuron tree may be spatially close to each other while having long "distance" in terms of topology (*e.g.*, number of hops from one node to another) and vice versa. The perturbed edges would create cyclic graphs on the one hand, and break the consistency between space (3d coordinates) and structure (nodes connectivity) on the other hand. Therefore, we argue the special consideration is needed for the design of data augmentations on tree-structured data.

Data augmentation for neuron morphology representation learning. Regarding the characteristics of neuron data, we propose 11 available augmentations for neuron morphology representation learning, which are briefly summarized in Table 1 and some implementations are provided in Appendix B.3. The rationale behind our design is that neuron representation should be invariant to augmentations — the golden rule applicable in many cases. For neuron data, such augmentations may occur in (i) node coordinates, (ii) overall morphology, (iii) attribute masking and (iv) topology. Thus we can augment neuron data from different perspectives as Table 1 shows. Jittering (4, 6) means adding random noise in a small range; deforming (7) means scaling each branch independently. We also include attribute masking (8) used in general graph augmentation. For (iv) topology, we propose three different augmentations. The first is (9) "truncated dropping subtrees" that recursively visits nodes from the root to leaves and drops a subtree with a small probability (*e.g.*, 0.05). The name "truncated" means at most m revisesubtrees will be dropped. Similarly, we have (10) "skipping parent node" that at most m parent nodes will be replaced by one of their subtrees (respectively), and (11) "swapping sibling child" that picks a subtree from a sibling node and swaps that subtree with another subtree in the current node; all pickings are random. For simplicity, we empirically use the same $m = 10$ for (9)-(11) in this paper. Figure 2 (d) illustrates how topology augmentation acts like.

Table 1: Overview of data augmentations for neuron morphology representation learning.

Category	Target	Augmentations
(i) Point Transformation	Coordinates	1) Scaling, 2) Rotating, 3) Shifting, 4) Jittering, 5) Flipping
(ii) Morphology	Branch feats	6) Jittering branches, 7) Deforming branches
(iii) Attribute Masking	Branch feats	8) Masking attributes
(iv) Topology	Edges	9) Truncated dropping subtrees 10) Truncated skipping parent node 11) Truncated swapping sibling child

4 Experiments

4.1 Experiment setups

Data. We download data from 3 public resources: (1) BICCN fMOST data from Brain Image Library (BIL, `https://download.brainimagelibrary.org/biccn/zeng/luo/fMOST/`) [31], (2) Janelia MouseLight (JML, `http://mouselight.janelia.org/`) [48], and (3) Allen Cell Types (ACT, `https://celltypes.brain-map.org/`) [1]. In total, 3358 three-dimensional mouse brain neuron reconstructions are collected (refer to Table 2 for details). To facilitate analysis, we exclude cell types with less than 50 reconstructions and the neurons with obvious reconstruction errors, yielding 2403 neurons in total. Some of the samples include both dendrite and axon reconstructions. For simplicity, we only keep dendritic arbors in this paper and leave the analysis of axons for future works. All three datasets have registered reconstructions to the 3D Allen mouse brain Common Coordinate Framework (CCFv3 [45]) and thus their scales are the same. Meanwhile, the absolute coordinates and orientations of registered neurons could also carry cell-type information that are

related to brain anatomy instead of neuron morphology. To eliminate such potential information leakage, all dendrites are shifted and rotated such that the root is placed at the origin and the X/Y/Z coordinates are determined based on principle component analysis (PCA). Data are then processed and analyzed following the steps introduced in the Section 3.1. In the interest of evaluating TreeMoCo's capability of classifying known cell types across data sources, we specifically use the cell types with rich coverage among 3 datasets only (grayed cell types in Table 2). We denote these sub-datasets as BIL-6, JML-4 and ACT, respectively. For supervised training, 80% samples in each dataset are randomly picked as training set and 20% are reserved for testing. For TreeMoCo pre-training, we pre-train on the joint training sets of BIL-6, JML-4, and ACT when evaluating the supervised downstream tasks and train on all samples except those from class "others" when analyzing the embedding space.

Table 2: Demographic table of reconstructions. Refer to Appendix A.2 for full names of brain regions.

Dataset	**Number of neuron reconstructions in each brain regions**											
	Total	**L2/3**	**L4**	**L5**	**L6**	**VPM**	**CP**	**VPL**	**SUB**	**PRE**	**MG**	**Others**
BIL [31]	1741	126	56	315	93	378	311	80	4	-	50	328
JML [48]	1107	64	-	179	114	12	-	5	70	60	-	602
ACT [1]	510	111	123	164	97	-	-	-	-	-	-	15
Total	3358	301	179	658	304	390	311	85	74	60	50	945

Implementations and evaluation. We postpone the detailed implementations and settings of hyper-parameters of TreeMoCo to Appendix B.1. For implementations of all the compared methods, we directly use the official codes and the default settings of MorphVAE [24], TRNN [57] and GraphCL [57] to obtain the results. Please refer to Appendix B.2 for their implementation details. All models share the same 29-d input features and are trained for 100 epochs. For evaluation, we report the sample-wise accuracy. During pre-training, we evaluate models every 5 epochs via a K Nearest-Neighbor (KNN) classifier for unsupervised methods or their inherent classification heads for supervised methods and report the best test set performance for all the studied methods. This strategy mainly follows [49, 53]. We set the number of neighbors to 20 for BIL-6 and ACT datasets and 5 for the JML-4 dataset since a minority class might only have 12 samples in total in JML-4.

4.2 TreeMoCo delineates neuron morphology in the embedding space

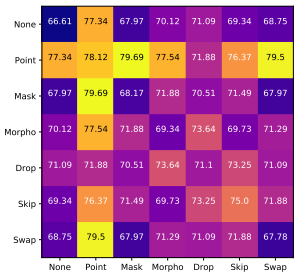
Visual inspection of cell types delineation. We first demonstrate that TreeMoCo can learn neuron tree representations that manifest different characteristics of neuron morphology. Self-supervised training is conducted on the full dataset, and the distribution of the embeddings is visualized by t-SNE [42] in Figures 1 and 4. By visual inspection, it is evidenced that TreeMoCo can separate major cell-types such as pyramidal neuron (Figs. 1(b-c)) and interneuron (Figs. 1(d-e)). Pyramidal neurons are the primary long-range output neurons of the brain [38] while interneurons mainly sculpt local network dynamics through a variety of circuit motifs. The asymmetric long projecting apical tuft is a unique feature that differentiates pyramidal neuron's morphology with interneuron's. TreeMoCo can further delineate subtypes of pyramidal neuron and interneurons. In Figure 1 and 4(a), we can clearly see the separation of layer 2/3 (Fig. 1(c)) and layer 5 (Fig. 1(b)) pyramidal neurons. Layer 2/3 neurons can be distinguished by their shorter apical dendrites with extensive oblique branching [47] while layer 5 neurons feature extended apical dendrites with extensive horizontal branching in layer 1 and sparse oblique branching in layer 2/3 [15]. The two primary classes of inhibitory interneurons (parvalbumin-expressing basket cells and somatostatin-expressing Martinotti cells) are also clearly distinguished with TreeMoCo. Parvalbumin-expressing basket cells have extensive branching field that extends radially around the soma (Fig. 1(d)) [32]. In contrast, somatostatin-expressing Martinotti cells tend to have unipolar or bipolar dendritic branching that extends vertically above and below the soma (Fig. 1(e)) [37]. It should be noted that such delineation is purely self-driven without any human labels or any prior knowledge of cell types. In contrast, the delineation is less clear in GraphCL's result and not observed in MorphVAE's result (details in Appx. D).

Quantitative evaluation. Here we quantitatively evaluate the quality of learned representations of different methods in classifying the known cell-types. For supervised methods, we directly train them on each dataset. For unsupervised methods, we pre-train them on the joint training sets. Table 3 reports the results, with the best test KNN accuracy during unsupervised pre-training or the best test

Table 3: Comparisons to state-of-the-arts neuron cell-type classification. We report the sample-wise accuracy (%). The numbers of classes in BIL-6, JML-4 and ACT are 6, 4, and 4, respectively. † Results obtained from officially released codes.

Methods	BIL-6	JML-4	ACT
Supervised training in individual datasets.			
†MorphVAE [24]	66.80	40.00	41.05
†TRNN [57]	76.56	51.43	46.32
Unsupervised pre-training on joint training sets.			
†GraphCL [53] frozen KNN	69.14	54.29	58.95
TreeMoCo frozen KNN	78.91	64.29	55.79
TreeMoCo fine-tuned	83.98	67.14	68.42

Table 4: Ablation on augmentations. We report the accuracy (%) of KNN on BIL-6 test set with models pre-trained on the full BIL-6 dataset with different augmentations.



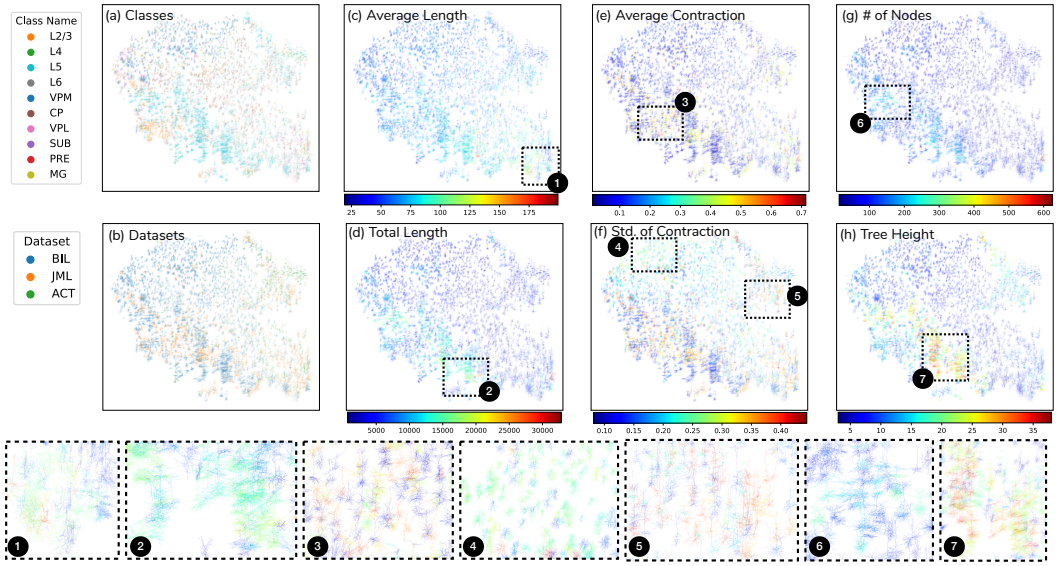


Figure 4: Thumbnails of neurons in the same embedding space of Figure 1 colored by different attributes. It is evidenced TreeMoCo encodes cell types (a) and neuron morphology features (c)-(h), *without human labels*. Refer to Appendix D for high-resolution figures.

accuracy during supervised training included. It can be observed that TreeMoCo trained without label information can already achieve better performance than supervised MorphVAE [24] and TRNN [57]. Fine-tuning the pre-trained neuron encoder further improves cell-type classification. This is mainly because MorphVAE [24] is based on random walk to capture topology information, which ignores tree architecture. While TRNN [57] has less description power than our neuron encoder in capturing high-order interaction between features. GraphCL [53] outperforms TreeMoCo in the ACT dataset but falls behind in BIL-6 and JML-4 datasets. This coincides with the data quality differences (ACT reconstruction is less complete than BIL and JML, more details in Appx. A.2). One explanation is that GraphCL pays more attention to local context and thus is less sensitive to global changes. It should be emphasized that although the overall test set performance is improved during training, we observe that the test KNN accuracy of both TreeMoCo and GraphCL fluctuates and suffers variations, especially for the smaller JML-4 and ACT datasets. GraphCL fails to improve models over time in the JML-4 dataset under some scenarios. Please refer to Appendix C for more details and discussions, where we provide the performance curves of different methods over training.

TreeMoCo manifests neuron morphology distribution. Figure 4 shows the t-SNE plots of neuron representations. Each plot is colored by a different morphology feature, *i.e.*, average/total branch length (c, d), average / standard deviation of contraction (e, f), number of nodes in a neuron (g) and the height of a neuron tree (h). Neurons that share similar morphology features are well clustered in

the embedding space (see (1) to (7)), demonstrating the good representations learnt by TreeMoCo. These morphology attributes have not been explicitly provided as input features but it turns out that TreeMoCo can mine such information by itself, *without human labels*. In addition, we can see it by cross-referencing Figure 4 (a) and (b) that neurons of a same class but from different datasets are clustered together in TreeMoCo's representation space, showing TreeMoCo's unbiasedness to data sources — a much desirable property for neuron morphology representation learning.

4.3 Ablation on augmentation methods and network architectures

In this subsection, we first assess the importance of data augmentations in contrastive neuron morphology learning. Then we ablate on several design choices in our neuron encoder.

To systematically study the effects of different categories of data augmentations, we first group the proposed augmentations in Table 1 by their categories, except for the topology augmentation, the one we are more interested in. Then, we apply them individually or in pairs to TreeMoCo framework with a higher applying probability due to the reduced number of augmentations (details in Appx. B.2.) when pre-training. Note that this ablation does not aim to figure out the best compositions of presented augmentations, but to unravel how different augmentations contribute to the final performance. For ablation purposes, in this experiment we train on the full BIL-6 dataset for 50 epochs. Table 4 shows the KNN evaluation results under each individual category or a composition of data augmentations. Here we do not specify the order of augmentations, thus resulting a symmetric performance matrix. Unlike the observation in visual or graph representation contrastive learning that no single augmentation performs the best [6, 53], we find in neuron morphology representation learning certain kinds of augmentations are good enough for pre-training (*e.g.*, point transformation, skipping parent node). Topology augmentations do not necessarily bring better performance. This could be caused by the mechanism of Tree-LSTM, which aggregates the whole tree to generate embeddings and thus is sensitive to the global changes caused by topology augmentation. Better neuron morphology representations might be excepted if a more proposer set of augmentations or a better network architecture were applied under the TreeMoCo framework.

Then, we study the effects of different components in our neuron encoder. For this experiment, we train neuron encoders for 100 epochs on three datasets and report their best test performance. Table 5 (left) shows the results, where a steady increasing trend of model performance can be observed. We also include the KNN accuracy (Tab. 5 right) for 50-epoch pre-training on the joint dataset and evaluated in BIL-6 dataset just for ablation interest. Here we find that batch normalization (`BN`) is beneficial for both contrastive self-supervised learning and supervised learning. No `BN` in the neuron encoder would give significantly worse results (first line of right table). Further adding `BN` in the projection head is helpful for TreeMoCo. This behavior has also been reported in non-contrastive SSL methods before, *e.g.*, BYOL [13, 34], where removing `BN` causes model collapse.

Table 5: **Ablation on model designs. (left)** Classification accuracy of supervised training for 100-epoch. **(right)** KNN accuracy of TreeMoCo pre-training for 50 epochs and evaluated in BIL-6 dataset. BN matters more in TreeMoCo. Gray rows indicate our default setting.

Settings			Classification Accuracy (%)		
BN	MLP	# LSTM Cells	BIL-6	JML-4	ACT
-	-	1	80.86	54.29	43.16
-	✓	1	83.98	57.14	50.53
✓	✓	1	84.77	58.57	**63.16**
✓	✓	2	**85.55**	**64.29**	62.11

Encoder BN	Projector BN	KNN accuracy (%) BIL-6
-	-	69.14
✓	-	75.00
✓	✓	**76.56**

5 Conclusion and Limitations

In this work, we have explored contrastive learning for neuron morphology representation learning via TreeMoCo. Our results on 2403 neuron reconstructions show that TreeMoCo has the unique potential to identify unbiased cell-types. Modern cell-type classification is heavily dependent on transcriptomic data [54, 9]. However, even neurons within the same mouse possessing identical genetic identity and birth-date demonstrate extremely variable dendritic morphology and synaptic distribution [20]. This variability underlies the need for high-throughput neuronal morphology classification techniques to probe for the existence of novel subtypes within classically established

neuronal identities [4]. TreeMoCo has a great potential to fill this need and to explore dendritic architecture. These findings could lead to more rigorous output measurements for drug discovery experiments and aid in the development of treatments for neurological disorders characterized by progressive dendrite degeneration such as Alzheimer's disease and depression [11, 8].

We note that TreeMoCo is still a preliminary effort of its kind. Certain limitations can be observed and need to be solved in the future. For instance, Tree-LSTM aggregates the whole graph and thus is sensitive to global changes. This results in its performance decay on incomplete neuron reconstruction (ACT dataset) and its favor of node-feature-based augmentation over topology-based augmentation. Moreover, compared to most image contrastive learning frameworks, TreeMoCo's performance on downstream tasks could oscillate after long training epochs on small datasets. An effective early stopping criterion is needed but absent when label information is not available.

Nevertheless, the tree graph is a common data structure representing objects varying from artery and lung vessels [23] to program source code [29] and HTML documents [5]. Another direction of our future work is to extend TreeMoCo into a more general framework for learning representations beyond neuron morphology. Following the success of contrastive learning in image analysis problems, we believe TreeMoCo could be a powerful tool to solve unsupervised and few-shot learning problems related to a wide range of tree graph data.

Acknowledgments and Disclosure of Funding

Daniel Maxim Iascone is sponsored by NIH (F32 MH125600-01).

Lijuan Liu is sponsored by a MOST (China) Brain Research Project, "Mammalian Whole Brain Mesoscopic Stereotaxic 3D Atlas" (2022ZD0205200, 2022ZD0205204).

References

[1] Allen cell types database. allen institute for brain science. http://celltypes.brain-map.org/data.

[2] Yoshua Bengio, Yann Lecun, and Geoffrey Hinton. Deep learning for ai. *Communications of the ACM*, 64(7):58–65, 2021.

[3] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments. *Advances in Neural Information Processing Systems*, 33:9912–9924, 2020.

[4] Mark S Cembrowski and Nelson Spruston. Heterogeneity within classical cell types is the rule: lessons from hippocampal pyramidal neurons. *Nature Reviews Neuroscience*, 20(4):193–204, 2019.

[5] Chia-Hui Chang, Mohammed Kayed, Moheb R Girgis, and Khaled F Shaalan. A survey of web information extraction systems. *IEEE transactions on knowledge and data engineering*, 18(10):1411–1428, 2006.

[6] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[7] Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. Improved baselines with momentum contrastive learning. *arXiv preprint arXiv:2003.04297*, 2020.

[8] David Cotter, Daniel Mackay, Sabine Landau, Robert Kerwin, and Ian Everall. Reduced glial cell density and neuronal size in the anterior cingulate cortex in major depressive disorder. *Archives of general psychiatry*, 58(6):545–553, 2001.

[9] Javier DeFelipe, Pedro L López-Cruz, Ruth Benavides-Piccione, Concha Bielza, Pedro Larrañaga, Stewart Anderson, Andreas Burkhalter, Bruno Cauli, Alfonso Fairén, Dirk Feldmeyer, et al. New insights into the classification and nomenclature of cortical gabaergic interneurons. *Nature Reviews Neuroscience*, 14(3):202–216, 2013.

[10] Michael N Economo, Nathan G Clack, Luke D Lavis, Charles R Gerfen, Karel Svoboda, Eugene W Myers, and Jayaram Chandrashekar. A platform for brain-wide imaging and reconstruction of individual neurons. *Elife*, 5:e10566, 2016.

[11] Eric Falke, Jonathan Nissanov, Thomas W Mitchell, David A Bennett, John Q Trojanowski, and Steven E Arnold. Subicular dendritic arborization in alzheimer's disease correlates with neurofibrillary tangle density. *The American journal of pathology*, 163(4):1615–1621, 2003.

[12] Hui Gong, Dongli Xu, Jing Yuan, Xiangning Li, Congdi Guo, Jie Peng, Yuxin Li, Lindsay A Schwarz, Anan Li, Bihe Hu, et al. High-throughput dual-colour precision imaging for brain-wide connectome with cytoarchitectonic landmarks at the cellular level. *Nature communications*, 7(1):1–12, 2016.

[13] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, et al. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in Neural Information Processing Systems*, 33:21271–21284, 2020.

[14] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[15] Kenneth D Harris and Thomas D Mrsic-Flogel. Cortical connectivity and sensory coding. *Nature*, 503(7474):51–58, 2013.

[16] Kaveh Hassani and Amir Hosein Khasahmadi. Contrastive multi-view representation learning on graphs. In *International Conference on Machine Learning*, pages 4116–4126. PMLR, 2020.

[17] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[18] R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler, and Yoshua Bengio. Learning deep representations by mutual information estimation and maximization. *arXiv preprint arXiv:1808.06670*, 2018.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[20] Daniel Maxim Iascone, Yujie Li, Uygar Sümbül, Michael Doron, Hanbo Chen, Valentine Andreu, Finola Goudy, Heike Blockus, Larry F Abbott, Idan Segev, et al. Whole-neuron synaptic mapping reveals spatially precise excitatory/inhibitory balance limiting dendritic and somatic spiking. *Neuron*, 106(4):566–578, 2020.

[21] Lida Kanari, Paweł Dłotko, Martina Scolamiero, Ran Levi, Julian Shillcock, Kathryn Hess, and Henry Markram. A topological representation of branching neuronal morphologies. *Neuroinformatics*, 16(1):3–13, 2018.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Cemil Kirbas and Francis Quek. A review of vessel extraction techniques and algorithms. *ACM Computing Surveys (CSUR)*, 36(2):81–121, 2004.

[24] Sophie C. Laturnus and Philipp Berens. Morphvae: Generating neural morphologies from 3d-walks using a variational autoencoder with spherical latent space. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 6021–6031. PMLR, 18–24 Jul 2021.

[25] Congping Lin, Yuanfei Huang, Tingwei Quan, and Yiwei Zhang. Modelling brain-wide neuronal morphology via rooted cayley trees. *Scientific reports*, 8(1):1–10, 2018.

[26] Bojan Mihaljević, Pedro Larrañaga, Ruth Benavides-Piccione, Sean Hill, Javier DeFelipe, and Concha Bielza. Towards a supervised classification of neocortical interneuron morphologies. *BMC bioinformatics*, 19(1):1–22, 2018.

[27] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020.

[28] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[29] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems*, 34:9343–9354, 2021.

[30] Hanchuan Peng, Jianyong Tang, Hang Xiao, Alessandro Bria, Jianlong Zhou, Victoria Butler, Zhi Zhou, Paloma T Gonzalez-Bellido, Seung W Oh, Jichao Chen, et al. Virtual finger boosts three-dimensional imaging and microsurgery as well as terabyte volume image visualization and analysis. *Nature communications*, 5(1):1–13, 2014.

[31] Hanchuan Peng, Peng Xie, Lijuan Liu, Xiuli Kuang, Yimin Wang, Lei Qu, Hui Gong, Shengdian Jiang, Anan Li, Zongcai Ruan, et al. Morphological diversity of single neurons in molecularly defined cell types. *Nature*, 598(7879):174–181, 2021.

[32] Frédéric Pouille and Massimo Scanziani. Enforcement of temporal fidelity in pyramidal cells by somatic feed-forward inhibition. *Science*, 293(5532):1159–1163, 2001.

[33] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. Gcc: Graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1150–1160, 2020.

[34] Pierre H Richemond, Jean-Bastien Grill, Florent Altché, Corentin Tallec, Florian Strub, Andrew Brock, Samuel Smith, Soham De, Razvan Pascanu, Bilal Piot, et al. Byol works even without batch statistics. *arXiv preprint arXiv:2010.10241*, 2020.

[35] Philipp J Schubert, Sven Dorkenwald, Michał Januszewski, Viren Jain, and Joergen Kornfeld. Learning cellular morphology with neural networks. *Nature communications*, 10(1):1–12, 2019.

[36] Ruggero Scorcioni, Sridevi Polavaram, and Giorgio A Ascoli. L-measure: a web-accessible tool for the analysis, comparison and search of digital reconstructions of neuronal morphologies. *Nature protocols*, 3(5):866–876, 2008.

[37] Gilad Silberberg and Henry Markram. Disynaptic inhibition between neocortical pyramidal cells mediated by martinotti cells. *Neuron*, 53(5):735–746, 2007.

[38] Nelson Spruston. Pyramidal neurons: dendritic structure and synaptic integration. *Nature Reviews Neuroscience*, 9(3):206–221, 2008.

[39] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[40] Fan-Yun Sun, Jordan Hoffmann, Vikas Verma, and Jian Tang. Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. *arXiv preprint arXiv:1908.01000*, 2019.

[41] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[42] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[43] Xavier Vasques, Laurent Vanel, Guillaume Villette, and Laura Cif. Morphological neuron classification using machine learning. *Frontiers in neuroanatomy*, 10:102, 2016.

[44] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[45] Quanxin Wang, Song-Lin Ding, Yang Li, Josh Royall, David Feng, Phil Lesnar, Nile Graddis, Maitham Naeemi, Benjamin Facer, Anh Ho, et al. The allen mouse brain common coordinate framework: a 3d reference atlas. *Cell*, 181(4):936–953, 2020.

[46] Yimin Wang, Qi Li, Lijuan Liu, Zhi Zhou, Zongcai Ruan, Lingsheng Kong, Yaoyao Li, Yun Wang, Ning Zhong, Renjie Chai, et al. Teravr empowers precise reconstruction of complete 3-d neuronal morphology in the whole brain. *Nature communications*, 10(1):1–9, 2019.

[47] Verena C Wimmer, Randy M Bruno, Christiaan PJ De Kock, Thomas Kuner, and Bert Sakmann. Dimensions of a projection column and architecture of vpm and pom axons in rat vibrissal cortex. *Cerebral cortex*, 20(10):2265–2276, 2010.

[48] Johan Winnubst, Erhan Bas, Tiago A Ferreira, Zhuhao Wu, Michael N Economo, Patrick Edson, Ben J Arthur, Christopher Bruns, Konrad Rokicki, David Schauder, et al. Reconstruction of 1,000 projection neurons reveals new cell types and organization of long-range connectivity in the mouse brain. *Cell*, 179(1):268–281, 2019.

[49] Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3733–3742, 2018.

[50] Dongkuan Xu, Wei Cheng, Dongsheng Luo, Haifeng Chen, and Xiang Zhang. Infogcl: Information-aware graph contrastive learning. *Advances in Neural Information Processing Systems*, 34, 2021.

[51] Fang Xu, Yan Shen, Lufeng Ding, Chao-Yu Yang, Heng Tan, Hao Wang, Qingyuan Zhu, Rui Xu, Fengyi Wu, Yanyang Xiao, et al. High-throughput mapping of a whole rhesus monkey brain at micrometer resolution. *Nature Biotechnology*, 39(12):1521–1528, 2021.

[52] Mang Ye, Xu Zhang, Pong C Yuen, and Shih-Fu Chang. Unsupervised embedding learning via invariant and spreading instance feature. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6210–6219, 2019.

[53] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. *Advances in Neural Information Processing Systems*, 33:5812–5823, 2020.

[54] Rafael Yuste, Michael Hawrylycz, Nadia Aalling, Argel Aguilar-Valles, Detlev Arendt, Ruben Armañanzas, Giorgio A Ascoli, Concha Bielza, Vahid Bokharaie, Tobias Borgtoft Bergmann, et al. A community-based transcriptomics classification and nomenclature of neocortical cell types. *Nature neuroscience*, 23(12):1456–1468, 2020.

[55] Hongkui Zeng and Joshua R Sanes. Neuronal cell-type classification: challenges, opportunities and the path forward. *Nature Reviews Neuroscience*, 18(9):530–546, 2017.

[56] Xiaohang Zhan, Jiahao Xie, Ziwei Liu, Yew-Soon Ong, and Chen Change Loy. Online deep clustering for unsupervised representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6688–6697, 2020.

[57] Tielin Zhang, Yi Zeng, Yue Zhang, Xinhe Zhang, Mengting Shi, Likai Tang, Duzhen Zhang, and Bo Xu. Neuron type classification in rat brain based on integrative convolutional and tree-based recurrent neural networks. *Scientific reports*, 11(1):1–14, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Additional information of neuron data and features

A.1 Limitations of handcrafted features

In this section, we will show some examples of neuron branches and neuron dendrites that cannot be differentiated by handcrafted features.

Limitations of branch features. We take a sequence of vertices between two bifurcations or a bifurcation (fork node) and a tip node (leaf) as a branch. A branch can be viewed as a sequence of vectors $E = [\vec{E_0}, \vec{E_1}, ..., \vec{E_{n-1}}]$, where $\vec{E_i} = V_i - V_{i+1}$ is the vector pointing from a child node to its parent node. Two commonly used measurements measuring the shape of a branch are *length* and *contraction*. The formulas of these two metrics are detailed in Sec. 3.1 and Figure A1 (a). Here, we elaborate on the limitations of both metrics. Based on the definition of length and contraction, it is evident that both metrics are invariant to rotation and shifting of the branch — two desired properties. What is less obvious is that both metrics are invariant to the order of branch vectors $\vec{E_i}$. As shown in Figure A1 (a), by only re-ordering of edge vectors, a branch's shape can be significantly changed from a bow shape to a zig-zag shape while its length and contraction remain the same.

Limitations of neuron features. Most classic neuron morphology analysis works adopt the mean and the standard deviation of handcrafted features related to branches or bifurcations to measure the shape of a neuron. Such measurement is taken in a bag-of-word (BOW) fashion. The context and spatial relationship between arbors are ignored under this measurement. Figure A1 (b) shows that the neurons of significantly different shapes could share similar features based on this kind of measurement. In this paper, our proposed neuron encoder can capture information from both the branch features as well as the context and spatial relation via the interaction between the embedding layer and the tree-LSTM-based path processor.

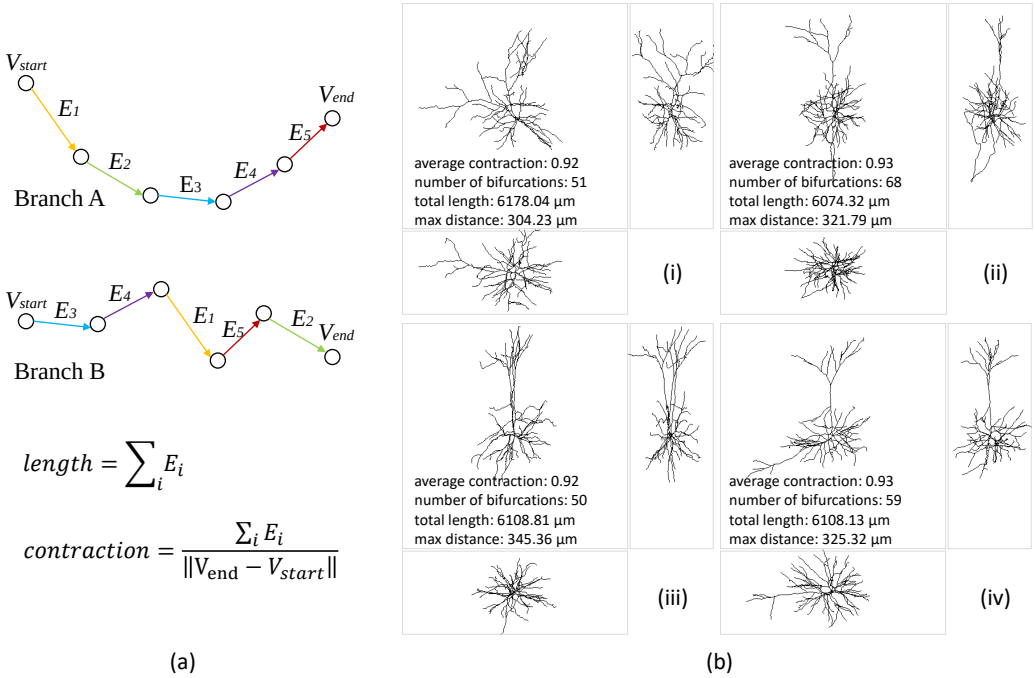


Figure A1: Illustration of limitation of handcrafted features. (a) Length and contraction are invariant to the order of edge vectors. Branch A and B are composed of the same set of edge vectors. They are of different shapes but share the same length and contraction value. (b) Neurons of different shapes may have similar features. Each neuron is viewed from three different angles. Feature values are listed in the subplot.

Table A1: Details of datasets used in this paper.

Dataset	BIL	JML	ACT
Full Name	BICCN fMOST data from Brain Image Library	Janelia Mouse Light	Allen Cell Types
Reference	[31]	[48]	[1]
Total reconstructions	1741	1107	510
Reconstructions used	1413	505	495
Complete reconstruction	Yes	Yes	No
Cell type label	Manual	Auto	Manual
Tissue labeling	TIGRE2.0 transgenic reporter lines coupled with Cre espression	A mixture of low-titer AAV Syn-iCre and a high-titer Cre-dependent reporter	Biocytin-filled
Image modality	fMOST	MouseLight	flourescent and brightfield
Reconstruction method	Automatic reconstruction followed by manual correction and automatic refinement	Automatic reconstruction followed by manual correction	Automated 3D reconstruction manually curated using the Mozak extension of Vaa3D.
Registration method	mBrainAligner	LandMark	Manual

Links for datasets:
BIL: https://download.brainimagelibrary.org/biccn/zeng/luo/fMOST/
JML: http://mouselight.janelia.org/
ACT: https://celltypes.brain-map.org/

A.2 Dataset details

Three public datasets are used in this study. All of them contain 3D neuron reconstructions from mouse brains. Reconstructions are registered to Allen mouse brain Common Coordinate Framework (CCFv3 [45]) in their original release. The cell type of a neuron is assigned based on its brain anatomical region. This information comes together with each dataset. Note that the labels of samples from the JML dataset are purely determined by the registration results without manual inspection. In contrast, in ACT and BIL datasets, such information has been visually inspected and corrected by neuroscientists and thus is more accurate. For JML and BIL datasets, whole neuron reconstructions are generated, including the long-range projection of axons. Due to the limitation of imaging techniques, the ACT dataset's neuron reconstructions are limited to a thin 3D slice of brain tissue and thus not complete. This type of data with incomplete morphology is typical in neuroscience studies due to the technical limitations and cost efficiency concerns. Thus, to show if and to which degree our proposed method could work on incomplete reconstruction, we keep the ACT dataset for analysis even if it is incomplete. To summarize, the BIL dataset has the best quality – complete 3D reconstructions and reliable cell type labels. More details of the datasets can be found in Table A1.

To facilitate analysis, we merge the isocortex cell types by layers and only keep 10 cell types with relatively broad coverage. The abbreviations and full names of these brain regions are listed in Table A2. Examples of neuron reconstructions of different types from different datasets can be found in Figures A2 and A3. It is visually evident that ACT's reconstructions are incomplete compared to BIL's and JML's. Also, with a visual impression, the dendritic morphology of some cell types, such as VPM, VPL and CP, are similar and thus challenging to be differentiated.

B Implementation details

Here we elaborate on the implementation details. Our code and pre-trained models will be made public once the paper is accepted.

B.1 Implementation details of TreeMoCo training.

Model architectures. For implementation, we set the layer sizes of embedding MLP to 256 and 128, and the cell state size of LSTM-processor to 128. Our detailed model architectures in PyTorch style are:

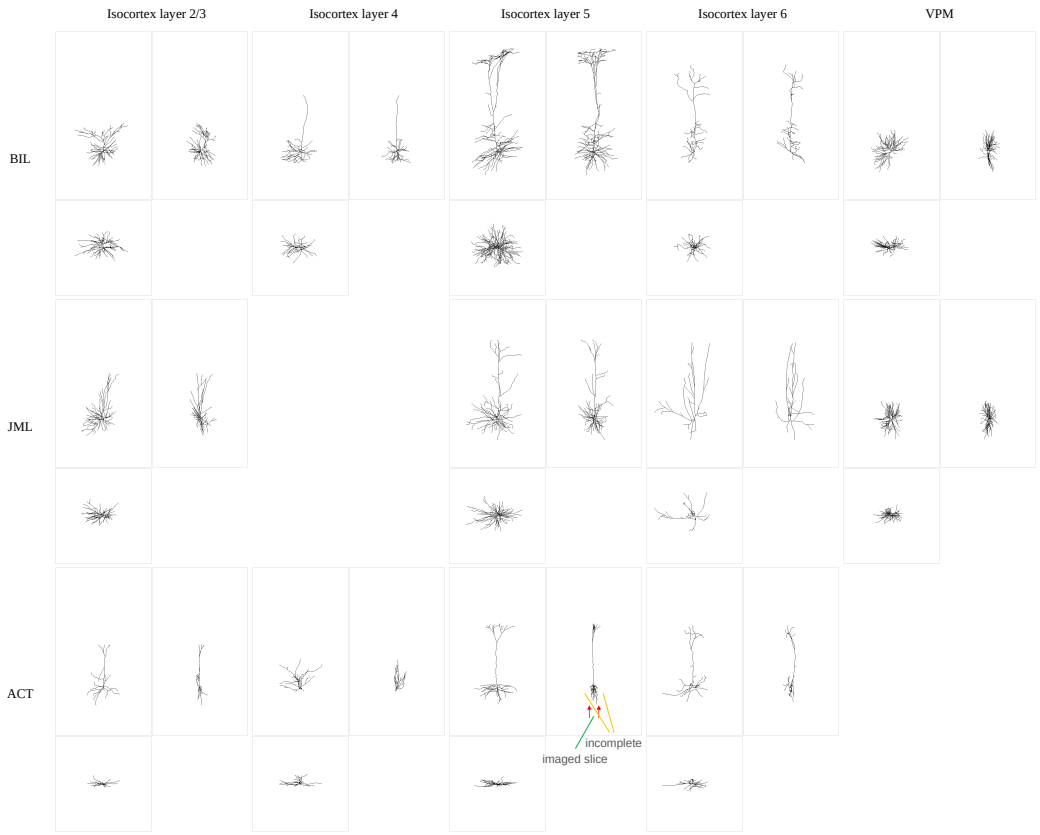


Figure A2: Example reconstructions of 5 neuron cell types that are common between datasets. Each column corresponds to a cell type. Each row corresponds to a dataset. Each subplot shows 3 different views (three-view) of the same neuron – top-left: front view, top-right: side view, bottom-left: top view. Thumbnails are taken on the same scale for all neurons. As evident from the side view, ACT's reconstructions are all flat compared to compared to BIL's and JML's. It is because ACT reconstructions are from a thin tissue slice and thus incomplete.

- Embedding layer ϕ:
```
embedding = Sequential(
            Linear(29,128), BatchNorm(128), ReLU(),
            Linear(128,256), BatchNorm(256), ReLU(),
            Linear(256,128))
```
- LSTM-based tree path processor η:
```
W1_iouf, U1_iouf = Linear(128,512), Linear(128,512) # first cell
W2_iouf, U2_iouf = Linear(128,512), Linear(128,512) # second cell
```
- Projection head h:
```
projector = Sequential(
            # non-linear projection head
            Linear(128,128), BatchNorm(128), ReLU(),
            Linear(128,128))
```

LSTM processor forward. We provide a pseudo-code for LSTM forward in Algorithm 1. Our implementation is based on Deep Graph Library (DGL) [44], an efficient library for graph propagation in deep learning.

Pre-training details. We provide a pseudo-code for TreeMoCo training in Algorithm 2. We train TreeMoCo with a batch size of 128 in one V100 GPU. We use an SGD optimizer with a weight decay of 5e-4 and a constant learning rate of 0.06. The temperature parameter τ is 0.1, the momentum

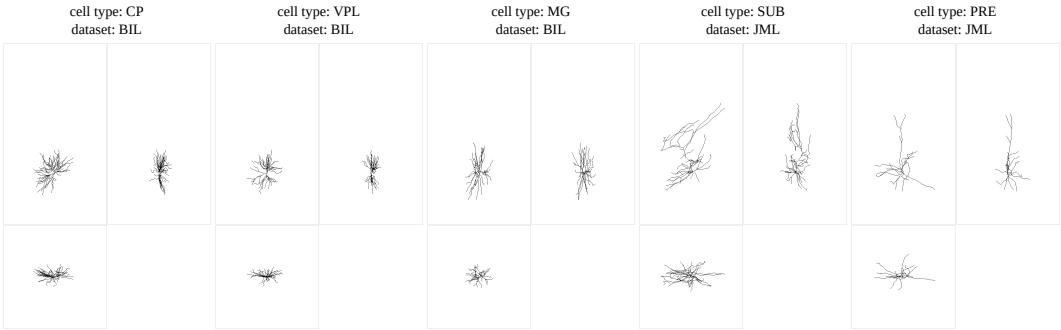


Figure A3: Example reconstructions of 5 neuron cell types that are only present in one dataset. Each subplot shows 3 different views (three-view) of the same neuron – top-left: front view, top-right: side view, bottom-left: top view. Thumbnails are taken on the same scale for all neurons.

Table A2: Abbreviation of brain region and cell types. All brain regions are defined following Allen Reference Atlas. Refer to the website for more details: http://atlas.brain-map.org/atlas?atlas=602630314

Abbreviation	**Full name**
l2/3	isocortex layer 2/3
l4	isocortex layer 4
l5	isocortex layer 5
l6	isocortex layer 6
VPM	Ventral posteromedial nucleus of the thalamus
VPL	Ventral posterolateral nucleus of the thalamus
CP	Caudoputamen
SUB	Subiculum
PRE	Presubiculum
MG	Medial geniculate complex

parameter to update the momentum encoder is 0.99, and the queue length to store negative keys is 1024. We use a smaller queue than MoCo [17] (1024 v.s. 65536) since the total number of neurons is significantly less than the number of images in visual representation learning [17, 7] (2.3k v.s. 1.28M). For data augmentations, we *have not searched* their combinations and hyper-parameters extensively. Therefore, our default setting of data augmentations may not be optimal for pre-training. We emphasize that even under our current naive combination of data augmentations, TreeMoCo has shown promising performance, showing its potential for gaining improved representations with better combined and parameterized data augmentations used during pre-training. The default data augmentations we use for pre-training are:

```
RandomScaleCoordsBranches(p=0.2, scales=[0.8, 1.2]),
RandomRotate(p=0.5),
RandomJitter(p=0.2, sigma=1, clip=5),
RandomShift(p=0.2, shift=[0.2, 0.2, 0.2]),
RandomFlip(p=1),
RandomMaskFeats(p=0.2),
RandomJitterBranches(p=0.2, sigma=0.1, clip=1),
RandomDeformation(p=0.2, scales=[0.8, 1.2]),
RandomDropSubTrees(p=0.05, max_cnt=10),
RandomSkipParentNode(p=0.05, max_cnt=10),
RandomSwapSiblingSubTrees(p=0.05, max_cnt=10)
```

Evaluation details. During pre-training, we extract features from the backbone and the projection head and evaluate them individually with the KNN protocol. We report the best KNN accuracy between them. Note that, in visual representation learning, the features extracted from the backbone are usually better than those from the projection head in terms of clustering performance [3, 56].

Algorithm 1 LSTM-based tree path processor ($\eta(\cdot)$) forward: PyTorch-like Pseudocode

```
# W_iouf, U_iouf: weights for input features and hidden states for a LSTM cell.
# nodes: data structure to store graph and data, used in DGL.
# embedding layer
feats = embedding(feats)
# initial states
nodes.data['iouf'] = W1_iouf(feats)
nodes.data['h1'], nodes.data['c1'] = zeros((N,128)), zeros((N,128))
nodes.data['h2'], nodes.data['z'] = zeros((N,128)), zeros((N,128))

def propagate(nodes):
    # nodes: [N, j, C], currently processing N nodes, each of which has j subtrees that own C-dim
         representations.
    h1, c1 = nodes.mailbox['h1'], nodes.mailbox['c1']
    h2, z = nodes.mailbox['h2'], nodes.mailbox['z']
    # sum over child nodes
    h1, c1, h2, z = h1.sum(-2), c1.sum(-2), h2.sum(-2), z.sum(-2)
    # first LSTM cell
    xi, xo, xu, xf = chunk(nodes.data['iouf'], 4, 1)
    hi1, ho1, hu1, hf1 = chunk(U1_iouf(h1), 4, 1)
    i, f, g, o = sigmoid(xi + hi1), sigmoid(xf + hf1), tanh(xu + hu1), sigmoid(xo + ho1)
    c1 = i * g + f * c1
    h1 = o * tanh(c1)

    # second LSTM cell
    xi, xo, xu, xf = chunk(W2_iouf(c1), 4, 1)
    hi2, ho2, hu2, hf2 = chunk(U2_iouf(h2), 4, 1)
    i, f, g, o = sigmoid(xi + hi2), sigmoid(xf + hf2), tanh(xu + hu2), sigmoid(xo + ho2)
    z = i * g + f * z
    h2 = o * tanh(z)
    return {'h1': h1, 'c1': c1,
            'h2': h2, 'z': z}
```

However, here we find the best KNN accuracy comes from the backbone for the BIL-6 dataset and the projection head for JML and ACT datasets. We leave the discussion and analysis for such an observation to future work. For fine-tuning evaluation, we initialize the neuron encoder with pre-trained weights and train it for 100 epochs using the Adam optimizer [22] with a learning rate of 2e-3 and a weight decay of 0.02. The projection head is removed and unused when fine-tuning.

Algorithm 2 TreeMoCo training: PyTorch-like Pseudocode

```
# f_q, h_q: encoder: backbone, projection head
# f_k, h_k: momentum encoder: momentum copies of backbone and projection head
# m: momentum coefficient
# tau: temperature

for batch_neuron in loader: # load a minibatch x with N samples
    x_q, x_k = aug(batch_neuron), aug(batch_neuron) # augmentation
    z_q, z_k = f_q(x_q), f_k(x_k) # positive pairs (query, key): [N, C] each
    q, k = h_q(z_q), h_k(k) # projection: [N, 128] each
    q, k = normalize(q), normalize(k) # l2 normalization

    logits_pos = dot(q, k) # similarity between positive pairs: [N, 1]
    logits_neg = mm(q, queue.t()) # similarity bewteen negative pairs: [N, K]
    logits = cat([logits_pos, logits_neg], dim=1) # Nx(1+K)
    labels = range(N)
    loss = CrossEntropyLoss(logits/tau, labels)
    queue.dequeue_and_enqueue(k) # update queue
    loss.backward()
    update([f, h]) # optimizer update: f, h
    [f_k, h_k] = m*[f_k, h_k] + (1-m)*[f_q, h_q] # momentum update: f_k, h_k
```

Notes: `dot` is vector dot-product. `mm` is matrix multiplication. `k.t()` is `k`'s transpose. `f_q, f_k, h_q` and `h_k` corresponds to f, f', h and h' in the main text respectively.

B.2 Implementation details of compared methods

MorphVAE. For MorphVAE [24], we adopt the officially released code [1] and use its default settings to obtain results. Instead of only using the x/y/z coordinate of nodes, we use the same set of processed neurons and 29-d features. MorphVAE is based on the variational auto-encoder (VAE) that aims to

[1] https://github.com/berenslab/morphvae

reconstruct from noisy inputs and further combines a classification head to utilize label information. Since MorphVAE is a path-based method that requires the "random walk" to sample paths from leaf nodes to the root node, we follow the original work to sample 256 paths with a maximum length of 32 from leaf nodes to the root node for each processed neuron tree reconstruction. The best testing set accuracy during 100-epoch training (evaluated every 5 epochs as other methods) is reported for a fair comparison. TRNN [57] and our neuron encoder encode the whole tree structure, while MorphVAE encodes the sampled paths inside the trees, which could ignore the tree topology information. And thus, it is not surprising that the performance of TRNN and our TreeMoCo is superior to MorphVAE (Table 3).

TRNN. For TRNN [57], we follow the officially released code [2] and use its default settings to obtain results. We change the input layer of the TRNN to take our processed tree graph with 29-d node features as inputs. Similar to evaluations of other methods, we monitor the test set performance every 5 epochs for 100 epochs during training and report the best test accuracy.

GraphCL For GraphCL [53], we use the officially released code[3] to obtain the results. We first convert our dataset to the TU dataset format [27] and directly use their default settings for unsupervised TU training. The input features are the same as other methods, *i.e.*, 29-d features. We train GraphCL for 100 epochs on the joint training sets, evaluate the KNN test accuracy every 5 epochs, and report the best one. In fact, the original setting of GraphCL code is to evaluate models every 10 epochs by training a linear SVM classifier on the training set and testing on the test set. Then, the best performance on the test set is reported. Here, we change it to the KNN evaluation protocol and the evaluation interval from every 10 epochs to every 5 epochs. The official code suggests choosing from three sets of augmentations (refer to the link for more details). We provide the results obtained from them in Table B1. We report the results under the "Random2" augmentation set in the main paper (Tab. 3) since it achieves the best results on two datasets.

Table B1: Frozen KNN evaluation results of GraphCL under different compositions of data augmentations. We report the sample-wise accuracy (%).

Augmentation Set	Detailed Augmentations	BIL-6	JML-4	ACT
Unsupervised pre-training on joint training sets.				
Random2	NodeDrop, Subgraph	**69.14**	54.29	**58.95**
Random3	NodeDrop, Subgraph, EdgePert	67.58	57.14	57.89
Random4	NodeDrop, Subgraph, EdgePert, AttrMask	67.58	**61.43**	55.79

B.3 Implementation details of data augmentation

Data augmentations in pre-training. A detailed overview of data augmentations is provided in Table B2. Most of the implementations can be inferred by names. Particularly, for (1) `RandomScaleCoordsBranches(p=0.2, scales=[0.8, 1.2])`, we sample a scale factor from `[scales[0], scales[1]]` uniformly, and multiply $[x, y, z; L, F_E]$ (refer to the last line in Sec. 3.1) with the scale factor; for (2) rotation, we uniformly sample three angles from $[0, 2\pi]$ for x, y, and z, respectively and do a rotation with those angles; for (3) and (4), shifting means the translations are isotropic — three shifting scalars are sampled for x, y, and z and added to all the point coordinates, while jittering means the translations are anisotropic — each point has its own shifting scalars. The (4) jittering process is: `jitters = np.clip(sigma * np.random.randn(coords.shape[0], 3), clip, clip); coords += jitters`; this process applies similarly to (6) but its targets are branches F_E. For flipping (6), the input will be flipped alongside the x and the y axes with 4 possibilities. For (7) deformation, a scale vector of the same shape as F_E will be uniformly sampled between the given range and be applied to F_E. For (8) random masking, a percentage of $(p * 100)\%$ features from F_E will be set to zero. For (9) to (11), please cross-reference to the descriptions in Section 3.3 (Data augmentation for neuron morphology representation learning), the illustrations in Figure 2 (d) and the python-style pseudo-codes in Algorithm 3.

[2] https://github.com/thomasaimondy/treestoolbox/tree/master/casia
[3] https://github.com/Shen-Lab/GraphCL/tree/master/unsupervised_TU

Table B2: Overview of data augmentations for neuron morphology representation learning.

Category	Target	Augmentations
(i) Point Transformation	Coordinates	1) `RandomScaleCoordsBranches(p=0.2, scales=[0.8, 1.2])`, 2) `RandomRotate(p=0.2)`, 3) `RandomShift(p=0.2, shift=[0.2, 0.2, 0.2])`, 4) `RandomJitter(p=0.2, sigma=1, clip=5)`, 5) `RandomFlip(p=1)`
(ii) Morphology	Branch feats	6) `RandomJitterBranches(p=0.2, sigma=0.1, clip=1)`, 7) `RandomDeformation(p=0.2, scales=[0.8, 1.2])`
(iii) Attribute Masking	Branch feats	8) `RandomMaskFeats(p=0.2)`
(iv) Topology	Edges	9) `RandomDropSubTrees(p=0.05, max_cnt=10)`, 10) `RandomSkipParentNode(p=0.05, max_cnt=10)`, 11) `RandomSwapSiblingSubTrees(p=0.05, max_cnt=10)`

Data augmentations in ablation study. In the ablation study on data augmentations (Sec. 4.3), we use the data augmentations in different categories individually or in pairs. Since the number of augmentations is reduced, we increase the applying probabilities of (6, 7, 8) in Table B2 from 0.2 to 0.5 to better inspect their influence. Point transformations' applying probabilities are not changed since they are a combination of five augmentations.

C Learning curves

Under our evaluation protocol, all methods' best test set performance is reported. However, the performance evaluated in small datasets like ours usually fluctuates. To get a complete view and the stability of different methods, we provide their performance curves here. Figure C1 shows the KNN test set accuracy during training (recorded every 5 epochs). The dashed lines of the same color denote the linear trend lines of different curves. Please refer to Appendix B.2-GraphCL and Table B1 for the meaning of "random2/3/4" in Figure C1. The sub-figures' y axes are aligned so that the curves can be directly compared.

Despite suffering variance, the overall performance in the BIL-6 dataset is improved over training in all four sub-figures (TreeMoCo and GraphCL). The performance fluctuation of JML-4 and ACT datasets is larger than that of BIL-6. The dataset size matters here since the number of test set samples would affect the granularity of accuracy intervals, exhibiting different degrees of variance. JML-4 and ACT datasets are much smaller than the BIL-6 dataset (about 1.2k/0.3k/0.3k samples for BIL-6, JML-4, and ACT datasets, respectively) and therefore suffer more performance variance. Collecting more data could alleviate this issue. Regarding stability, TreeMoCo and GraphCL perform less stable in the JML-4 and ACT datasets. However, TreeMoCo can still improve the accuracy in the JML-4 dataset over training. In contrast, GraphCL fails to improve that with longer training.

With observations from Figure C1, we emphasize the particularity of the neuron tree morphology learning problem and the datasets. Existing large-scale benchmarks in visual or lingual domains usually have abundant training/testing samples and thus can evaluate models precisely with minor variance. However, neuron morphology learning is a new ground to explore. Current insufficient testing samples might make the evaluation noisy and less accurate. We hope this early work could contribute to future exploring in designing a more reliable evaluation protocol.

D t-SNE plot of different self-supervised methods

We train models on the full dataset without "others" for 100 epochs to analyze and compare the embeddings learned by different self-supervised methods. The distribution of the same set of neurons in the embedding space is then visualized by projecting the encoded representations into a 2D space with t-SNE [42]. The visualization results are shown in Figures 1 and 4 and below (Figs D1 and D2).

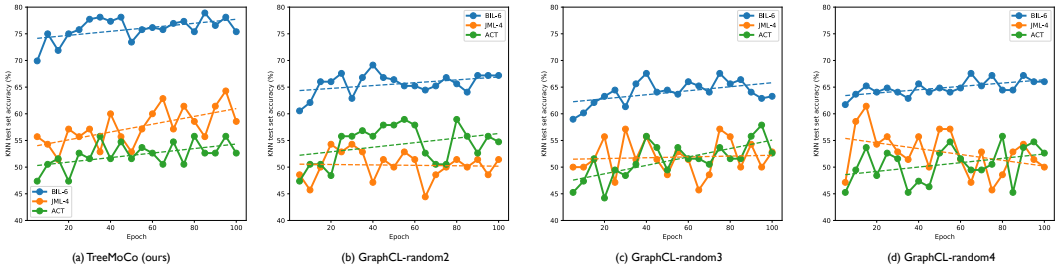


Figure C1: **KNN test set accuracy during training.** The dashed lines denote the linear trend lines.

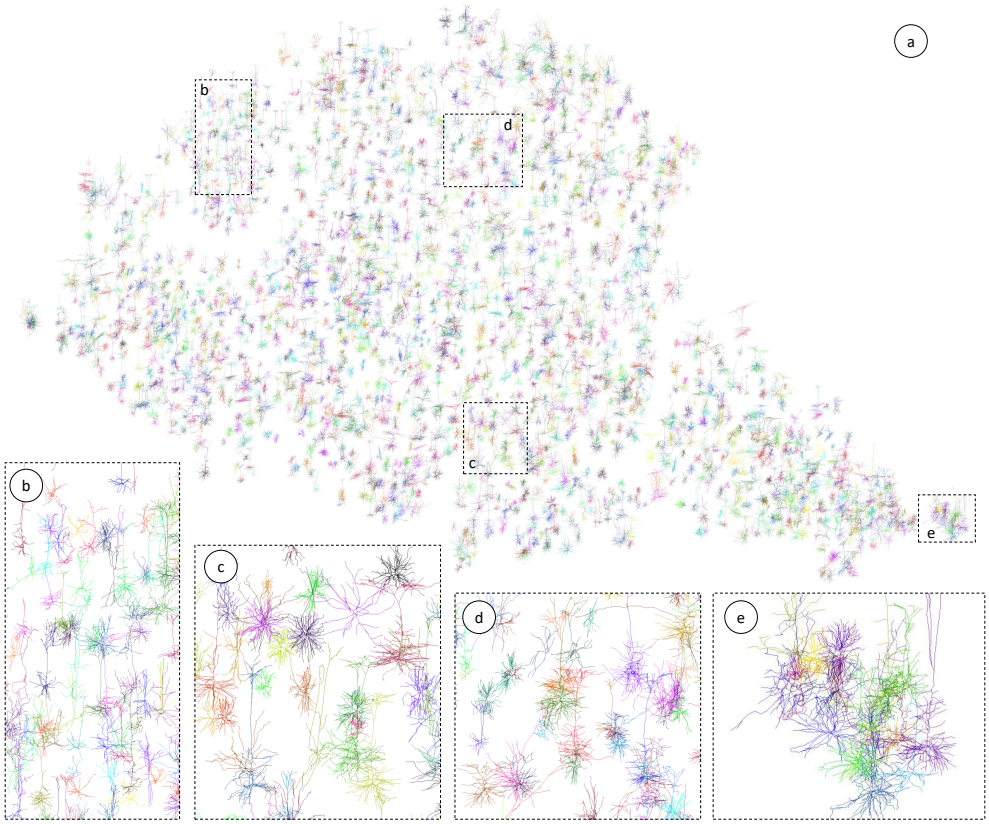


Figure D1: Distribution of neurons in the embedding space of MorphVAE visualized by t-SNE [42]. The thumbnails of dendrites are randomly colored. (b-e) Zoom-in view of regions in (a) showing the mixture of neurons with different morphologies.

D.1 t-SNE plot of MorphVAE

When training MorphVAE in a self-supervised fashion, the classification loss is set to zero, and only the reconstruction loss is computed. The resulted t-SNE plot is shown in Figure D1. It is obvious that MorphVAE largely fails to generate meaningful embedding to differentiate neuron morphology. This is in consensus with the observation in the original MorphVAE paper [24] that without a classification loss to train the encoder, MorphVAE's performance on downstream classification tasks will significantly degenerate.

D.2 t-SNE plot of GraphCL

For GraphCL, we can have similar observations as TreeMoCo in Figure 1 — neurons with different morphology can be largely separated in the embedding space. However, as shown in Figure D2(c-e),

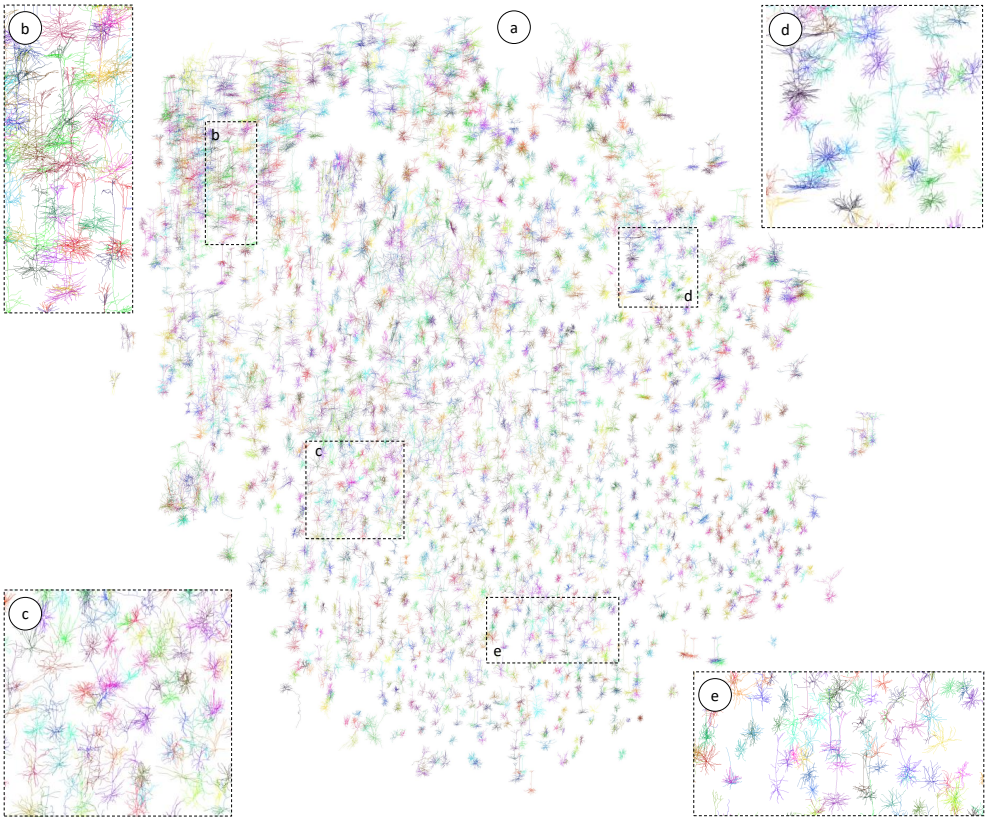


Figure D2: Distribution of neurons in the embedding space of GraphCL visualized by t-SNE [42]. The thumbnails of dendrites are randomly colored. (b-e) Zoom-in view of regions in (a). (c-e) shows the mixture of neurons with and without apical dendrite.

GraphCL's separation is not as clear as TreeMoCo's. Certain levels of the mixture of neurons with or without apical dendrite can be observed. Since GraphCL relies on local aggregation to generate graph representation, the unique global morphology patterns of apical dendrite could be ignored by it.

D.3 t-SNE plot of TreeMoCo

Figures D3 to D10 show the high-resolution version figures in Figure 4.

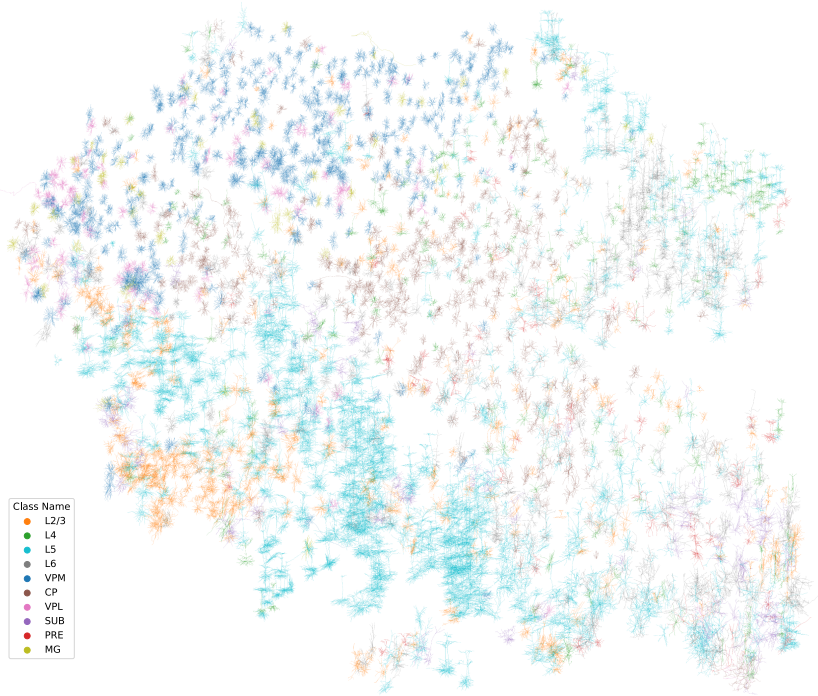


Figure D3: The t-SNE plot of neuron representations colored by class labels. A high-resolution figure for Figure 4-(a).

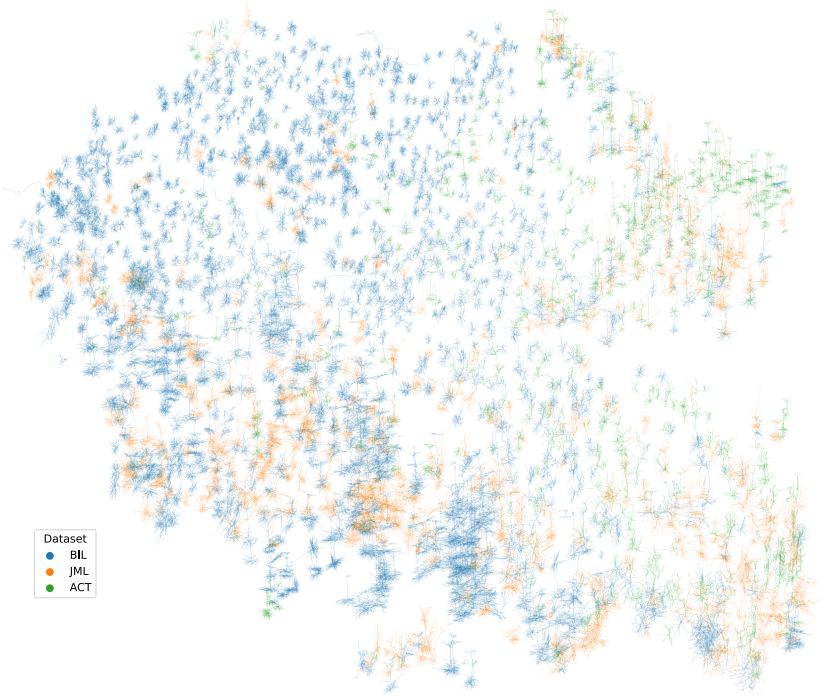


Figure D4: The t-SNE plot of neuron representations colored by dataset sources. A high-resolution figure for Figure 4-(b).

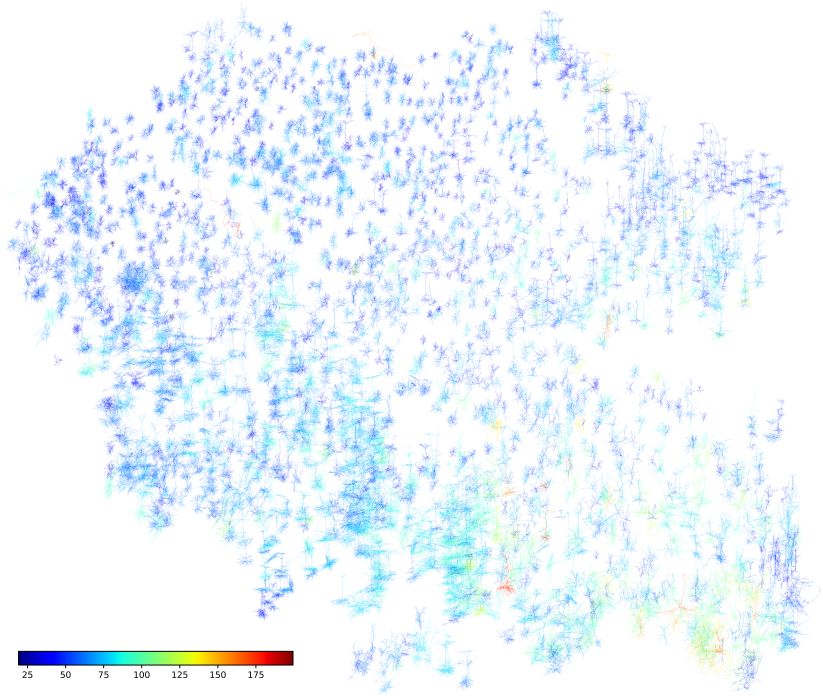


Figure D5: The t-SNE plot of neuron representations colored by the average length of each individual neuron. A high-resolution figure for Figure 4-(c).

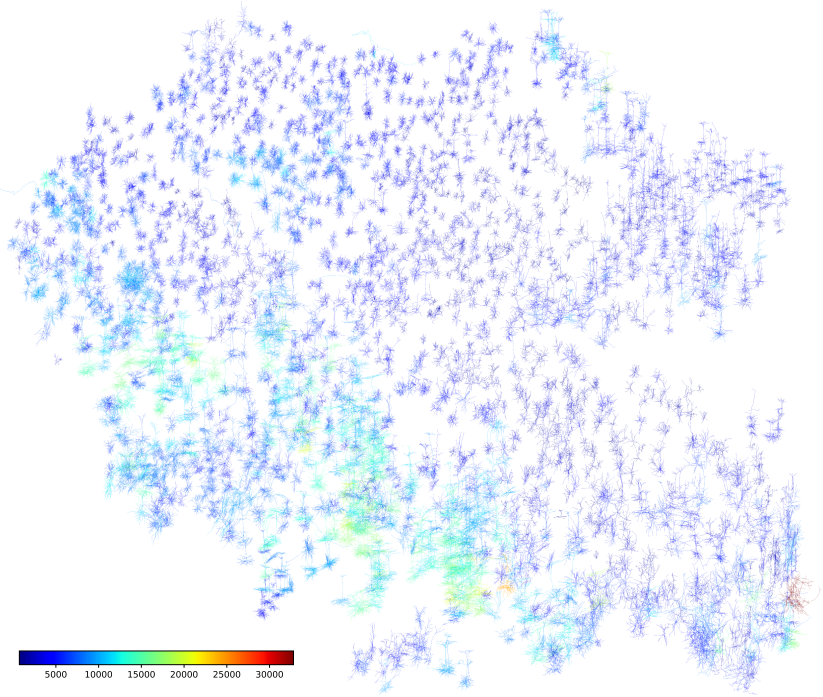


Figure D6: The t-SNE plot of neuron representations colored by the total length of each individual neuron. A high-resolution figure for Figure 4-(d).

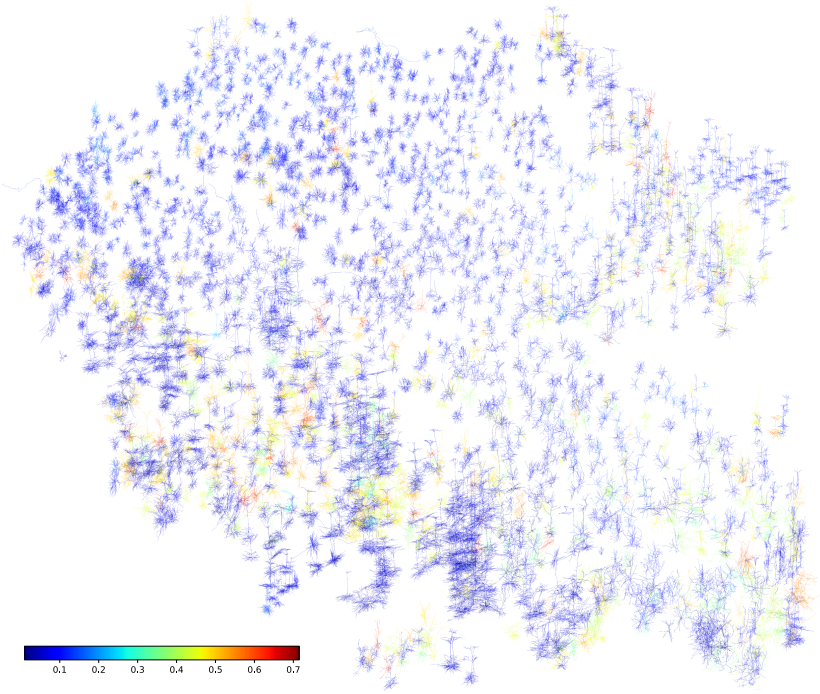


Figure D7: The t-SNE plot of neuron representations colored by the average contraction. A high-resolution figure for Figure 4-(e).

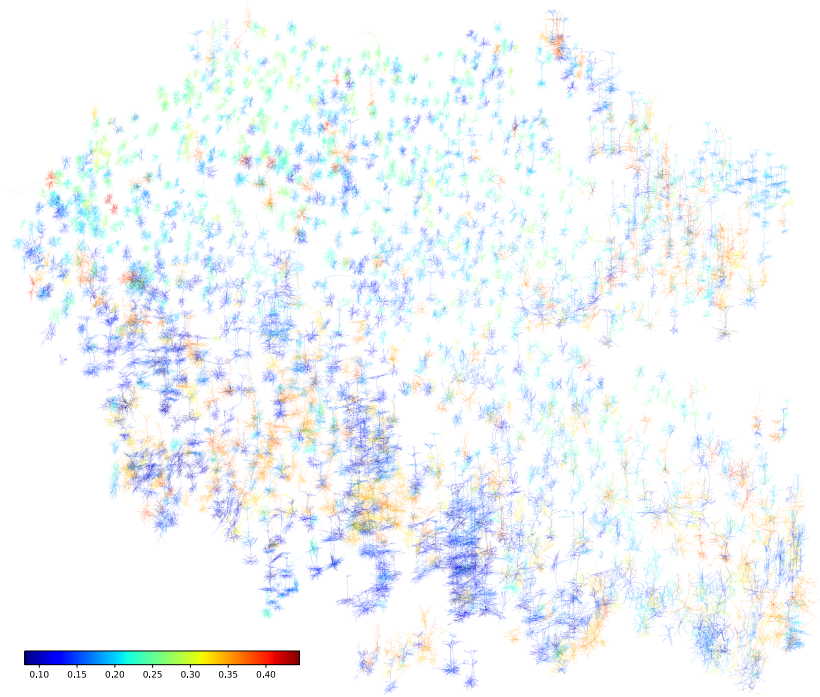


Figure D8: The t-SNE plot of neuron representations colored by the standard deviation of contraction. A high-resolution figure for Figure 4-(f).

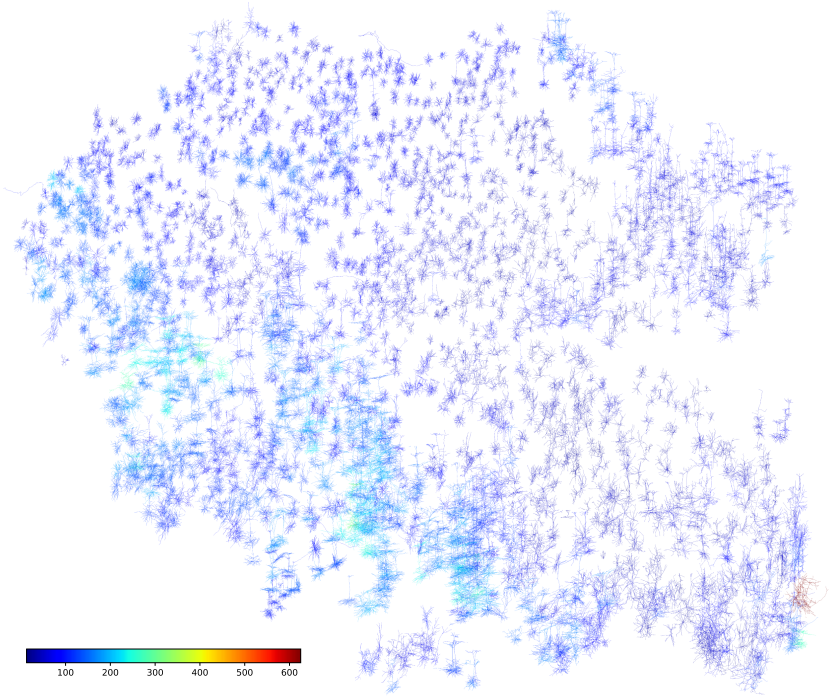


Figure D9: The t-SNE plot of neuron representations colored by the number of nodes inside each individual neuron. A high-resolution figure for Figure 4-(g).

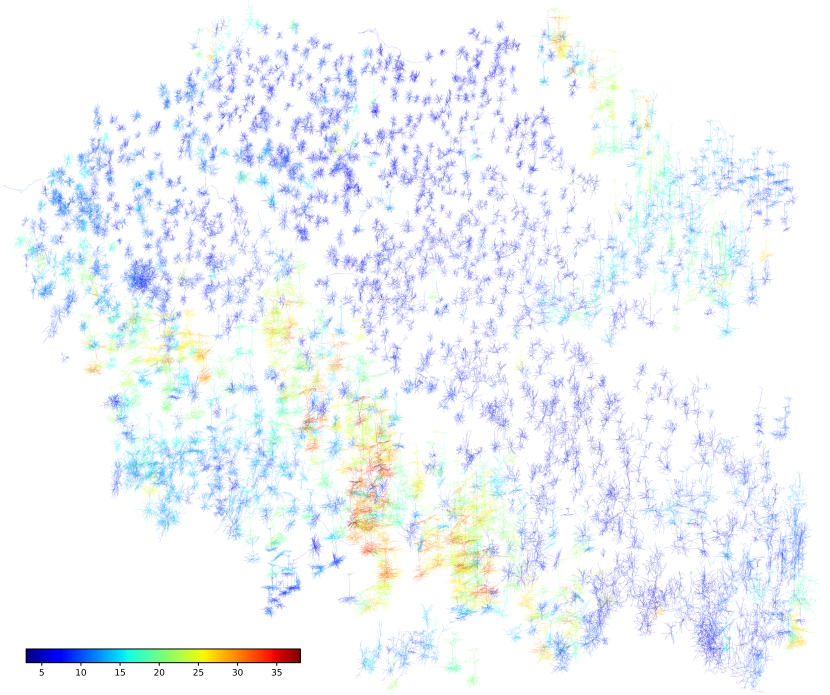


Figure D10: The t-SNE plot of neuron representations colored by the height of each individual neuron tree. A high-resolution figure for Figure 4-(h).

Algorithm 3 Topology augmentations: Python-like Pseudocodes

```
# p: applying probability
# max_cnt = 10: the truncation constraint.

cnt = 0 # cnt will be set to 0 at the first call.
def RandomDropSubTrees(self, root):
    reduced_root = Tree(root) # initialize a tree
    if len(root) == 0:
        return reduced_root
    p = np.random.uniform(0, 1, len(root))
    for child_idx in range(len(root)):
        if cnt > max_cnt: # no longger dropping
            reduced_root.append(root[child_idx])
            continue
        else:
            if p[child_idx] > p:
                reduced_root.append(RandomDropSubTrees(root[child_idx])) # recursively traverse
            else:
                cnt += 1 # not appending this child = dropping this child
    return reduced_root


cnt = 0 # cnt will be set to 0 at the first call.
def RandomSwapSiblingSubTrees(root):
    if len(root) < 2:
        return root
    p = np.random.uniform(0, 1, len(root))
    for child_idx in range(len(root)):
        if cnt >= max_cnt:
            break
        if p[child_idx] < p:
            if len(root[child_idx]) < 2:
                continue
            else:
                my_subtree_idx = random.randint(0, len(root[child_idx])-1)
                sibling_idx = random.randint(0, len(root)-1)
                if len(root[sibling_idx]) == 0:
                    continue
                sibling_subtree_idx = random.randint(0, len(root[sibling_idx])-1)
                my_subtree = root[child_idx][my_subtree_idx].copy()
                sibling_subtree = root[sibling_idx][sibling_subtree_idx].copy()
                root[child_idx][my_subtree_idx] = sibling_subtree
                root[sibling_idx][sibling_subtree_idx] = my_subtree
                cnt += 1
        else:
            root[child_idx] = RandomSwapSiblingSubTrees(root[child_idx])
    return root


cnt = 0 # cnt will be set to 0 at the first call.
def RandomSkipParentNode(root):
    if len(root) == 0 or len(root)==1:
        return root
    p = np.random.uniform(0, 1, len(root))
    for child_idx in range(len(root)):
        if cnt >= max_cnt:
            break
        if p[child_idx] < p:
            if len(root[child_idx]) == 0 or len(root[child_idx])==1:
                continue
            else:
                idx = random.randint(0, len(root[child_idx])-1) # picking a child node
                root[child_idx] = root[child_idx][idx] # replacing the parent node with the child node
                cnt += 1
        else:
            root[child_idx] = RandomSkipParentNode(root[child_idx]) # recursively traverse
    return root
```