TU WIEN Informatics

Visual Analytics für Deep Learning mit Graphen: Fallstudie zum Bündeln von Neuronen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Marie-Sophie Pichler, BSc
Matrikelnummer 11933235

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing.in Dr.in techn. Astrid Berg
Dipl. math.in Dr.in techn. Katja Bühler

Wien, 8. September 2024

Marie-Sophie Pichler | Eduard Gröller

TU WIEN Informatics

Visual Analytics for Graph Deep Learning: Case Study Neuron Clustering

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Marie-Sophie Pichler, BSc
Registration Number 11933235

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing.in Dr.in techn. Astrid Berg
Dipl. math.in Dr.in techn. Katja Bühler

Vienna, 8th September, 2024

Marie-Sophie Pichler | Eduard Gröller

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Marie-Sophie Pichler, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. September 2024

Marie-Sophie Pichler

Danksagung

An erster Stelle danke ich meiner großartigen Familie für die unerschöpfliche Unterstützung während meiner beruflichen Orientierung und Ausbildung sowie in allen anderen Lebensbereichen.

Ich danke meinen Betreuern Astrid Berg, Katja Bühler und Eduard Gröller für die umfangreiche Betreuung meiner Arbeit. Ein besonderer Dank gilt meiner Betreuerin Astrid Berg für die vielen Stunden, die in die Ausarbeitung des Themas geflossen sind, sowie für die sofortige Präsenz wann immer ich Fragen hatte. Du warst mein Vorbild in Bezug auf wissenschaftliche Arbeit und Zusammenarbeit. Danke auch an Katja Bühler für die Aufnahme am VRVis, ihre thematischen Anregungen und das Korrekturlesen. Und ein herzliches Dankeschön an Herrn Gröller für die sorgfältige Korrektur der Arbeit und die Ermutigungen und Positivität.

Vielen Dank auch dem tollen VRVis Team für die Wissenstransferangebote und die Integration. Ich will die Gelegenheit nutzen, auch Thomas Torsney-Weir zu danken, der mir die Graphdaten der Drosophila-Melanogaster-Larve bereitgestellt hat und mich ermutigt hat, in Richtung Visual Analytics zu gehen sowie Markus Töpfer für seine Hilfestellung bei Fragen zu der technischer Umsetzung des User Interfaces.

Diese Arbeit wurde durch das Kompetenzzentrum VRVis ermöglicht und das Projekt I 4836 des Österreichischen Wissenschaftsfonds (FWF) gefördert. VRVis wird von BMK, BMAW, Steiermark, SFG, Tirol und der Wirtschaftsagentur Wien im Rahmen von COMET - Competence Centers for Excellent Technologies (879730) gefördert, das von der FFG geleitet wird.

Acknowledgements

First and foremost, I would like to thank my wonderful family for their inexhaustible support during my professional orientation and education, as well as in all other areas of my life.

I would like to thank my supervisors Astrid Berg, Katja Bühler and Eduard Gröller for their extensive supervision of my thesis. Special thanks go to my supervisor Astrid Berg for the many hours she spent working on the topic and for her immediate presence whenever I had questions. You were my role model in terms of scientific work and cooperation. Thanks as well to Katja Bühler for welcoming me at VRVis, her thematic suggestions and proofreading. And a big thank you to Mr. Gröller for the thorough correction of the thesis and the encouragement and positivity.

Many thanks also to the great VRVis team for the knowledge transfer offers and the integration. I would also like to take this opportunity to thank Thomas Torsney-Weir, who provided me with the graph data of the Drosophila melanogaster larva and encouraged me to move towards visual analytics, and Markus Töpfer for his help with questions regarding the technical implementation of the user interface.

This work was enabled by the Competence Centre VRVis and funded by the Austrian Science Fund (FWF) Project: I 4836. VRVis is funded by BMK, BMAW, Styria, SFG, Tyrol and Vienna Business Agency in the scope of COMET - Competence Centers for Excellent Technologies (879730) which is managed by FFG.

Kurzfassung

Viele Deep-Learning-Anwendungen basieren auf Graphdaten, um Beziehungen oder Strukturen zu analysieren. Das Annotieren dieser Daten ist teuer und erfordert oft Expertenwissen. Im Kontext von Graphen-Bündelung erzeugt die Stand-der-Technik-Methode GraphDINO Graphen-Repräsentationen durch Selbst-Überwachung und bündelt diese in einem nachgelagerten Prozess. Wir beobachten an einem besonders anspruchsvollen Neuronendatensatz, dass diese Methode nicht zu zufriedenstellenden Bündelungsergebnissen führt. Daher verwenden wir die von GraphDINO erzeugten Graphen-Repräsentationen als Ausgangspunkt um die Modell-Architektur und das Modell-Training zu verbessern. Dazu haben wir das Visual-Analytics-Framework *NetDive* entwickelt. Die Benutzer*in kann die Graphen-Repräsentationen analysieren und einzelne Neuronen kennzeichnen, die falsch zugeordnet sind. Diese Annotationen werden daraufhin zum Trainieren eines semi-überwachten Modells verwendet. Wir haben die Netzwerk-Architektur *GraphPAWS* entwickelt, welche die Annotationen verarbeitet. *GraphPAWS* enthält Komponenten von GraphDINO und der semi-überwachten Netzwerk-Architektur *PAWS*. Das Modell-Training kann aus der Visual-Analytics-Anwendung NetDive heraus gestartet werden, und die resultierenden Graphen-Repräsentationen sind in NetDive verfügbar, sobald das Training abgeschlossen ist. Wir demonstrieren, wie wir das Modell mit NetDive und GraphPAWS iterativ trainieren, und vergleichen die Ergebnisse unseres Modells mit den Ergebnissen mit dem selbst-überwachten Stand der Technik für unseren Datensatz.

Abstract

Many deep learning applications are based on graph data in order to explore relationships or to analyze structures. Labeling this data is expensive and often requires expert knowledge. For the application of graph clustering to neuron data, the SOTA method GraphDINO generates self-supervised graph embeddings combined with the downstream task of clustering these embeddings. We observe on a particularly challenging neuron dataset that this method does not lead to satisfying clustering results. Therefore we use the graph embeddings generated by GraphDINO as an initial starting point to improve the network and to guide the network training. To achieve this, we developed the visual analytics framework *NetDive*. The user can analyze the graph embeddings and label single neurons that are falsely clustered. This annotation information is then used to train a semi-supervised model. To this end, we developed a network architecture, named *GraphPAWS*, that assembles components of GraphDINO and of the semi-supervised network architecture *PAWS*. The model training can be started from within the visual analytics application NetDive and the resulting graph embeddings are available in NetDive as soon as the retraining is completed. We demonstrate how we iteratively train the model using NetDive and GraphPAWS and evaluate our model against the self-supervised SOTA for our dataset.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Many deep learning applications are based on graph data in order to explore relationships or to analyze structures. The use case investigated in this thesis is to cluster unlabeled spatial graph data that represents drosophila melanogaster larval Level 1 neurons to reproduce meaningful cell types. This use case poses the challenge of clustering data without initially having labels to train the deep learning model with a supervised objective function. Furthermore we initially do not know what the network should learn, but we want to be able to steer the training while gathering new knowledge about the resulting clusters. We disassemble and describe the use case in the following paragraphs:

Unlabeled Data

Training techniques that do not depend on labeled data are an interesting alternative to supervised learning, as depending on the label type and the domain, labels can require expert knowledge, can be costly, and can be noisy and biased. The network either learns to extract information from the input data distribution or the network developer can steer the network training using self-supervised learning strategies like contrastive learning, which derive supervisory signals from the input data to guide the learning process. Contrastive learning aims to ignore specific data features while learning representations based on the remaining features themselves. This task to guide the training with domain knowledge encoded in the network architecture is cheaper than generating labels for supervised learning in some cases. At the same time it is difficult to improve the self-supervised trained network performance, if the network does not output the expected results. While supervised learning can often be improved by increasing the amount of training data (while ensuring that the training data is not biased), improving a contrastive learning setup requires an analysis of the input data and domain knowledge

in order to identify the features that should be irrelevant to embed an input sample as expected and to further process the input sample in a downstream task.

Biological Motivation

The biological motivation for the thesis roots in the objective of neuroscience to understand the correlation between nerve cells, also named neurons, and behavior [Nat89, mw:, WPB+23]. One step to meet this objective is to assign a *cell type* to each neuron correlating to functionalities in the brain. The European Human Brain Project and the American BRAIN initiatives announced cell type classification to be one of their highest priorities in order *to complete a comprehensive cell census of the human brain* [AA15].

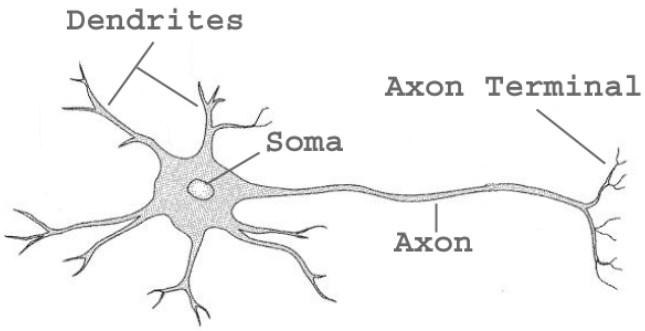


Figure 1.1: The compartments of a nerve cell [SMS09].

Cell Types

The cell type is an annotation that a neuron receives based on predefined features. However, the feature selection that specifies the classification varies and therefore no universal definition of a cell type exists. Despite this lack of an unambiguous specification, cell types are fundamental to reveal organizational patterns. Key features are morphology, genetic markers, the neuron position within the nervous system, connectivity and intrinsic electrophysiological signatures [CMO+16]. In this thesis we cluster the neurons solely based on their morphology and aim to find correlations to meaningful cell type assignments. The morphology of a prototypical neuron is depicted in figure 1.1. The figure also displays the compartments that compose the neuron, namely the *soma*, the *axon*, and *dendrites*. The soma is the cell body of the neuron and from there branches reach out to transport information. The dendrites receive information and transport it to the soma. The branch forwarding information is called axon. Neurons are not physically connected with each other. Instead the information is released in form of neurotransmitters into the synaptic gap between neurons from where multiple neighbouring neurons can collect the information.

Cell Type History

Cell type annotations were initially performed using visual criteria by experts [DLCBP+13]. Nowadays, due to the complexity and magnitude of the neuron data that is available, manual cell type assignment is infeasible [SESM+20, WHLE21] and shows a high variance of annotations depending on the expert [DLCBP+13].

This led to algorithms like the segment-wise similarity comparison of neurons implemented by NBLAST [CMO+16] and to machine learning solutions based on predefined features, e.g., diameter, length, and angles of the neurons [SPA08], and node statistics [Ham20]. The features and statistics are processed using kernel functions, e.g., the Weisfeiler-Lehman kernel [Ham20]. Yet, predefined features encode biases and are often not adaptable to other species [WHLE21]. Deep learning models on the other hand can be adapted using transfer learning and model refinement and can find correlations using feature learning that are not intuitive to experts.

Drosophila Melanogaster

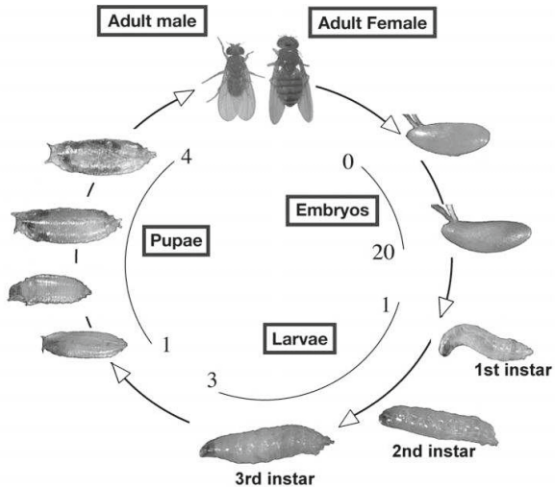


Figure 1.2: The four stages of the drosophila melanogaster life cycle: embryo, larva, pupa and adult [FMFKG07].

Figure 1.2 depicts the four stages of the drosophila melanogaster. The larval stage is divided in three morphological stages, titled Level 1 (L1), Level 2 (L2), and Level 3 (L3). The brain structures between the stages remain homologous and the neuronal connectivity stays almost stable within the larval stages [WPB+23].

The drosophila melanogaster, commonly known as fruit fly, is a simple organism compared to humans [MP19, WPB+23]. A human brain contains around 86 billion neurons, while the drosophila melanogaster carries approximately 250 thousand neurons. This number reduces to 8000 neurons in the L3 drosophila melanogaster larval brain and to 3000 in the earlier L1 drosophila melanogaster larval stage. Also the number of connections

varies drastically between species and between morphological stages. At the same time 60 percent of the drosophila melanogaster genes correlate to human genes and for genes known to cause health issues the percentage goes up to 90. Exploring the biological structure of the drosophila melanogaster therefore is of great interest for science. Other reasons are the short reproduction cycle of the drosophila melanogaster and better conditions for species-appropriate keeping compared to the keeping of other animals for scientific reasons [mp].

Figure 1.3 depicts three rendered graph representations of drosophila melanogaster larval neurons, taken from `www.larvalbrain.org`.

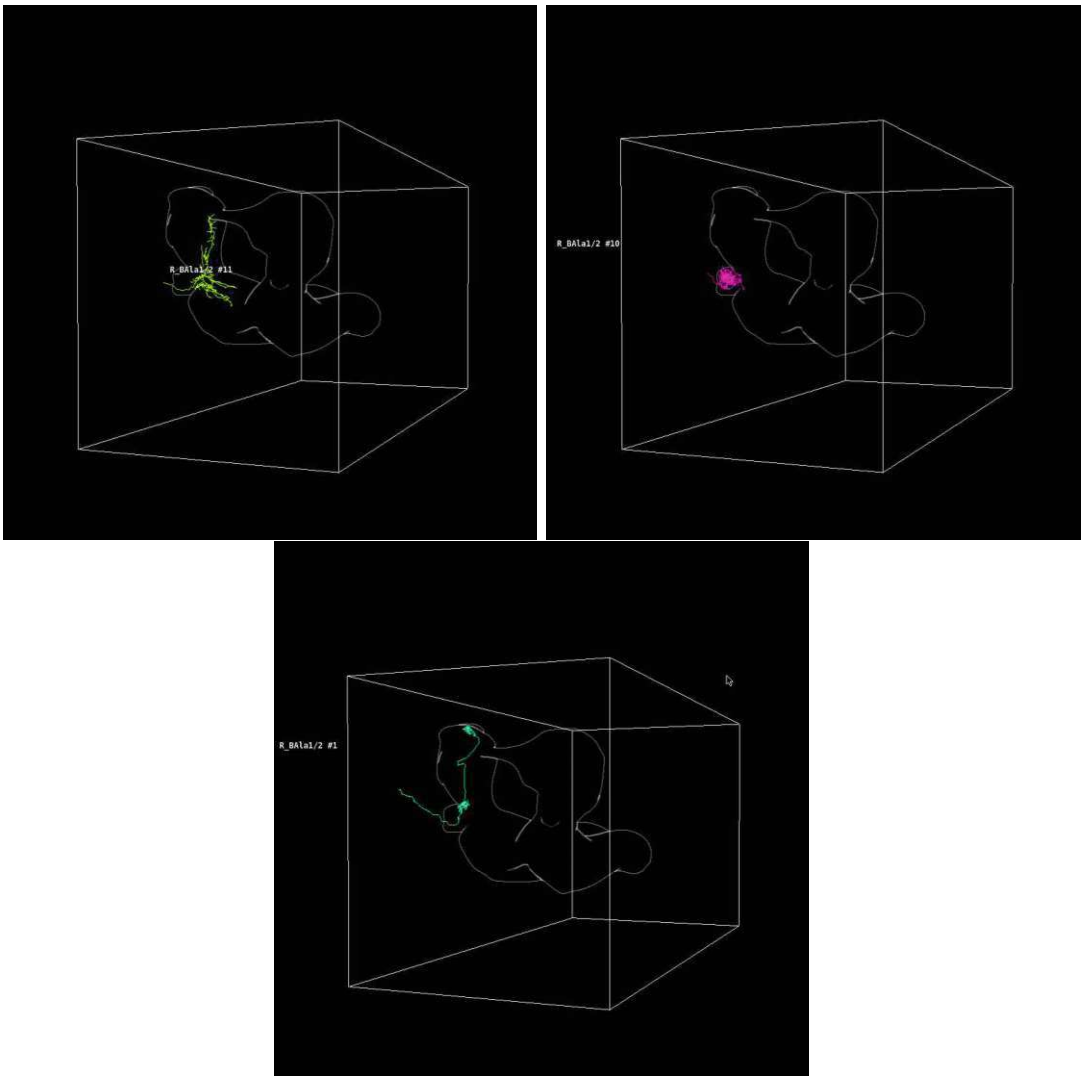


Figure 1.3: Rendered graph representation of three drosophila melanogaster larval neurons. The images depict screenshots from `www.larvalbrain.org`.

The drosophila melanogaster graphs that form the basis of the thesis are unregistered. The process of registering data entails embedding the data in a standardized coordinate system. In the case of the drosophila melanogaster larval neurons, it describes the process of mapping neurons to a standardized brain volume, to discard the shape variance of different specimen of drosophila melanogaster larvals that were used to trace the neuron graphs. Registered data is often easier to process, however, registered data is not always available.

1.2 Aim of the Thesis

The aim of the thesis is to train a network on the drosophila melanogaster neuron graph representation data and to cluster the network latent embeddings, such that the clusters represent cell types. We solely train on the graph morphology, i.e., we dismiss compartment information encoded in the neuron graphs and we do not have a ground truth available.

Therefore we defined the problem statement of improving the network training for graph data with initially no labeled data by iteratively guiding the training. Our use case is applicable to a broader range of graph clustering tasks. The approach we take in this thesis addresses the issue of incrementally improving a model based on new knowledge about the underlying data. This approach is data type agnostic and therefore applicable also to representations of other input data types.

1.3 Contribution

To improve the results of self-supervised graph clustering without performing an exhaustive analysis of the graph input data, we investigate how to iteratively improve the model. We start out by training on our dataset with self-supervised training and improve the results by adding labels for data subsets and train a semi-supervised model that processes the labels.

We adapt the self-supervised deep learning architecture GraphDINO [WPLE23], discussed in Section 3.3.1, and the semi-supervised deep learning architecture PAWS [ACM+21], discussed in Section 3.3.2. GraphDINO is implemented to cluster the neuron graph representations of mice and rats, namely the Allen Brain Atlas (ABA) dataset and respectively the Blue Brain Project (BBP) dataset. PAWS is a semi-supervised architecture that computes a similarity score between the latent embeddings of input samples and provided support samples. We develop a semi-supervised network architecture *GraphPAWS* that adopts the graph encoder of the self-supervised deep learning architecture GraphDINO and the processing of support samples of the semi-supervised deep learning architecture PAWS.

The resulting graph latent embeddings are visualized in a visual analytics (VA) web application we title *NetDive* that we developed to analyze the embeddings, to iteratively

add new support samples if needed and to retrain a model with this new information. We evaluate our method analytically on a manually labeled subset of the drosophila melanogaster dataset that we get from the *collaborative annotation toolkit for massive amounts of image data* (CATMAID) [SCHT09]. We combine the analytical evaluation with visual inspection and comparative analysis enabled by the VA application.

1.4 Outline

The thesis broadly divides into the topics deep learning and visual analytics. Chapter 2 covers related work in the fields of deep learning and visual analytics (VA). Chapter 3 first discusses our methodology to set up the pipeline including the network architecture GraphPAWS to train the graph latent embeddings. The chapter follows up with the presentation of the visual analytics web application NetDive, developed to analyse the training results and to start the retraining with the new support samples.

Chapter 4 focuses on implementation details, regarding data preparation, the ground truth preparation, GraphPAWS and NetDive, and regarding the concept of interchanging information between GraphPAWS and NetDive. Chapter 5 documents the data subsets, i.e., specific lineages, we use for training and the experiments we conducted with self-supervised or semi-supervised training. The chapter concludes with a demonstration of NetDive to incrementally improve the network performance. Chapter 6 discusses the results of the previously performed experiments and Chapter 7 provides a summary of the findings in this thesis and an outlook on improvements and future work regarding visual analytics and deep learning for graph clustering.

CHAPTER 2

Background and Related Work

This chapter is dedicated to the background and related work regarding the concepts throughout the thesis. Section 2.1 covers topics regarding machine learning: Sub-section 2.1.1 provides an overview of deep learning components and terminology, as well as related work regarding regularization. Sub-section 2.1.2 discusses clustering algorithms, as varying algorithms can produce different results, which is relevant to consider for the cell type clustering. Sub-section 2.1.3 looks at state of the art research about supervised, self-supervised, and semi-supervised learning. Sub-section 2.1.4 covers graph neural networks and developments in the field and functionality of transformer networks.

Section 2.2 covers visual analytics (VA) topics relevant to this thesis. Sub-section 2.2.1 explains the concept of VA applications and Sub-section 2.2.2 highlights VA user interactions. Sub-section 2.2.3 discusses applications of VA in deep learning, in particular visualization of embeddings.

2.1 Machine Learning

AI was defined by Barr & Feigenbaum in 1981 as follows: "(AI) is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior — understanding language, learning, reasoning, solving problems, and so on" [BF81]. Figure 2.1 shows the relation between *machine learning* and *deep learning*. Deep learning is embedded in machine learning and both are types of *artificial intelligence* (AI). Machine learning includes classical algorithms like support vector machines, decision trees and the k-nearest neighbor algorithm. Another implementation of machine learning are artificial neural networks, that are titled deep neural network if the networks consist of multiple layers.

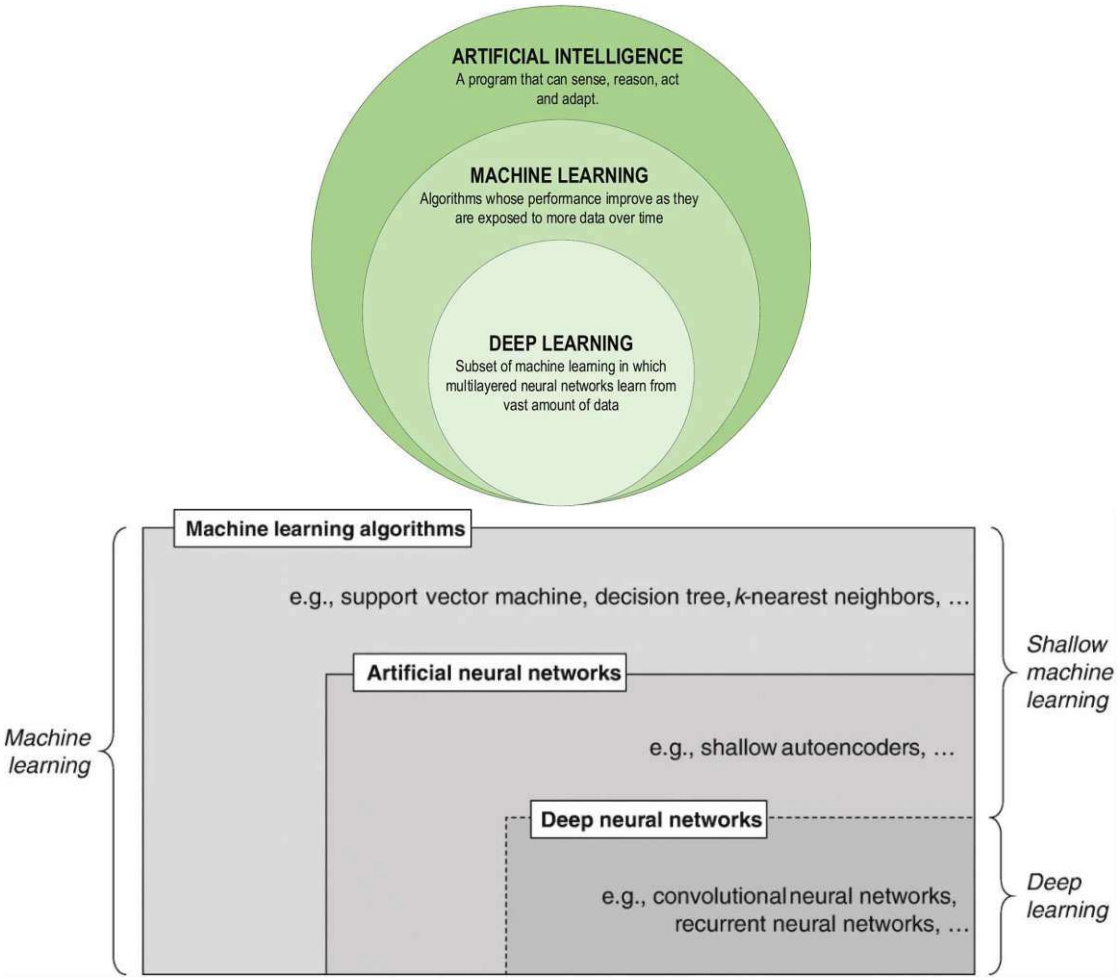


Figure 2.1: The relationship between artificial intelligence, machine learning, and deep learning [AZH+21] (left). Technologies associated with machine learning and deep learning [JZH21] (right)

2.1.1 Deep Learning

Deep learning was first mentioned in the 1940s, had its first big application in speech recognition after the GPU was developed by NVIDIA in 1999 [LBH15], and has been increasingly referenced in paper submissions since the 2010s [HKPC19] with a breakthrough in 2012, when the deep learning model AlexNet won the *ImageNet Large-Scale Visual Recognition Challenge* (ILSVRC) [KSH17] against hand-designed systems. Many application fields evolved for deep learning, including visual object recognition, object detection, speech recognition, and graph embeddings.

Deep learning is implemented with computational models consisting of an input layer, an output layer and hidden layers that learn a representation of data. Each layer has

neurons, representing computational units.

The number of hidden layers sets deep learning networks apart from artificial neural networks. If a network consists of multiple hidden layers with multiple neurons on each layer, the network is called *deep*.

Each neuron is represented by learnable parameters, which are coefficients of a linear hyperplane representing the decision boundary. Parameters are composed by multiple weights (w) and a bias (b). The parameters are applied to the input data (x) as a weighted sum (z), with N being the number of connected input neurons:

$$z = \sum_{i=1}^{N} w_i * x_i + b. \tag{2.1}$$

The *activation function* defines how significant z is for the prediction. Activation functions are grouped in linear and non-linear activation functions. The first artificial neural network was the Rosenblatt Perceptron, introduced 1969 by M. Minsky and S. Papert [MP69]. It includes an input layer, an output layer and a single hidden layer. The activation function of the Rosenblatt perceptron is linear and therefore can only divide the input space in two half-spaces separated by a hyperplane [LBH15]. It took 17 years until the Rosenblatt perceptron was revolutionized by changing the activation function to a non-linear function in 1986 [RHW86]. The substitution of the activation function made an expansion of the single perceptron to a multiperceptron possible. The features learned by the network neurons in a multiperceptron are titled *low level features* in the first layers, which are combined to *high level features* in the subsequent layers.

The features are learned by iterative parameter updates. The parameters are usually updated using *backpropagation* [LBH15]. After the network processes the input data, a predefined *optimization function* is optimized and produces a surface in $\mathbb{R}^n$, with n being the number of parameters. The *optimization function* is also refered to as *error function*, *loss function*, or *training objective*. The surface represents a comparison of the predicted network outputs and the expected network outputs. Backpropagation attempts to update the weights in the negative direction of the steepest gradient, such that the network learns to find the global minimum of the optimization surface.

A common issue with deep learning training is *overfitting*. To avoid this, regularization techniques are applied that support the generalization of the network. The optimal regularization depends on the model architecture and input data. Dropout layers, for example, randomly set a fraction of the neural network units to zero during training and batch normalization normalizes layer outputs within a neural network. Early stopping terminates the training if the training loss increases [SK23]. Another regularization technique that is intrinsic to contrastive learning, discussed in Sub-section 2.1.3 and throughout the thesis, are augmentations.

We will also look at regularization terms that are added to the optimization function in this thesis. Popular terms are the L1 loss, which introduces a penalty based on the

absolute values of the weights, encouraging sparsity by pushing some weights towards zero and the L2 loss, which adds the square of the weights to the objective function to penalize high weights [DKKAK11]. Other regularization terms are based on the output entropy. This will be discussed in more detail in the following sections. Peer et al. [ACM+21] also implement batch-entropy regularization. They apply it to address the *degradation* problem, in which increasing the number of layers leads to a worse generalization.

Finding the optimal model architecture and hyperparameters is an iterative trial-and-error process [PHG+18]. Recent research in the field of *neural architecture research* aims to automate the design of neural networks.

2.1.2 Clustering

Clustering is a technique to separate data in groups of high similarity based on specific properties. Clustering is applied in varying fields, including image analysis, pattern recognition, statistics and graph theory [KGH+21].

Exclusive clustering, also called *hard clustering*, is a separation without overlapping, whilst overlapping clustering techniques, also called *soft clustering*, allow data points to be mapped to more than one cluster. If data points are overlapping they have a membership degree assigned, that has a value between 0 and 1 representing the percentage with which the data point belongs to a specific cluster [BG14].

This thesis focuses on exclusive clustering for the case study of neuron correspondences as each neuron should get one cell type assigned. Clustering algorithms can be divided into five categories: *hierarchical clustering*, *partitional clustering*, *density-based clustering*, *grid-based clustering*, and *model-based clustering* [DW22].

Depending on the cluster algorithm and the parametrisation of the chosen algorithm the cluster sets can significantly vary [KGH+21]. *Scikit-learn* [scia], a popular machine learning library in Python, offers a comparison of different clustering algorithms on their documentation website, referenced in Figure 2.2 that depicts how the choice of algorithm affects the cluster results.

Hierarchical clustering aims to build a hierarchy of clusters, commonly visualized with a tree-like structure [KGH+21], e.g., with a *dendrogram*. Relationships between hierarchy levels in a dendrogram are visualized with edges that depict which clusters on a lower level are merged on a higher level. Hierarchical clustering does not require the user to specify the number of clusters upfront, which can be useful in situations where the number of clusters is not known. While the algorithm can handle both continuous and categorical data, we will use categorical data to assign cell types to neurons. Hierarchical clustering is a good choice for exploratory data analysis, as it allows the user to easily identify patterns and relationships in the data on different granularity levels. Michael Winding [WPB+23] does apply hierarchical cell type clustering to the drosophila melanogaster neurons and the publication NBLAST also applies hierarchical clustering for cell type clustering [CMO+16].

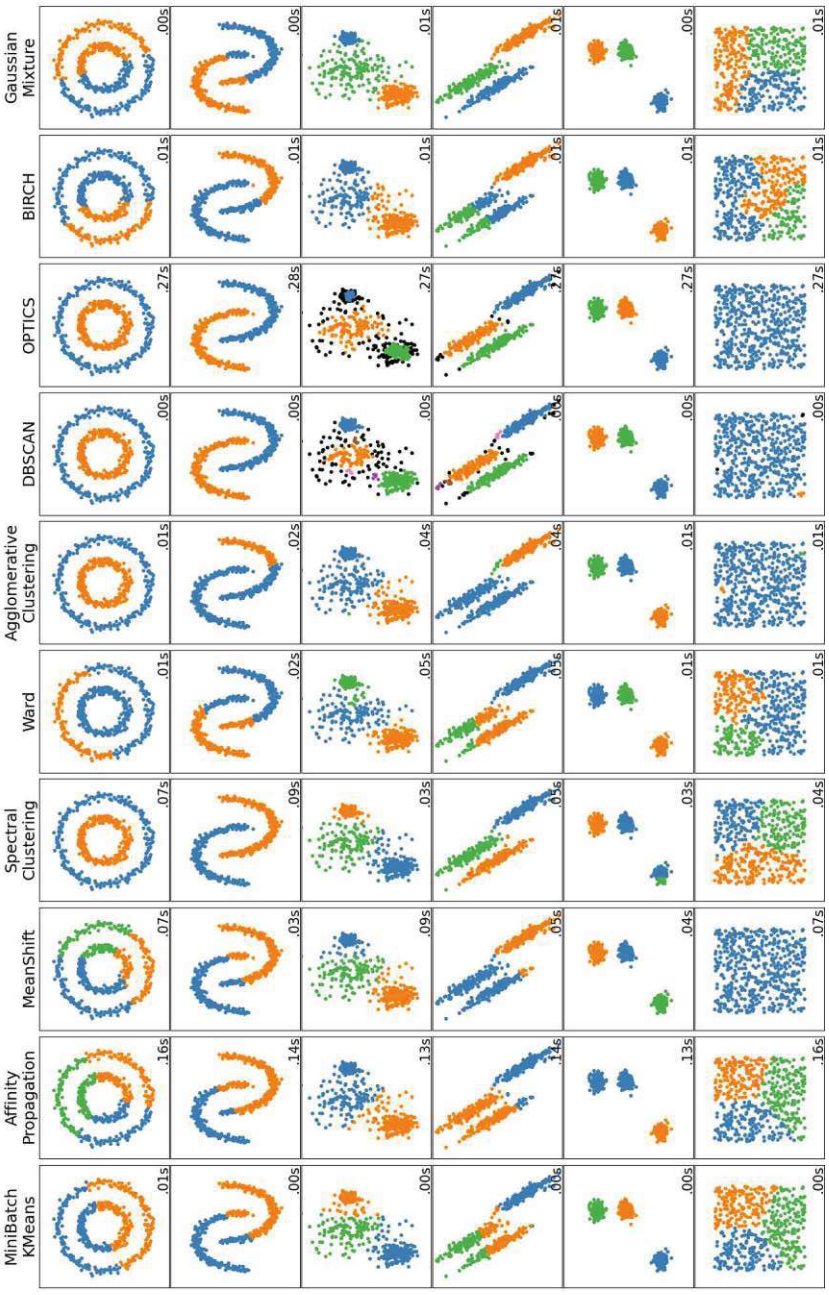


Figure 2.2: *Scikit-learn*, a popular machine learning library in Python, provides a comparison of different clustering algorithms in the online user guide under sub-section 2.3.1 [scia].

Partitional clustering algorithms are hard clustering algorithms based on the minimization of an objective function [KGH+21]. A prominent example for partitional algorithms is *k-means*.

The k-means algorithm partitions N data points into k clusters. Initially, k centroids are chosen randomly, and then iteratively adjusted with the objective to minimize the distances between each data point and its closest centroid. Each iteration consists of two steps: the *assignment step* and the *update step*. During the assignment step, each data point is assigned to its nearest centroid based on a distance metric, originally based on the Euclidean distance. All data points assigned to the same centroid constitute a cluster. In the subsequent update step, the centroids are recalculated by calculating the mean over the data points within their respective cluster.

The algorithm terminates after a predefined number of iterations or after the objective function converges. The original objective function that builds on the Euclidean distance measure is called sum-of-squared error (SSE). Changing the objective function leads to significant shifts in clustering behavior [FS19].

Scikit-learn optionally combines the Euclidean distance measurement with weighing by sample weights [scia]. Other commonly used objective functions are built on the Manhattan distance, the Chebyshev distance, the Minkowski distance, the Mahalanobis distance, the angle cosine and the correlation coefficient. Also new distance measurements are introduced, like the view-distance in 2022 for high-dimensional data spaces [ZL22].

The following limitations of k-means are under the assumption, that an Euclidean distance is used, which is relevant for us as we use the scikit-learn k-means implementation: If the cluster is non-circular and intersecting with another shape it is very likely that k-means clustering fails. The density of shapes is not considered by k-means, such that clusters that vary in density can not be distinguished.

An example for failing k-means is depicted in Figure 2.2 at the top of the first column. The example fails, because the Euclidean distances between the centers and the data points do not represent the patterns.

Model-based clustering refers to statistical techniques to cluster data [BS10]. An example for model-based clustering are Gaussian Mixture Models (GMMs). While a GMM has a higher computational cost than k-means clustering, it can reveal more fine-grained patterns. K-means clusters data points based on the distance to the centroids and GMM divides the data points into the underlying Gaussian distributions. Figure 2.2 depicts in Row 4, that k-means can not find the elongated clusters while GMM succeeds.

2.1.3 Learning Strategies

Various learning strategies address different prerequisites regarding the available training data and the task.

Representation Learning

Representation learning describes the encoding of input data into a meaningful representation at a lower dimension. The attribute *meaningful* sets representation learning apart from other dimensionality reduction algorithms that do not learn a representation that generalizes to new data samples. Phuc Le-Khac et al. [LKHS20] state that the core principles of a good representation include *abstraction*, referring to deriving abstract concepts from input data, *invariance*, referring to invariance towards small changes in the input data, *distribution*, referring to a wide spectrum of representations that can be expressed with the representation, and *disentanglement*, referring to a separation of contributing features in the representation. The representations can be used for downstream tasks, such as classification and regression. Representation learning differs from traditional machine learning techniques that required expensive and inflexible engineering of hand-crafted features [ZCH+20].

Supervised Learning

Supervised learning is a machine learning technique, used to train a model to make predictions or decisions based on labeled training data. The training data consists of a set of input samples and their associated outputs. The model aims to learn a function that maps the input samples to the expected outputs. The performance measure is encoded in the optimization function that compares the expected output with the predicted output [GBC16].

Unsupervised Learning

Unsupervised learning is used to train a model to discover patterns or relationships in a dataset without requiring labeled training data. The advantage of using learning strategies that are independent of labels is, that labels are often not available or expensive to generate. Furthermore, labels can be noisy or biased. For example, an image showing both a cat and a dog might only have a label for the cat. The model learns to find relationships and structures in the data itself. One use case for unsupervised learning is the reconstruction of the entire probability distribution that underlies a dataset [GBC16]. Autoencoders are usually trained unsupervised. They consist of an encoder that learns a compact representation of the input data in a lower-dimensional latent space and a decoder that reconstructs the original input from the latent representation [Bal11].

Evaluation techniques for unsupervised learning include internal evaluation metrics, external evaluation metrics [VG20], visual inspection, comparative evaluation, and expert evaluation. Each of these strategies has downsides or impediments in our use case. The internal evaluation metrics are based on uncertainty of cluster assignments, by measuring the distances to other cluster centers and measuring the similarity of the objects within a cluster. But first we have to evaluate if the clusters even cluster graphs together that indeed are similar. We do not know if the embeddings reflect relevant features for our use case. External evaluation works under the assumption that we do have data labels for the evaluation which in our case we do not have. Visual inspection of all neurons to determine if the clusters are correct, can be very time consuming and even if we had

experts to evaluate the clusters we have to prepare the output clusters in a way to make the results easily accessible to the experts. Comparative evaluation is an additive step to compare multiple clustering algorithms, clustering networks, and parameter settings to find the best settings.

Self-Supervised Learning

Like unsupervised learning, self-supervised learning is based on unlabeled data. But it involves the derivation of *supervisory signals* from the input data to guide the learning process [LLH+22].

Contrastive learning belongs to the most successful self-supervised learning methods. Contrastive learning dates back to the 1990s and has since then been applied to tasks in the fields of computer vision, natural language processing, and audio processing. It is a subcategory of representation learning.

Contrastive learning techniques optimize the model output by embedding the latent representations of variations of the same input sample close to each other, while increasing the distance between the embeddings of different input samples. The pairs of samples that are either attracted or repelled by each other are denominated *positive pair* or *negative pair* respectively [LLH+22]. Phuc Le-Khac et al. [LKHS20] explain, that contrastive learning is not about learning from individual samples, but instead from *comparing* multiple samples. Positive pairs are generated by applying data augmentations, discussed in Sub-section 2.1.5, to an input sample to get variants of input data that are considered *similar*. The original non-augmented input sample is called *anchor view* and the augmented variant is referred to as the *positive view*. Negative pairs are generally formed by comparing the anchor view with all the other input samples. If contrastive learning is solely based on positive views [WHLE21], the model architecture needs to ensure that the latent representations do not collapse to a single node in the embedding space. This phenomenon is called *node collapsing*. Another force needs to increase the space between different samples.

A contrastive model includes an *encoder* that maps the input view x $\in$ X to a representation vector v $\in \mathbb{R}^d$ and a *transform head* $h(v; \Phi_h) : V \to Z$, with $v \in V$ being the the feature embedding, and where Φ represents the model parameters, that are either used to aggregate features from multiple representation vectors or to reduce the dimensionality of a feature representation vector [LKHS20].

An advantage of contrastive learning compared to other self-supervised learning methods is, that it does not require a pretext task that has to be adjusted for transfer tasks [LKHS20]. Prominent models that use contrastive learning to learn image representations are SimCLR [CKNH20], MoCo [HFW+20], BYOL [GSA+20], SwAV [CMM+20], PIRL [VMS+18], and DINO [CTM+21]. GraphCL [YCS+20] and GraphDINO [WHLE21] are examples for contrastive models that process graph data.

Semi-Supervised Learning

Semi-supervised learning leverages the benefits of supervised and self-supervised learning by combining a large unlabeled training dataset $M_{dataset}$ with a smaller labeled dataset $N_{dataset}$, such that $N_{dataset} \leq M_{dataset}$. The goal of a semi-supervised trained model is to accurately predict the expected outputs for the labeled examples and generalize to the unlabeled examples.

The datasets can be used simultanuously during the training, while the optimization function considers the representations of both the self-supervised and the supervised network stream. Additionally or instead the training can be split in pre-training and fine-tuning. PAWS [ACM+21] uses both datasets M and N during pre-training and only the labeled dataset N for fine-tuning. PAWS implements the semi-supervised method *Few Shot Learning*. In few shot learning, each class is represented by just a few labeled samples.

Supervised, unsupervised, and semi-supervised learning are all widely used machine learning techniques, and they each have their own strengths and limitations. In general, supervised learning is effective if a large amount of labeled training data is available and the problem being solved is well-defined. Unsupervised learning is useful if there is a large amount of data available but it is not labeled, and the goal is to discover patterns or relationships within the data. Semi-supervised learning can be useful if there is a limited amount of labeled data available, and it can often improve the performance of a model compared to using either supervised or unsupervised learning alone [EGLH22].

This thesis will use semi-supervised learning. The semi-supervised technique that we use is adapted from PAWS [ACM+21], discussed in Sub-section 3.3.2, and has characteristics of few-shot learning.

2.1.4 Graph Neural Networks

Graph

Graph neural networks (GNNs) are a type of neural network that is designed to process data that is organized in the form of a graph. A graph G = (V, E) is a data structure that consists of a set of nodes, also known as vertices $V = \{v_i\}_{i=1}^{N}{}_{vertices}$ and edges $E = \{e_{ij} = (v_i, v_j)\}$ that connect these nodes. A *simple graph* has no multi-edges, i.e., each pair of nodes is connected by at most one edge, and each edge is undirected. Many applications of GNNs are based on simple graphs [Ham20]. Furthermore literature distinguishes between heterogeneous data, i.e., data with *node type* and *edge type* information, and homogeneous data without type information. Our use case of neuron graph representations processed homogeneous simple graphs.

Graphs are represented with data structures that capture the feature information and the structural information for deep learning. Common representations are the adjacency matrix A, the Laplacian L in combination with a feature matrix F which stores features of each node. The adjacency matrix stores for each node the relationship to each other

node, i.e., 0 if there is no direct connection and 1 if the nodes are connected by an edge. The Laplacian is defined as $L = D - A$, with D being the degree matrix that stores the number of outgoing edges of each node in the diagonal. The normalized Laplacian has the advantage, that it is real symmetric positive semi-definite and can be decomposed to specify the eigenvalues and eigenvectors. It is calculated by $L_{normalized} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ [ZCH+20], with I being the identity matrix.

Applications

GNNs are most widely used for (1) node classification, (2) relation prediction, and (3) graph classification [Ham20]. These tasks are embedded in social network analysis, network compression, traffic prediction, e-commerce, combinatorial optimization [ZCH+20], in chemistry for the analysis of molecular structures [GMP21], in neuroscience for neuron classification [WHLE21] and brain connectome analysis [VCL+18, YJK+19].

Node classification assigns a label to each node of an input graph. Application examples are the detection of bots in a social network or the classification of the function of proteins in the interactome. Similar use cases apply to relation prediction. In a social network a GNN model can suggest new friends or find new biological interactions in biomedical datasets. The field of node classification and relation prediction covers clustering and community detection to find nodes and relations that are *similar* depending on varying metrices [Ham20].

While node classification and link predictions focus on the individual components of the graphs, graph classification is applied to entire graphs. All node features are aggregated and the representation embedding of the features is applied to the whole graph. The embedding is a vector representation that encodes the topological graph information [WPC+21]. Besides graph classification also graph regression and graph clustering are common applications [Ham20]. In this thesis we train a deep neural network for entire graphs.

Computational Modules

GNNs operate on graph data, i.e., data that is embedded in a non-Euclidean space. In contrast, Euclidean data in the context of neural networks is arranged in an underlying grid like images that are arranged in a 2D grid and text that is arranged as an 1D sequence [VCL+18, WPC+21, ZCH+20, YCL+21]. To generalize neural networks to process graph data, these grid-like Euclidean data structures can be considered to be graphs.

Figure 2.3 depicts on the left side an image with 4×4 pixels represented as a graph. The pixels are represented by nodes and each pixel is connected to the adjacent pixels with edges. The light-blue circled subgroup of nodes in Figure 2.3 of size 3×3 pixels indicates a convolution operation that aggregates the information from the image patch and uses a pooling operation like *mean*, *sum*, or *max* to compute the new hidden representation for the corresponding red node. Analogous to convolutions applied for image data,

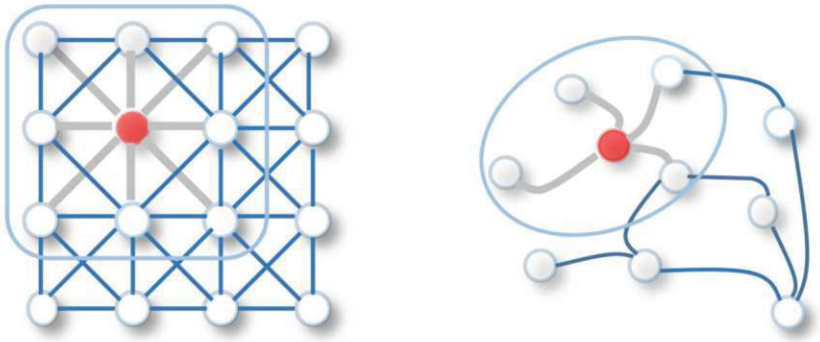


Figure 2.3: Image data visualized in a 2D grid and represented as a graph (left) and a non Euclidean graph (right). With both data structures a selection of adjacent nodes is used to perform a convolution to compute the value for the centered node that is colored red [WPC+21].

convolutions can be generalized to graph data by aggregating the feature information of neighboring nodes, visualized on the right side of Figure 2.3.

Bronstein et al. [BBL+17] use the term *geometric deep learning* to provide an overview for deep learning problems, applications and challenges regarding non-Euclidean data, including graphs and manifolds. Common computational modules to build GNNs are summarized and grouped by Zhou et al. [ZCH+20]. Figure 2.4 provides an overview of these modules and popular examples that implement them. The computational modules are divided in propagation modules, sampling modules, and pooling modules.

Propagation modules aggregate information of neighboring nodes of node v_i to encode the feature information and the topological information. The convolutional operator aggregates information about neighboring neurons over the course of multiple layers and each layer learns its own parameters, i.e., weights and biases, to combine features from the previous layer. The recurrent operator on the other hand iteratively uses the same set of parameters to process the input data. The output is passed back into the feedback loop until an equilibrium is reached [VCL+18]. The skip connection operation is used to *skip* layers and to process information from historical layers to solve the problem of exploding and vanishing gradients.

The convolutional operator is applied to *spectral* and to *spatial* representations of graph data. The graph Fourier transformation transforms graph data in the spectral domain. Spatial convolutions are directly applied on the graph data, more specifically on the matrices representing the graph data like the Laplacian and the feature matrix.

The number of nodes and the importance of each node that is considered for the propagation of structural and feature information depends on the implementation. GraphSAGE [HYL17] propagates information from a fixed-size set of neighboring nodes. Figure 2.5 depicts the node aggregation as implemented by GraphSAGE. In the image, the sample

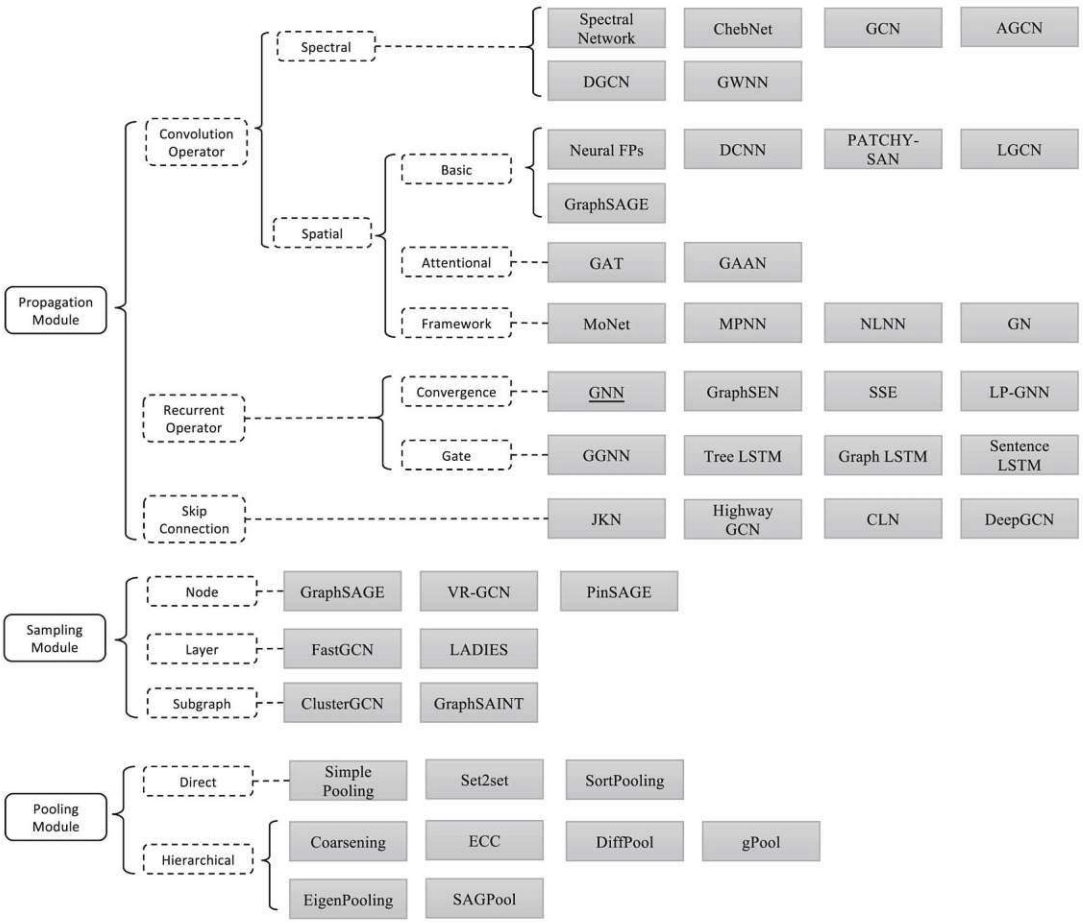


Figure 2.4: Overview of computational modules to process graph data [ZCH+20].

neighborhood is set to k=2, therefore all nodes that have a path length equal or smaller than two are used for the feature aggregation. The aggregator functions that sum up the information to compute the new representation of the red node v_i are learned. Step 3 assigns a new label computed with the aggregator functions 1 and 2 to node v_i. This new label is stored in the next layer, such that each layer aggregates information of neighboring nodes from the previous layer.

Attention-based operators assign different weights to the node features from neighboring nodes, leading to *adaptive aggregation*. The graph attention networks (GAT) presented by Veličković et al. [VCL+18] follow a *self-attention* strategy. For each node pair (v_i, v_j) an attention coefficient is learned that indicates the importance of v_j for v_i. To consider the graph structure, a *masked attention* masks the relevant neighborhood for each node, such that the attention coefficient is only computed for node pairs that share a neighborhood. Veličković et al. apply softmax and the nonlinear LeakyReLU activation to the aggregated node features $\overrightarrow{h_i}$, $\overrightarrow{h_j}$ by

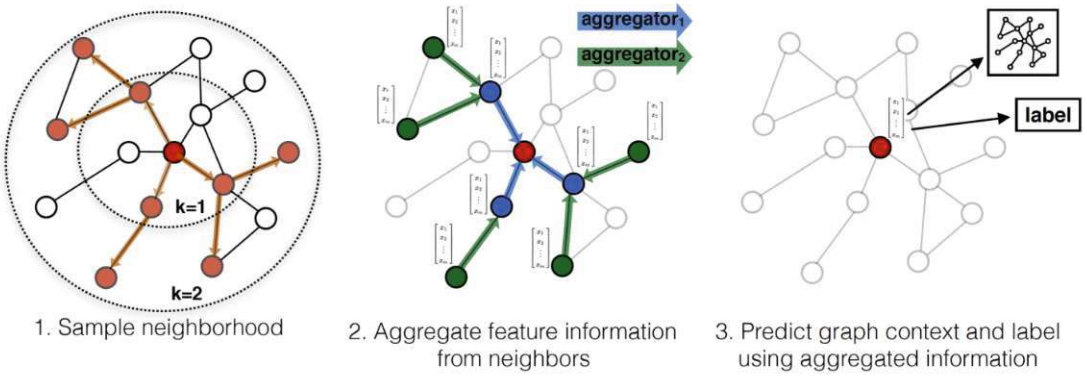


Figure 2.5: Feature aggregation as implemented by GraphSAGE [HYL17]

$$\alpha_{ij} = softmax(a(\boldsymbol{W}\overrightarrow{h_i}, \boldsymbol{W}\overrightarrow{h_j})), \tag{2.2}$$

where $\boldsymbol{W}$ is the weight matrix.

Figure 2.6 depicts the aggregation implemented by GAT. On the left, Equation 2.2 is visualized and on the right, the Figure illustrates the multi-head attention for three heads for node feature $\overrightarrow{h_1}$. Each attention computation is colored differently. The aggregated features are then concatenated or averaged to receive $\overrightarrow{h_1'}$.

The advantages of the self-attention are, that it is (1) parallelizable for the representation computation of each node, (2) it can be applied to all nodes with varying node degrees by adapting the edge weights and (3) the attention mechanism can adapt to previously unseen graph data.

Sampling modules are usually required to perform feature propagation on large graphs to address the *neighbor explosion* issue that arises if node features from multiple previous layers are aggregated. The number of contributing features grows exponentially with each additional layer dimension. Sampling reduces the number of nodes used for the feature aggregation. *Node sampling* reduces the number of contributing nodes to a subset, either stochastically or by using a predefined fixed number. *Layer sampling* reduces the nodes on each layer, again by reducing the nodes to a fixed number by using importance sampling or by using a trainable sampler that is conditioned on the former layer. The third subsampling approach is to *sample subgraphs* and to propagate feature information solely within these subgraphs.

Pooling modules extract information from nodes in order to create a subgraph or graph representation. *Direct* pooling modules are in some cases referred to as *readout functions*. They learn the subgraph / graph representation directly from a subset of the node features. *Hierarchical* pooling leverages the hierarchy of the graph structure to compute the subgraph / graph representation.

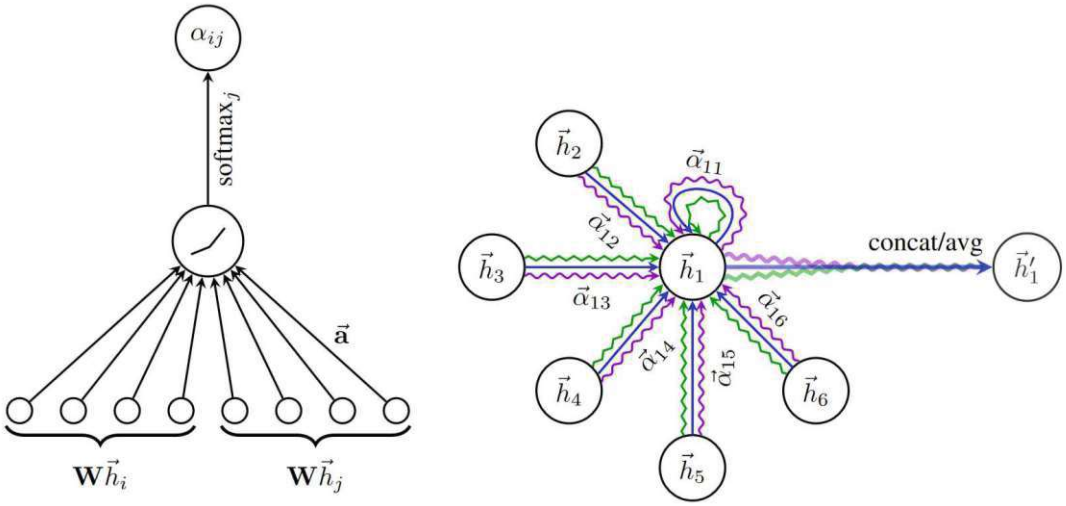


Figure 2.6: Feature aggregation as implemented by GAT [VCL+18].

Graph Transformer

The transformer architecture was developed to overcome the RNN bottleneck of sequential processing within the domain of natural language processing (NLP) by Google Research. It achieves state-of-the-art performance on a wide range of NLP tasks, including text analysis and translation [VSP+17, DB20, YCL+21]. The transformer architecture was introduced in the paper *Attention is All you Need* [VSP+17] published in 2017. While the attention mechanism was already used previously additional to convolutional or recurrent operations, the transformer architecture is the first publication that relies entirely on *self-attention*.

Figure 2.7 depicts the transformer architecture. It is a encoder-decoder architecture, the left box displays the encoder and the right box contains the decoder network.

The encoder encodes the input sequence to a representation vector using N encoding layers, each of which including a *multi-head self-attention* and a *fully connected feed-forward* sub-layer. Vaswani et al. [VSP+17] use residual connections to improve the network optimization. The residual connection is added to the processed signal and normalized in the *Add & Norm* blocks.

The decoder generates an output using N decoding layers, using the attention sub-layer and the feed-forward sub-layer analogous to the encoder and with an additional masked multi-head attention sub-layer. The masking and the shift operation ensure that the prediction for an output position relies solely on data that was already processed and not on subsequent positions. The decoder is trained to generate the output sequence one element at a time, based on the encoded input sequence and the previously generated output elements.

Transformers can aggregate information across long contexts, due to the **self-attention**

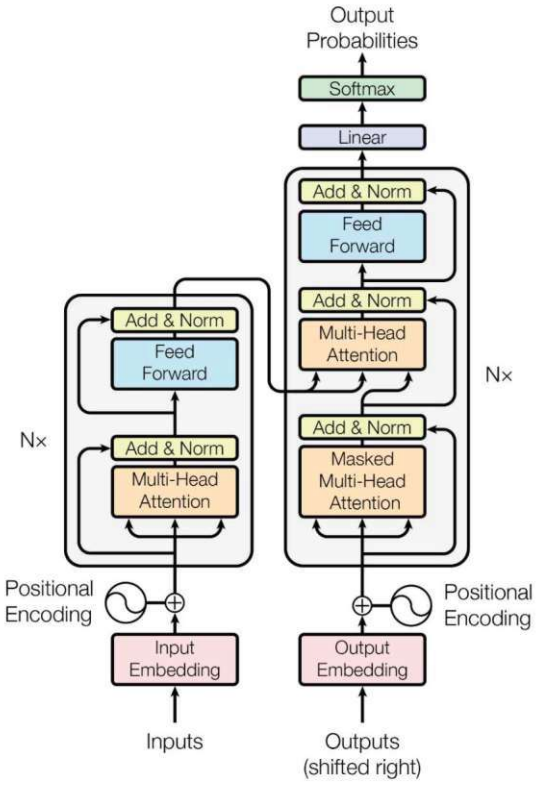


Figure 2.7: Transformer architecture [VSP+17].

mechanism. Each node potentially communicates to each other node. Vaswani et al. [VSP+17] use a scaled self-attention, denoted by

$$\text{Attention}(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V. \tag{2.3}$$

The matrices Q (queries), K (keys) and V (values) are learned parameters. The queries and keys have dimension d_k, while the values have dimension d_v. The classical self-attention is divided with $\sqrt{d_k}$ to compensate for very large values of d_k as softmax did not perform well without the scaling. This leads to the name *scaled self-attention*.

The self-attention mechanism is invariant to input permutations and therefore ignores structural information [MCSM21]. Vaswani et al. [VSP+17] add **positional encodings** to each input token based on cosine and sine functions that add up to unique positional encodings.

The transformer excels over recurrent and convolutional networks regarding the (1) parallelisation of computations, (2) the computational complexity per layer, and regarding (3) the communication between tokens of the input sequence that are far away from each other. Ad (1): The parallelisation of computations is limited in recurrent and

convolutional networks, as the feedback loop of recurrent networks and the later layers of convolutional networks depend on the previous computations. Transformers are able to process input sequences in parallel, rather than sequentially. This makes them much more efficient and allows them to scale to very large input sequences. Ad (2): The complexity per layer is $O(n^2 * d)$ for self-attention, $O(n * d^2)$ for recurrent layers, and $O(k * n * d^2)$ for convolutional layers, with n being the sequence length, d being the representation dimension, and k being the kernel size of convolutions. Add (3): Recurrent networks process the data sequentially and relations between tokens that are spatially far away from each other are weakened. The same applies to convolutional layers in which local neighborhoods are processed and combined to bigger neighborhoods in the subsequent layers. Relationships on a lower scale between long-range dependencies are missed. On the contrary, the transformer architecture sets every token in relation to all other tokens in the input data [VSP+17].

Since the publication of *Attention is all you need*, the transformer architecture was widely adopted for other large-scale language models, such as BERT and GPT-2, GPT-3, and GTP-4. Following the success of transformers in NLP, transformer networks were applied in other fields. The Vision Transformer (ViT) model proposed by Dosovitskiy et al. [DBK+20] achieved state-of-the art performance in the image domain and recent research adopts the transformer architecture for graph data.

As the self-attention mechanism ignores structural information, graph transformers add positional encodings and *structural encodings*. The positional and structural encoding for graphs is more complex than for sequential data since the concept of node positions in graphs is ill-defined [MCSM21]. Structural encodings can be node features representing the graph structure, while positional encodings store the node position within a graph. Research differentiates between *absolute position encoding* as implemented in [VSP+17] and *relative positional encoding* that encodes the distance between two elements [SUV18].

Rampasek et al. [RGD+22] propose a *recipe for a general, powerful, scalable graph transformer* as they named their paper, that includes a summary of positional encoding (PE) and structural encoding (SE) strategies and a categorization thereof in *local*, *global* and *relative* encodings. Local PEs encode the embeddings of nodes within a local cluster. This can be implemented with node distances from the local centroid. Global PEs encode the global position of a node within the graph, for example implemented with the adjacency matrix or the Laplacian. The relative PE can be expressed with pair-wise node distance measurements. Local SE encodes the substructure, e.g, the node degree. Global SE encodes the global structure of a graph that is for instance represented by the eigenvalues of the Laplacian and the relative SE encodes the similarity measure between two nodes.

GraphiT [MCSM21] uses relative positional encoding and local structural encoding to solve classification and regression tasks. The relative positional encoding is implemented with positive definite graph kernels. The local structural encoding is done with graph convolutional kernel networks that capture small sub-structures.

Dwivedi et al. [DB20] use eigenvectors of the graph Laplacian for the positional encoding and apply the attention mechanism to neighboring nodes. They assign the first k coordinates of the node eigenbasis of the normalized graph Laplacian to each node. Dwivedi et al. called their positional embedding *LapPE*. The eigenvectors oscillate more with decreasing eigenvalues and are interpreted as Fourier coefficients. The local attention is analogue to a weighted message passing that is trained on node feature similarity.

Chen et al. [COB22] introduce the Structure-Aware Transformer that adapts the self-attention to incorporate structural information. Therefore a subgraph is extracted for each node to compute the attention.

The graph transformer network (GTN) [YJK+19] is trained on heterogeneous data to generate new graph structures. GTNs transform the heterogeneous graph into a homogeneous graph by encoding the type information in meta-paths. In previous work the meta-paths were manually generated while GTN learns the meta-paths. Yun et al. [KW17] use a graph transformer network to learn the meta-paths. The node representations of the graph are learned with a graph convolutional network (GCN). The authors describe their network as *an ensemble of GCNs on multiple meta-path graphs learned by GT layers*.

Graphormer [YCL+21] also handles heterogeneous data by learning the edge type encodings. The graph structure is encoded using *centrality encoding*, that learns the node importance within the graph and with a spatial encoding.

GraphBERT [ZZSX20], an adoption of the transformer network BERT for graph data, is trained on sampled fixed-size linkless subgraphs instead of being trainined on the whole graph. As the links are omitted (*linkless subgraphs*), GraphBERT encodes absolute structural and relative positional information in the node representations.

2.1.5 Data Augmentation

Augmentations were previously mentioned in the context of contrastive learning. Augmentations are also used for other learning strategies, to add variance to the training data in order to ensure that the model generalizes, i.e., does not overfit to the training data. Depending on the task, examples for applied augmentations in image processing are geometric transformations, color space augmentations, kernel filters, mixing images, random erasing, and feature space augmentation [SK19]. In graph processing, the augmentations result in slightly changed graphs or synthetic graphs based on the input graph [DXTL22]. Figure 2.8 depicts augmentation strategies for graphs.

Ding et al. [DXTL22] differentiate between *structure-oriented*, *label-oriented*, and *feature-oriented* augmentation techniques. Techniques that we will use for our experiments are *node dropping*, *graph sampling* and *feature corruption*.

Node dropping, also known as node masking, is the deletion of a set of nodes and the associated edges. It is formally denoted with: $\hat{A} = \{V \backslash \hat{V}, E \backslash \hat{E}\}$, with V, E being the vertices and edges of the original graph and $\hat{V}$ and $\hat{E}$ being subsets of the vertices and

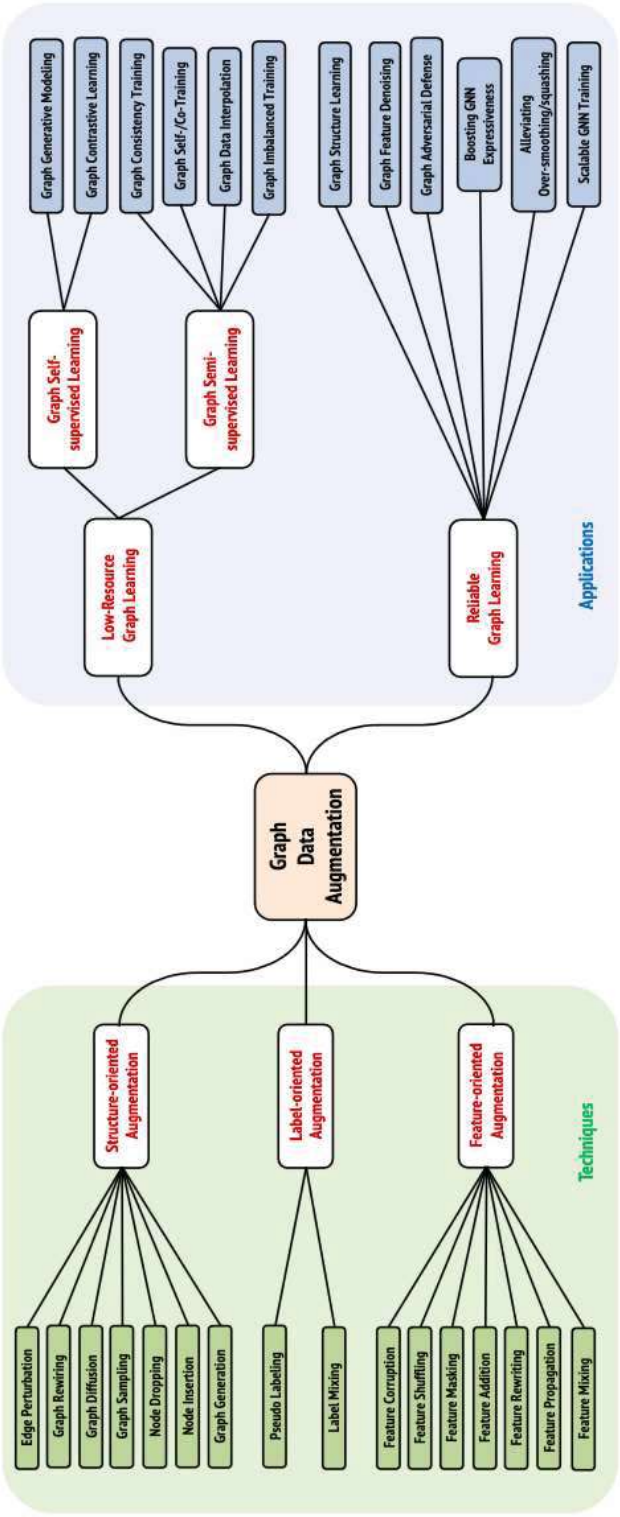


Figure 2.8: Taxonomy of Graph Data Augmentation techniques and applications proposed by Ding et al. [DXTL22].

edges. In GraphDINO [WHLE21] this augmentation is implemented in the form of branch deletion, such that not single nodes are randomly removed, but branches that are connected to leave-nodes.

Graph sampling is implemented based on vertex sampling, edge sampling, or traversal based sampling. Graph sampling commonly returns a connected subgraph induced from the sampled nodes. Given a graph $G = (A, X)$, where denotes the adjacency matrix and X denotes the node feature matrix, it is formally denoted with: $\hat{G} = \{\hat{A}, \hat{X}\} = \{A[idx, idx], X[idx, :]\}$, with idx being a list of indices to select elements from A and X. GraphDINO implements *graph sparsification* by trying to retain the underlying structure of the graph while reducing the graph to a predefined number of nodes.

Feature corruption adds noise to either the nodes or the graph. It is formally denoted by $\hat{x}_i = x_i + r_i$ with x_i being a feature x at index i and r_i being the noise added at index i. GraphDINO implements *node jittering*, such that each node is perturbed in its position with a random factor and *translation*, which is a fixed position vector that is added to each node.

2.2 Visual Analytics in Deep Learning

2.2.1 Visual Analytics

Visual analytics (VA) is "the science of analytical reasoning facilitated by interactive visual interfaces" [WT04]. Users can focus on certain aspects of the dataset or restructure it in a way that new information is revealed by reducing the complexity of big datasets and processes. Visual analytics aims to close the gap for tasks that are too dependent on user judgement and interpretation to be solved exclusively analytically, but handle too complex data to be solved exclusively visually [KBB+10].

Working with visual analytics consists of data collection, data preprocessing, visual representation and incremental data manipulation that leads to visual updates [KBB+10]. Figure 2.9 illustrates the feedback loop for interaction with the visual representations. The Figure depicts the components *data*, *visualization*, *models*, and *knowledge*. It further includes the actions *transformation*, *mapping*, *data mining*, *model building*, *model visualization*, *user interaction*, *parameter refinement*, and *feedback loop*.

The data component represents the unfiltered input data. The visualization is generated by mapping the data to a visual application and the model is generated by applying *data mining* techniques. Data mining is used to explore large, complex data with computational methods in order to find patterns, to build models that describe the data, and to make predictions based on these models [KT13]. In this thesis we use artificial intelligence to find meaningful latent representations of the neuron graphs that encode important information and we use clustering to group similar neuron graphs.

The analyst can interact with the visualization and apply analytical techniques to the underlying model. The interaction techniques are described in Sub-section 2.2.2. Keim

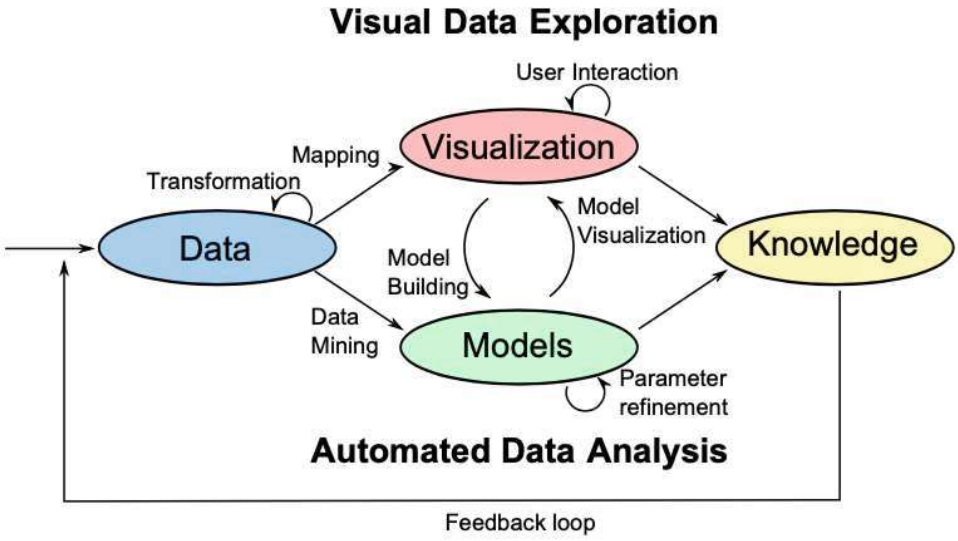


Figure 2.9: The feedback loop for user-interaction on visually represented data. Injected data is mapped to a model and to a visualization. The user interacts with the visualization and incrementally adjusts the model to make a decision or to gain insights on the data [KBB+10].

et al. [KBB+10] call this combination of analytical and visual analysis *advanced*, while both the visual and the analytical component themselves can be simple.

With iterative updates to the visualization that potentially lead to rebuilding the underlying model the user gains knowledge from both the visual and the analytical components and can accordingly adapt the input data. In our use case, the adaptation of the input data is performed by retraining the deep learning model with new labeled data.

2.2.2 User Interactions

Shneiderman defines the Visual Information-Seeking Mantra: *Overview first, zoom and filter, then details on demand* [Shn96].

Overview First

We have to distinguish between varying visualization techniques. Commonly, data is displayed in 2D [STN+16, ZCAW17, PHG+18] and 3D [STN+16]. While 2D visualizations use icons and project the data to a mostly flat surface, 3D visualizations use stereographic depth effects [AB12]. In this thesis we visualize the graph latent representations with 3D scatter plots. If data is stored in a database, queries are linked to UI elements, including buttons and selection-boxes to request, update, and delete information. This kind of

interaction is referred to as **dynamic query** [Shn99]. We use this technique to load data on demand according to parameter values that the user specified.

Zoom and Filter [and Other Analysis Methods]

A popular technique to analyse the data is **brushing**, which is the selection of a subset of the original dataset. It is commonly used in combination with **linking**, that links multiple views of the same data in the sense, that selections and changes made within a view are reflected in the other view. For example, a user might use a scatter plot to represent the underlying data model and in parallel they work with a region-map to display the same data. If the user selects a subset of data-points in the scatter plot with the brush, the region-map will have a visual selection applied as well [IF09].

Brushing and linking is common for various visualizations, e.g., heat map visualizations [IF09], scatter plots [NMA+20, FH18], parallel coordinate plots [Eds03], and interactive graph exploration [Guc17]. In the latter case, the user can rearrange the nodes to detect patterns and correlations. The reassembly does not affect the relations between data points.

The important criteria for brushing techniques are *efficiency* and *accuracy*. The user should be able to select data fast, accurately, and in a fluid manner. The most common brushing technique is *brushing using simple geometries*, i.e., selecting the points that intersect with a geometric shape, for example a one-, two-, or three-dimensional line, respectively bounding-box. Other techniques include *lassoing* that lets the user draw a closed curve that encloses points, *logical combinations* to combine multiple brushing techniques and iteratively refine the selection, and *sketch-based brushing* that applies heuristics to a shape sketched onto a visualization [FH18].

While brushing and linking is used to analyse the visible data, **filtering** is used to focus on specific data subsets and to hide insignificant data points. It is common, to analyse huge data in smaller fractions [IZJ18].

Details on Demand

Details are usually presented in separated panels [STN+16] or with fish-eye enlargements [YXL+22].

2.2.3 VA Application in Deep Learning

While deep learning opens up new possibilities in research, it sparks new research questions as well. A deep learning network in general is a *black box*, which is difficult to interpret. Enlightening the black box is interesting for many applications, as the results need to be comprehensible in order to be reliable and to avoid misjudgement due to an encoded bias.

Hohman et al. [HKPC19] define the following questions to design and evaluate a VA application for deep learning:

1. **Why** do we want to use visualizations in deep learning? Why and for what purpose would one want to use visualization in deep learning? This question can be answered with *Interpretability & Explainability* [STN+16, ZCAW17, PHG+18, WGYS18], *Debugging & Improving Models* [PHG+18, WGYS18], *Comparing & Selecting Models* [WGYS18], and *Teaching Deep Learning Concepts* [KTC+19].

2. **Who** wants to visualize deep learning? Who are the types of people and users that would use and stand to benefit from visualizing deep learning? Possible answers that Homan et al. [HKPC19] propose are *Model Developers & Builders* [PHG+18, WGYS18], *Model Users* [STN+16], and *Non-experts* [KTC+19].

3. **What** can we visualize in deep learning? What data, features, and relationships are inherent to deep learning that can be visualized? Possible answers include *Computational Graph & Network Architectures* [WGYS18, KTC+19], *Learned Model Parameters* [WGYS18], *Individual Computational Units* [STN+16, PHG+18], *Neurons in High-dimensional Space* [STN+16, PHG+18, KTC+19], and *Aggregated Information* [STN+16, PHG+18, WGYS18].

4. **How** can we visualize deep learning? How can we visualize the aforementioned data, features, and relationships? This can be answered with *Node-link Diagrams for Network Architectures* [KTC+19], *Dimensionality Reduction & Scatter Plots* [STN+16, PHG+18, WGYS18], *Line Charts for Temporal Metrics* [PHG+18, WGYS18], *Instance-based Analysis & Exploration* [STN+16, PHG+18, WGYS18], *Interactive Experimentation*, or with *Algorithms for Attribution & Feature Visualization*.

5. **When** can we visualize deep learning? When in the deep learning process is visualization used and best suited? Homan et al. [HKPC19] reference research that applied VA *During Training* [PHG+18, WGYS18, KTC+19] and *After Training* [STN+16].

6. **Where** is deep learning visualization being used? Possible answers are: In *Application Domains & Models*.

We answer these questions for our VA application in Section 3.6.

Our visual analytics application handles graph embeddings. Therefore, in the following paragraphs we look at visual analytics applications developed to explore and compare embeddings, i.e., vector representations for the input data, generated with various models.

Embeddings are widely used in ML applications. Common ways to trace the decision making of a network that generates the embedding are input feature importance, saliency, and neuron activations. In the previous years another line of work was analyzing the embeddings using visual analytics applications to interactively find differences between embedddings, find interesting objects, and investigate interesting objects [HKMG22].

The research regarding visual analytics for embeddings typically includes a user study regarding the general requirements and specific tasks to create the design of the VA application. Boggust et al. [BCS22] identified two user groups: model-driven users that compare model performances and data-driven users that study properties of the underlying data. They further conclude that crucial criteria are *global check and local check*.

Addressing this, a common approach to compare embedding spaces is the comparison of local neighborhoods of individual objects in combination with a global comparison of the embeddings [HKMG22, BCS22]. The global embedding comparisons are typically implemented using scatter plots that are interlinked with detail views of selected objects [HKMG22]. Therefore dimensionality reduction algorithms are used to map the high-dimensional data into 2D or 3D. The most common dimensionality reduction algorithms are PCA, t-SNE, and UMAP [SWP22]. Boggust et al. [BCS22] discovered that users prefer deterministic dimensionality reduction algorithms and that they distrust t-SNE and therefore use PCA dimensionality reduction as the default setting for the global projection. The visual analytics tool EmbComp [HKMG22] implements a binning feature for the scatter plots to manage the scale of big datasets. The scatter plots can be investigated via single object selection or multiple object selection using for example a rectangle selection tool [LWBM22].

The investigation of local neighborhoods is built upon varying metrics. EmbComp [HKMG22] visualizes point-wise comparison metrics, i.e., the amount of overlap between the neighborhoods of an object in two different embedding spaces and the spread metric that defines how far a neighborhood extends from one embedding to another one. Furthermore EmbComp implements distribution comparison metrics, i.e., the distance between nodes within a neighborhood and the local density within a neighborhood. The metrics are visualized in bins which can be selected by the user to identify the corresponding objects. This design concept represents a top-down analysis through the actions *summarizing local metrics to find similarities and differences*, *subset selection*, and *sequentially scanning*.

The Embedding Comparator [BCS22] displays the k-nearest neighbors for a chosen data point to visualize the local neighborhoods. The comparison of two models is performed with the computation of similarity scores between the local neighborhoods of a chosen data point using the cosine similarity or Euclidean distance. The results are visualized with a histogram of scores, color-encoding in the global embedding plots, and with local neighborhood dominoes, i.e., multiple small visualizations. These small visualizations can be filtered and linked views enable the comparison between visualizations. The similarity metric is computed for every embedded object.

The Embedding Comparator highlights data points with least and highest similarities to address the concern of users stating that they make object selections in an unprincipled way and might miss important correlations between the embedding spaces. Emblaze [SWP22] states, that the Embedding Comparator lacks in finding relevant neighborhoods and addresses this issue in their application.

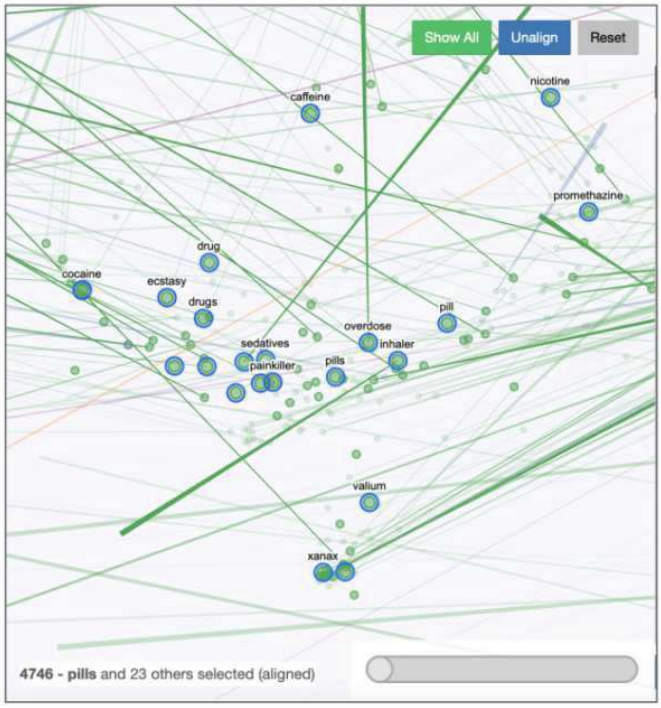


Figure 2.10: Star Trail visualization of Emblaze [SWP22] that compares a selection of drug-related words within two embedding spaces. One embedding space is based on Google News data and the other on Twitter data. The line opacity is dependent on the relative number of nearest neighbors that changed between the two embedding spaces.

Emblaze can be integrated in a computational notebook environment. This combines the support of predefined tasks with the freedom to implement own metrics. The novel approach of Emblaze is the comparison of embedding spaces using Star Trail augmentation, as depicted in Figure 2.10. The embedding spaces contain the object embeddings as scatter plots. The trails connect the embeddings of the same object in different embedding spaces and the transition between the spaces can be animated using a slider. The connection lines, i.e., Star Trails, between the object embeddings quickly reveal data points that vary the most between multiple embedding spaces. To align the embedding spaces, Emblaze computes the Procrustes transformation, which minimizes the distances between the objects in the embedding spaces. The Procrustes transformation can also be applied to single selected objects and their neighborhoods. The neighbor sets are compared using the Jaccard distance, a common metric to compare sets, which computes the intersection of two sets over the junction of the two sets. Emblaze further applies color-coding to visualize if nodes are only present in one of the nearest neighbor sets under comparison.

While partly being data type agnostic, Embedding Comparator, EmbComp and Emblaze as well as many other lines of research regarding visual analytics for embeddings focus

on NLP use cases. In the field of graph embeddings the tools EmbeddingVis [LNH+18], CorGIE [LWBM22], GEMvis [CZG+22], and BiaScope [RSL+22] were developed. While we implement an application for whole graph embeddings, the aforementioned applications are developed for node embeddings. CorGIE states that they might extend the scope of the application with whole graph embeddings in future work.

CorGIE [LWBM22] encodes the graph nodes and trains a GNN to embed the nodes in the latent space. The user can interact with the node embeddings and select clusters of nodes using a rectangle selection tool. The selection leads to a topology space and feature space analysis. Regarding the topology space, the k-hop neighbors, i.e., the neighbors that are reachable by walking along a path by passing k nodes, of the selected nodes are depicted within a visualization of the original graph. The user can evaluate whether the node embeddings correspond to the topological closeness, i.e., whether nodes that have similar embeddings are within each other‘s k-hop neighborhoods. The feature space analysis panel shows histograms of feature value distributions of the selected nodes.

GEMvis [CZG+22] also interlinks a visualization of the original graph and the node embeddings as depicted in Figure 2.11. The selection of nodes can be applied regarding predefined node metrics. Chen et al. [CZG+22] define nine node metrics, including the node degree, node eccentricity, and the node closeness. The metric values for each node are depicted in parallel coordinate plots. The user can interact with these plots to select the corresponding nodes in the original graph and in the embedding space.

While advanced applications exist to leverage VA to compare and analyze the embeddings generated with deep learning models, we employ the component of dynamically adding new labels to retrain the model while exploring the latent space that the input graphs are embedded in. We focus the usage of VA for AI to the specific case, in which ground truth is difficult to gather and can only be provided to nudge the training in the right direction. We furthermore integrate detail views specific to the use case of exploring graph embeddings.

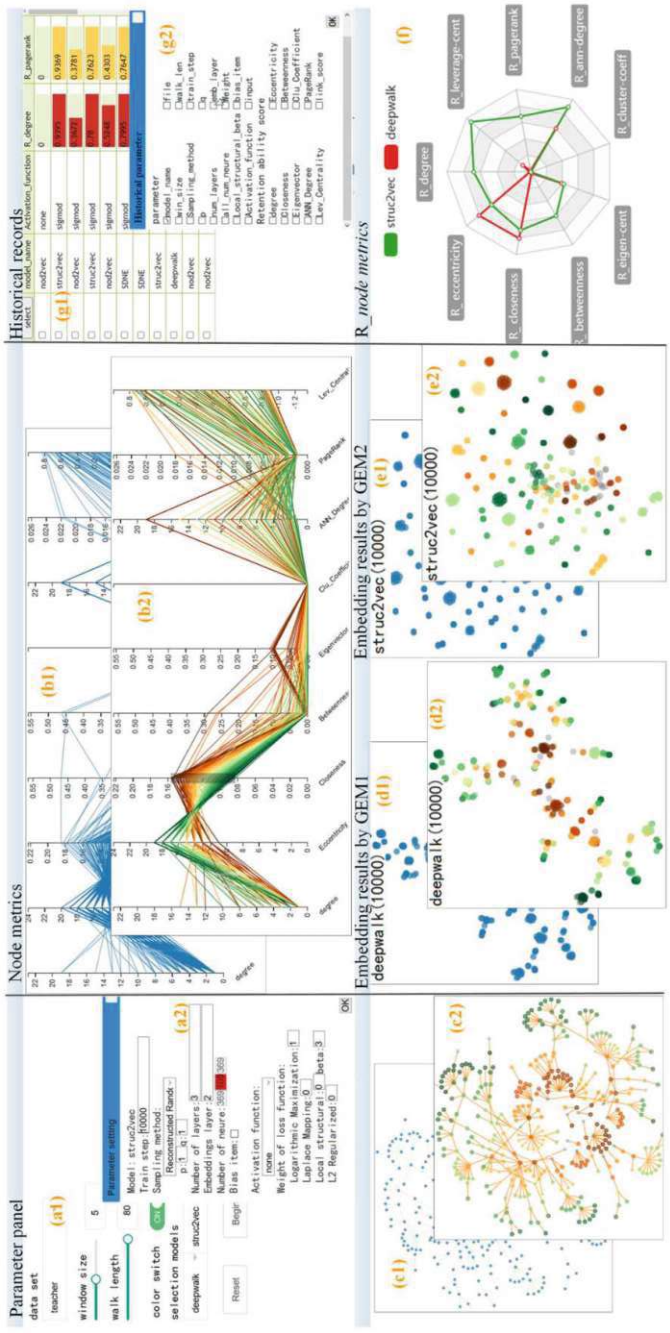


Figure 2.11: The GEMvis [CZG+22] interface includes (a) a parameter panel to select two models for comparison, (b) parallel coordinate plots to choose data based on feature values through brushing, (c) the original input graph, (d)(e) the node embeddings of the input graph, (f) a panel depicting the node metrics in a radar plot, and (g) a record view that stores previous analyses. Overlaying panels annotated with 2 are changes that are applied to the panels beneath during user interactions.

CHAPTER 3

Materials and Methods

This chapter covers materials and methods that we apply in our thesis. We discuss the pipeline to incrementally gain new knowledge to improve GraphPAWS using NetDive in Section 3.1. Section 3.2 addresses the the drosophila melanogaster dataset. Section 3.3 covers the development of our deep learning network architecture *GraphPAWS* based on the architectures GraphDINO and PAWS. Section 3.4 discusses the clustering methods we use and the evaluation we perform based on the predicted clusters. Section 3.5 covers dimensionality reduction algorithms to reduce the graph latent embeddings to three dimensions that can be visualized in NetDive and Section 3.6 explains the concept and design of NetDive.

3.1 Pipeline

Figure 3.1 depicts the pipeline that we set up to incrementally gain new knowledge in order to cluster graph data. The first step is to process the graph data, such that the graphs are aligned. The preprocessed data serves as input data to train, validate, and test the GraphPAWS model. The model outputs latent embeddings of the input graphs. We store the latent embeddings on the filesystem. The visual analytics application NetDive accesses the data and provides the user with visualizations and user interactions to explore the latent embeddings and the associated neurons. The user can filter and update the underlying data by choosing an inference subset and by varying the dimensionality reduction algorithm. The data exploration with NetDive leads to new knowledge which aids the user to generate annotations for a subset of data points. The user can start the retraining of the GraphPAWS model within NetDive, leveraging the new annotations.

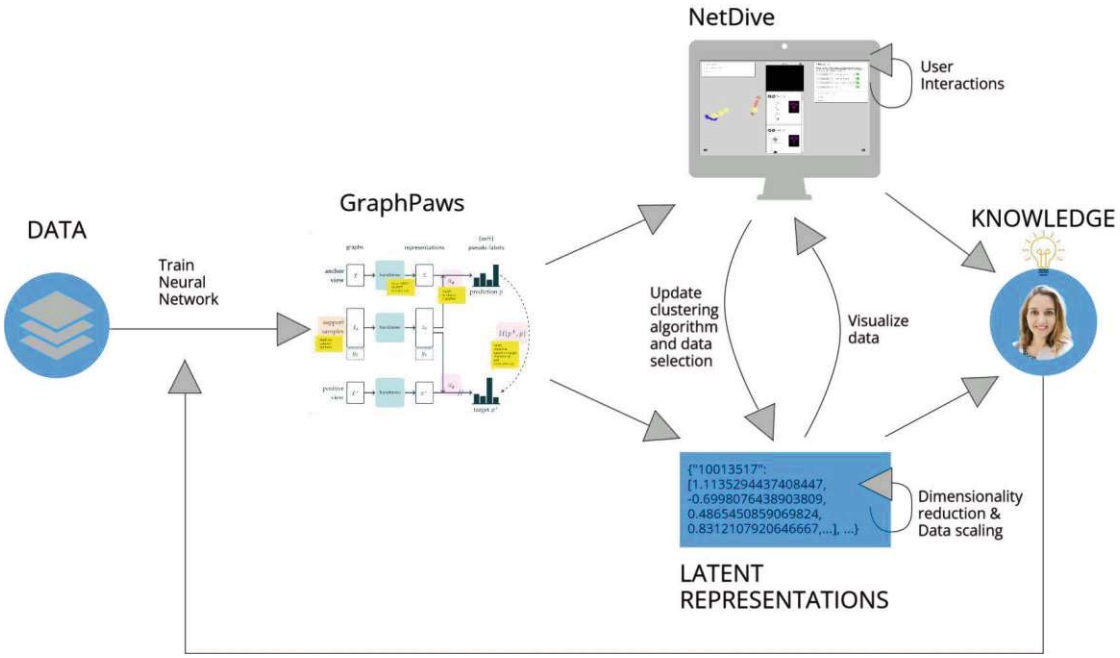


Figure 3.1: The pipeline including our visual analytics tool NetDive and our semi-supervised model architecture GraphPAWS.

3.2 Data

We obtain the drosophila melanogaster larval neuron dataset we use for our experiments from CATMAID [SCHT09]. CATMAID is a platform for a collaborative reconstruction and annotation of data and stands for *The **C**ollaborative **A**nnotation **T**oolkit for **M**assive **A**mounts of **I**mage **D**ata*.

The dataset contains 7297 drosophila melanogaster larval neurons from CATMAID. We restricted our analysis to a subset of 2970 neurons that was annotated by Michael Winding [WPB+23] (further discussed in Section 4.2). We chose the latter, as the subset contains more reliable neuron traces compared to the remaining 4327 neurons. We reduced this subset to 2541 neurons by removing all neurons that do not contain exactly one node annotated as *soma* and by removing all neurons with less than 200 nodes.

The drosophila melanogaster larval neurons are represented as undirected, acyclic graph in three dimensions with the root node representing the soma. To extract the graph information, the larval brain is sliced in thin layers with a diamond knife and each layer is scanned with an electron microscope. Each layer of the microscopic imagery is traced by experts. They draw points along the curve that belong to the neuron. The points represent nodes in the graph and the nodes are connected with edges. The graph is traced over the whole image stack that composes the brain. The traces of each layer are connected to the neighboring layers, such that the resulting graph is spanned in three dimensions. This process is depicted in Figure 3.2.

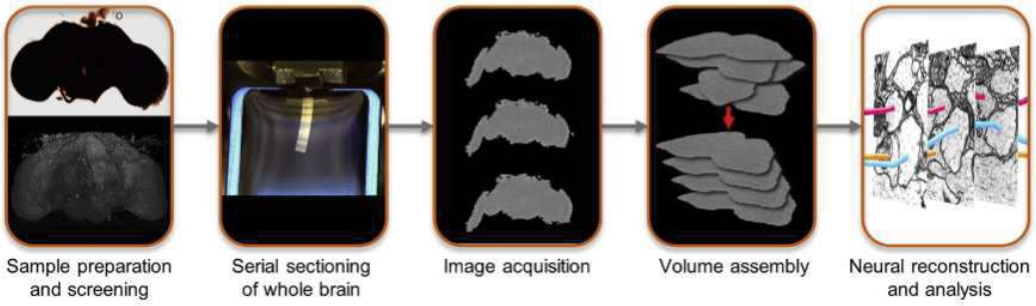


Figure 3.2: The process from electron imagery to graph data: The brain tissue is prepared, sectioned in thin layers and the layers are scanned with transmission electron microscopy (TEM). The resulting scan images are assembled to a volume and the neurons are traced over the image stack [ZLP+18].

3.3 Deep Learning

3.3.1 GraphDINO: Self-Supervised Learning

The GraphDINO network [WPLE23] utilizes self-supervised contrastive learning to find similarities between graphs based on their shapes. The graph representation learning network *GraphDINO* was published in 2021. It is an adaptation of the DINO network, developed at Facebook earlier that year [CTM+21]. While DINO operates on image data, GraphDINO operates on graph data. Both network architectures employ contrastive learning based on transformer networks. While DINO integrates vision transformers, GraphDINO integrates graph transformers. GraphDINO applies complexity reductions to the DINO architecture like the elimination of L2 normalization [WHLE21]. The name DINO stems from the descriptive name *self-**di**stillation with **no** labels*, referring to the self-supervision of the network architecture.

GraphDINO is trained on the datsets *rat somatosensory cortex neurons* that are part of the *Blue Brain Project* (BBP) [RCA+15], *M1 PatchSeq* of *mouse motor cortex neurons* [SKB+21], *mouse visual cortex neurons* that is part of the *Allen Brain Atlas* (ACT) [All16], *Brain Image Library* (BIL) [PXL+21] of *mouse neurons (based on the whole brain)*, *Janelia MouseLight* (JML) [WBF+19] of *mouse neurons (based on the whole brain)*, and *Botanical Trees* [SDS21].

The following sections discuss GraphDINO in more detail.

Architecture

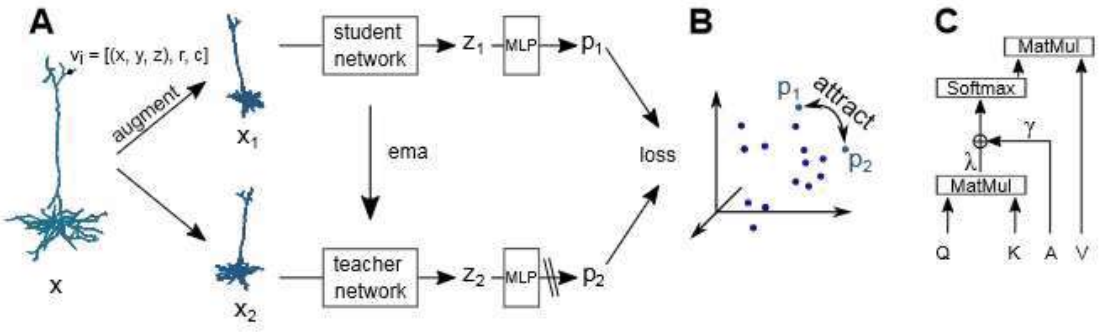


Figure 3.3: GraphDINO architecture developed by Weis et al. [WHLE21].

Figure 3.3 depicts the GraphDINO architecture. The model builds upon a student-teacher architecture that is used to generate latent embeddings of an input graph x. Both the teacher and the student process variations of x. The variations x_1 and x_2 are sub-sampled to a fixed number of nodes. Graph x_2 is passed to the teacher encoder and graph x_1 is augmented before being passed to the student encoder. The augmentations that are used are subsampling, rotation, node jittering, subgraph deletion, cumulative jittering, and a random translation of the soma depth.

The student and the teacher network are identically initialized transformer networks. The student parameters are learned with gradient descent, while the teacher parameters are updated with an exponential moving average (ema) of the student parameters. A centering operation computed with a mean over the batch is applied on the output of the teacher network. In this architecture the student learns to discard errors from the teacher network. The outputs of the student and the teacher network are the latent embeddings z_1 and z_2 respectively. The multi-layer perceptron (MLP) utilizes a normalization layer and a linear layer to translate the latent embeddings z_1 and z_2 to p_1 and p_2. The objective of the network is to decrease the loss that measures the similarity of p_1 and p_2, while not resulting in node collapsing.

Node Collapsing Avoidance

In self-supervised learning, it is essential to avoid mapping all inputs to the same latent embedding when using only positive samples. This effect is called node collapsing. Weis et al. [WHLE21] do not explicitly discuss how to prevent node collapsing, but experimentally demonstrate that the student-teacher architecture and the use of batch normalization help to prevent learning a single latent embedding for all input graphs. As we discard the student-teacher network for our semi-supervised architecture adaptation that is discussed in Sub-section 3.3.3, we have to pay attention to leverage a different

node collapse avoidance strategy.

Encoder

GraphDINO encodes the graph inputs with an adaptation of the transformer attention mechanism introduced with the paper *Attention is All you Need* [VSP+17] that was originally developed for natural language processing (NLP) as discussed in Subsection 2.1.4.

Each token x_i in the input sequence corresponds to a linear transformation of the features f_i of a node v_i. The keys K, queries Q, and values V are learned linear projections of these tokens. In order to train not only on feature information but also on structural graph information, GraphDINO incorporates a bias towards the adjacency matrix A. The adjacency matrix of a graph is a symmetric matrix of size $n \times n$, with n being the number of vertices in the graph. A connection between two nodes is represented with a one in the corresponding cell and a missing link between two nodes with a zero. Equation 3.1 denotes how GraphDINO calculates the attention,

$$\text{Attention}(Q, K, V, A) = \sigma(\lambda \frac{QK^T}{\sqrt{d_k}} + \gamma A)V, \text{with} |\lambda_i, \gamma_i| = exp(Wx_i), \tag{3.1}$$

where σ is the softmax function, W is a learned weight matrix, λ and γ assign the ratio of relevance of the neighboring nodes versus all nodes in the graph. For $\lambda = 1$ and $\gamma = 0$ this is equal to the classical transformer attention and for $\lambda = 0$ and $\gamma = 1$ it is equal to the GNN message passing algorithm. The token of node v_i with the added positional encoding is denoted by x_i.

For the positional encoding, GraphDINO uses the normalized graph Laplacian L, as denoted in Equation 3.2,

$$L = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} = U^T \Lambda U, \tag{3.2}$$

where I is the identity matrix, D the degree matrix, A the adjacency matrix. The Laplacian encodes the neighborhood information with the adjacency matrix and structural information on the matrix diagonal with the degree matrix. The Laplacian without normalization has positive integer values on the diagonal, i.e., if i=j, with i being the row index and j being the column index, that represent the count of connected edges. If i≠j, the value is 0 if the nodes v_i and v_j are not connected and -1 otherwise. For the normalized Laplacian as denoted above all entries are divided by $\sqrt{deg(v_i)deg(v_j)}$, which renders the diagonal to have 1 in each entry. $U^T \Lambda U$ is the eigenvalue decomposition of the normalized Laplacian with U being the eigenvector matrix and Λ the eigenvalue matrix. Using the eigenvalue decomposition, it is possible to select the first k eigenvectors (representing the Laplacian) with the highest eigenvalues for the positional encoding.

The graph transformer used by GraphDINO is composed by seven multi-head attention modules and eight heads.

Training Objective

GraphDINO uses a cross-entropy loss as shown in Equation 3.3. The training objective is to decrease the loss

$$\text{loss} = -\frac{1}{N}\sum_{N}(p_1 * log(p_2 + eps)), \tag{3.3}$$

where p_1 and p_2 are the latent embeddings that are projected from the latent embeddings z_1 and z_2 using the MLP head as shown in Figure 3.3. The MLP head consists of linear layers, GELU layers and a normalization layer to project z_1 and z_2 to the predefined number of classes, i.e., classes that are assigned to the graphs for classification, and the predefined number of dimensions.

3.3.2 PAWS: Semi-Supervised Learning

PAWS [ACM+21] deploys a semi-supervised deep learning architecture based on contrastive views and support samples to assign one-hot encoded pseudo labels to input images. The authors evaluate the network with 1% labeled samples and 10% labeled samples and achieve 66.5% and 75.5% accuracy on ImageNet. With 10% labeled samples, PAWS outperforms state-of-the-art networks that are strictly self-supervised, namely BYOL [GSA+20], SwAV [CMM+20] and others. The following sections discuss PAWS in more detail following the structure of Sub-section 3.3.1.

Architecture

The high level network architecture of PAWS is depicted in Figure 3.4. The figure shows three processing streams. The first stream processes the anchor view, which is a sample image $\hat{x}$ of the unlabeled image data. The positive view, represented by the bottom stream, processes an augmented version of the image $\hat{x}$, denoted as $\hat{x}^+$. PAWS implements the image augmentations random crop, horizontal flip, color distortion, and blur. The middle input stream processes a mini-batch of labeled support samples. PAWS expects each mini-batch to be composed by an equal number of instances for each sampled class.

All three streams are encoded with the same parameterized encoder f_θ, which is the trunk of a deep residual network. The encoder outputs z, respectively z_S and z^+ are the learned embeddings of the input data. The unlabeled latent embeddings z and z^+ are then compared to the latent embeddings of the labeled support mini-batch, using the similarity classifier π_d. The comparison outputs vectors p, respectively p^+ that represent a probability distribution of class labels. The objective function $H(p^+, p)$ ensures, that the probability distributions of p^+ and p are similar. To avoid node collapsing the target

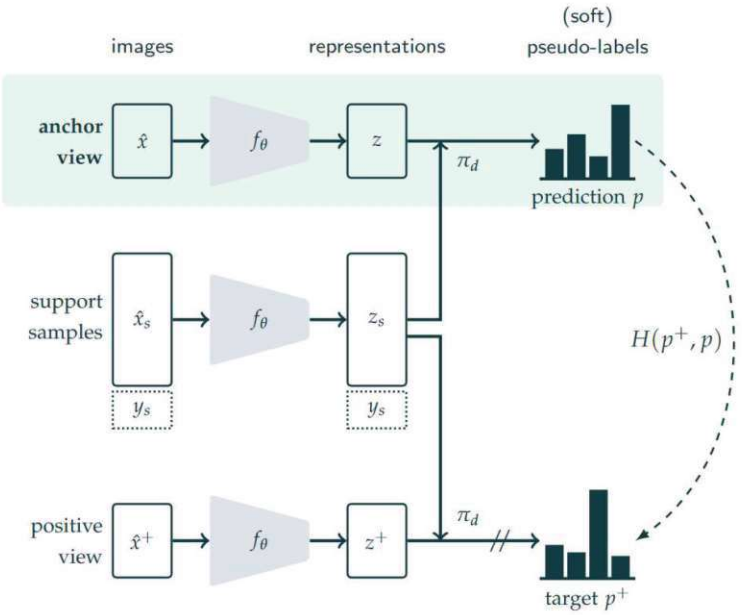


Figure 3.4: PAWS architecture developed by Facebook AI Research. The blue rectangles represent the value distribution for the vectors p and p^+ [ACM+21].

p^+ is sharpened before being passed to $H(p^+, p)$. The sharpening function is discussed in the paragraph *Node Collapsing Avoidance*.

Similarity Classifier π_d

The classifier function $\pi_d(z_i, \boldsymbol{z_S})$ (Equation 3.4) outputs a one-hot encoded vector that indicates how similar z, respectively z^+ is to the support samples in the mini-batch $\boldsymbol{z_S}$. The function π_d sums up the distances $d(.,.) \geq 0$ between a sample embedding z_i of the unlabeled samples and the labeled mini-batch latent embeddings z_S. The one-hot ground truth label that is associated with a labeled sample z_{S_j} of the support mini-batch z_S is denoted with y_j. Therefore y_j has the dimensions $N_{classes} \times N_{supportsamples}$.

PAWS uses the similarity metric $d(a, b) = \exp(a^T b / ||a|| ||b|| \tau)$ to calculate π_d. This is the exponential temperature scaled cosine similarity metric that considers the direction and the magnitude of the representation vectors a and b, divided by the similarity temperature τ.

Equation (3.4) applies this similarity metric to the matrix z_S and the vector z_i in order to calculate the similarity for multiple vector comparisons, i.e., the comparison of the support sample vectors stored in z_S with z_i, in parallel.

The resulting vector p_i stores the highest entry for the class with the highest similarity to z, respectively z^+. This result is conditioned by the prerequisite that each class is represented with an equal number of samples in the mini-batch of support samples.

$$p_i = \pi_d(z_i, \boldsymbol{z_S}) = \sum_{(z_S j, y_j) \in z_S} \left(\frac{d(z_i, z_S j)}{\sum_{z_S k \in z_S} d(z_i, z_S k)} \right) y_j. \tag{3.4}$$

Node Collapsing Avoidance

The PAWS sharpening function $\rho(.)$, given by

$$[\rho(p_i)]_k := \frac{[p_i]_k^{\frac{1}{T}}}{\sum_{j=1}^{K} [p_i]_j^{\frac{1}{T}}} \tag{3.5}$$

is used to steer the network towards eliminating all trivial solutions that map all input data to the same latent embedding. T is the target sharpening temperature parameter, set > 0. The temperature factor is used to change the distance between probabilities of the soft pseudo label. For example, the vector $[0.5, 0.25, 0.25]$ becomes $[0.8889, 0.0556, 0.0556]$.

Training Objective

The training objective of PAWS is to minimize

$$H(p^+, p) = \frac{1}{2n} \sum_{i=1}^{n} (H(\rho(p_i^+), p_i) + H(\rho(p_i), p_i^+)) - H(\bar{p}). \tag{3.6}$$

The objective function uses the cross entropy function H to measure how similar the distributions of a sample p_i and its corresponding positive view p_i^+ are. The cross-entropy is calculated twice for each sample, once p_i is sharpened and once p_i^+ is sharpened. The first term $\frac{1}{2n} \sum_{i=1}^{n} (H(\rho(p_i^+), p_i) + H(\rho(p_i), p_i^+))$ is the average over all cross-entropy calculations for all the unlabeled samples within a batch.

The second term $H(\bar{p})$ is a regularization term, called *mean entropy maximization (ME-MAX)* that aims to increase the entropy of an unlabeled training-batch. The parameter $\bar{p}$ denotes the component-wise average of sharpened latent embeddings p and p^+ over a batch, shown in Equation 3.7. Equation 3.8 uses ρ to compute the ME-MAX regularization. ME-MAX is inversely proportional to the entropy of ρ.

$$\bar{p} = \frac{1}{2n} \sum_{i=1}^{n} (\rho(p_i) + \rho(p_i^+)), \tag{3.7}$$

$$\text{ME-MAX}(p^+, p) = \sum_{i=1}^{n} \log(\bar{p}^{\,-\bar{p}}). \tag{3.8}$$

While sharpening aims to increase the confidence in the probability distributions, i.e., to decrease the entropy, ME-MAX is calculated over a batch and leads towards a uniform distribution, to ensure that each label is getting predicted. More concretely, distributions like $[[1., 0., 0.], [1., 0., 0.], [1., 0., 0.]]$ are penalized and distributions like $[[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]$ are favoured. This regularization assumes that indeed a uniform representation of class samples is present in an unlabeled training batch. If the batch rightfully predicts the class $[1., 0., 0.]$ for each label, the ME-MAX might be misleading the network.

3.3.3 GraphPAWS

The following comparison of GraphDINO and PAWS is leveraged to draft our GraphPAWS architecture.

Comparison architecture GraphDINO and PAWS		
	PAWS	GraphDINO
Input Type	Image	Graph
Anchor Encoder	f_θ := Deep Residual Network	Teacher Transformer
Pos. View Encoder	f_θ := Deep Residual Network	Student Transformer
Latent Repres. p_i	$p_i := \pi_i$ [Equation 3.4]	$p_i := MLP(z_i)$
Training Objective	Minimize Equation 3.6	Minimize Equation 3.3
Node Collapse Avoidance	Prediction Sharpening	Student-teacher Architecture
Semi-Supervised Learning Components		
Similarity Classifier	Equation 3.4	-
Similarity Metric	$d(a, b) = exp(a^T b/\|\|a\|\|\|\|b\|\|\tau)$	-

While GraphDINO is based on a student and a teacher network to embed the positive sample and respectively the anchor sample, PAWS uses the same encoder to embed the anchor sample, the positive sample, and the support samples. The outputs z / z^+ of the GraphDINO student - teacher encoders are transformed with a MLP and immediately passed to the objective function. The PAWS embeddings are further processed with the sharpening operation and with the previously discussed similarity classifier. Both GraphDINO and PAWS implement cross-entropy for the objective function. PAWS additionally implements the ME-MAX regularization.

Our architecture GraphPAWS is an adaptation of GraphDINO and PAWS. We extend the PAWS architecture by three input streams to leverage manually annotated support samples. The stream that processes the anchor view and the stream that processes the positive view use the same encoder. This replaces the GraphDINO student-teacher architecture. We employ the similarity classifier to compare the latent embeddings of unlabeled samples to the latent embeddings of the labeled support samples and we experiment with different objective functions that integrate the sharpening function and

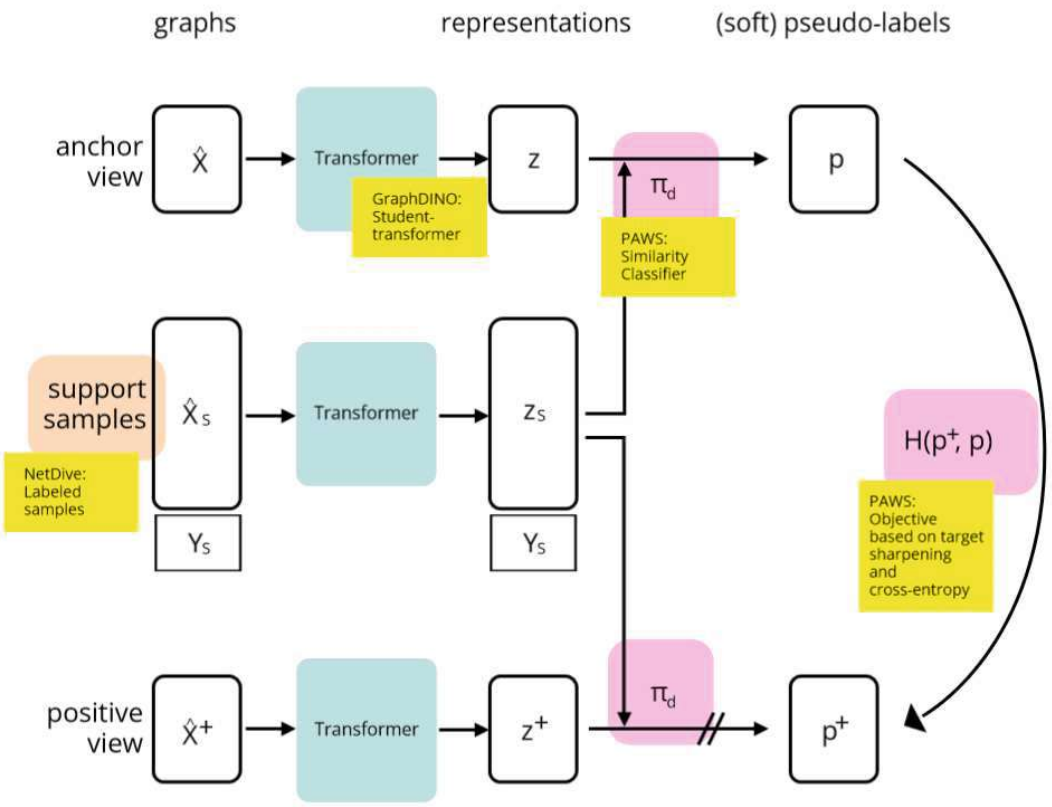


Figure 3.5: The GraphPAWS architecture developed in this thesis. Components with a blue background are adopted from GraphDINO, components with a pink background are adopted from PAWS and the component with the orange background, i.e., the support samples are generated in NetDive.

a regularization term, which we vary during the experiments. As PAWS is developed for images, we rely on the GraphDINO components to process graph data and embed them in the GraphPAWS architecture. We use the GraphDINO dataloader, the augmentations, and the graph transformer. The GraphPAWS architecture is depicted in Figure 3.5.

3.3.4 Training Objective

GraphPAWS can be trained with different objective functions. Besides the cross-entropy objective function employed in the original network of Weis et al. [WHLE21] we added a mean-squared error (mse) objective function to train models for both loss functions and to compare the performance of the resulting models.

The regularization term is added to the cross-entropy / mse loss. Equation 3.9 depicts the

cost function C in relation to the hyperparameters λ and γ that determine the relevance of the regularization terms *ME-MAX* and *One-Hot-Enforcement*,

$$C(p^+, p) = \text{loss}(p^+, p) + \lambda * \text{ME-MAX}(p) + \gamma * \text{One-Hot-Enforcement}(p). \tag{3.9}$$

The ME-MAX regularization term is adopted from PAWS, as denoted in Equation 3.8. While PAWS computes ME-MAX based on the average of sharpened latent embeddings p and p^+ over a batch, we oly use the average of sharpened latent embeddings p. We add an additional term that we name One-Hot-Enforcement, given by

$$\text{One-Hot-Enforcement(p)} = \frac{\sum_{i=1}^{N} \sum_{k=1}^{M} \log(\rho(p_{ik})^{-\rho(p_{ik})})}{N}, \tag{3.10}$$

with N being the number of samples and M being the dimension of the embedding vector p. This regularization term enforces one-hot encodings of the embedded vector p. One-Hot-Enforcement computes the logarithm of p_{ik} (represented by the parameter i in Figure 3.6) to the power of $-p_{ik}$ as depicted in Figure 3.6 for every component of the sharpened vector $\rho(p)$. If the vector is one-hot encoded, the result is 0, otherwise it is greater than 0. This is computed for every sample in the batch and the results are averaged.

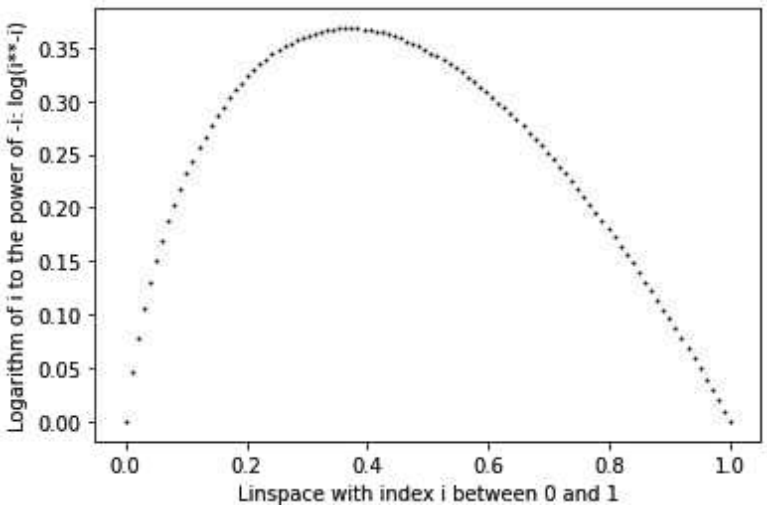


Figure 3.6: Logarithm of i to the power of -i

While ME-MAX operates over a batch of samples, One-Hot-Enforcement is applied to single training samples and averaged over a batch.

3.3.5 Graph Augmentation

The augmentations are a central component of contrastive learning. The network learns that the original graph and its augmented variation should be mapped closely in the

embedding space. In this manner, the network learns to become invariant to specific features that vary between the samples within a cluster.

Domain knowledge is necessary to know which variances we expect within a cluster.

Weis et al. [WHLE21] utilize the following augmentations :

1. **Subsampling**: Reduction of the graph to a fixed number of nodes. Nodes are randomly removed that have a maximum of two neighbors and are therefore no branching points. If the deleted node had one neighbor, no further actions are necessary, if the deleted node had two neighbors those are connected. Subsampling is applied in order to make the network invariant towards the manual tracing granularity of the neuron graph.
2. **Rotation**: 3D rotation around coordinate axes. Weis et al. [WHLE21] rotate along the y axis as the y axis is orthogonal to the pia, which is part of a protective membrane covering the brain and spinal cord. Rotation is applied in order to make the network invariant towards neuron rotations as Weis et al. [WHLE21] want to cluster neurons with similar shapes independent from their orientation in the brain.
3. **Jittering**: Random translation of node positions with a scaled Gaussian noise factor. Jittering is applied to make the network invariant towards small local structure changes.
4. **Branch Deletion**: Deletion of subbranches that contain a leaf node, which is not the soma. A subbranch starts from a branching point and does not contain further branching points. These subbranches are named *terminal branches*. Branch deletion is applied as variations of neuron types have varying numbers of dendrites.
5. **Translation**: Random translation of the whole graph with a scaled Gaussian noise factor. Translation is applied as the network should be invariant towards the position of the neuron within the brain.

We complement these augmentations with the augmentation *flip* to steer that neurons from the left and the right hemisphere with a similar but mirrored morphology are mapped to the same cluster. Figure 3.7 depicts such a pair of counterpart neurons from the left and right hemisphere. A flip along the yz plane would approximately align these neuron graphs.

We also discussed some further subsampling strategies. Subsampling is important, as experts trace neurons with varying accuracy. A good subsampling strategy teaches the network that the number of nodes and the density of nodes is less important than the global graph shape. Further potential subsampling approaches are mentioned in Chapter 7 under *future work*. For the experiments we kept the subsampling implementation by Weis et al. [WHLE21].

We document and discuss the experiments we did to choose augmentations and the values for the augmentations in Section 5.2 and Chapter 6.

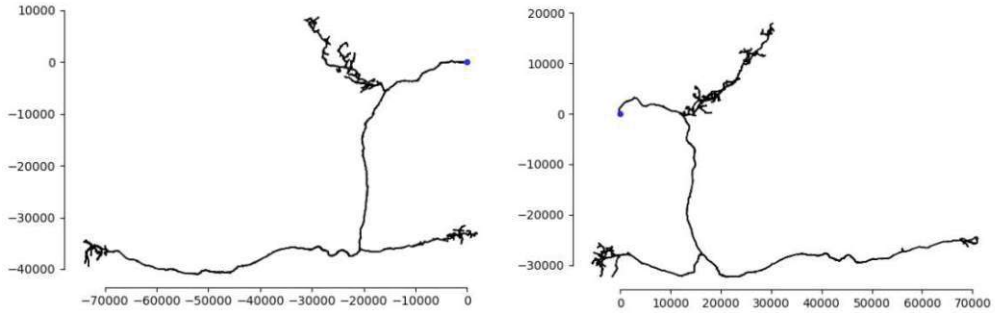


Figure 3.7: Counterpart neuron graphs from the left and right brain hemisphere.

3.3.6 Graph Alignment

In the previous Sub-section 3.3.5 we listed the augmentations we utilize to ensure that variations of morphologically similar graphs are embedded close to each other. The augmentations *translation*, *rotation*, and *flip* depend on the location of the neuron graph within the coordinate system. We need to ensure that we have previous knowledge about the optimal rotation axis, the optimal flip plane, and whether and when translation has to be applied. In order to enforce prerequisites on the orientation and position of neurons within the coordinate system, we experiment with alignment strategies as a preprocessing step.

We perform a principle component analysis (PCA) on all graph nodes and align the axis with the highest variance with the x axis. We call this alignment strategy *PCA alignment*. Another strategy we test is PCA based on solely the nodes that belong to the main branch of the neuron, which is the longest path that we find starting from the soma. We then align the main branch with the x axis, again by aligning the axis with the highest variance. We call this alignment strategy *Main branch PCA alignment*. These approaches are further elaborated in Section 5.3 and discussed in Chapter 6.

3.4 Clustering

We experiment with two clustering algorithms, k-means and Gaussian Mixture Models (GMM), both discussed in Sub-section 2.1.2. While k-means is based on distance measurements, GMMs are based on data distributions and therefore are able to reveal more complex patterns than k-means.

The choice of a clustering technique is interlinked with the objective function that we use for model training, as we want to cluster accordingly to what the model learned and considers to be *similar* or *dissimilar*. For the training we experiment with the mean-squared error and the cross-entropy loss. The mean-squared error loss minimizes the Euclidean distance between latent embeddings while the cross-entropy loss optimizes towards finding the optimal decision boundary between data distributions. Therefore we expect the GMM to perform better on models trained with the cross-entropy loss.

GMM depends on random seeds for the initialization and k-means is initialized with k-means+, which is optimized to find good clusters and is not fully deterministic. Both GMM and k-means are initialized with a predefined number of clusters. Weis et al. [WHLE21] fit 1000 GMMs with varying random seeds for 2-30 clusters using five-fold cross-validation to find the optimal number of clusters. For the sake of simplicity, we let the number of clusters be defined by the users (which can use NetDive to explore the graphs and can set the number of clusters based on their findings), therefore we omit this step.

We adapt the method developed by Weis et al. [WHLE21] to find the optimal model. They fit 100 GMMs with varying random seeds and perform a five-fold cross-validation to determine the performance of each model. Weis et al. choose the GMM with the highest average adjusted rand index (ARI).

The adjusted rand index (ARI) is calculated as

$$\mathrm{ARI} = (\mathrm{RI} - \mathrm{RI}_{Expected})/(max(\mathrm{RI}) - \mathrm{RI}_{Expected}), \tag{3.11}$$

with the Rand Index (RI) being the ratio of the number of pairs of data points that are classified the same way in both clusterings to the total number of pairs. ARI has a value between -0.5 and 1, with values close to 0.0 for random labeling, values approaching 1.0 for consistent labeling, and approaching -0.5 for especially discordant clusterings [scib].

3.5 Dimensionality Reduction

In order to visualize the clusters in three dimensions we take advantage of varying dimensionality reduction techniques that focus on different aspects of the data patterns. Principal component analysis (PCA) computes the axes with the highest variances and projects the data onto these axes. T-SNE reduces the dimensionality by stochastically preserving nearest neighbor relationships between points. UMAP is similar to t-SNE as it also preserves local structures, but scales better for larger datasets and nonuniform distributions. PCA is a deterministic algorithm while t-SNE and UMAP are probabilistic algorithms, though the reproducibility of the results can be improved by using a fixed seed for the initialization of t-SNE and UMAP.

3.6 NetDive

The visual analytics (VA) tool NetDive addresses the topics *Model Selection*, *Model Analyses*, and *Model Improvement*. During the network training with GraphPAWS we encountered the problem that the network did not produce embeddings that cluster well.

Whilst computing the ARI score gives us a performance metric (which is not always reliable as discussed in Section 5.4), we aim to visualize the input data contributing to a cluster to identify outliers, and to visually evaluate the quality of the cluster separation.

The visual inspection provides knowledge that numeric metrics like the ARI score can not provide. We embedded NetDive in the pipeline discussed in Section 3.1, such that the user can add new annotations to specific data points and restart the training within NetDive.

3.6.1 Goals and Tasks

NetDive supports the user to analyze and compare latent embeddings and to subsequently improve the model by adding new labels. The VA tool is developed for model engineers that design and refine the model and for domain experts to explore complex graph data and to choose a model from the pre-trained model database. Model engineers can use NetDive during the training to fine-tune a model based on the user input, whilst domain experts can use NetDive after the training to explore the embeddings. NetDive is designed for the case study of exploring similarity scores in neuroscience. The concept of NetDive is data type agnostic and can be adapted for other use cases.

The goals of NetDive are (G1) to select a specific model, (G2) to give the user an analyses tool to explore the clustering and the cluster contents for specific model embeddings, and (G3) to tightly integrate the developer into the training process (human in the loop) in order to improve the model. Table 3.1 depicts the goals and associated tasks to achieve these goals. The goals and tasks were developed based on related work and discussions within the Biomedical Image Informatics department at VRVis.

In the subsequent sub-sections we discuss the design of the user interface and the implementation of the tasks in order to achieve the goals G1-G3.

3.6.2 User Interface

According to Shneiderman's mantra *overview first, zoom and filter, then details on demand* [Shn96], we present the data point embeddings corresponding to a specific model as scatter plots and the user can request details for selected neurons that are depicted in a separate panel.

The user interface (UI) consists of four panels. Two global views as depicted in Figure 3.8(1) and Figure 3.8(2) and a detail panel as depicted in Figure 3.8(3). The two view panels enable the user to load the latent embeddings processed by two (different) models to compare the results. Each view includes a detail view realized as accordion menu, depicted in Figure 3.8(4) to set the parameters. Both views include a canvas that displays the dimensionality reduced latent vectors of the neurons in three dimensions as scatter plots.

The user can expand and collapse components of the UI to maximize the space available to interact with the components of interest. The dynamic components are the detail panel, the view panels, the legends, depicted in Figure 3.9 on the bottom right image, and the accordion menu. Figure 3.9 shows different layout setups that the user can configure.

G1	G2	G3	
■			**T1** *Selection* of a specific model based on training-data, hyperparameters, and version. We want to be able to select and load latent embeddings of specific models using dropdown menus that display the available data. The model should be uniquely identifiable by its training hyperparameters, the training dataset, and a version name.
	■		**T2** *Visualization of the latent embeddings* in a scatter plot.
	■		**T3** *Analyses of the cluster contents*. The user should be able to get input data information about data points on demand. The user should be assisted in identifying and analysing clusters.
	■		**T4** *Comparative analyses* based on model and attribute data. We want to compare latent embeddings generated by two (different) models or compare the cluster assignments for latent embeddings generated by the same model based on different dimensionality reduction algorithms or ground truth assignments.
		■	**T5** *Relabeling* of misclassifications. We want to manually assign cluster labels to input data. The user should be able to assign labels to data points that are misclassified and to data points that are prototypical for a cluster, i.e., represent a cluster well.
		■	**T6** Starting the *retraining* with new cluster labels. We want to process the new labels in the training.

Table 3.1: NetDive goals and their associated tasks. **G1**: Model Selection, **G2**: Cluster Analyses, **G3**: Model Improvement

Global View

NetDive requests the pre-computed latent embeddings from the filesystem using the API of our NetDive backend server. On demand, the backend applies dimensionality reduction to the requested feature vectors. The dimensionality reduced data points are displayed as a scatter plot in the global view. The global views embed parameter panels presented as accordion menus to set parameters in order to select and analyze the embeddings generated by a specific model.

Parameter Panel

The parameter settings are divided into four cards. The first card *Data* regards the training data, the training data preprocessing, and the inference-data. The second card *Deep Learning* covers deep learning parameters. The third card *Analysis* provides options to analyse the loaded data, and the fourth card *Presets* includes presets to reproduce results that are discussed in this thesis. The settings are depicted in Figure 3.10.

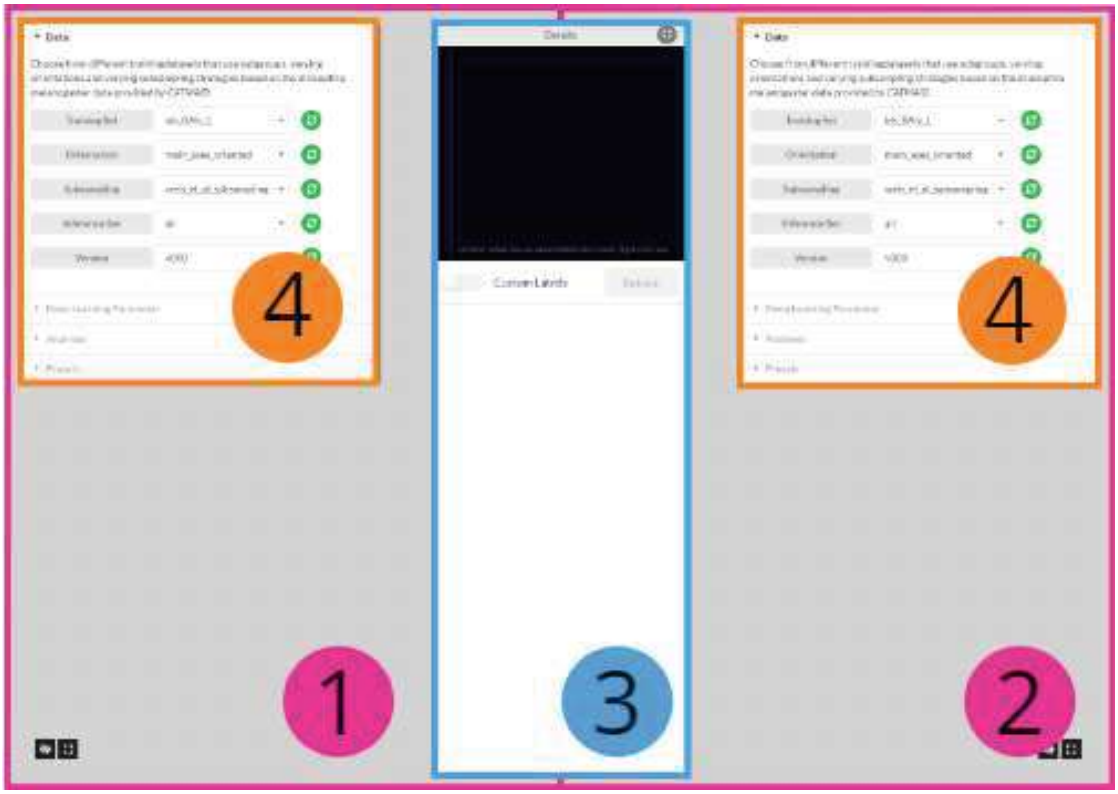


Figure 3.8: NetDive layout: (1) First global view, (2) second global view, (3) detail view, (4) parameter panel.

Detail View

The detail panel has two parts, as depicted in Figure 3.11. In the top area, marked with A in Figure 3.11, the user can interact with a 3D graph representation of a neuron, which is selected in the bottom half of the detail panel. The bottom half, marked with B in Figure 3.11, is a scrollable container that lists tiles, marked with D in Figure 3.11, depicting image galleries of pre-rendered preview-images for each selected neuron. Between the 3D graph representation and the tile area, marked with C in Figure 3.11, the user finds elements to activate the relabel feature and to start the network training using the relabeled data.

3.6.3 Task Implementation

T1: Model Selection

To select the embedding data generated with a specific model, the user can set the parameters in the accordion cards *Data* and *Deep Learning Parameters* to match that specific model. The options in the dropdown menus for these parameters correspond to the available models that the backend server can access.

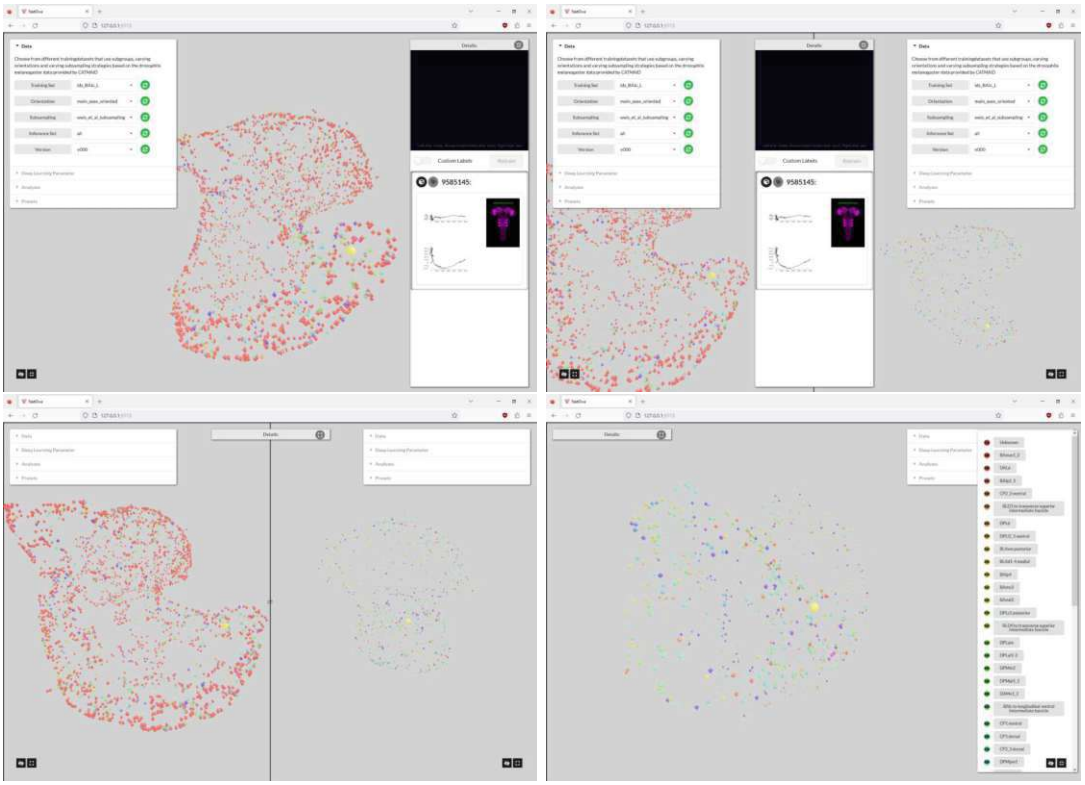


Figure 3.9: Layout configurations in NetDive.

The *Data* card provides parameters regarding the training data and the preprocessing techniques. The pre-processing includes *Orientation* and *Subsampling*. The options for these parameters depend on the models that are available to the user and the metadata that is encoded for these models. In chapter 5 we discuss the pre-processing techniques we implemented for our use case. The user can also define, which subset of the data to use for the inference. The *Version* refers to different versions of a model. Version v000 is the model that was initially trained with the self-supervised GraphDINO model, while other versions are generated with the same hyperparameters and the same training data, but with additional support samples set in NetDive. The user can specify the version name before starting the retraining.

The *Deep Learning Parameter* card displays parameters that refer to the deep learning hyperparameters, set during the model training. Each available model encodes these hyperparameters within the storage path, with each position in the path corresponding to the value of a specific parameter. The same applies to the storage path of the pre-computed embeddings. Therefore NetDive can associate the embeddings with the models that were used to generate the embeddings.

If the user changes settings that effect other settings, default parameter values are chosen for the settings that are invalidated by the change. This is the case, if e.g., the user

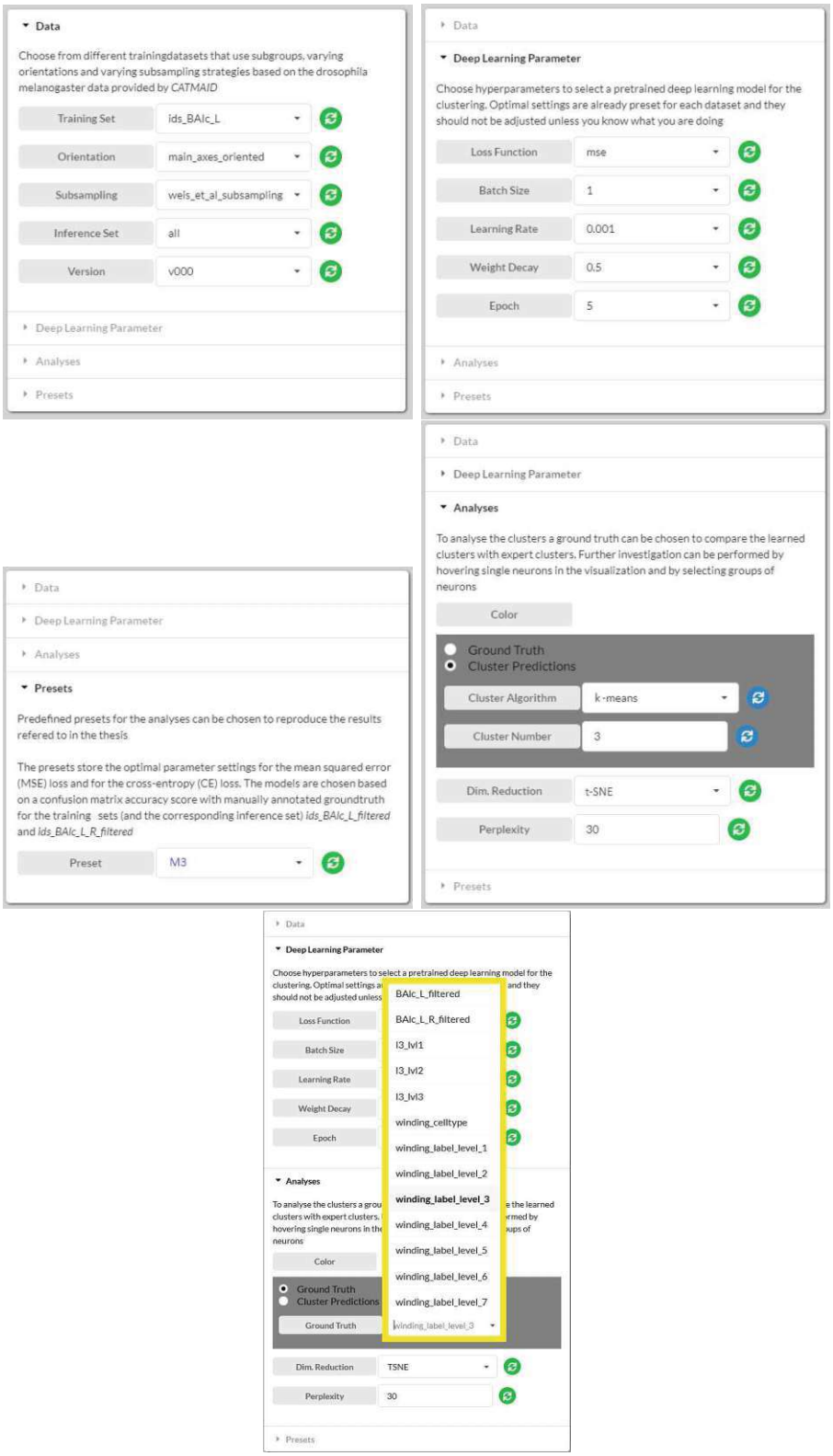


Figure 3.10: NetDive: Parameter panel with accordion layout. Different cards are expanded in the individual screenshots. The yellow rectangle highlights the ground truth clusterings we integrated for our use case.

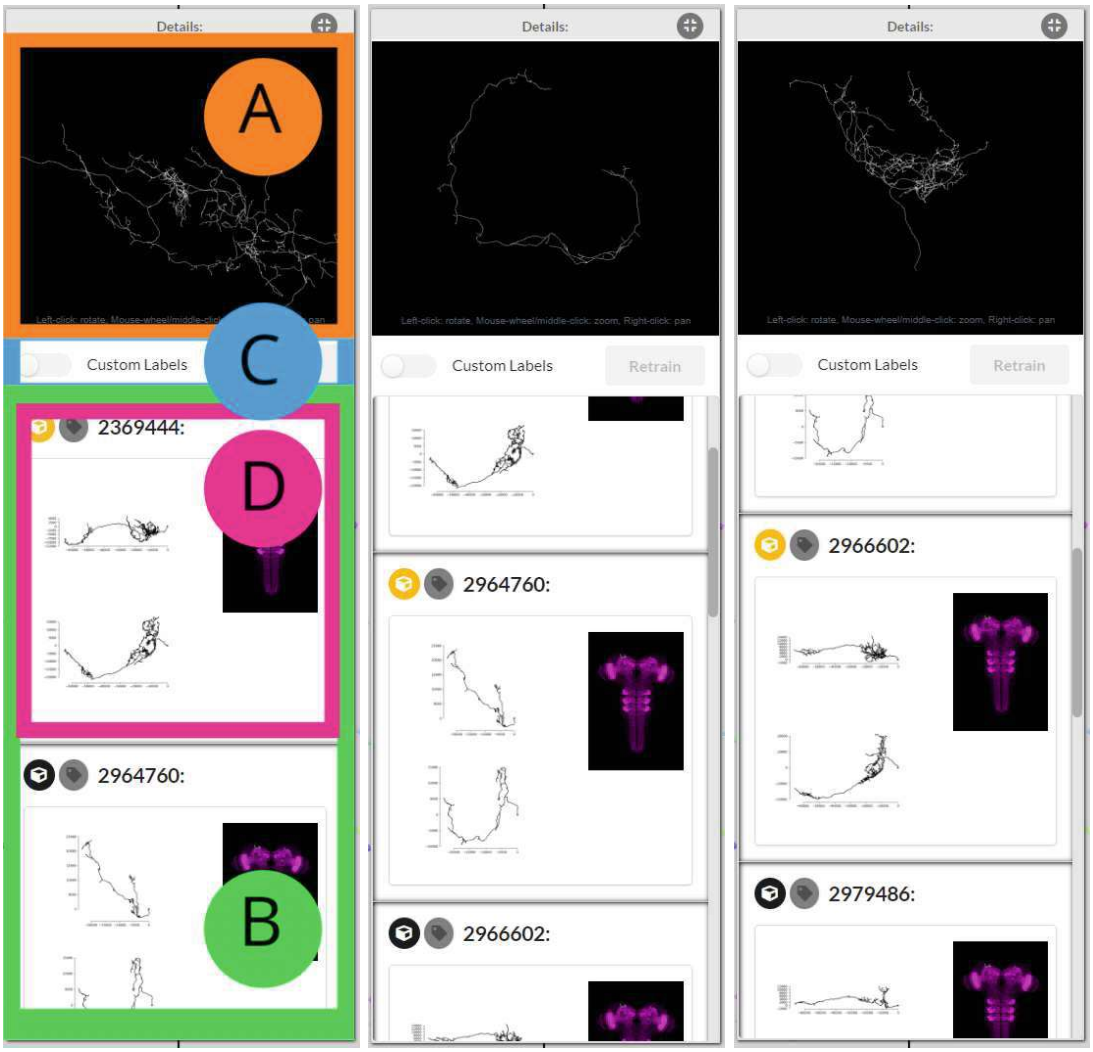


Figure 3.11: NetDive: Detail views.

selects a batch size and no model exists with the selected batch size and the learning rate that was previously chosen. In this case the learning rate is set to a default learning rate that exists for the selected batch size.

T2: Visualization of the Latent Embeddings

NetDive visualizes the embeddings as a scatter plot in three dimensions. We chose to visualize the data in three instead of two dimensions, as we can keep more information after the dimensionality reduction. This can come with the downside of distortion and occlusion. As clustering is not dealing with issues of length and angle preservation, and the points representing the neurons are not covering a lot of space, which limits the issue of occlusion, we accepted these downsides.

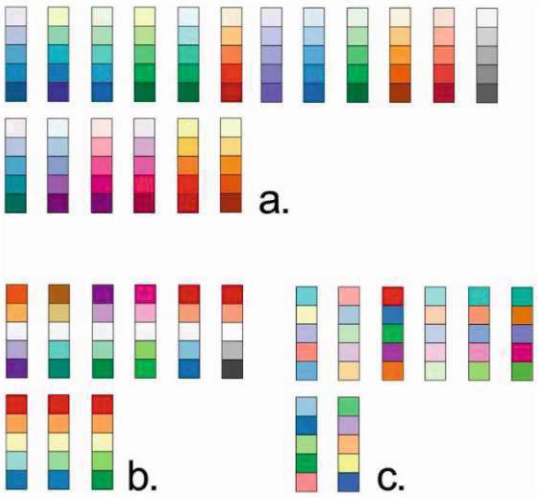


Figure 3.12: 35 colour schemes defined by Brewer et al. [HB03] divided in the categories: (a) sequential, (b) diverging and (c) qualitative.

The scatter plots visualize spatial and attribute information. In contrast to spatial data, attribute data includes any information that is unrelated to spatial information. We visualize the cluster labels as attribute information for each data point with color assignments.

Choosing the right color scheme for the label encoding is essential, as the human perception is biased to associate colors with meaning. We refer the reader to ColorBrewer `https://colorbrewer2.org`. The website is based on research by Cynthia Brewer, and helps the user to select color schemes that e.g., consider potential color-blindness and varying display environments and media (digital and print) [HB03]. Brewer et al. divide the color schemes in *sequential*, *diverging*, and *qualitative* ones as depicted in Figure 3.12. Since sequential and diverging color schemes suggest that the colors have an order or a certain degree of similarity to each other, we make sure to use qualitative colors to color-code the cluster labels.

The scatter plots are depicted in three linked views, implementing the concept of multiple linked views (MLVs), displayed in juxtaposition. MLVs link the information displayed in each view to the others based on user interactions.

T3: Analyses of the Cluster Contents

The user can vary the dimensionality reduction algorithm and its configuration as well as the color coding of the scatter plots to analyze the scatter plots within the *Analyses* card in the parameter panel.

The latent embeddings of each processed neuron graph have 32 dimensions and therefore can not be displayed in the user interface without a dimensionality reduction. The user can choose the dimensionality reduction algorithm. We integrate UMAP (Uniform

Manifold Approximation and Projection), t-SNE (t-Distributed Stochastic Neighbor Embedding), and PCA (Principal Component Analysis). The user can set the number of neighbors for the UMAP dimensionality reduction and the perplexity for the t-SNE algorithm.

The color is either assigned based on a chosen ground truth or based on a clustering algorithm. The available ground truth labels are the neuron annotations by Volker Hartenstein [LNO+13], by Michael Winding [WPB+23], and our manual annotations. The Winding clusters are hierarchical, therefore the user can choose the granularity level for the Winding clusters. We integrate the Volker and Hartenstein clusters as options to be selected in the parameter panel to aid the user to explore correlations between these clusters and our generated morphology based clusters. If the color is assigned based on the clustering algorithm, the user chooses between the k-means algorithm and Gaussian Mixture Models (GMMs) and can select the number of clusters.

Besides the color coding and the dimensionality reduction, we provide user interactions to explore the embeddings. We implement the interaction techniques *filtering*, *multi-selection*, and *camera movement*.

The user can hide and show neuron clusters by opening the legend at the bottom corners of the views as displayed in Figure 3.13. The legend displays each color group with the corresponding name, which is the cluster name of the ground truth cluster or a generic name that is assigned to each cluster after the cluster algorithm was applied. Next to each cluster is an eye icon that can be toggled to hide and show the data points belonging to the respective cluster.

The user can select single or multiple data points in the scatter plot or select a neuron cluster from the legend, displayed in Figure 3.14. Selected neurons are colored yellow. The three panels in the UI (two views and one detail panel) are linked. Therefore a neuron selection in one view also leads to a color change in the other view and details about the selected data point or data points are listed in the detail view. The user can interact with the canvas elements using zoom, panning, and orbiting.

T4: Comparative Analyses

By loading the embeddings of two models in the first global view and the second global view, depicted in Figure 3.8, the user can compare embeddings, either by assigning the same ground truth to two different model embeddings or different ground truth colorings to the same embedding.

T5: Relabeling

If the relabeling feature is initially activated, all neuron data points are rendered in gray. The relabel feature disables the color encoding that is set in the parameter panel. After the activation, the user can select neurons from the detail panel and open the relabeling modal, i.e., an overlay window that disables the interaction with the underlying website

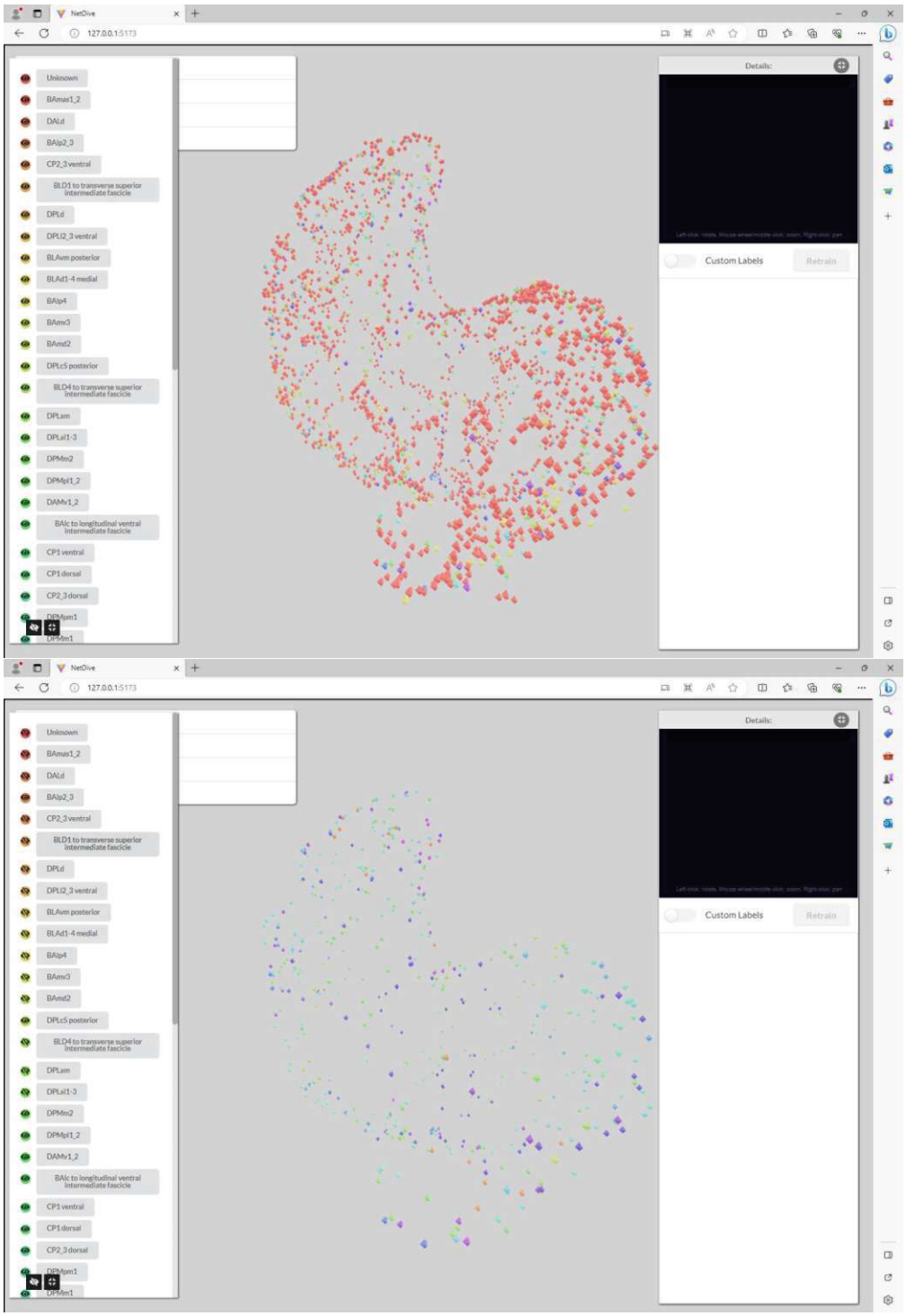


Figure 3.13: NetDive: Filtering.

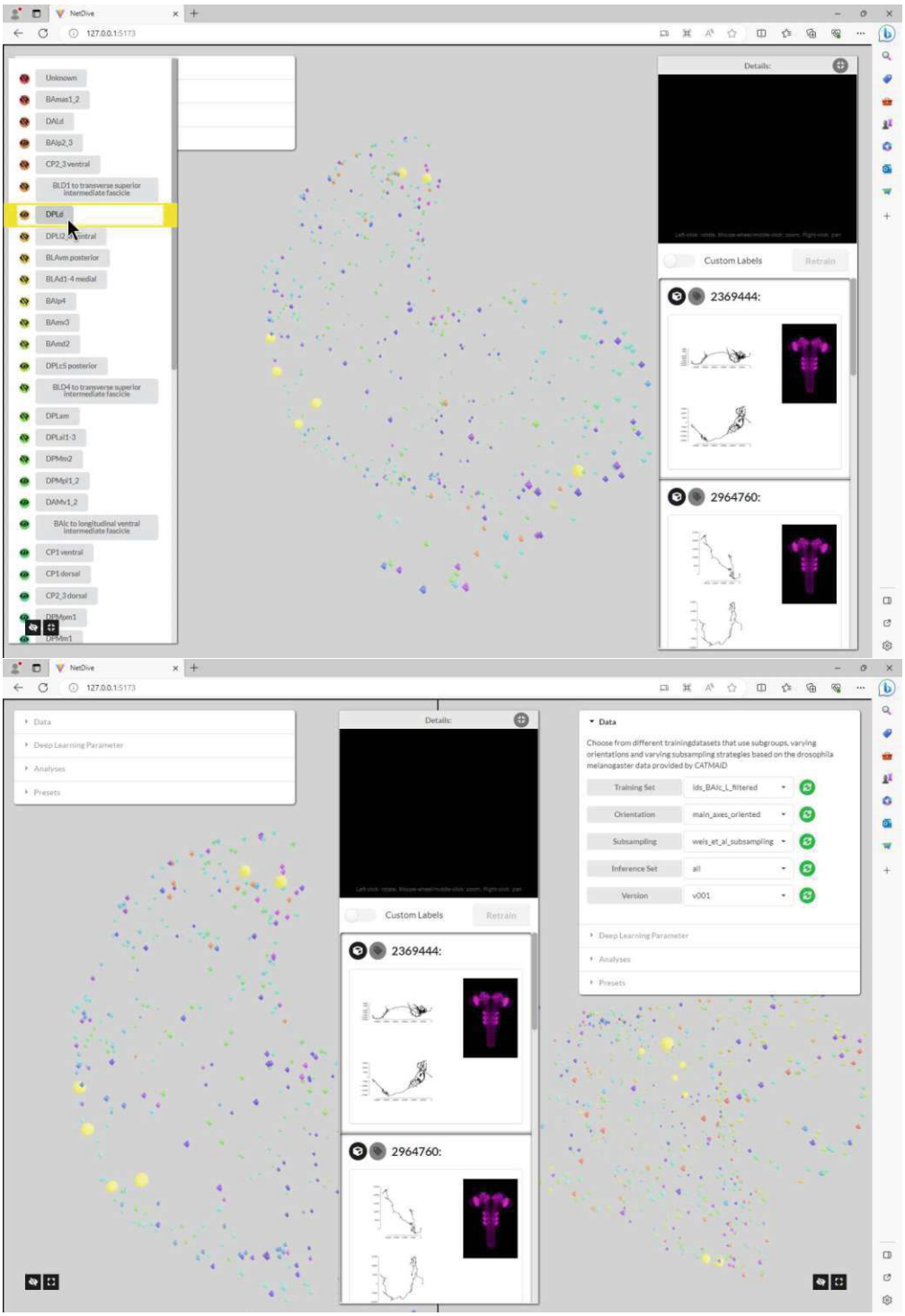


Figure 3.14: NetDive: Selection.

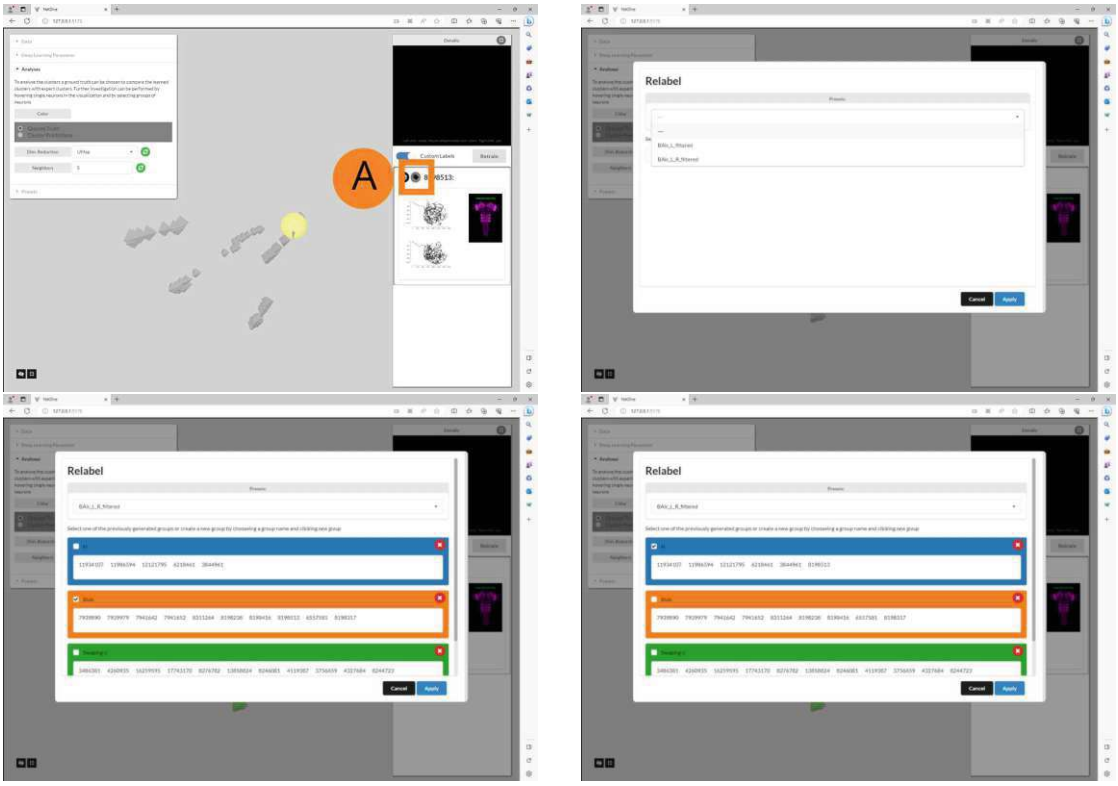


Figure 3.15: NetDive: The user can assign a color to a selected node. The relabel button on the neuron tile within the detail view, maked with A, opens a modal (top left). The user can choose a preset to color the neuron or they can manually generate a new label group (top right). The user can assign the currently selected node to a group by clicking the checkbox next to the group (bottom left and right).

until the user actively prompts the window to close. The modal is depicted in Figure 3.15. With the *New Group* button, visible in Figure 5.25, the user creates a new cluster group with the previously entered group name. If the user tries to create a group without a name or with a taken name they are prompted to update the group name first. The new group is added to a color-coded list of hand-labeled neuron groups. The user can choose one of the groups to add the currently selected neuron. After closing the modal, the user sees that the previously gray neuron is colored with the group color as depicted in Figure 3.16 (left). PAWS [ACM+21] requires the semi-supervised training to use an equal number of annotated data samples for each cluster. To ensure a minimum number of samples for each hand-labeled group, we could have added a check that ensures an equal number of labels for each cluster before starting the retraining. To be more flexible with our experiments we did not add this check and adapted the PAWS model architecture to allow unequally distributed support sample labels in each batch.

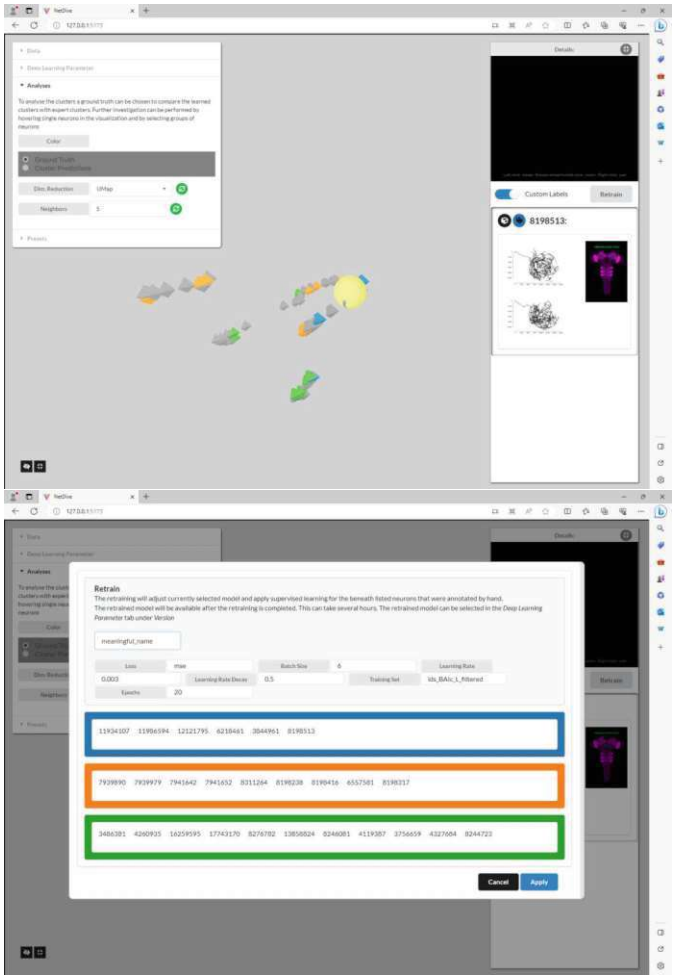


Figure 3.16: NetDive: The user can retrain the model with the currently loaded hyperparameters and the previously assigned labels that act as support samples during the training. The user needs to click the Retrain button (left), to provide a version name, and to confirm the hyperparameters and support samples (right).

T6: Retraining

If the user triggers the retraining, a confirmation window, displayed in Figure 3.16 (right), with a list of the relabeled neurons and the currently set parameter settings is prompted to avoid unintentional actions. The parameter settings that are passed on to the retraining process are: *loss*, *batch size*, *learning rate*, *learning rate decay*, *training set* and the number of *epochs* until the training terminates. The data in the confirmation window is sent to the backend server. A *subprocess call* activates the retraining environment and calls the retraining script. After the retraining, the embeddings generated with the retrained model are available to be selected in the parameter panel.

CHAPTER 4

Implementation

This chapter includes the data preparation, ground truth preparation, the setup of the filesystem, implementation details for the deep learning architecture GraphPAWS, presented in Sub-section 3.3.3, and for the visual analytics application named NetDive, presented in Section 3.6.

4.1 Data Preparation

The training data for the architectures GraphDINO and GraphPAWS has to be in a specified format. We export the neuron graph .swc files from CATMAID and change the node attributes to fit our use case. The extracted files from CATMAID store seven features for each node. The node features are [swc]:

- SampleID
- TypeID
- x
- y
- z
- r
- ParentID

The SampleID is a unique ID to identify each node in the graph, the TypeID encodes the neuron compartment (depicted in Figure 1.1), x, y, z encode the spatial data of each

node, r is the radius of a node, as a neuron can have varying thicknesses along the curves and the ParentID is used to find the node that is connected to the current node. The dataset we obtained from CATMAID contains radius features with zero values, indicating that this information has not been annotated yet. The TypeID values are: undefined, soma, postsynaptic node, and presynaptic node. The connectivity is not encoded in the neuron data. We solely keep the features SampleID, x, y, z, and ParentID to reconstruct the neuron morphology, we reduce the TypeID information to *soma* or *not soma* and we omit the radius feature.

4.2 Ground Truth

For cluster analysis in NetDive, we organize the available ground truth annotations, allowing the user to seamlessly switch between different ground truth colorings. We generated a single json file that stores multiple ground truth cell type labels for each neuron id. For clarification: The neuron id is an id that identifies the whole neuron, while the SampleID identifies a specific node in the neuron graph representation. The annotation files include manually labeled cell types and expert cell type annotations by Michael Winding and Volker Hartenstein [LNO+13] that are exported from CATMAID.

4.2.1 Michael Winding et al.

Michael Winding et al. [WPB+23] published connectdome annotations for the drosophila melanogaster on CATMAID and describe them in their paper. Winding et al. generated a dataset with all central nervous system (CNS) neurons, sensory neuron axons, and motor neuron dendrites. While we want to only operate on the graph morphology, i.e., the shape, topology, and spatial information of the graph, Winding et al. solely consider the synaptic connectivity. They assigned cell types to 3.016 neurons and additionally assigned connection-types and brain-wide circuit motifs to the connectome. The neurons of the two drosophila melanogaster larval brain hemispheres are mostly mirror-symmetric. We also leverage the mirror-symmetry for the manual annotations of cell types based on morphology later on. Winding et al. paired the corresponding mirror-symmetric neurons and identified 90 neuron clusters using an unbiased hierarchical clustering method. They state that the cell types based on connectivity did correlate with cell types annotated for other features like morphology and functionality.

4.2.2 Volker Hartenstein et al.

Volker Hartenstein [LNO+13] analyzed lineages, that describe neurons deriving from the same stem cells called *neuroblasts*. He states that neurons within a lineage do not only share the same stem cell, but are also alike regarding the morphology. In this thesis we define datasets based on lineages, i.e., the Datasets 1-3 described in Section 5.1. Dataset 1 and 2 are based on lineage BAlc and Dataset 3 corresponds to lineage CM4. Lineage BAlc neurons are located in the lateral surface of the antennal lobe and lineage CM4 is located in the postero-medial brain cortex. As depicted later, the morphology of neurons

in lineage BAlc visually divides the lineage in three clusters that we manually annotated. Other lineages like lineage DILP have a visually coherent morphology. Similar to lineage BAlc, the neurons in lineage CM4 fall into four clusters regarding the morphology that we manually labeled.

```
{
    "10013517": {
        "l3_lvl1": "Unknown",
        "l3_lvl2": "Unknown",
        "l3_lvl3": "Unknown",
        "winding_celltype": "ct PNs",
        "winding_label_level_1": "1_level-1_clusterID-0_walksort-0.593",
        "winding_label_level_2": "3_level-2_clusterID-3_walksort-0.719",
        "winding_label_level_3": "5_level-3_clusterID-9_walksort-0.638",
        "winding_label_level_4": "12_level-4_clusterID-21_walksort-0.693",
        "winding_label_level_5": "26_level-5_clusterID-42_walksort-0.778",
        "winding_label_level_6": "42_level-6_clusterID-81_walksort-0.769",
        "winding_label_level_7": "68_level-7_clusterID-142_walksort-0.744",
        "BAlc_L_filtered": "Unknown",
        "BAlc_L_R_filtered": "Unknown"
    },
    "10017944": {
        "l3_lvl1": "CM4 longitudinal ventroposterior fascicle",
        "l3_lvl2": "Centro-medial",
        "l3_lvl3": "Unknown",
        "winding_celltype": "ct pre-dSEZs",
        "winding_label_level_1": "1_level-1_clusterID-0_walksort-0.593",
        "winding_label_level_2": "3_level-2_clusterID-3_walksort-0.719",
        "winding_label_level_3": "7_level-3_clusterID-8_walksort-0.849",
        "winding_label_level_4": "15_level-4_clusterID-19_walksort-0.851",
        "winding_label_level_5": "29_level-5_clusterID-39_walksort-0.859",
        "winding_label_level_6": "48_level-6_clusterID-74_walksort-0.828",
        "winding_label_level_7": "76_level-7_clusterID-131_walksort-0.822",
        "BAlc_L_filtered": "Unknown",
        "BAlc_L_R_filtered": "Unknown"
    },
    "10018150": {
        "l3_lvl1": "CM4 longitudinal ventroposterior fascicle",
        "l3_lvl2": "Centro-medial",
        "l3_lvl3": "Unknown",
        "winding_celltype": "ct dVNCs",
        "winding_label_level_1": "1_level-1_clusterID-0_walksort-0.593",
        "winding_label_level_2": "3_level-2_clusterID-3_walksort-0.719",
        "winding_label_level_3": "7_level-3_clusterID-8_walksort-0.849",
        "winding_label_level_4": "14_level-4_clusterID-18_walksort-0.848",
        "winding_label_level_5": "27_level-5_clusterID-36_walksort-0.821",
        "winding_label_level_6": "45_level-6_clusterID-69_walksort-0.817",
```


Figure 4.1: Screenshot of the json file that stores the ground truth annotations for each neuron id.

Figure 4.1 depicts a screeshot of the ground truth data. The json file is embedded in NetDive and the user can select a ground truth coloring in the UI as depicted in Figure 3.10.

The available annotations in NetDive are the Michael Winding clusters (hierachical clustering named *winding_label_level_<1-7>* and the winding_celltype cluster), the

Volker Hartenstein [LNO+13] clusters (*hartenstein_level_<1-3>*) and the manually annotated ground truth labels (*BAlc_L_R_filtered* and *CM4_L_R_filtered*). The manual annotations are described in Section 5.1.

4.3 Filesystem

Both the GraphPAWS training and NetDive heavily depend on predefined folder structures on the filesystem to write and read data depending on a given set of hyperparameters.

Models are stored to a path that encodes training parameters *version*, *fold*, *loss*, *training set*, *alignment strategy*, *subsampling strategy*, *batch size*, *learning rate*, *learning rate decay* and *epoch*:

```
<version>/<fold>/<loss>/<training_set>/<alignment_strategy>/<subsampling_strategy>/<batch_size>/<learning_rate>/<learning_rate_decay>/<epoch>.ckpt
```

The latent representations of the inference data are stored in json files titled *latent.json* and are embedded in a folder structure identical to the folder structure for the models within a different root directory.

4.4 GraphPAWS

4.4.1 Implementation Environment

We adapted the codebase of GraphDINO [WHLE21] and combined it with semi-supervised training components of PAWS [ACM+21] according to the architecture discussed in Subsection 3.3.3. Next to the package ssl_neuron from GraphDINO, we generated a package named graphdino_paws_neuron. We duplicated files of GraphDINO that we needed to adapt and included them to the graphdino_paws_neuron package. Files that did not need to be adjusted are imported from ssl_neuron. This setup preserves the original code of GraphDINO [WHLE21] and highlights the adaptations we made.

4.4.2 Data Loading

The data loading requires preprocessing steps in which files referencing the neuron data are generated that should be loaded during training, validation, and testing. We store this information in numpy files and load these during training. Therefore the files have to be embedded in a previously defined folder structure and follow a naming schema.

The graph data that corresponds to the previously saved neuron ids will then be requested from the filesystem during runtime. Each neuron graph is composed by the **features** file, which stores the spatial data, the **neighbors** file, which stores the topology, and a

file named **parents** which stores the parent node of each node with the axon being the root node. The latter file is utilized to calculate the longest axis for the graph alignment preprocessing, discussed in Sub-section 3.3.6.

The features are stored in separate files and updated according to the orientation. We store the spatial features of the graphs as provided by CATMAID, the main branch PCA aligned feature data, and the PCA aligned feature data.

The feature orientation could have been computed during runtime instead of determining it in a preprocessing step. We decided to outsource the processing to save computation time during training and to be more flexible in case we want to leverage the oriented data also in the NetDive visualization.

4.4.3 Class Balancing

We adjusted the similarity classifier of PAWS [ACM+21]. The similarity score calculation in the similarity classifier is an unweighted cosine similarity. It causes graphs being classified as more similar to samples that are represented more frequently in the support sample batch. Therefore, PAWS states that the support samples have to be evenly distributed over all classes in each batch. We want to enable the user to provide an unbalanced number of samples for each identified class. To compensate for the class imbalance we calculate class weights and multiply them with the results. First, we normalize the query and support vectors, then we count how often each class is represented in the support vectors and based on this number of instances of each class ($N_{instances}$) we calculate the class weight of each class as noted in Equation 4.1,

$$\text{class\_weights} = N_{\text{support\_samples}}/(N_{classes} * N_{instances}), \tag{4.1}$$

with $N_{support\_samples}$ being the number of support samples in the support sample batch and $N_{classes}$ being the number of classes represented in the support samples. We ensure that each support sample is represented at least once in each support sample batch, otherwise we would divide by zero in Equation 4.1. Besides that it would be counter-productive to compute the similarity between the query and the support samples if some classes are not represented. Equation 4.2 denotes the computation of the weighted similarity score for a query vector that we referenced with z / z^+ in Figure 3.5,

$$\text{similarity\_score} = \sigma\left(\frac{query \cdot \text{support\_samples}}{\tau} \cdot \text{labels} \cdot \text{class\_weights}\right). \tag{4.2}$$

We generate a new support sample batch for each iteration of a training epoch. We therefore implement an iterator that endlessly cycles through elements from the support sample loader. The iterator processes a pytorch *DataLoader* object and yields a new batch of support samples on demand, i.e., by calling NEXT. To ensure that each class is represented in the batch, we run a check and generate a new batch of support samples in case that a class is missing.

4.4.4 Architectural Details

The graph transformer returns a vector embedding of 32 dimensions and a projection of these 32 dimensions to 1000 classes. For the semi-supervised training we pass the embeddings of the 32 dimensions to the similarity classifier to compute the label for the input neuron graph. The vector length of the label corresponds to the vector length of the support vector labels and is determined on demand in the similarity classifier.

4.4.5 Evaluation

We implemented jupyter notebook scripts to analyse the latent embeddings of the inference sets and to visualize different aspects of the models and the model training. We solely pre-computed the latent embeddings for the subsets that we used for the inference as the computation of the latent embeddings for the whole drosophila dataset consisting of 2541 neuron graphs for one model takes approximately one hour (hardware details are listed in Chapter 5).

We decided not to compute the latent embeddings for varying inference sets on demand whilst interacting with NetDive as the computing time disrupts the user experience. The available inference sets are listed in the *Data* card in the NetDive accordion menu.

While Weis et al. [WHLE21] train with five-fold cross-validation we perform four-fold validation to have sufficiently many annotated samples to validate the results. We report the results on a test set that was not previously used for training and validation. We choose the model with the highest average performance across folds by fitting 100 GMMs to each fold and computing the ARI between the ground truth labels and the labeled neurons in the validation set.

We implemented scripts to compute the ARI, denoted in Equation 3.11, based on the manually annotated ground truth and the inference latent embeddings. We visualize the ARI scores for multiple models in plots to find patterns of which hyperparameters lead to better results. We further tracked the loss curves during training, collect those, and visualize them in image grids.

4.5 NetDive

4.5.1 Frontend

The frontend renders the NetDive application as depicted in Figure 3.8.

The frontend implementation tech stack is:

- **React**: React is a popular JavaScript library to build user interfaces. It provides a component-based architecture that allows developers to easily create reusable UI elements. The modularity of React allows to reuse the *view component* for NetDive to implement two views for comparison of the neuron data points. React

implements efficient DOM (Document Object Model) updates using a virtual DOM, which is a lightweight copy of the DOM that exists entirely in memory. Updates are first applied to the virtual DOM and only if the updates lead to a difference between the updated virtual DOM and the previous state of the virtual DOM, the actual DOM is rerendered. Therefore React is a good framework to build complex, interactive web applications. In comparison to the Angular Javascript framework, React is more lightweight and produces less boilerplate code when being bootstrapped.

- **Vite.js**: Vite is a build tool to create a React application. Until 2021, the most popular React build tool, developed at Facebook, was *Create React App*. Create React App, however, installs over 140 MB of dependencies, which is often irrelevant overhead. A lightweight alternative tool is *Vite* with 31 MB of dependencies. Additionally, Vite does not rebuild the whole bundle after each change by making use of the browser-native ES (ECMAScript) modules. This makes Vite a fast tool to create, update, and build React apps [Cho]. Since 2021 Vite is installed more often than Create React App [npm]. NetDive is built with Vite.

- **Semantic UI**: Semantic UI is a UI framework to style web applications. Semantic UI provides user-friendly syntax to define composed classes. Predefined colors and styling elements support the developer to create a coherent design. Additionally, Semantic UI uses *behaviors* that trigger Javascript functionality to create responsive components. Behaviors specify the functions and properties that an element implements. They can be accessed using spaced words, camelcase, or dot notation. Semantic UI has an official React integration named Semantic UI React [rea]. Semantic UI React provides React components, which wrap the styling and behavior of Semantic UI elements. We use Semantic UI to style all the components in NetDive, including the buttons, the accordion menu, the split between the views, and the tiles that embed the neuron details.

- **Three.js**: Three.js is a JavaScript library to create 3D graphics within a web application. It includes a wide range of methods and classed to generate geometries and materials. A competitor of Three.js is Babylon.js, released three years after Three.js in 2013. Both libraries are highly customizable while offering a higher abstraction layer than for example D3. Other applications that embed 3D visualizations work with even more advanced 3D engines, for example Unity and the Unreal Engine. Those are preferable, if the developer wants to generate complex 3D scenes with simulations. As this is not the case for NetDive, Three.js provides the adequate amount of complexity to visualize and interact with the data points. Both views of NetDive have a canvas element embedded that contains the data points, rendered with Three.js.

- **3d-force-graph** [for]: 3d-force-graph is a library to visualize graphs. We use it to render the interactive three dimensional representation of the neuron in the detail component. The library offers a variety of force layouts that we surpass by

specifying the specific spacial locations of each node in the graph. The user can orbit around the neuron and zoom in and out to explore the neuron shape.

- **Axios**: Axios is a promise-based *HTTP Client* for node.js. NetDive uses Axios to communicate with the backend that provides local data and processes data before passing it on to the React application.

4.5.2 Backend

The frontend needs to access data on the local filesystem. Therefore I implemented a backened server that serves the data and applies preprocessing to the data if necessary.

The backend implementation tech stack is:

- **Flask**: Flask is a Python web framework with a built-in development server that supports url routing. We use flask to access the filesystem to serve the inference latent data, images of neurons, graph data for the graph 3D rendering and the ground truth json file described in Section 4.2. It runs per default on `localhost:5000`.
- **Swagger** [swa]: The flask API provides a swagger documentation, which is available under: `localhost:5000/docs` after starting the flask server. Swagger provides the user with a description about each REST endpoint, the schema of the return value, and the input parameters. The user can process test calls with custom values that match the input parameter schema.

Application Programming Interface

Figure 4.2 depicts the swagger specification of the Flask server.

`GET /api/datapoints`: This endpoint returns a dictionary of neuron ids as the keys and the three dimensional spatial position of the neuron latent representations as values. The caller has to provide the following information as body parameters to select the inference data points accordingly from the filesystem:

- Loss function used for the training?
- Training set used for the training?
- Inference set used to select the latent representations?
- Batch size used for the training?
- Learning rate used for the training?
- Weight decay for the training?

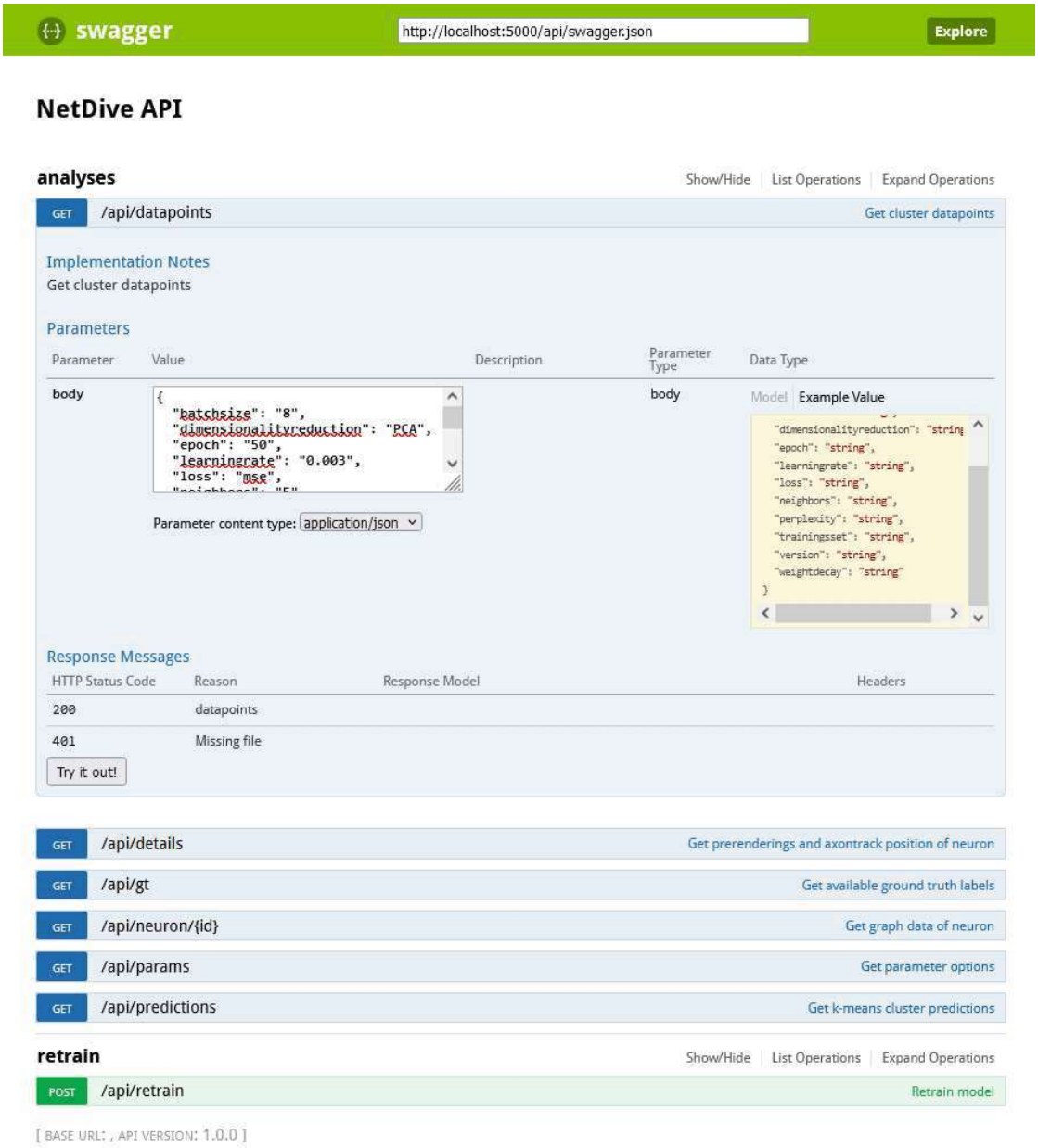


Figure 4.2: Swagger API visualization of NetDive flask REST endpoints

- Based on which training epoch should the latent representation be selected?
- Which version based on the previous parameters should be selected?
- Which dimensionality reduction algorithm should be used to reduce the 32 dimensions of the latent representations to three dimensions for visualization?

- Depending on the dimensionality reduction algorithm further parameters are required: The neighbor parameter needs to be set for UMAP and the perplexity needs to be set for t-SNE.

`GET /api/details/{id}`: This endpoint expects a neuron ID and returns a zip file containing two pre-renderings of the input neuron from different angles, along with an image showing the axon tract location within the drosophila melanogaster brain dome through which the neuron passes.

`GET /api/gt`: This endpoint returns varying ground truth annotations for each neuron as a dictionary, depicted in Figure 4.1.

`GET /api/neuron/{id}`: This endpoint expects a neuron id and returns the 3D graph representation of the corresponding neuron. The return format is parsed, such that 3d-force-graph can process it: nodes: Array<Object>, links: Array<Object>

`GET /api/params`: This endpoint returns all available parameter options that are then displayed in the UI in the accordion menu. Therefore the filesystem is scanned for available paths and the location of a substring specifies the feature. The caller can pass optional values for some features, so that the options for the other features are selected accordingly. For example, the caller can set the option *version* to *v002* and this endpoint will return option values for other hyperparameters that are available for version v002. The hyperparameters are identical with the options for endpoint `/api/datapoints` without the dimensionality reduction parameters and the inference set.

`GET /api/predictions`: This endpoint returns a dictionary of the neuron ids mapped to labels. The labels are predicted using the clustering algorithm k-means or the clustering algorithm GMM, depending on the caller input. The caller also has to provide the training hyperparameters and the inference set as body parameters to choose the latent representations to cluster.

CHAPTER 5

Experiments

The experiments chapter includes an introduction to the datasets (section 5.1), and the augmentations (section 5.2) and the graph alignment (section 5.3) we use. Based on this we perform the training experiments for self-supervised learning and semi-supervised learning (section 5.4). We conduct an ablation study regarding the augmentations during the semi-supervised learning and demonstrate the iterative training using NetDive. The experiments are built on top of each other, therefore parts of the discussion Section in Chapter 6 are already briefly covered in order to provide a reasoning for the subsequent experiments.

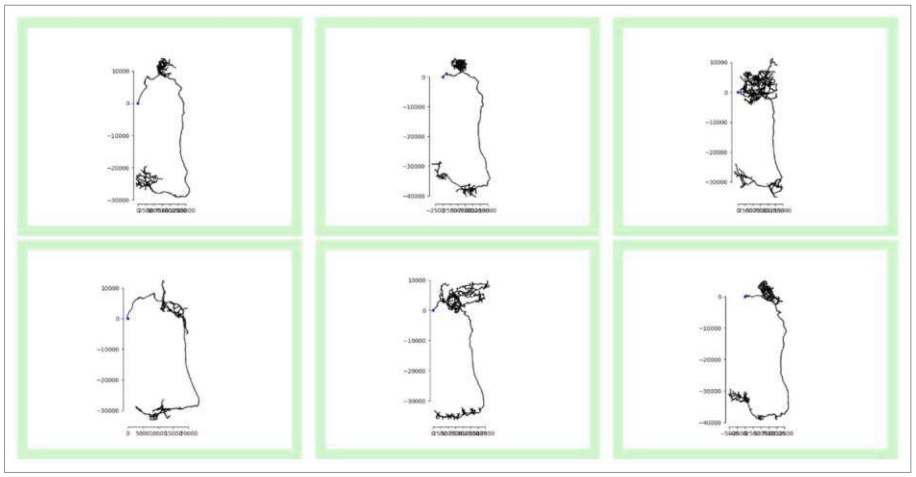
5.1 Datasets

5.1.1 Drosophila Melanogaster

We defined four subsets of the drosophila melanogaster dataset, that is exported from CATMAID [SCHT09], to train and to test the pipeline defined in Section 3.1:

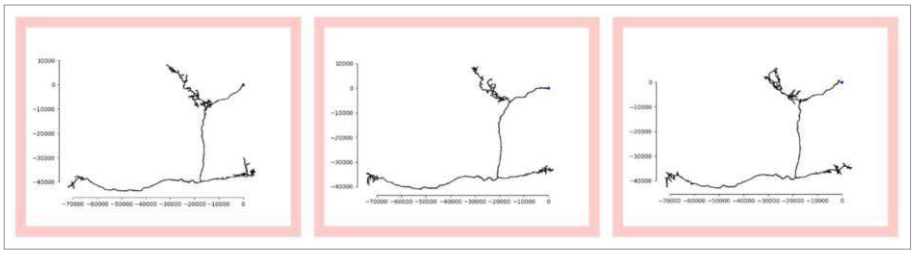
- Dataset *BAlc L / R filtered*: A subset of lineage BAlc with hand-selected neurons that we can visually divide and annotate with cluster labels grouping morphologically similar neurons. Lineage BAlc contains neurons from the left brain hemisphere (BAlc L) and the mirrored neurons of the right brain hemisphere (BAlc R). The neurons are depicted as image collections in the Figures 5.1-5.3. Cluster 1 neurons resemble a semi-circle, Cluster 2 neurons have an H shape and Cluster 3 neurons have a very dense topology within a sphere. The dataset consists of 26 neurons, 13 in each brain hemisphere.
- Dataset *BAlc L / R*: The unfiltered lineage BAlc. This dataset consists of 81 neurons, including the 26 neurons from dataset *BAlc L / R filtered*.

(a) Neuron graphs of left hemisphere

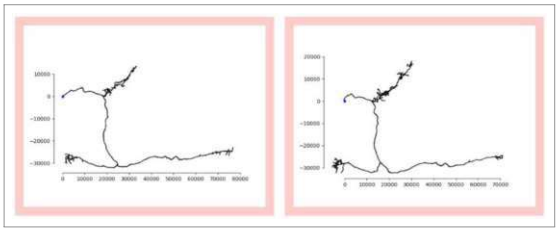


(b) Neuron graphs of right hemisphere

Figure 5.1: ***BAlc L / R filtered* Cluster 1** neuron graphs with the ids: **(a)** 3486381, 4260935, 16259595, 17743170, 8276782; **(b)** 13858824, 8246081, 4119387, 3756659, 4327684, 8244723



(a) Neuron graphs of left hemisphere



(b) Neuron graphs of right hemisphere

Figure 5.2: ***BAlc L / R filtered* Cluster 2** neuron graphs with the ids: **(a)** 11934107, 11986594, 12121795; **(b)** 6218461, 3844961

- Dataset *CM4 L / R*: The unfiltered lineage CM4. Lineage CM4 consists of 66 neurons, 33 in each brain hemisphere. We separate this lineage into 4 classes, as color coded in Figure 5.4. The neuron graph representations without a border in Figure 5.4 are not assigned to a class.
- Dataset *All*: All available neurons. After removing all neuron graphs with less than 200 nodes and less than 25 synapses, as well as the ones that have no unique soma assigned, 2519 neurons remain.

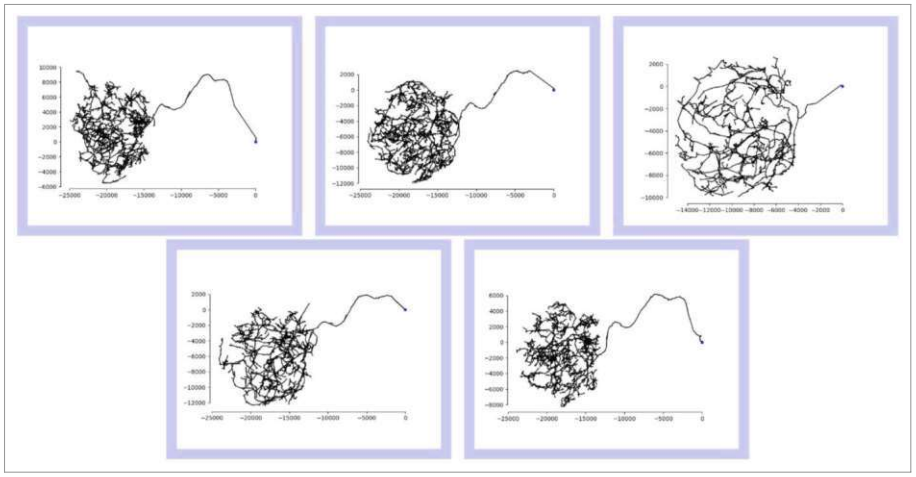
For the self-supervised learning we initially used the dataset with all 2519 neurons. As the self-supervised training did not output distinct clusters, we concluded that we have to guide the network training and train on subsets that we can evaluate.

For the the following experiments we evaluate the latent embeddings of the self-supervised and semi-supervised models trained on dataset *BAlc L / R* by computing the ARI scores between the predicted clusters and the manually annotated clusters of dataset *BAlc L / R filtered*. We demonstrate the visual analytics tool NetDive on dataset *CM4 L / R*.

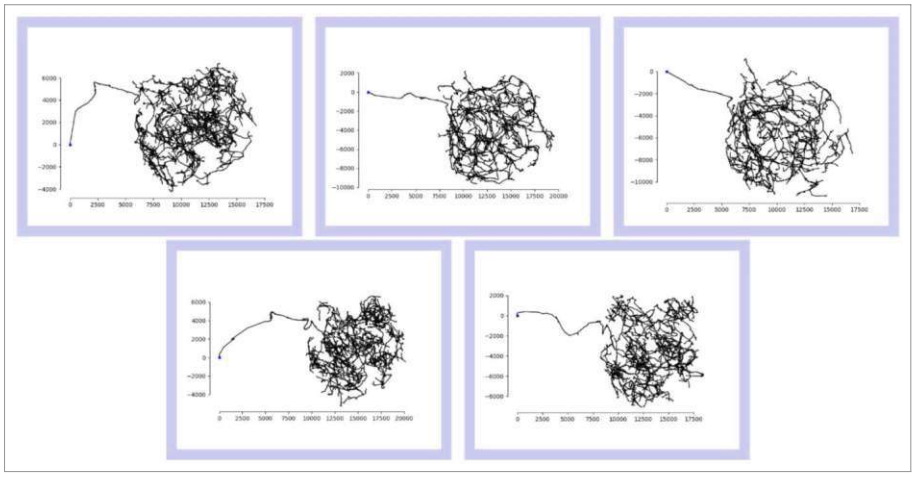
5.1.2 Comparison with Allen Brain Atlas (ABA): Mouse Visual Cortex

The original GraphDINO [WHLE21] was trained and evaluated on the ABA [aba] dataset.

The neuron graph representations used by GraphDINO encode the features $v_i = [s_i, r_i, c_i]$ with s_i being the respective xyz-coordinates, r_i being the radius of node v_i and c_i being a (1x4) one-hot encoded vector that determines the compartment that a node represents,

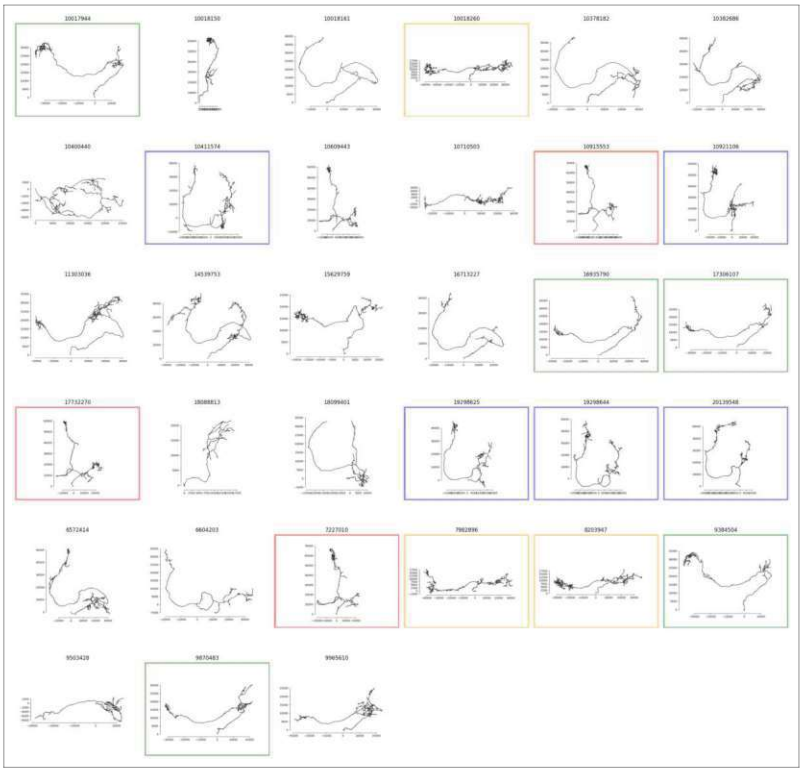


(a) Neuron graphs of left hemisphere

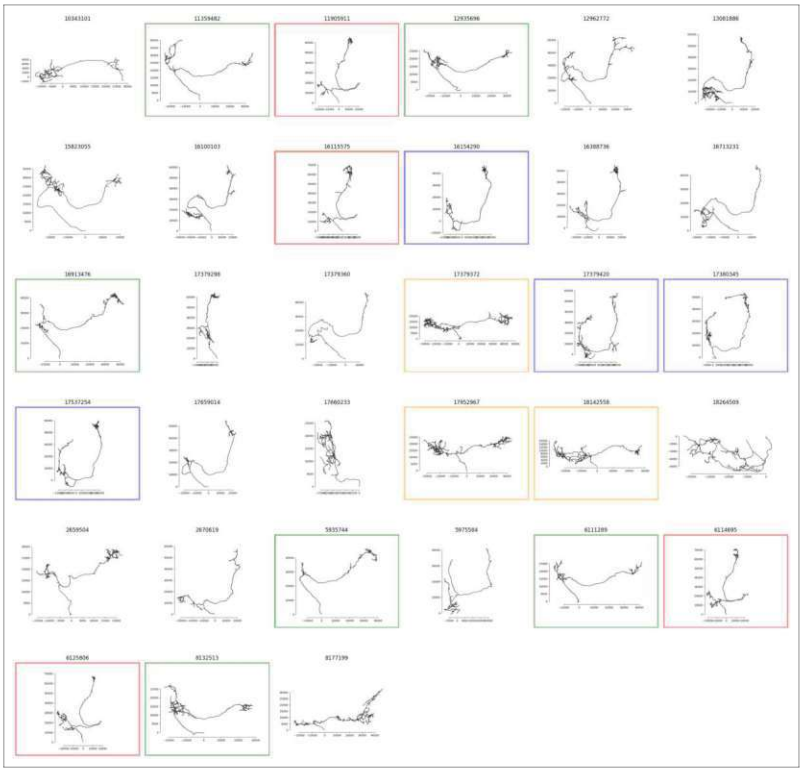


(b) Neuron graphs of right hemisphere

Figure 5.3: ***BAlc L / R filtered* Cluster 3** neuron graphs with the ids: **(a)** 7939890, 7939979, 7941642, 7941652, 8311264; **(b)** 8198238, 8198416, 8198513, 6557581, 8198317

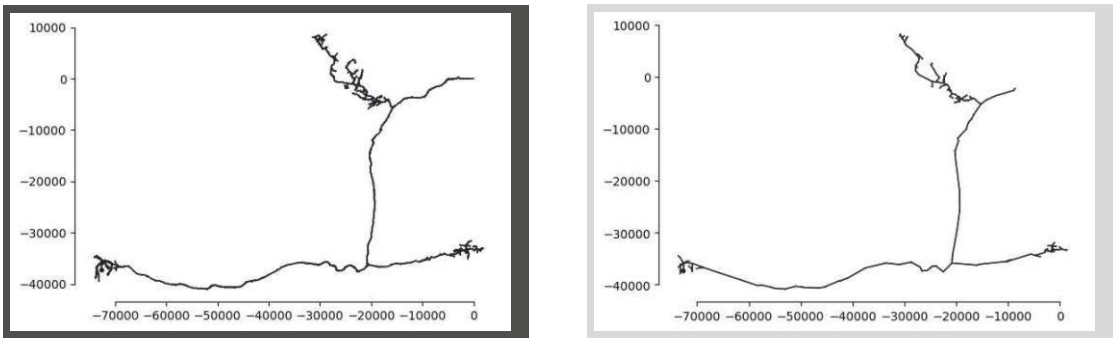


(a) Neuron graphs of left hemisphere



(b) Neuron graphs of right hemisphere

Figure 5.4: ***CM4 L / R*** neuron graph representation of (a) the left hemisphere and (b) the right hemisphere



(a) original graph representation (b) reduced graph representation

Figure 5.5: Neuron with id 11986594 reduced to 200 nodes. Both graphs are not augmented.

i.e., a soma, an axon, a basal dendrite, or an apical dendrite. In comparison, we only use the xyz-coordinates of a neuron representation graph as the radius information for the drosophila melanogaster data is not available and the specification of a node compartment is not sufficiently reliable and sparse.

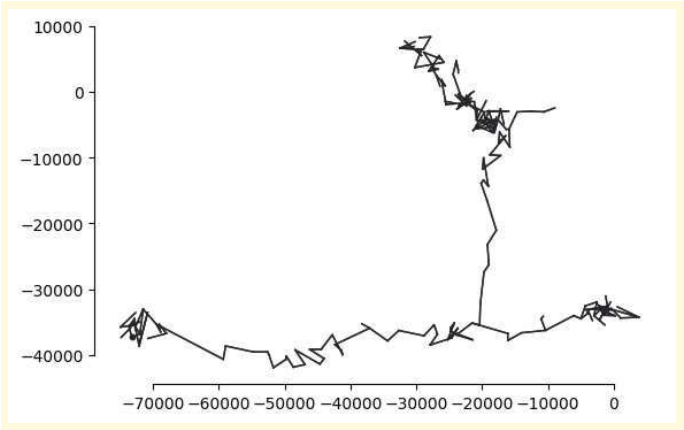
Weis et al. [WHLE21] performed an ablation study to test the effect of architectural and input data choices. The effect of leaving out the neural compartment information did improve the model performance from 51,5% to 54,3% accuracy. The authors did not elaborate why they did not drop this compartment information even though the results were better without. The omission of neural compartment information seemingly did not have a negative effect and this limitation of the drosophila melanogaster dataset therefore should not be critical.

5.2 Data Augmentations

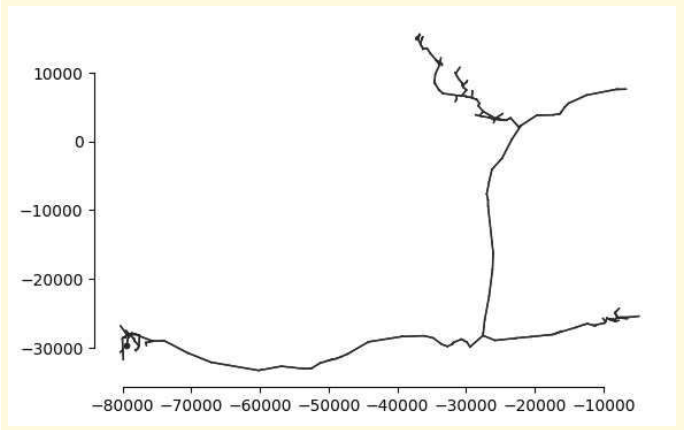
We experimented with different augmentations in order to vary the input sufficiently to generalize to graphs of the same cluster. We want the model to be invariant to rotations and positions of graphs and to generalize to graphs with similar shapes. We used the augmentations of the GraphDINO implementation and updated the configuration settings to fit our data.

The adopted and implemented augmentations are (1) Subsampling, (2) Rotation, (3) Jittering, (4) Branch deletion, (5) Translation.

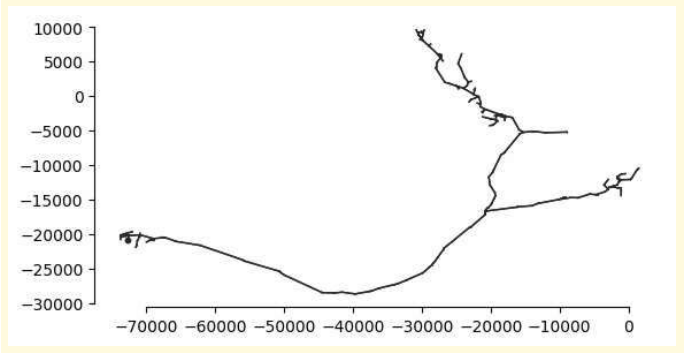
Figure 5.5 depicts a graph with 3103 nodes (a) that is subsampled and reduced to 200 nodes (b). Figure 5.6 (a) depicts a graph that is augmented using node jittering. Each node is randomly displaced with a normal distribution with mean 0 and variance 1, multiplied by a scaling factor that we set to 1000. Figure 5.6 (c) depicts a random rotation along the x-axis and Figure 5.6 (b) a translation with factor 10.000 along x,y,z



(a) reduced jittered graph representation

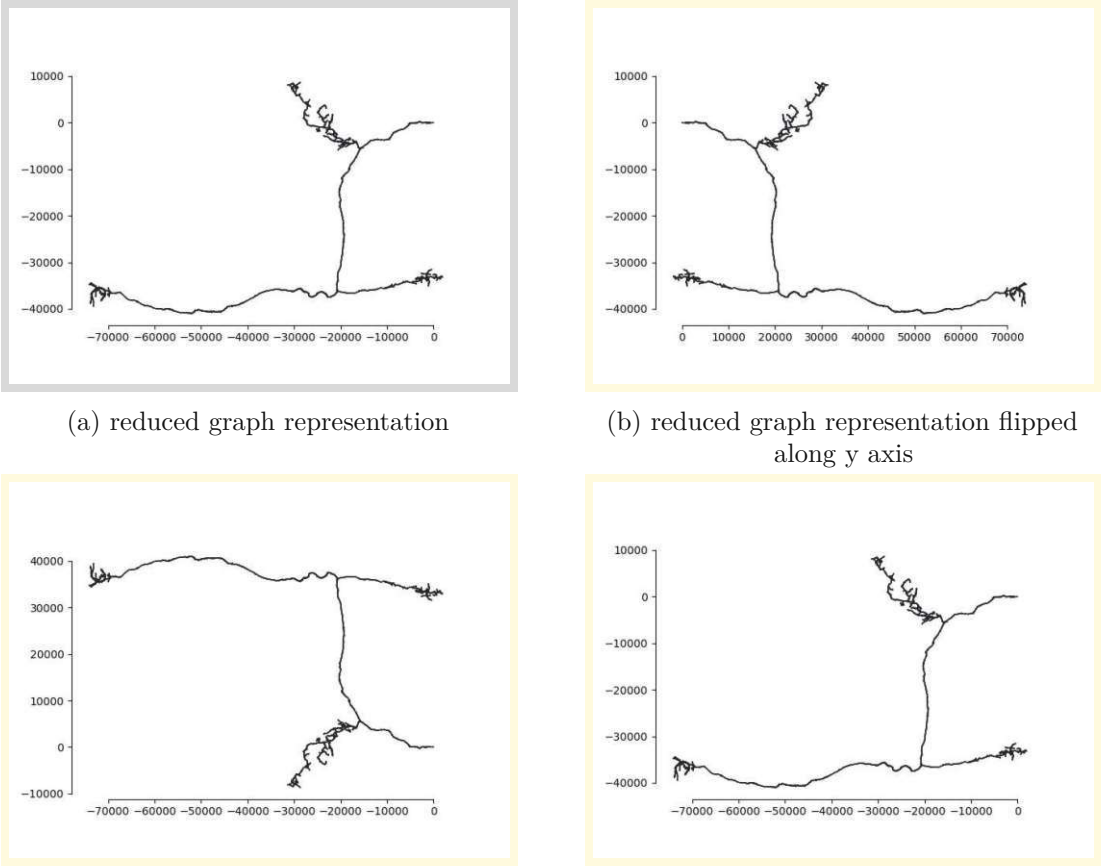


(b) reduced translated graph representation



(c) reduced rotated graph representation

Figure 5.6: Neuron with id 11986594, augmented, (a) with node jittering variance = 1000, (b) with translation = 10000, and (c) with rotation around the x axis.



(a) reduced graph representation

(b) reduced graph representation flipped along y axis

(c) reduced graph representation flipped along x axis

(d) reduced graph representation flipped along z axis

Figure 5.7: Neuron with id 11986594 reduced to 200 nodes. (a) Not augmented and (b), (c), (d) augmented with flips along three main axes.

axes. While the translation is set to 10.000 here, it is applied randomly during runtime with a maximum of 10.000.

Additionally to the augmentations implemented by Weis et al. [WHLE21] we decided to implement *flip*, so that we can align the mirrored neurons from the left and the right hemisphere, depicted in Figure 5.7. An augmentation study can be found in Sub-section 5.4.5.

5.3 Graph Alignment

We use a preprocessing step to align the neurons in the coordinate system. This preprocessing is necessary to align the mirror symmetric neurons of the left and right brain hemisphere during the training with the previously discussed *flip* augmentations.

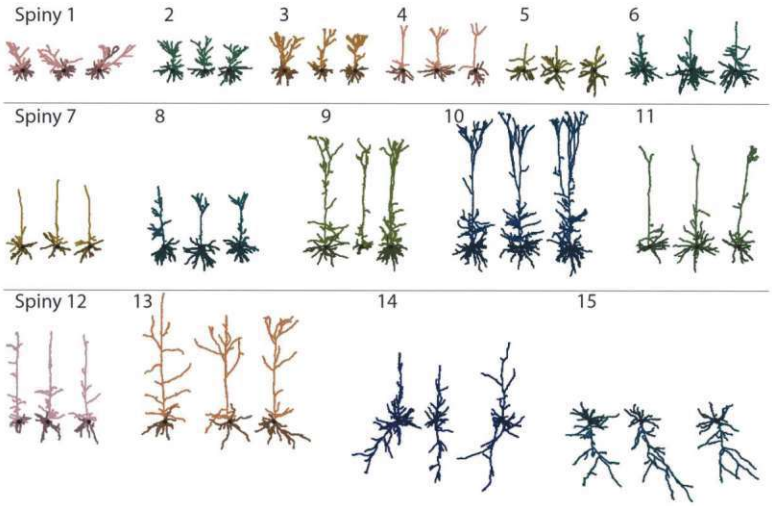


Figure 5.8: Example neurons for each spiny clusters of the BBP dataset, with apical dendrites in lighter color and basal dendrites in darker color. Image from Weis et al. [WPLE23].

The augmentation *rotation*, discussed in Sub-section 3.3.5, is then applied after the alignment to make the model invariant towards rotational changes. We use the terms *alignment* and *orientation* to reference the graph preprocessing and the term *rotation* to reference the augmentation.

It is not trivial to decide, how a neuron should be oriented in the coordinate system. Weis et al. [WHLE21] aligned neurons using the neurons' first principal component with the y-axis. This led to visually unambiguous results as depicted in Figure 5.8

We chose two approaches to orient the neuron graphs, elaborated in Sub-section 5.3.1 and Sub-section 5.3.2. To demonstrate examples of the neuron alignment, and the following augmentation techniques, we use the neuron with id 11986594. It is displayed in Figure 5.5 in its unprocessed form as stored in CATMAID.

5.3.1 PCA Alignment

The principal component analysis (PCA) is used to find the axes with the most relevant information in a point cloud. These are the axes with the largest data distributions. The axes form an orthonormal basis of the feature space, titled *eigenvectors* and the corresponding *eigenvalues* indicate the significance of an eigenvector.

We apply the PCA orientation on all the nodes of the unreduced graph representation. We use *scikit-learn* to perform PCA and to project the neuron node positions to the first three components, as we want to align the neuron with the three-dimensional space. The projection includes *mean centering*.

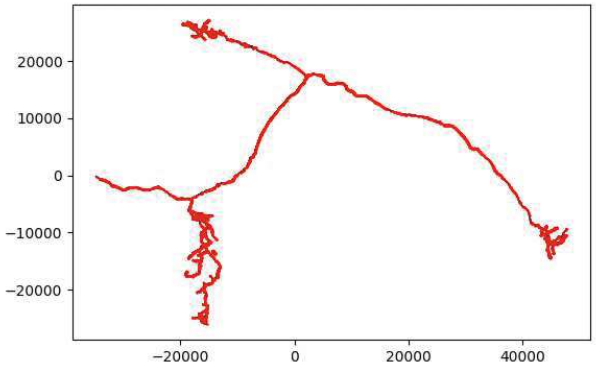
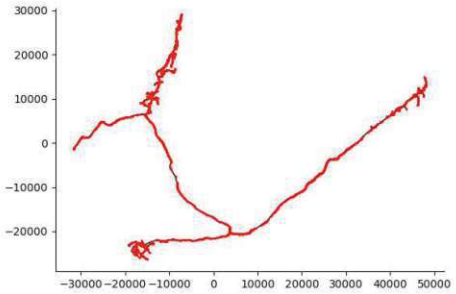
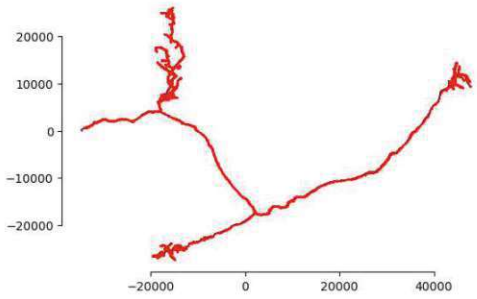


Figure 5.9: Neuron with id 11986594 aligned with PCA over all nodes.



(a) PCA aligned neuron with id 6218461



(b) PCA aligned and flipped neuron with id 11986594

Figure 5.10: (a) PCA aligned graph with id 6218461 from the right hemisphere and (b) PCA aligned graph with id 11986594 from the left hemisphere that is augmented using a *flip* along the x axis.

Figure 5.9 depicts the PCA aligned neuron graph with id 11986594. Figure 5.10 visualizes how we can align this neuron graph from the left hemisphere with a neuron graph from the right hemisphere, e.g., the neuron graph with id 6218461. Both neuron graphs are depicted without alignment and augmentations in Figure 5.2. Looking again at Figure 5.10, we see, that we can compare them by applying PCA alignment to both and by then flipping the graph with id 11986594 along the x axis.

Flipping the neuron graph, which is mean centered and PCA orientated, preserves the main branch PCA alignment. The neuron is still aligned with the x axis. Furthermore the mean centering prevents jumps within the coordinate system, i.e., the center of gravity

of the neuron graph is always in the center of the coordinate system and the neuron does not move from one quadrant of the coordinate system to another one after the flip augmentation.

We have to apply the PCA alignment before flipping the neuron, as we initially would not know the symmetry plane in unregistered data. After aligning the neuron, the symmetry planes xy, xz and yz all approximately intersect the origin of the coordinate system [0, 0, 0].

5.3.2 Main Branch PCA Alignment

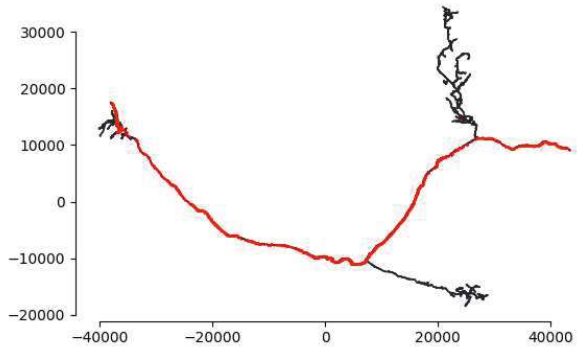
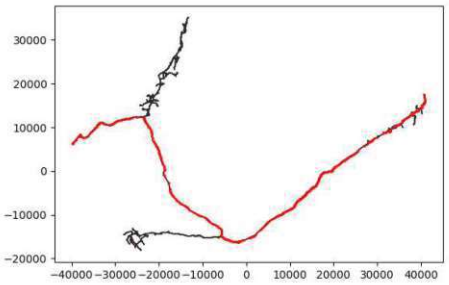
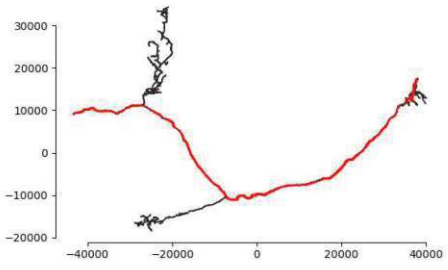


Figure 5.11: Neuron with id 11986594 aligned with PCA over all nodes that belong to the main branch, depicted in red.



(a) Main branch PCA aligned neuron with id 6218461



(b) Main branch PCA aligned and flipped neuron with id 11986594

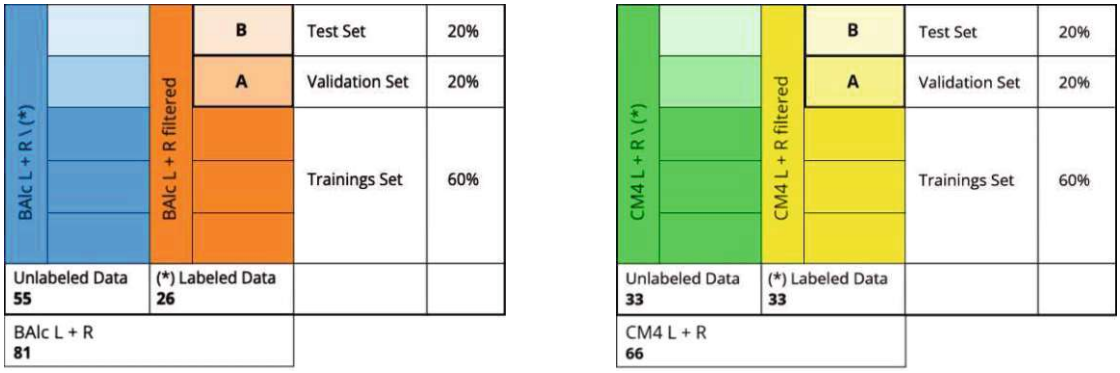
Figure 5.12: (a) Main branch PCA aligned graph with id 6218461 and (b) main branch PCA aligned graph with id 11986594 that is augmented using *flip* along y axis.

For the main branch PCA alignment we extract the longest path (called main branch) in the neuron graph representation. We perform PCA based on all the nodes on the main branch accordingly to the PCA alignment in Sub-section 5.3.1.

To find the longest branch we start from the leave nodes and follow the branches towards the soma, which is the graph root node. We compute the lengths of all the edges along each branch and compare the lengths to get the longest branch, i.e., the main branch.

Figure 5.11 depicts the main branch PCA aligned neuron graph with id 11986594. Given the graphs with id 6218461 and with id 11986594, depicted without alignment and augmentations in Figure 5.2, we see, that we can compare them by applying main branch PCA alignment to both and by then applying a flip along the y axis to the graph with id 11986594, as depicted in Figure 5.12.

5.4 Training



(a) Training, validation, and testing on dataset ***BAlc L / R***

(b) Training, validation, and testing on dataset ***CM4 L / R***

Figure 5.13: Split between training set, validation set, and test set for varying data subsets used for the experiments. Validation is performed on the labeled validation data A and results are reported on labeled test data B.

The training was performed on a server with an Intel(R) Xeon(R) Gold 5118 CPU (2.30 GHz, 12 cores) and a partitioned A100 utilizing 20 GB of memory.

Figure 5.13 visualizes the datasets, listed in Section 5.1, we use for the experiments and the division between training data, validation data, and test data.

We conduct an experiment that follows the pipeline discussed in Section 3.1 consisting of the following steps:

1. Self-Supervised Training: We perform a grid search to find the optimal hyperparameters for the self-supervised trained model. The accuracy of each model

is determined with the ARI score, noted in Equation 3.11, that computes the similarity between the predicted clusters and the manually generated ground truth. We train and evaluate the models on a subset of the data, discussed in Section 4.2. The best performing model is chosen to be explored in NetDive.

2. Semi-Supervised Training: Analogous to the grid search performed for the self-supervised trained model architecture, we perform a more exhaustive grid search for the semi-supervised trained model based on more features and feature values. The grid search is also executed for a subset of the data.

3. NetDive: We demonstrate the iterative fine-tuning of the model with the help of our visual analytics tool NetDive. We visualize the dimensionality reduced latent representations produced with the self-supervised trained model.

4. Human in the Loop: To improve the currently loaded model, we:

   a) Allow the assignment of custom labels to neurons that serve as labeled samples for the refinement of the model.

   b) Retrain the model with semi-supervised learning by utilizing the custom labels. We can leverage the results of the semi-supervised grid search and use the optimal hyperparameters for the retraining.

   c) Analyze the results. Therefore we:

      i. Compute the model accuracy based on ARI after each model retraining

      ii. Qualitatively analyse the model increments using NetDive

   d) We iteratively repeat step 4.

First, we explain the training setup based on dataset *BAlc L / R* in Sub-section 5.4.1. We follow up with the hyperparameter grid search runs for the self-supervised model in Sub-section 5.4.2 and the self-supervised model in Sub-section 5.4.3. We run an ablation study to evaluate the effect of single augmentation techniques in the context of semi-supervised training in Sub-section 5.4.5. We close the training experiments with the demonstration of NetDive based on dataset *CM4 L / R* in Sub-section 5.4.6.

5.4.1 Setup

We initially split the dataset *BAlc L / R* into the five subsets depicted in Figures 5.14-5.18 (four subsets for training and validation, and one subset for testing) and store the subsets so that we can rerun single folds, i.e., runs that leave out single subsets for validation, and investigate the data that is used for each fold. Each subset is a collection of 11 unlabeled neurons and 5 labeled neurons. To obtain collections of the same size, some of the neuron graphs remain unused. Instead of 26 labeled samples, we only use 25 labeled

neuron graphs, while we use all 55 unlabeled neuron graphs of the BAlc L / R dataset. Figures 5.14-5.18 depict the content of each subset.

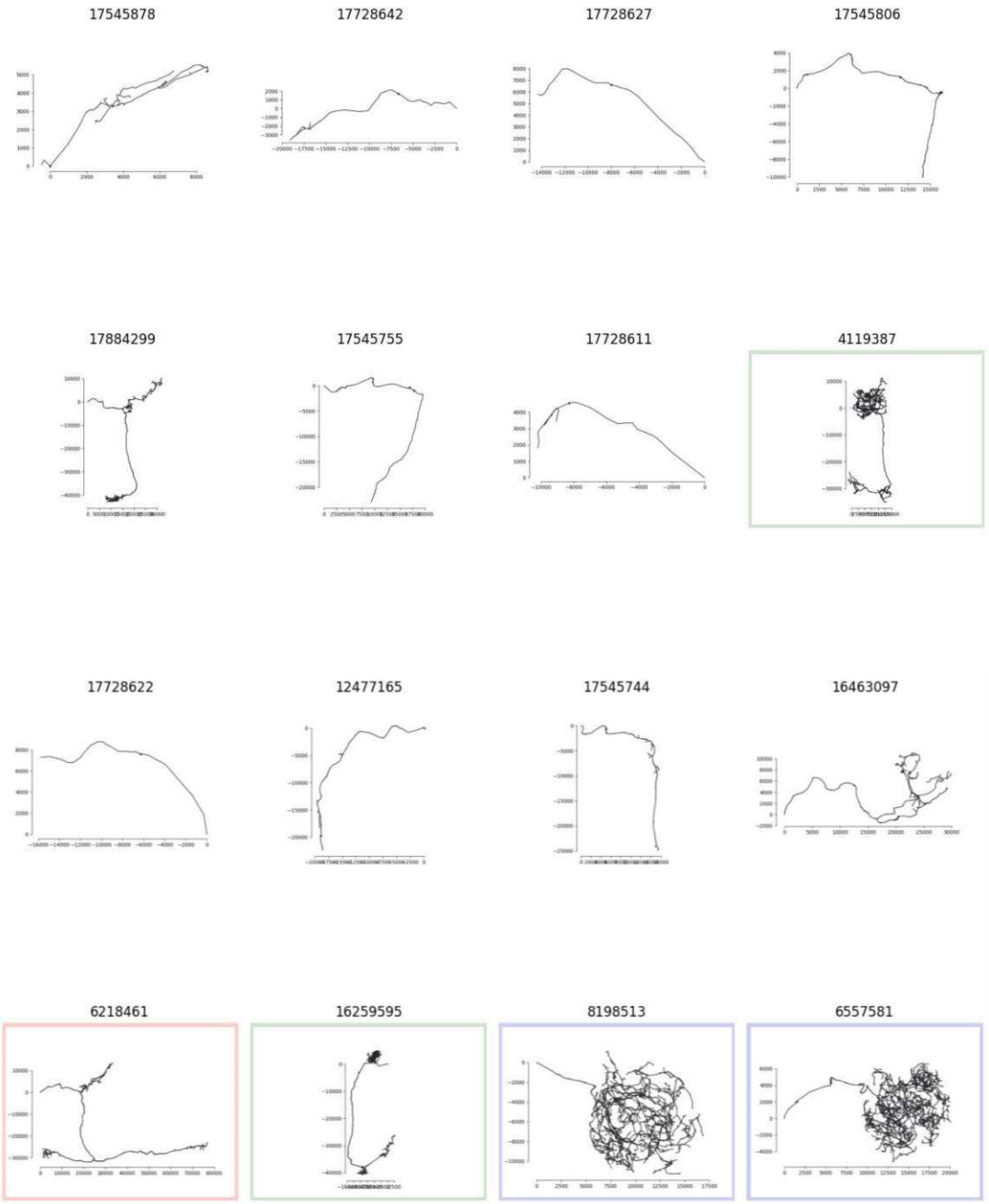


Figure 5.14: Subset 1 of dataset *BAlc L / R*. Cluster 1 neuron graphs are framed in green, Cluster 2 neuron graphs are framed in red, and Cluster 3 neuron graphs are framed in blue.

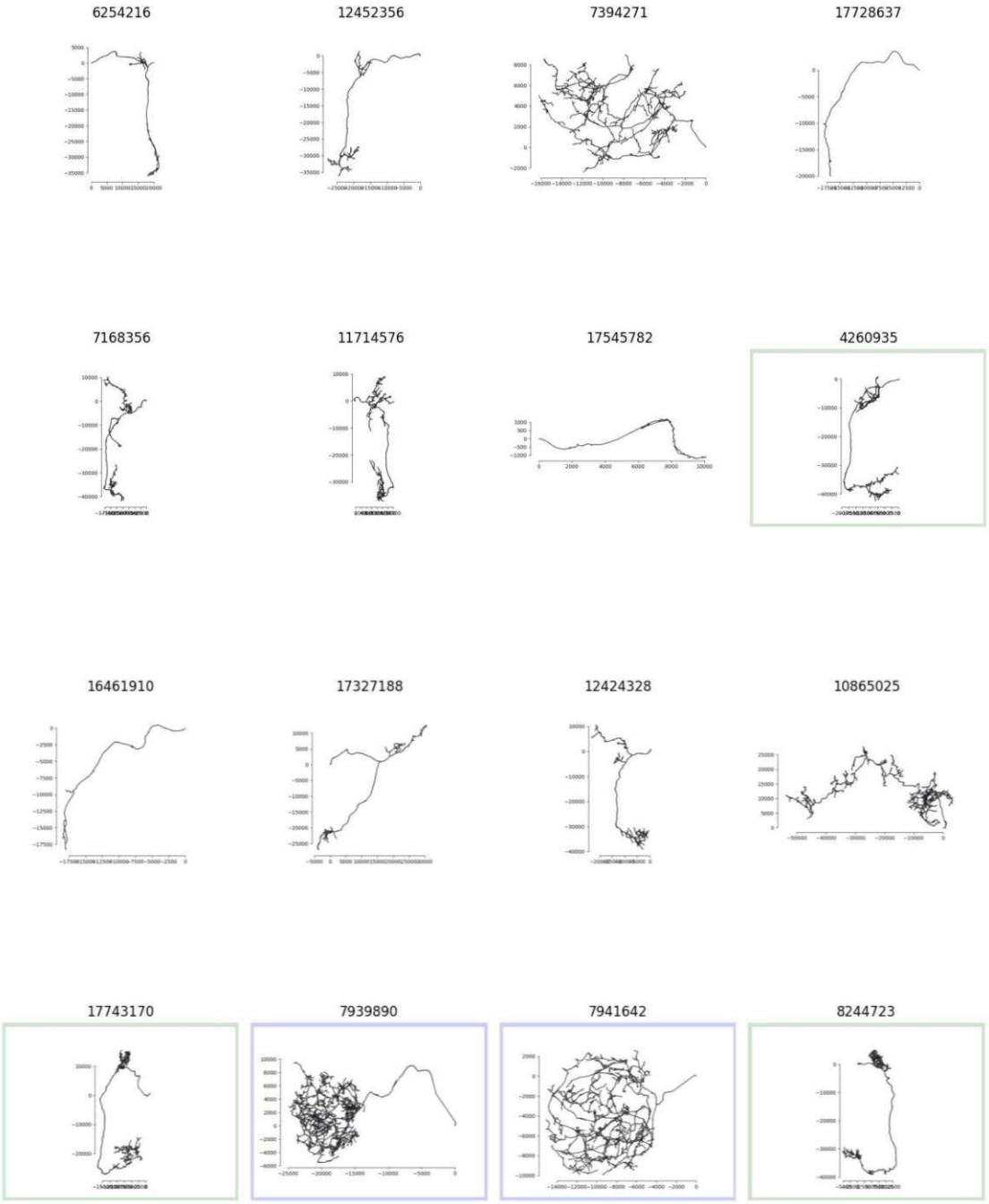


Figure 5.15: Subset 2 of dataset *BAlc L / R*. Cluster 1 neuron graphs are framed in green, Cluster 2 neuron graphs are framed in red, and Cluster 3 neuron graphs are framed in blue.

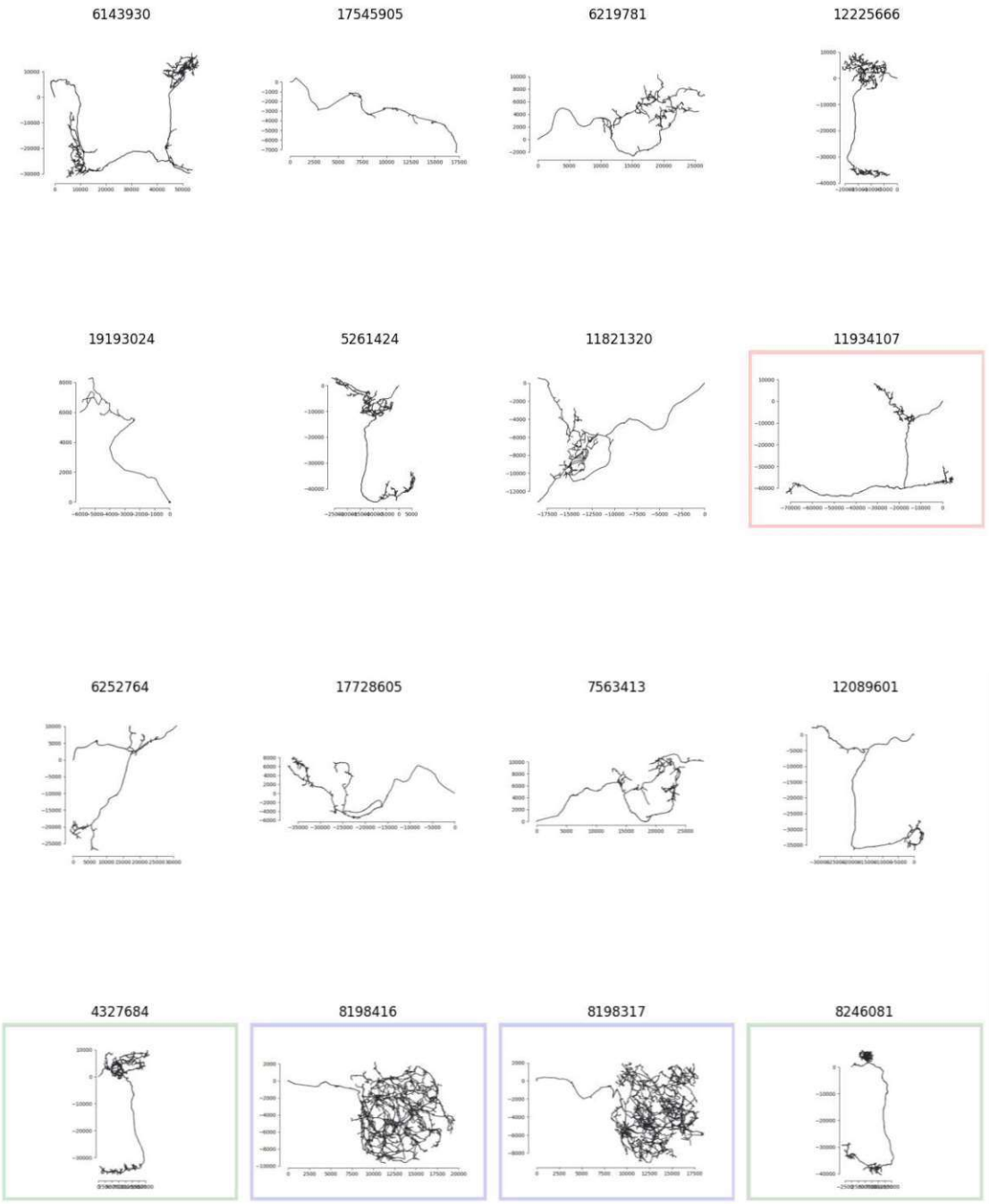


Figure 5.16: Subset 3 of dataset *BAlc L / R*. Cluster 1 neuron graphs are framed in green, Cluster 2 neuron graphs are framed in red, and Cluster 3 neuron graphs are framed in blue.

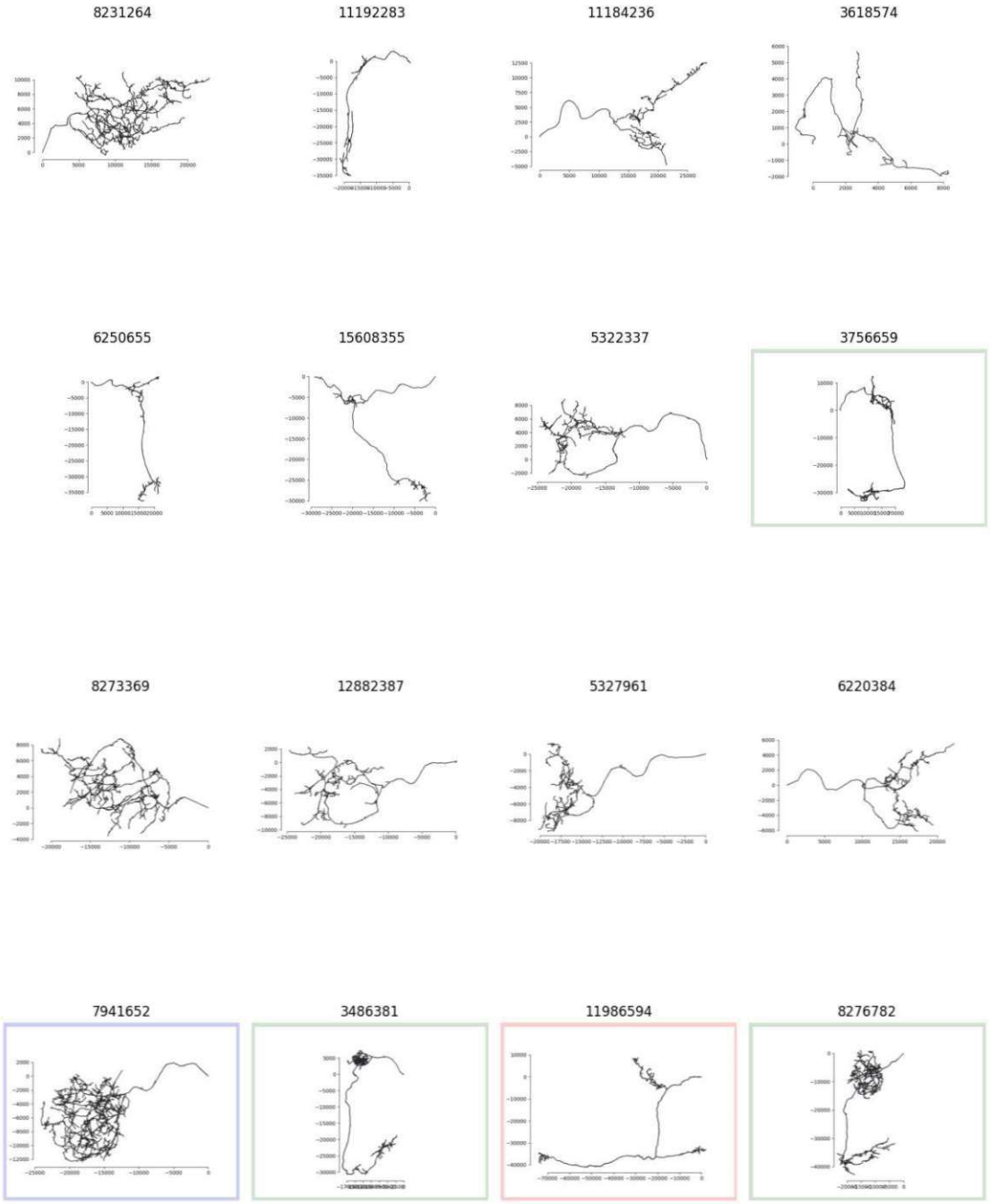


Figure 5.17: Subset 4 of dataset *BAlc L / R*. Cluster 1 neuron graphs are framed in green, Cluster 2 neuron graphs are framed in red, and Cluster 3 neuron graphs are framed in blue.

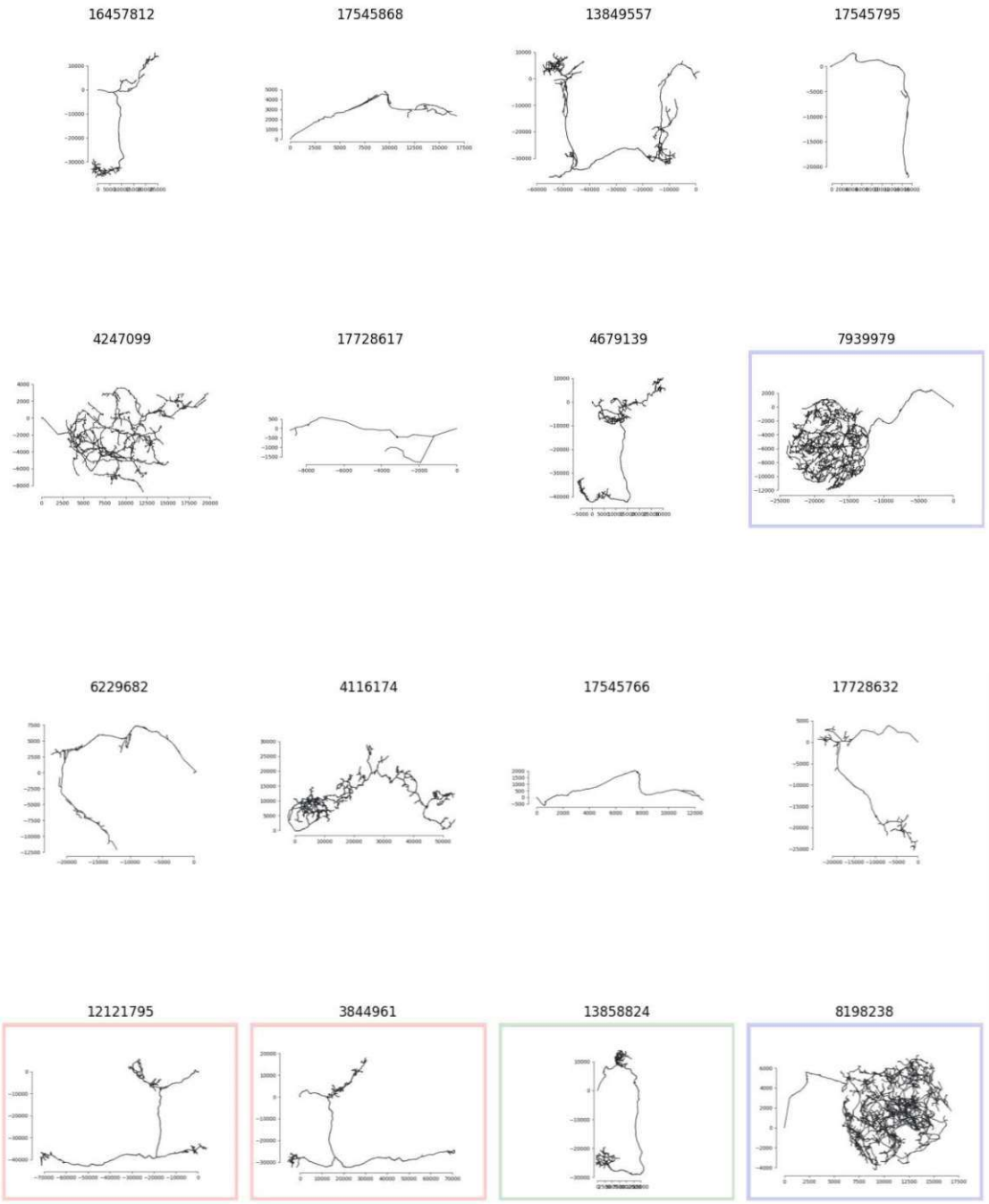


Figure 5.18: Subset 5 of dataset *BAlc L / R*. Cluster 1 neuron graphs are framed in green, Cluster 2 neuron graphs are framed in red, and Cluster 3 neuron graphs are framed in blue.

We train on 60% training data and evaluate on 20% labeled validation data, marked

with **A** in Figure 5.13. We do this using four-fold cross-validation, i.e., three subsets, resulting in 60% training data, are combined to the training data in each fold, whilst the remaining subset which is not the test subset is used for validation (resulting in 20% validation data). We select the model with the highest average ARI across the four folds.

We select Subset 1 to be the test set that remains unused for training and for validation. We do not use this subset to choose the best model, only to report the results. For the hyperparameter grid search we vary the features learning rate, learning rate decay, batch size, and λ and γ that determine the relevance of the regularization terms ME-MAX and One-Hot-Enforcement. We use the cost function 3.9.

The loss functions we use are mse and cross-entropy. We add the following regularization terms:

- No Regularization: We only use the objective function without a regularization term, i.e., $\lambda = 0$ and $\gamma = 0$.
- ME-MAX Regularization: We use the ME-MAX regularization term from PAWS to increase the entropy of the graph embeddings within a training batch, i.e., we want every class to be learned by the network and avoid mapping all the samples to one of the classes. We run a hyperparameter search, setting λ to 0.1, 0.5, and 1. γ is set to 0.
- One-hot encoding Regularization: We use our custom regularization term, which enforces one-hot encodings by reducing the entropy within a single embedding. While the ME-MAX regularizer operates over a batch of samples, the one-hot encoding regularizer is applied to each single isolated training sample. We run a hyperparameter search, setting γ to 0.1, 0.5 and 1. λ is set to 0.
- ME-MAX Regularization + One-hot encoding Regularization: We combine both regularization terms resulting in two hyperparameters λ and γ to add the respective terms to the loss. We explore the hyperparameter space for both variables and test combinations of the values 0.1, 0.5, and 1 for both variables.

Additionally, we address the issue of finding an objective function and a clustering algorithm that consider similar features. For the evaluation we fit 100 GMMs with varying random seeds to each fold for each model, leading to 400 GMMs for each of the models.

5.4.2 Self-Supervised Training

Weis et al. [WHLE21] train the self-supervised GraphDINO architecture for five different datasets. They run three grid searches with the focus points *hyperparameter search*, *augmentation strength*, and *model architecture*.

We train GraphDINO for the same learning rates, i.e., learning rate ∈ {0.001, 0.0001, 0.00001}. We train on the datasets *BAlc L / R* with a 60-20-20 training-validation-test split as depicted in Figure 5.13. While Weis et al. [WHLE21] train on batch sizes ∈ {32, 64, 128} we train on batch sizes ∈ {16, 32} due to the smaller training dataset.

The model with learning rate 0.0001 and batch size 16 has the overall best ARI performance on k-means clustering with a score of 0.495. We name this model M0.

5.4.3 Semi-Supervised Training

We run a grid search on the dataset *BAlc L / R*. We choose a graph alignment method and we run an extensive hyperparameter search based on this graph alignment. We then analyse the grid search and use the best performing model to run the augmentation search.

Graph Alignment

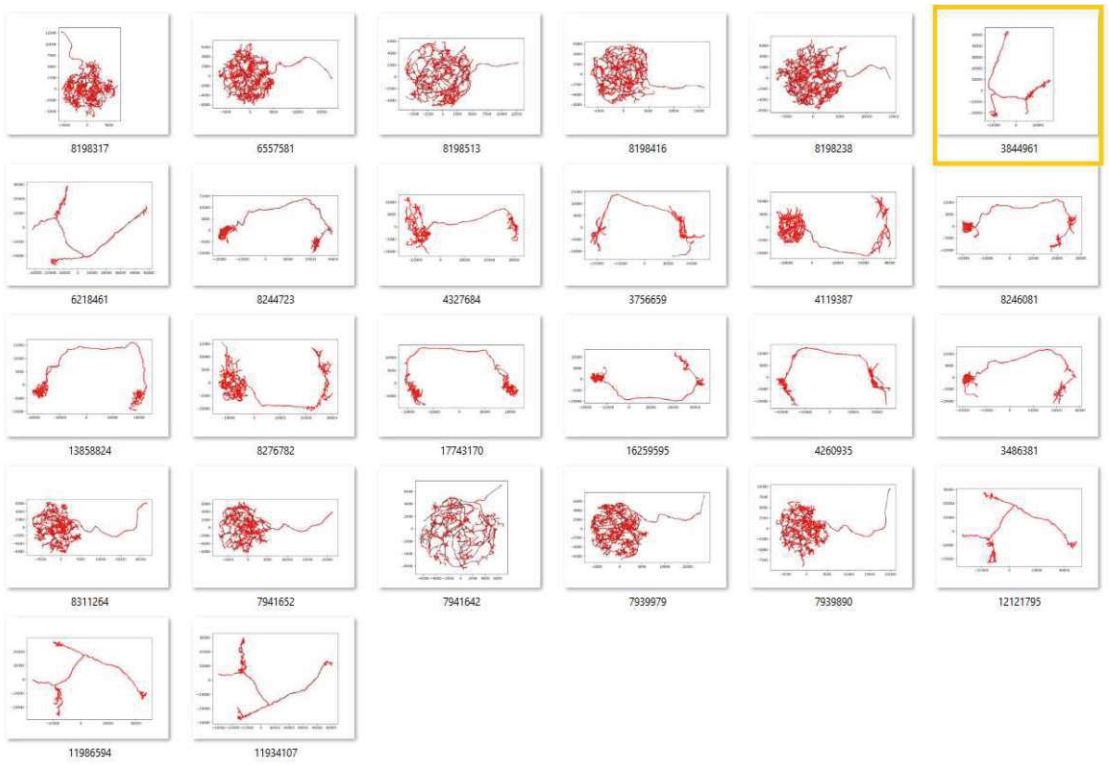


Figure 5.19: Graph representations of lineage BAlc L and BAlc R neurons with PCA alignment. The neuron with id 3844961, marked with a yellow rectangle, is not aligned via PCA and this misalignment cannot be corrected for by flip augmentations

We applied the PCA alignment and the main branch PCA alignment to the neuron graphs of the lineages BAlc L / R as depicted in Figure 5.19 and Figure 5.20 and visually

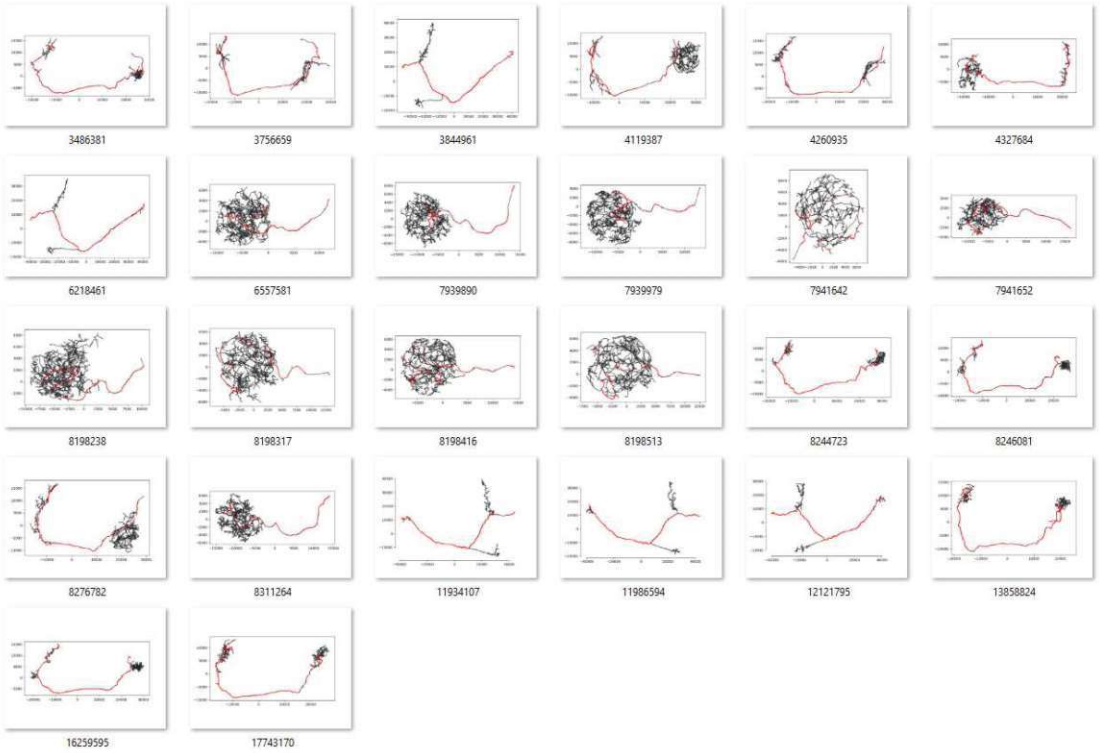
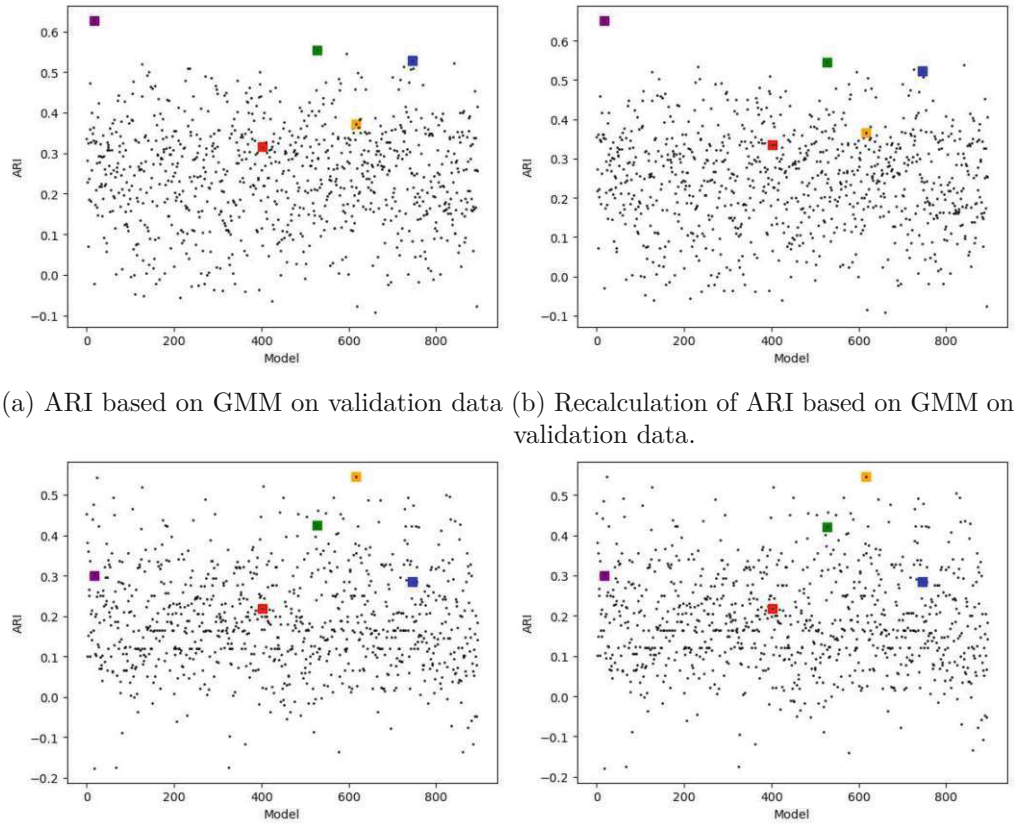


Figure 5.20: Graph representations of lineage BAlc L and BAlc R neurons with main branch PCA alignment.

evaluated the alignments. We discovered that one of the PCA aligned neurons, namely the neuron with id 3844961, was not aligned as we had expected as the distribution of nodes is different from other visually similar graphs. We therefore chose the main branch PCA alignment to perform the augmentation search to factor in the branch length besides the node density.

Hyperparameter Search

We trained 896 models using a grid search for the hyperparameters loss function, ME-MAX influence λ, One-Hot-Enforcement influence γ, batch size, and learning rate. We used the values ['cross_entropy', 'mse'] for the loss, the values [0, 0.1, 0.5, 1] for λ and γ, the values [0.001, 0.003, 0.006, 0.0001, 0.00006, 0.00003, 0.00001] for the learning rate, and the values [4, 8, 16, 32] for the batch size. We ran the hyperparameter search for 100 epochs. We evaluate with four-fold cross-validation for k-means and for GMM on the validation data.



(a) ARI based on GMM on validation data (b) Recalculation of ARI based on GMM on validation data.

(c) ARI based on k-means on validation data (d) Recalculation of ARI based on k-means on validation data.

Figure 5.21: (M1): Model trained on cross entropy with learning rate=0.0001, batch size=32, λ=0, and γ=0. (M2): Model trained on mse with learning rate=0.006, batch size=16, λ=0, and γ=1. (M3): Model trained on mse with learning rate=3e-05, batch size=32, λ=1, and γ=1. (M4): Model trained on cross entropy with learning rate=3e-05, batch size=4, λ=0.1, and γ=1. (M5): Model trained on mse with learning rate=0.001, batch size=8, $\lambda = 0.5$, $\gamma = 1$.

Figure 5.21 depicts the ARI scores for all 896 models. The ARI scores vary between -0.5 and 1, with -0.5 being especially discordant, 0.0 represting random clusterings and 1.0 being a perfect match between the ground truth and the predicted clusters. We computed the ARI scores for k-means and GMM twice to ensure that the computation produces stable results. We address this in chapter 6. The ARI scores are listed in tabular form in the Tables 5.1 and 5.2 for GMM and k-means clustering. The tables are sorted in descending order according to the ARI score. We highlight five models in Figure 5.21 and Table 5.1 and 5.2 that we will reference in this section and in chapter 6. We either choose the highlighted models for further experiments (M3) or we use these models as examples to discuss problems that can occur during training (M1, M2, M5) or we conduct

experiments that result in the hyperparameters of the highlighted model (M4).

Model M1 trained with cross entropy and the hyperparameters $\lambda = 0$, $\gamma = 0$, batch size=32, and learning rate=0.0001 achieves the highest ARI achieved with GMM clustering. As shown in Table 5.1, the ARI score is 0.627.

The highest ARI score achieved with k-means clustering is 0.545 for the model M5 trained on mse and the hyperparameters $\lambda = 0.5$, $\gamma = 1$, batch size=8, and learning rate=0.001, listed in Table 5.1.

We analyze the models M1 and M5 to ensure that the models are not performance outliers by chance, as the following experiments are built upon this model. We rerun the training with random initialization but the same hyperparameters for both models five times. The results are listed in Table 5.3.

We see that the results for the M1 model vary between average ARI scores from 0.122 - 0.627 based on GMM, while the results for the M5 model vary between average ARI scores from 0.206 - 0.545 based on k-means. This is an indicator for us, that the chosen model hyperparameters do not lead to stable results and that another selection of hyperparameters might be a better choice.

We further analyze the loss curves and the feature value distributions to evaluate the model performance. The analysis is detailed in Chapter 6. The loss curves of model M1 do not decrease. As the model does not learn over time, this model is not optimal, but a statistical outlier. The loss curves of the model M5 have the same issue for the folds 1, 3 and 4. The loss in fold 2 almost immediately drops down to the minimum loss value. Looking at the feature value distribution for fold 2 we see that the model suffers from node collapsing, i.e., maps all input data to the same output. We will discuss the issue *node collapsing* in chapter 6 based on model M2, which is also named and color coded in Table 5.1.

In order to find the optimal hyperparameters for the augmentation study, we made an analysis of the hyperparameter value distributions and the means and variances of each model. We therefore averaged all the ARI scores of the 896 trained models based on a specific value for a specific hyperparameter and chose the best performing values for each hyperparameter. The resulting optimal hyperparameter values are loss=cross entropy, learning rate=3e-05, batch size=4, λ=0.1, and γ=1 and in average GMM clustering performs better than k-means clustering. The analysis is covered in Section 6.4 in more detail. We retrained five models with the best performing values for each hyperparameter. We name the model M4. The results for the model reruns are denoted in Table 5.3. As shown in the table, the combination of these hyperparameter values turned out to be not optimal, as the ARI score is low for GMM clustering.

In the next step we evaluated the loss curves and value distributions for the top scoring models. We go through Table 5.1 and Table 5.2 line by line and we eliminate models that suffer from learning incapabilities and node collapsing until we find a model that does not suffer from one of these two issues. The rows with green background color correspond

ARI	Loss	Learning Rate	Batch Size	λ	γ	Evaluation	
0.627	cross_entropy	0.0001	32	0	0	No loss decrease	M1
0.554	mse	0.006	16	0	1	Node collapsing	M2
0.545	mse	6e-05	8	0.5	0.5	No loss decrease	
0.527	mse	3e-05	32	1	1	Works	M3
0.521	mse	0.001	8	0.1	1	Node collapsing	
0.520	cross_entropy	0.001	32	0.5	0	No loss decrease	
0.514	mse	3e-05	16	1	0.5	Small loss decrease	
0.509	mse	6e-05	32	1	1	Small loss decrease	
0.509	cross_entropy	3e-05	4	1	0	Small loss decrease	
0.509	cross_entropy	0.001	4	1	0	Node collapsing fold 1 + 2	
0.506	mse	1e-05	32	1	1	Works	
0.501	cross_entropy	3e-05	8	0.1	1	No loss decrease	
0.500	cross_entropy	0.0001	8	0.5	0.5	Small loss decrease	
0.498	mse	3e-05	8	0.5	1	No loss decrease	
0.491	cross_entropy	6e-05	16	0.5	0	No loss decrease fold 1 + 2	
0.490	cross_entropy	0.003	32	1	0.5	Node collapsing fold 2 + no loss decrease fold 1 + 3	
0.488	mse	1e-05	8	1	0.5	Small loss decrease	
0.486	mse	0.003	4	0.5	1	Node collapsing	
0.486	cross_entropy	3e-05	8	1	1	No loss decrease after 20 epochs	
0.484	mse	0.0001	32	0.5	0	No loss decrease	
0.481	mse	3e-05	32	1	0.5	Works	
0.479	cross_entropy	0.0001	8	1	0	No loss decrease after 20 epochs	
0.479	cross_entropy	0.001	4	0.1	0	Node collapsing	
0.476	cross_entropy	0.0001	16	0	0.1	Small loss decrease	
	... 12 Missing Rows ...						
0.459	cross_entropy	0.0001	32	1	1	Works	
	... 222 Missing Rows ...						
0.317	cross_entropy	3e-05	4	0.1	1	No loss decrease	M4
	...						

Table 5.1: Analysis of the loss curves and feature value distributions of models sorted by highest ARI based on the hyperparameter grid search results. The ARIs are computed using GMM clustering

ARI	Loss	Learning Rate	Batch Size	λ	γ	Evaluation	
0.545	mse	0.001	8	0.5	1	No loss decrease fold 1 + 3 + 4, node collapsing fold 2	M5
0.544	cross_entropy	0.001	16	0	0	Node collapsing	
0.521	cross_entropy	6e-05	4	0.1	1	No loss decrease	
0.52	cross_entropy	0.001	32	0.5	0	Small loss decrease	
0.499	mse	3e-05	4	0.1	0.5	Poor feature value distributions	
0.495	mse	0.003	32	0.1	0.5	Node collapsing	
0.494	cross_entropy	3e-05	8	0	1	No loss decrease	
0.494	mse	0.001	8	0	0	Node collapsing	
0.494	mse	3e-05	8	0.5	0	No loss decrease	
0.494	mse	1e-05	4	1	1	No loss decrease	
0.490	cross_entropy	0.003	32	1	0.5	No loss decrease fold 1 + 3, node collapsing fold 2	
0.488	mse	1e-05	8	0.5	0.1	No loss decrease	
0.487	mse	1e-05	8	1	0	Small loss decrease	
0.476	cross_entropy	0.001	32	0	0	Node collapsing	
0.475	mse	0.003	32	1	1	Node collapsing fold 1 + 3 + 4	
0.467	cross_entropy	1e-05	16	0	0.1	Small loss decrease	
0.466	mse	6e-05	16	1	0.5	Works	
0.462	mse	3e-05	16	0	1	Poor feature value distributions	
0.457	cross_entropy	1e-05	16	0.1	0.1	Small loss decrease	
		...					

Table 5.2: Analysis of the loss curves and feature value distributions of models sorted by highest ARI based on the hyperparameter grid search results. The ARIs are computed using k-means clustering

	M#	Initial	Rerun 1	Rerun 2	Rerun 3	Rerun 4	Rerun 5
k-means	M1	0.3	0.153	0.194	0.094	0.138	0.092
GMM	M1	0.627	0.219	0.346	0.196	0.383	0.122
k-means	M2	0.424	0.206	0.285	0.252	0.227	0.291
GMM	M2	0.554	0.206	0.285	0.252	0.227	0.291
k-means	M3	0.285	0.269	0.423	0.422	0.248	0.165
GMM	M3	0.527	0.379	0.513	0.591	0.442	0.507
k-means	M4	0.218	0.494	0.303	0.022	0.494	0.08
GMM	M4	0.317	0.253	0.223	0.25	0.189	0.131
k-mean	M5	0.545	0.206	0.285	0.252	0.227	0.291
GMM	M5	0.372	0.294	0.334	0.51	0.306	0.249

Table 5.3: Average ARI scores based on k-means and GMM for the models hyperparameter reruns. (M1): Model trained on cross entropy with learning rate=0.0001, batch size=32, λ=0, and γ=0. (M2): Model trained on mse with learning rate=0.006, batch size=16, λ=0, and γ=1. (M3): Model trained on mse with learning rate=3e-05, batch size=32, λ=1, and γ=1. (M4): Model trained on cross entropy with learning rate=3e-05, batch size=4, λ=0.1, and γ=1. (M5): Model trained on mse with learning rate=0.001, batch size=8, $\lambda = 0.5$, $\gamma = 1$.

to loss curves that have a downwards trend and feature value distributions that indicate a proper distribution of output representations for varying input samples. We assign the attribute *working* to these models, highlighted with a green background, though we can not guarantee that the learned representations are indeed meaningful. This approach returns us the top scoring model that *works*. We find model M3 to be the working model with the highest ARI. We also retrain five times with these hyperparameters. The results are listed in Table 5.3. We see that the results are more stable than for the other models, based on GMM clustering, and the loss curves consistently decrease throughout this experiment. Therefore we choose this model for the augmentation study.

5.4.4 Training Results

Table 5.4 compares the ARI scores of the optimal self-supervised trained model M0, reported in sub-section 5.4.2, with the optimal semi-supervised trained model M3, reported in sub-section 5.4.3.

	Self-Supervised Training GraphDINO	Semi-Supervised Training GraphPAWS
Loss	cross entropy	mse
Learning Rate	0.0001	3e-05
Batch Size	16	32
Gamma	-	1
Lambda	-	1
ARI	0.495 (k-means)	0.527 (GMM)

Table 5.4: Results of optimal self-supervised trained model and semi-supervised trained model

5.4.5 Ablation Study

We conduct an ablation study in order to evaluate the impact of different augmentations.

The search is inspired by the ablation study by Weis et al. [WHLE21] that is depicted in Figure 5.22.

Model	Accuracy
Ours	51.5 ± 1.3
– 3D rot.	32.6
– cum. jit.	56.8
– node jitter	54.0
– soma jitter	48.9
– drop branch	52.0
100 nodes	51.8
Minimal	29.7
+ cum. jit.	28.3
+ node jitter	33.8
+ soma jitter	30.2
+ drop branch	33.8
200 nodes	31.5

Figure 5.22: Screenshot of the ablation study of GraphDINO [WHLE21]

Figure 5.22 shows, that the optimal GraphDINO model [WHLE21] does not contain all augmentations (Accuracy=51.5), but exceeds for example if the cummulative jittering is removed (Accuracy=56.8). We also start by training a model that has all augmentations applied and remove one augmentation after another and then run the reverse process by subsequently adding augmentations to a minimal model that implements no augmentations.

Ablation Study Results

Table 5.5 depicts the results of our ablation study. The study is performed by starting with a minimal and a maximal set of augmentations and by removing and adding single augmentations.

The ablation study is also based on a four-fold cross-validation using the average result of 100 clustering models for each fold. The results of Table 5.5 are reported on the test sets of the labeled data, marked with **B** in Figure 5.13.

Model: Main branch PCA aligned	**ARI k-means**	**ARI GMM**
Ours	0.336	0.428
- flip (y,z)	0.225	0.396
- flip (x,y,z)	0.248	0.360
- 3D rotation	0.285	0.373
- cum. jit.	0.218	0.550
- node jit.	0.267	0.296
Minimal	0.142	0.301
+ flip (x)	0.574	0.601
+ flip (x,y,z)	0.237	0.426
+ 3D rotation	0.445	0.472
+ cum. jit.	0.445	0.660
+ node jit.	0.442	0.494

Table 5.5: Ablation study on main branch PCA aligned data

5.4.6 Demonstration of Pipeline using NetDive

In this sub-section we demonstrate the pipeline described in Section 3.1 and visualized with Figure 3.1. We aim to show that the annotations and the retraining does iteratively improve the clustering with minimal user input.

We demonstrate this on lineage CM4. We train a new model on lineage CM4. Instead of performing cross-validation, we train on the whole CM4 dataset and evaluate on a manually annotated CM4 subset. This approach simulates a real use case, where the user iteratively adds knowledge to the model.

Initially we simulate the case that no labeled data is available and therefore train self-supervised. We use the hyperparameter that performed best for the unsupervised architecture, i.e., learning rate 0.0001 and batch size 16 on the BAlc dataset. In this way we simulate a domain transfer from a previous dataset for the initial parameter settings. The results for the training are listed in Table 5.6 in column *Self-Supervised*. We then load the dimensionality reduced latent representations of the CM4 neuron graphs into NetDive and analyze the embeddings.

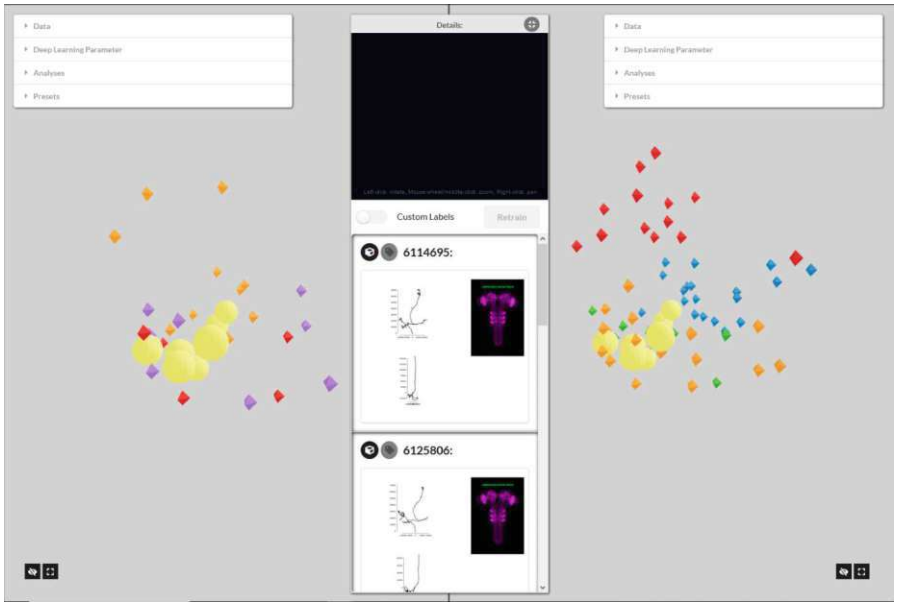


Figure 5.23: Analysis of latent embeddings. The left view depicts the filtered CM4 neurons, colored by their ground truth cluster assignment. The right view depicts the unfiltered CM4 neurons, colored by their prediction based on GMM clustering. Ground truth Cluster 1 is selected.

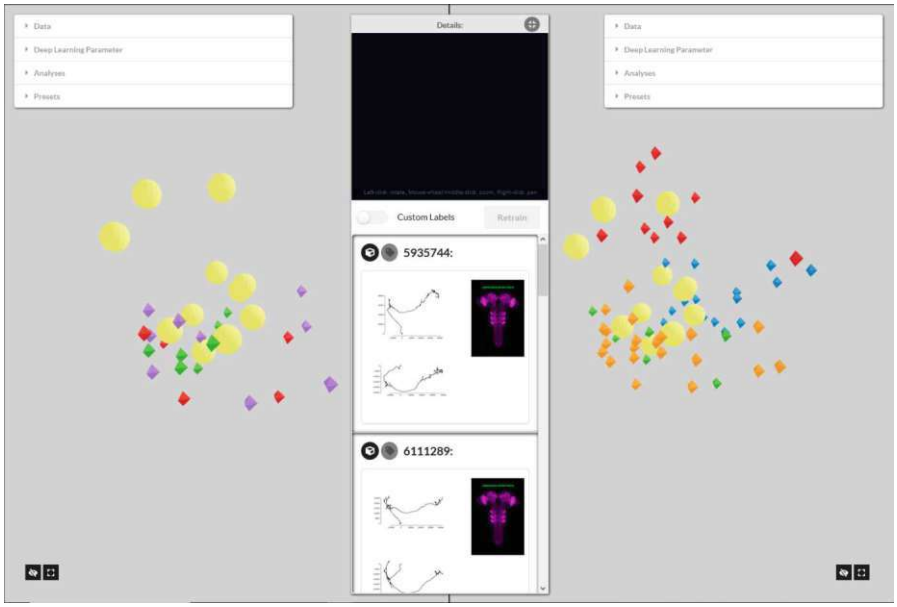


Figure 5.24: Analysis of latent embeddings. The left view depicts the filtered CM4 neurons, colored by their ground truth cluster assignment. The right view depicts the unfiltered CM4 neurons, colored by their prediction based on GMM clustering. Ground truth Cluster 2 is selected.

	Self-Supervised [WPLE23]	Iteration 1	Iteration 2	Iteration 3
ARI: **GMM**	0.152	0.145	0.143	0.225
ARI: **k-means**	0.117	0.211	0.191	0.317
Support Samples	-	17732270, 12935696, 10411574, 7982896	17732270, 11905911, 12935696, 11359482, 10411574, 19298625, 7982896, 10018260	17732270, 11905911, 7227010, 12935696, 11359482, 17306107, 10411574, 19298625, 16154290, 7982896, 10018260, 18142558

Table 5.6: ARI scores of incremental training with NetDive. The ARI computation is based on the manual ground truth for evaluation purposes only.

Figure 5.23 and Figure 5.24 depict the visualisation of the graph embeddings. On the left UI view of both images we color the neurons according to their ground truth clustering. We generate the ground truth ourselves by visual inspection. We chose the dataset lineage CM4 as the shapes of the neuron graphs of this lineage are visually distinguishable for non-experts. We only assign clusters to neuron graphs that have a distinct shape that is comparable to other similar shapes. Neuron graphs with indistinguishable shapes have no label assigned. The right views of both images depict the neurons colored by their cluster predictions based on their embedding.

On the left views of both images we filtered the embeddings to include only those that have been assigned to a cluster. CM4 includes 33 unlabeled neurons as visualized in Figure 5.13(b). NetDive applies the filtering based on color groups. As we use the predictions instead of the ground truth cluster assignments on the right UI view, we can not apply the same filter on the right UI view.

In both Figure 5.23 and Figure 5.24 we select a ground truth cluster using the cluster selection feature depicted in Figure 3.14. We see how the selected neuron clusters are distributed throughout the embedding space. We see that the embeddings of ground truth Cluster 1 in Figure 5.23 are spatially close together. Still, looking on the right UI view in Figure 5.23, the neurons of ground truth Cluster 1 do not all fall in the same predicted cluster, but are distributed across three predicted clusters. We see that the embeddings of ground truth Cluster 2 in Figure 5.24 are spatially more distributed. Looking at the right UI view in Figure 5.24, we see that the neurons of ground truth

Cluster 2 are also distributed across three predicted clusters.

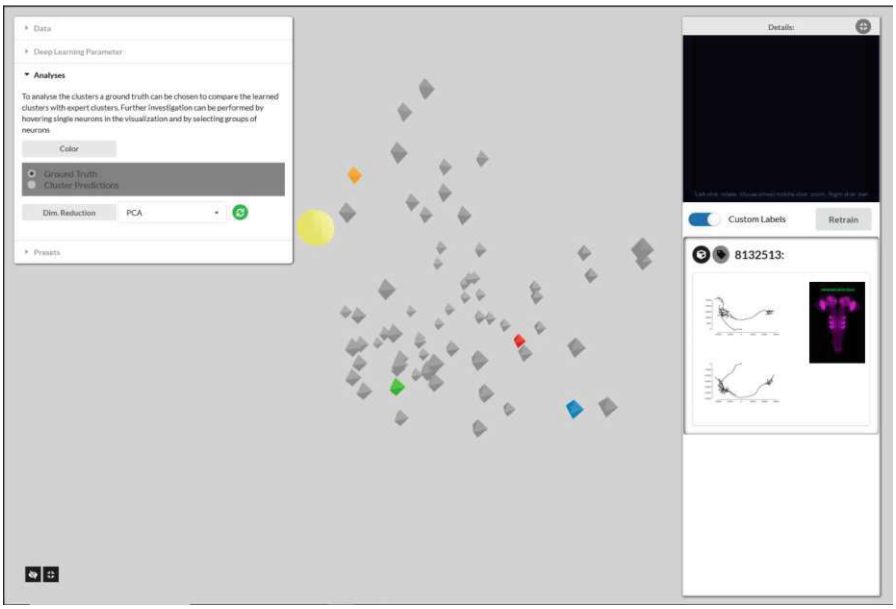
In our case we can visualize, which clusters we expect, but this ground truth is usually not available. Instead, we expect we would observe the predicted clusters and analyze the graphs assigned to each cluster.

After loading and analyzing the dimensionality reduced latent representations of the CM4 neuron graphs into NetDive, we relabel misclassified and prototypical neuron graphs. We start by relabeling one prototypical neuron graph representation for each cluster that we want to be learned.

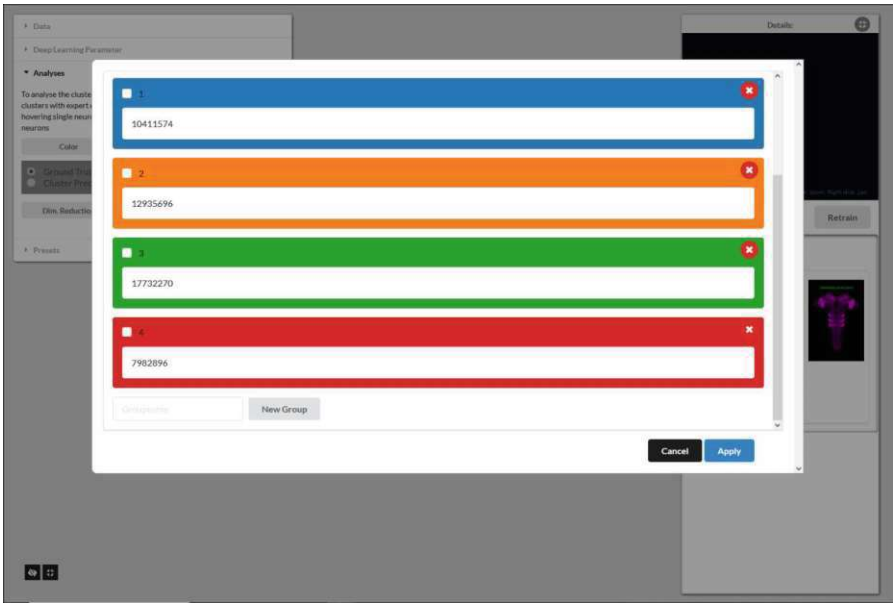
Figure 5.25(a) shows the labeled neurons, each cluster represented by another random color. These labeled samples, i.e., support samples, are used for the retraining. Figure 5.25(b) shows the relabel modal window that allows the user to create new labels and to assign the labels to the neuron graphs. By pressing *Retrain*, the retraining is started with a subprocess call. The retraining uses the GraphPAWS network architecture.

Figure 5.26 and 5.27 show the process respectively for the annotation of 2 and 3 neurons for each cluster. Figure 5.26(b) and 5.27(b) depict the confirmation window that prompts after triggering the retraining. This confirmation window shows the support samples used for the retraining and the adjustable hyperparameters that will be used for the retraining. We adjust these parameters during the experiments and ensure that they correspond to the optimal hyperparameters found for the semi-supervised GraphPAWS training, i.e., the hyperparameters corresponding to the M3 model (Table 5.1).

Figure 5.28 depicts the embeddings after each iteration, colored based on the CM4 ground truth. We see patterns according to the CM4 ground truth labels, but we cannot recognize a clear subdivision into clusters. We must therefore be cautious in assessing the slightly positive trend in the improvement of ARI scores, reported in Table 5.6. While iteration 3 outputs the best ARI scores, the other iterations fluctuate between better and worse results, there the positive trend itself is also questionable.

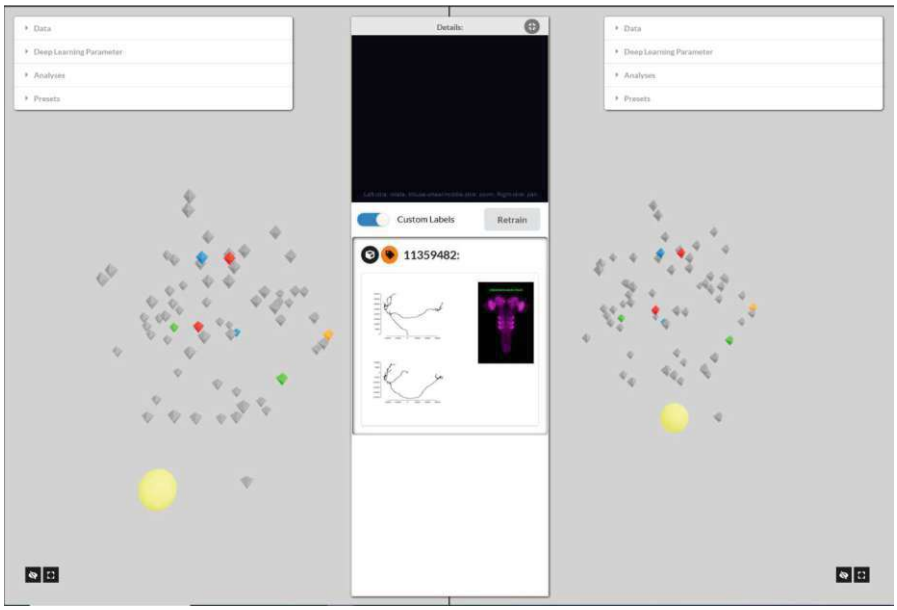


(a) Neurons colored according to their assigned classes

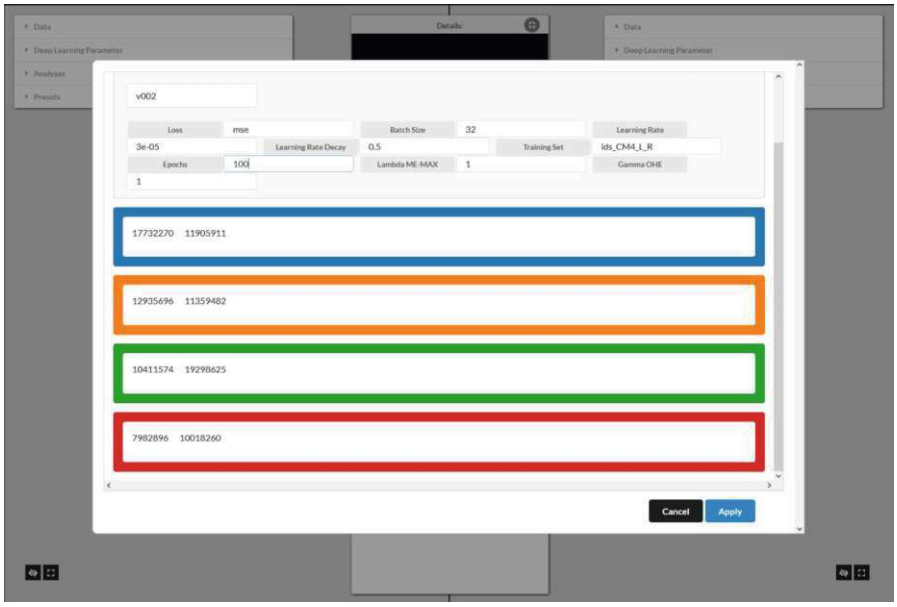


(b) Relabel modal

Figure 5.25: Manual assignment of classes to the latent representations of lineage CM4 neuron graphs with NetDive. The loaded graph embeddings are generated with the self-supervised GraphDINO model. Image (a) shows the neurons colored according to their assigned classes. Image (b) shows the relabel modal to generate clusters and assign neurons to these clusters.



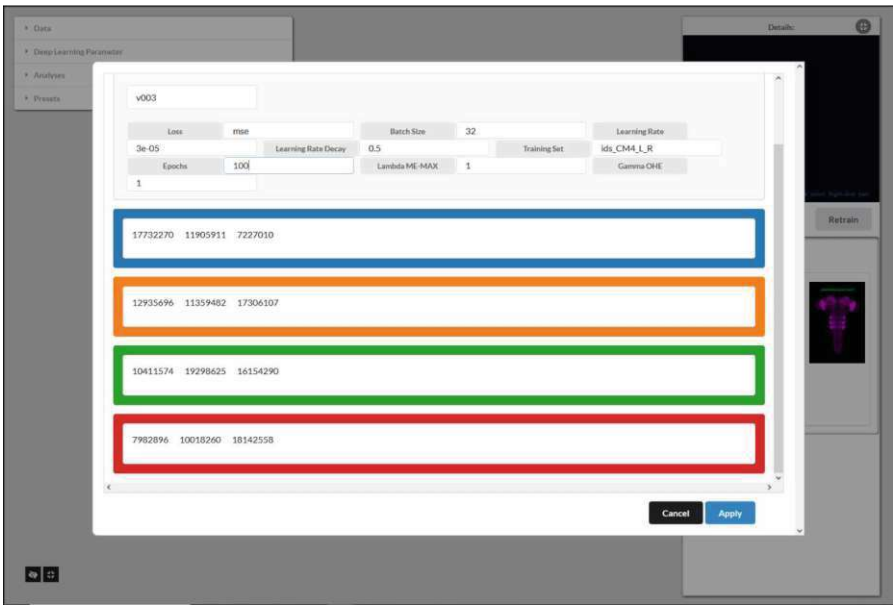
(a) Neurons colored according to their assigned classes



(b) Retrain confirmation modal

Figure 5.26: Manual assignment of classes to the latent representations of lineage CM4 neuron graphs with NetDive. The loaded graph embeddings are generated with the semi-supervised GraphPAWS model, trained with the support samples as annotated in Figure 5.25. Image (a) shows the neurons colored according to their assigned classes. Image (b) shows the retrain confirmation modal.

(a) Neurons colored according to their assigned classes



(b) Retrain confirmation modal

Figure 5.27: Manual assignment of classes to the latent representations of lineage CM4 neuron graphs with NetDive. The loaded graph embeddings are generated with the semi-supervised GraphPAWS model, trained with the support samples as annotated in Figure 5.26. Image (a) shows the neurons colored according to their assigned classes. Image (b) shows the retrain confirmation modal.

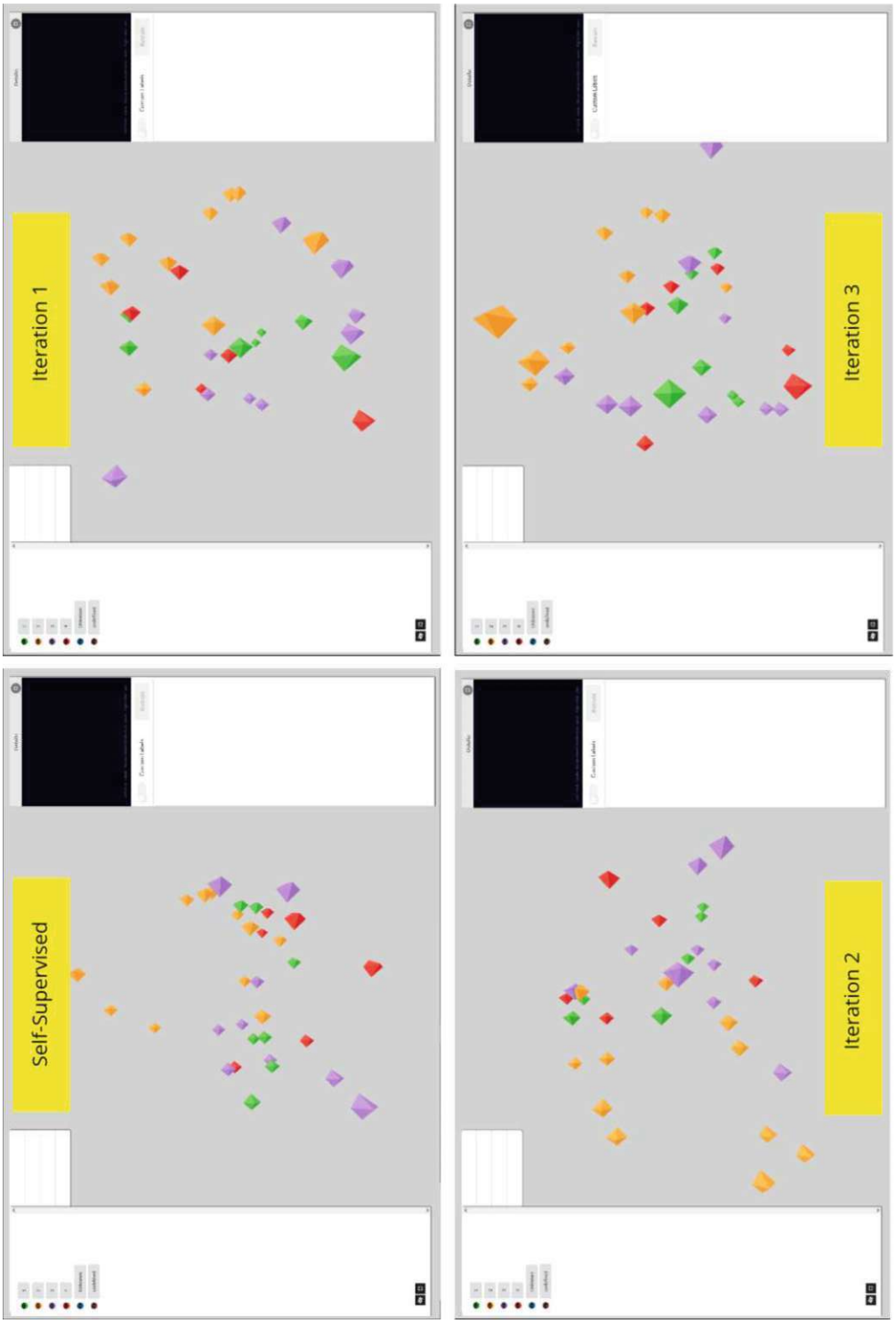


Figure 5.28: Neuron graph embeddings after each iteration denoted in Table 5.6. The initial embeddings are generated using self-supervised training. For the subsequent iterations we incrementally add more support samples to evaluate whether the embeddings improve. Using the numeric values in Table 5.6 and the depicted visual embeddings, we can not affirm this unequivocally.

CHAPTER 6

Discussion

The discussion addresses problems we faced during the experiments and guides through them in more detail than in the chapter 5 on experiments. We start with the limitations of self-supervised learning, continue with model learning incapabilities due to node collapsing or missing regularization, we then analyze the hyperparameter space, discuss how clustering effected our experiments, and close the discussion with an evaluation of the NetDive experiments.

6.1 Self-Supervised Learning

For this thesis, we originally started out with a purely self-supervised approach. We tested GraphDINO with our data for the same learning rates that Weis et al. [WHLE21] use during the hyperparameter grid search and with smaller batch sizes, i.e., 16 and 32.

The model with learning rate 0.0001 and batch size 16 has the overall best ARI performance on k-means clustering with a score of 0.495, as noted in Table 5.4. On GMM the best performing model has batch size 32 and learning rate 1e-05. Figure 6.1 depicts the loss curves and the feature distributions of each fold training for 100 training epochs.

We retrain this model for 400 epochs as we see a downward trend in the loss curve and want to see if the results improve. The loss curves and the feature distributions of this longer retraining are depicted in Figure 6.2.

The loss curve continues to go downwards and seems to start flattening after 300 epochs. The ARI result is 0.423 for k-means and 0.365 for GMM clustering. We see that the model increases confidence in what it learns as the loss decreases and the feature value distribution shows that the graph representations are mapped to the feature space without node collapsing. But the results do not align with what we expect the model to learn, as the clusters obtained with this method did not resemble our manually generated ground truth.

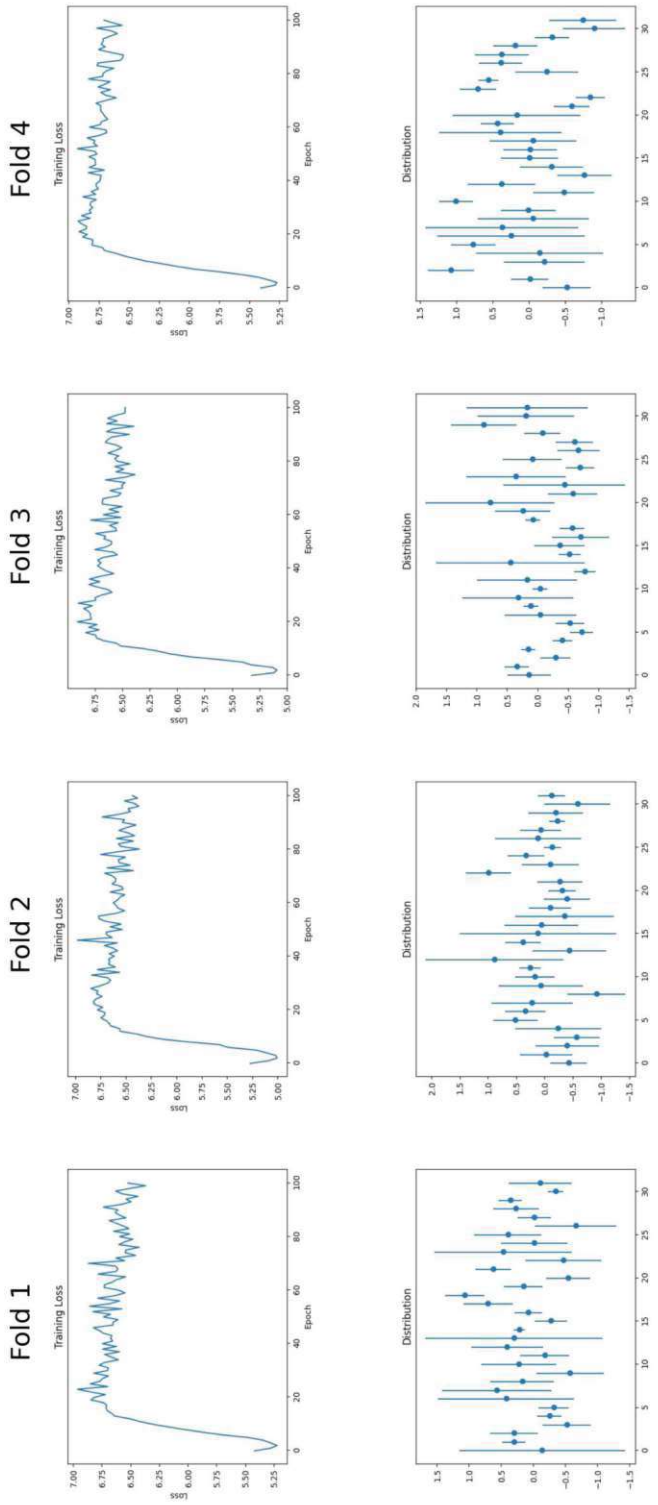


Figure 6.1: Loss curves and the feature distributions of each fold training for 100 training epochs for the hyperparameters learning rate=0.0001, and batch size=16. The average ARI on GMM is 0.453. The average ARI on k-means is 0.495.

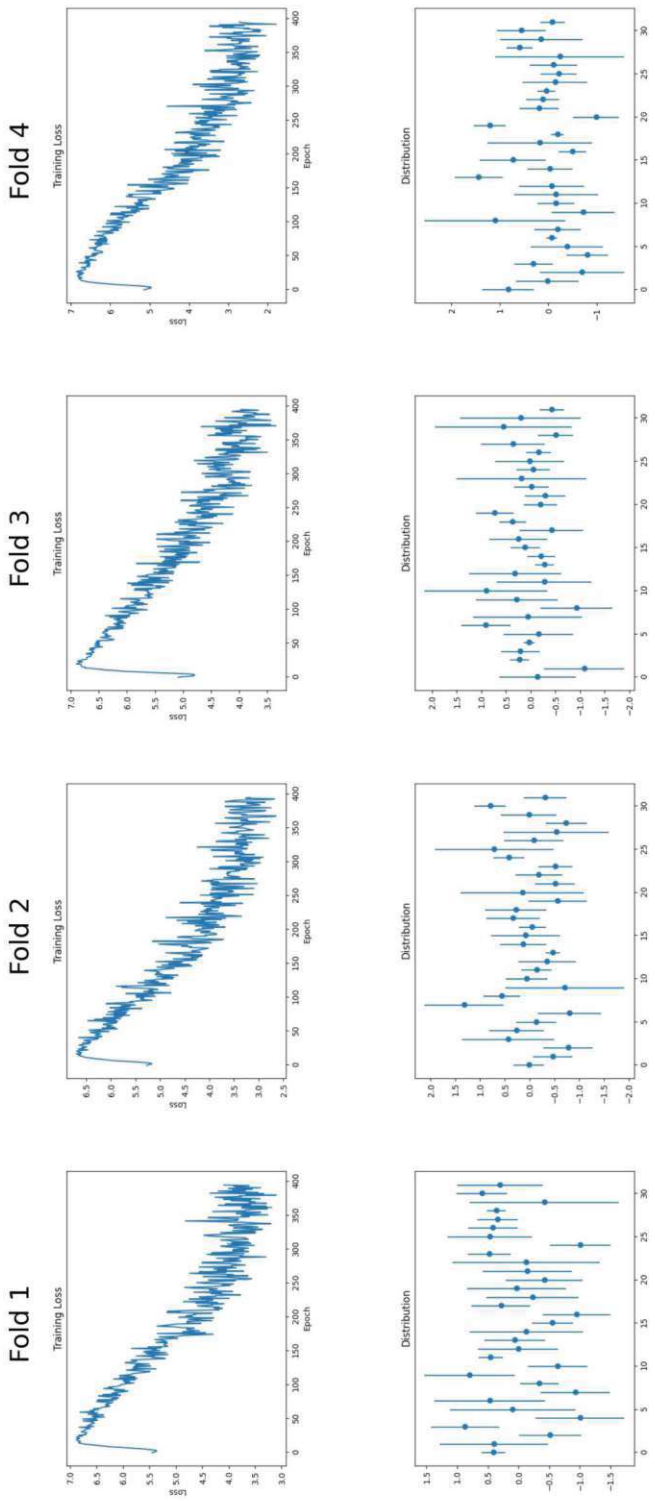


Figure 6.2: Loss curves and the feature distributions of each fold training for 400 training epochs for the hyperparameters learning rate=0.0001, and batch size=16. The average ARI on GMM is 0.365. The average ARI on k-means is 0.423.

Therefore we want to guide the network during the training process and add as much annotation data as required. We also want to be able to increase the complexity of the classes that the network should be able to distinguish, as we want to train models not only for one lineage, but for multiple lineages.

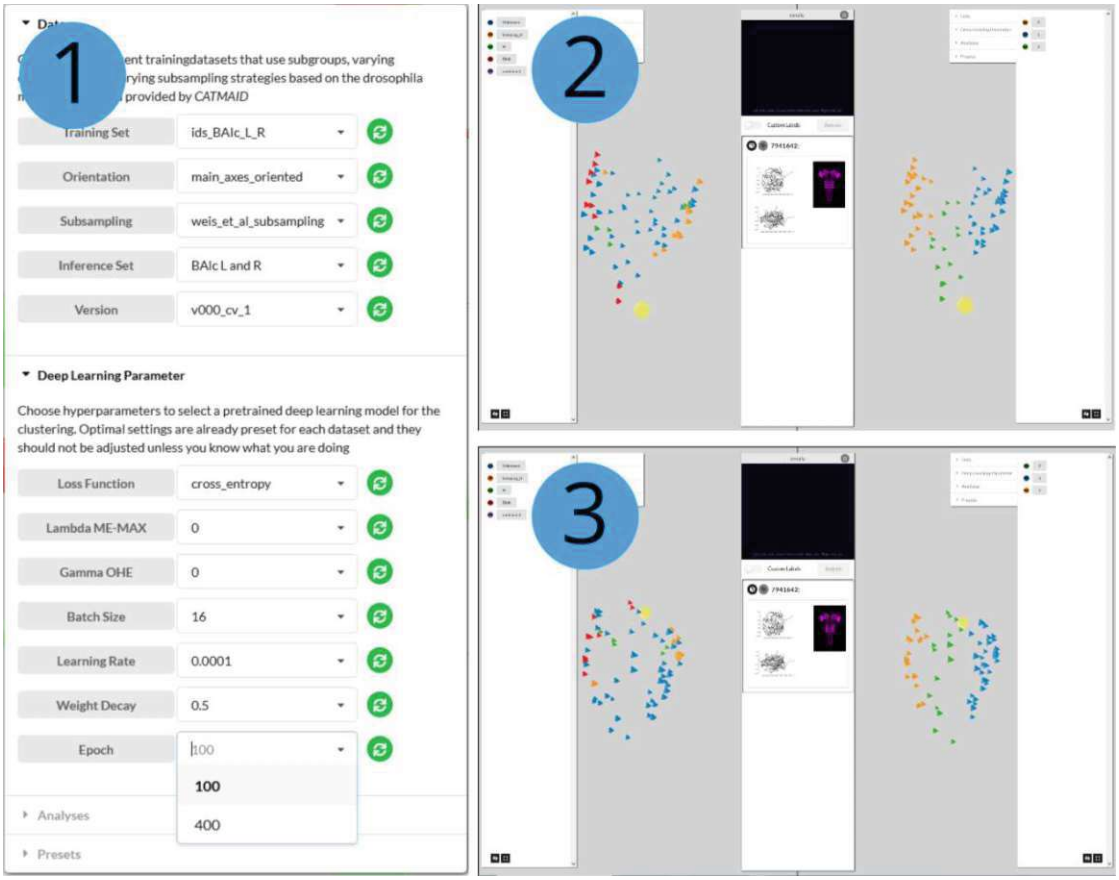


Figure 6.3: Graph embeddings generated with GraphPaws trained with the parameters depicted in the accordion menu (1) for 100 epochs (2) and for 400 epochs (3).

We elaborated the visual analytics NetDive in combination with semi-supervised learning to let the user drive the learning process. Figure 6.3(1) depicts the parameter settings of the model M0, that we trained for 100 and for 400 epochs. Figure 6.3(2) shows the graph embeddings of the model M0 trained for 100 epochs and Figure 6.3(3) shows the graph embeddings of the model M0 trained for 400 epochs. The left views of the NetDive interface color the embeddings according to the BAlc ground truth and the right views color the embeddings according to embedding cluster predictions computed with GMM. The color assignments for the clusters are random, therefore we have different cluster colors on the left and the right views. We also have a different number of cluster colors, as we only assigned three different cluster labels to the neuron graphs and some neuron graphs have no cluster label. These unlabeled neurons are colored blue on the left views.

On the right views we cluster the neurons in three groups. We expect that the labeled neurons are clustered correctly. On the left views we see that the red colored neurons form a denser cluster after training with more iterations, but neurons colored green according to their ground truth mix up with the red and orange colored embeddings. We can conclude that the network learns a more unambiguous representation for the shapes of neuron graphs belonging to the red cluster.

6.2 Node Collapsing

Though we implemented regularization techniques to avoid node collapsing we experienced node collapsing throughout the experiments. We study the model M2, named and color coded in Table 5.1, with the hyperparameters $\lambda = 0$, $\gamma = 1$, batch size=16, and learning rate=0.006 trained on mse with ARI 0.554 as an example. We train the model five times with the same hyperparameters and random initialization and compare the ARI results. The results are depicted in Table 5.3.

Figure 6.4 shows the loss curves and the feature value distribution of each fold of the first retrained model based on the whole dataset BAlc L / R. The loss curves immediately drop down to a fixed value, in this case approximately -1.1. The corresponding feature value distributions represent the latent space of 32 dimensions. They show the mean value and the standard deviation (std) for each latent feature for a set of input graphs. In this experiment we computed the feature value distributions for the dataset BAlc L / R. The distributions depicted for each fold model show, that each graph is mapped to the same 32 dimension values, as there is no variance indicated by the std bars.

On the contrary, if models do learn features, patterns become visible. Figure 6.5 depicts the loss curves and feature value distributions of the model M3, named and color coded in Table 5.1, trained on mse with the hyperparameters learning rate=3e-05, batch size=32, λ=1, γ=1. The loss curves have a downwards trend and the feature value distributions demonstrate that the model is capable of learning distributed representations, even though the datasets are small.

Figure 6.6 depicts the foldwise average ARI values for each of the five retrained models, i.e., M1, M2, M3, M4, and M5. Each fold model is evaluated 100 times and the depicted ARI values for each fold are the averaged values. The folds are represented by different colors. Fold 1 is colored cyan, Fold 2 is colored violet, Fold 3 is colored magenta and Fold 4 is colored gray for each model. The model numbers 1-5 are given on the horizintal axis. Figure 6.6(a) is based on k-means clustering, Figure 6.6(b) is based on GMM clustering.

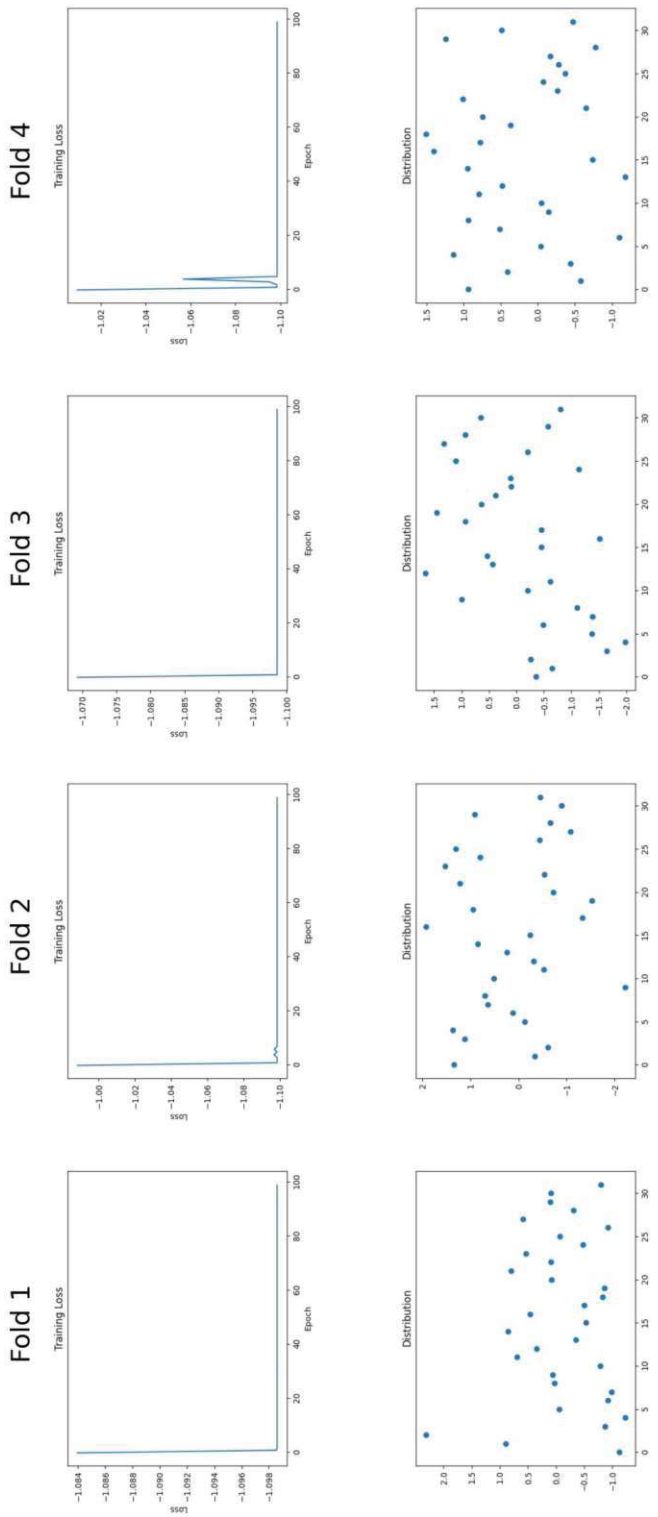


Figure 6.4: Foldwise analyses of the M2 model trained with mse for the hyperparameters learning rate=0.006, batch size=16, λ=0, γ=1. The first row depicts the loss curves and the second row the feature value distributions.

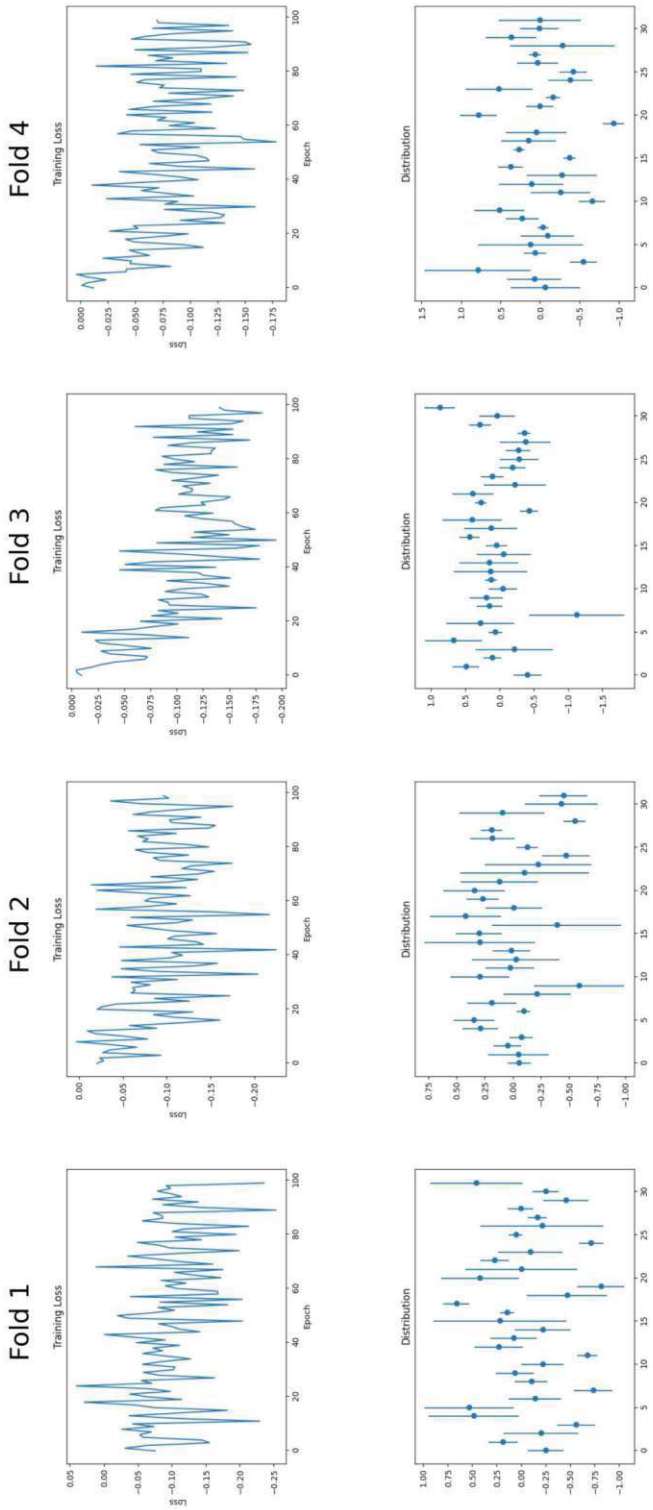
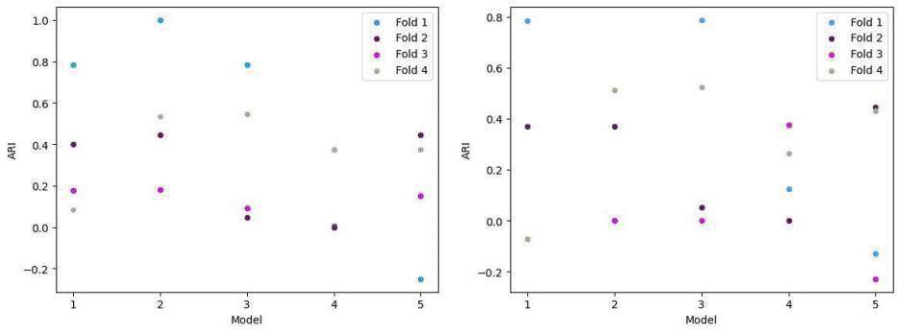


Figure 6.5: Analyses of the M3 model trained with mse for the hyperparameters learning rate=3e-05, batch size=32, λ=1, γ=1



(a) Based on k-means clustering (b) Based on GMM clustering

Figure 6.6: Fold analyses of the model M2 trained with mse for the hyperparameters learning rate=0.006, batch size=16, λ=0, γ=1. Each color represents a fold, each column represents a model, and the vertical axis indicates the average ARI score.

We see, that the averaged fold ARI values vary between <-0.2 and 1 and no pattern is visible that one fold out-performs another fold. Due to node collapsing, all inputs are mapped to almost the same output and the clustering then divides the output representations based on insignificant changes and produces random results. We would expect a higher performance consistency between the models if they were capable of learning.

6.3 Regularization

While we see the downward trend on the model training analysis in Figure 6.5, we miss this downward trend for the error curves demonstrated in Figure 6.7. The plots correspond to the model M1, named and color coded in Table 5.1.

We observed a correspondence between the model learning capability and the regularization parameters λ and γ. The parameter λ is responsible for increasing the influence of the ME-MAX regularizer and gamma determines the influence of the One-Hot-Enforcement. ME-MAX increases the averaged entropy across a batch, i.e., ensures, that each class is represented in a batch. The One-Hot-Enforcement regularizer prevents node collapsing by enforcing one-hot-encodings for each class label.

The following Figures 6.8-6.14 depict the loss curves and feature distributions of the semi-supervised training described in Sub-section 5.4.3 for Fold 1. The figures show that there is a correlation between loss curves that immediately fall to a fixed value and feature distributions that collapse to single representations. We also see, that increasing the values for λ and γ does improve the learning capability. For the learning rates 0.006, 0.003 and 0.001 we see node collapsing if the regularization terms are set to zero.

We also see for some learning rates, that the downwards trend of the loss curves is dependent on the value of λ, i.e., the influence of the ME-MAX regularizer. This effect

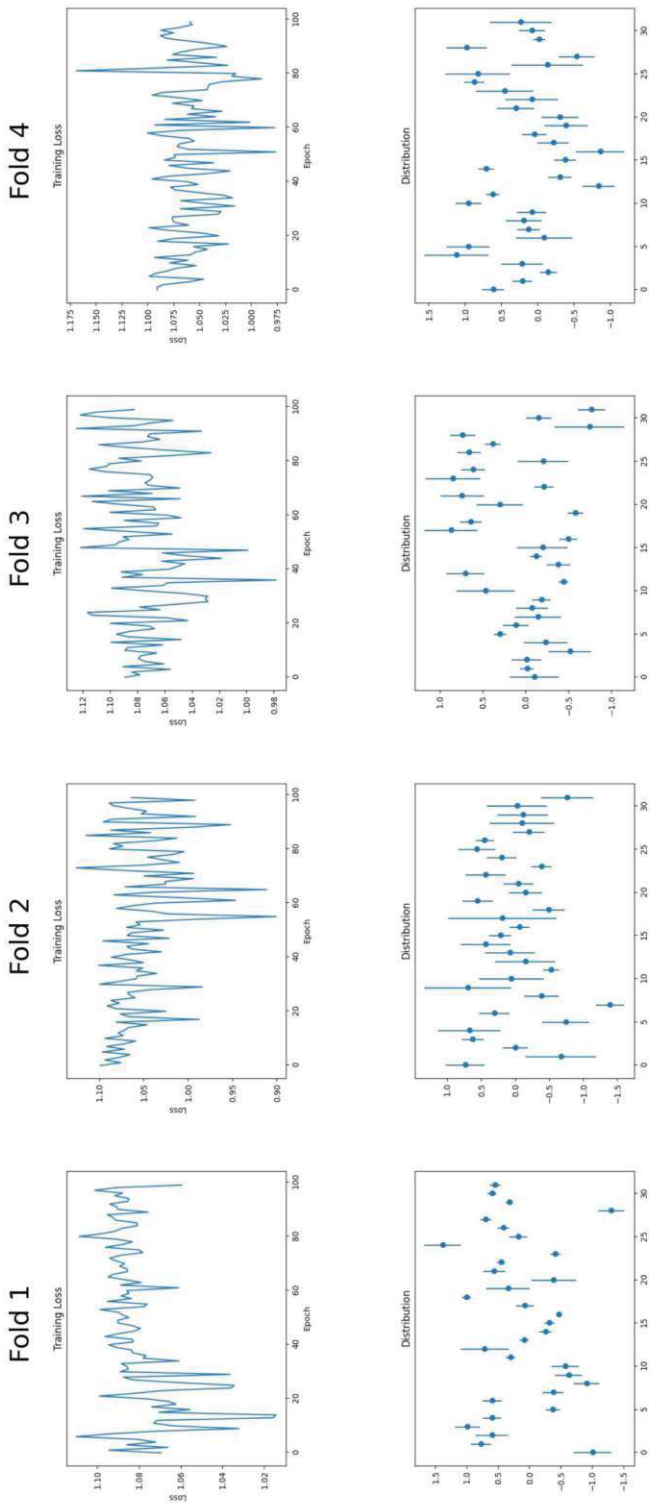


Figure 6.7: Analyses of model M1 trained with cross-entropy for the hyperparameters learning rate=0.0001, batch size=32, λ=0 and γ=0.

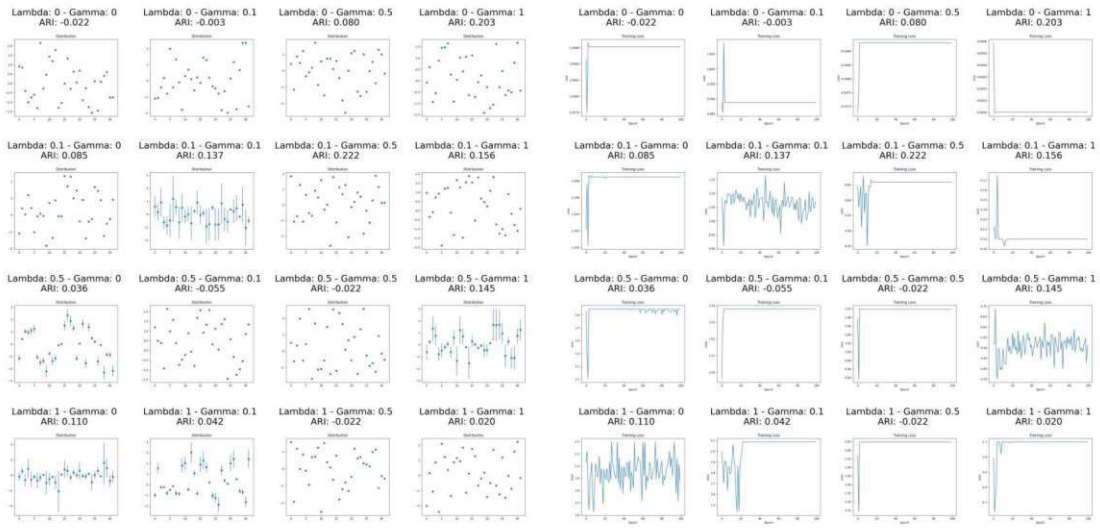


Figure 6.8: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=0.006**.

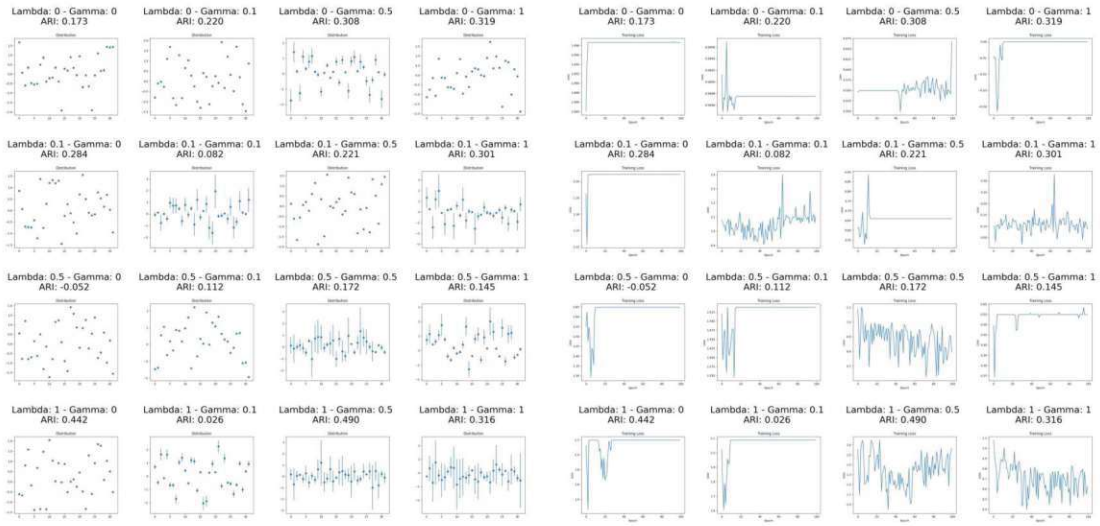


Figure 6.9: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=0.003**.

is visible for learning rates 1e-05, 3e-05, 6e-05, and 0.0001. The third observation is, that we see an increasing feature value distribution depending on both λ and γ. This holds true for all learning rates depicted in the Figures 6.8-6.14. Model M1 is trained with λ=0 and γ=0 and supports the hypothesis, that the missing regularization leads to insufficient learning.

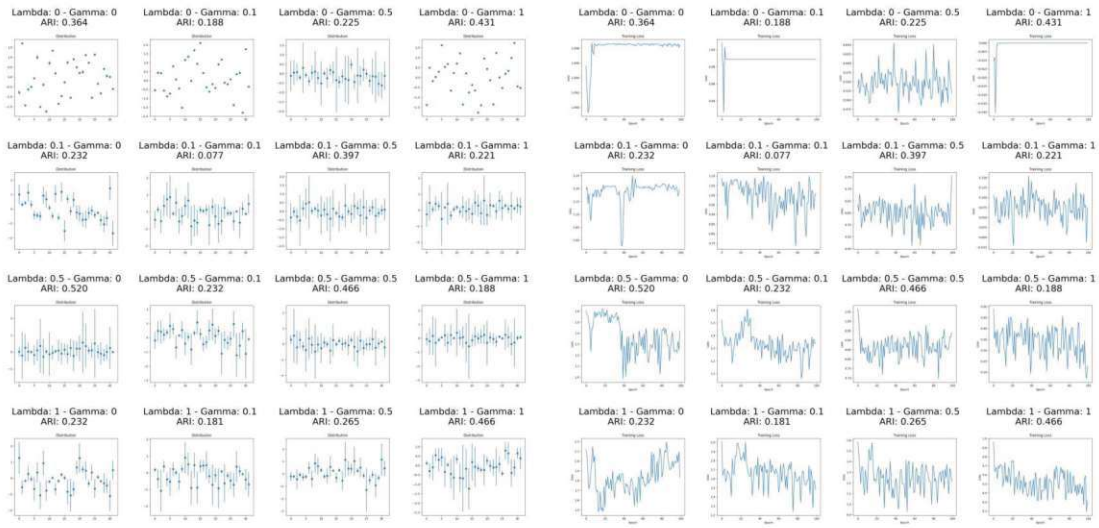


Figure 6.10: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=0.001**.

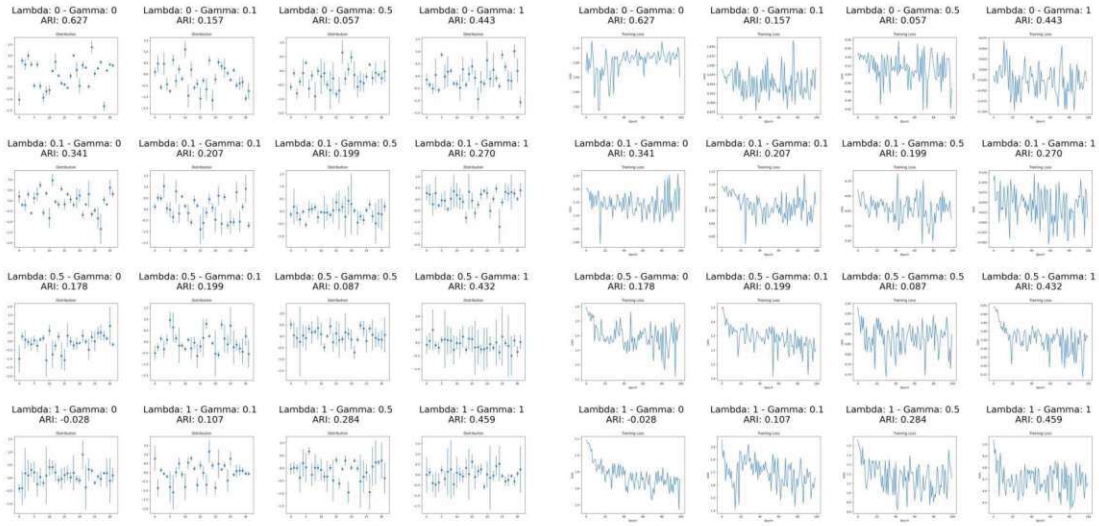


Figure 6.11: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=0.0001**.

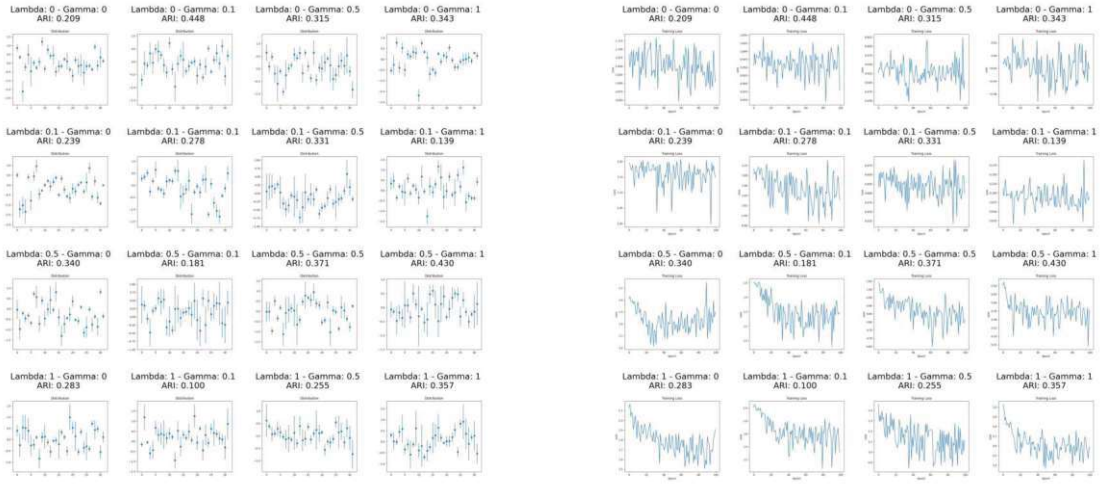


Figure 6.12: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=6e-05**.

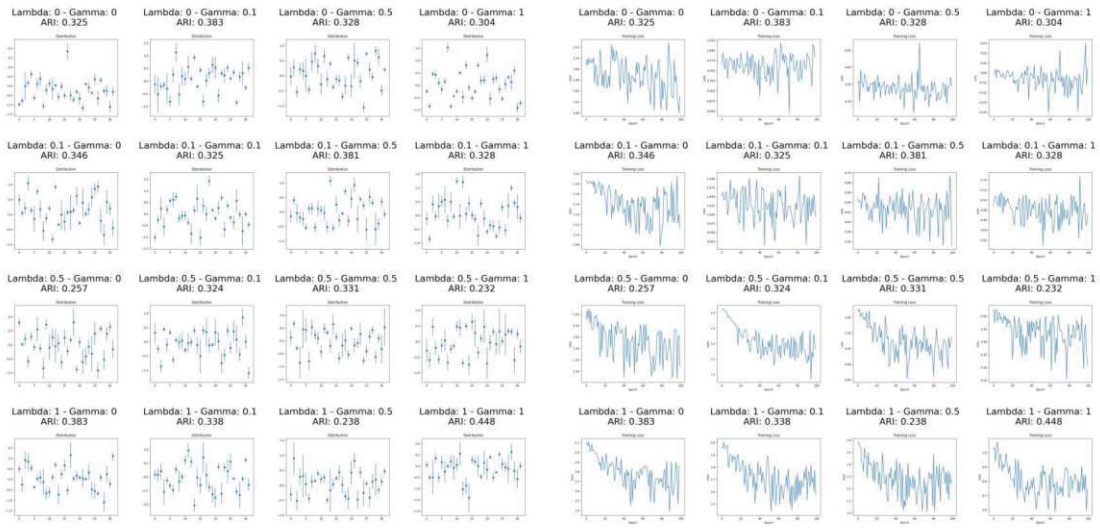


Figure 6.13: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=3e-05**.

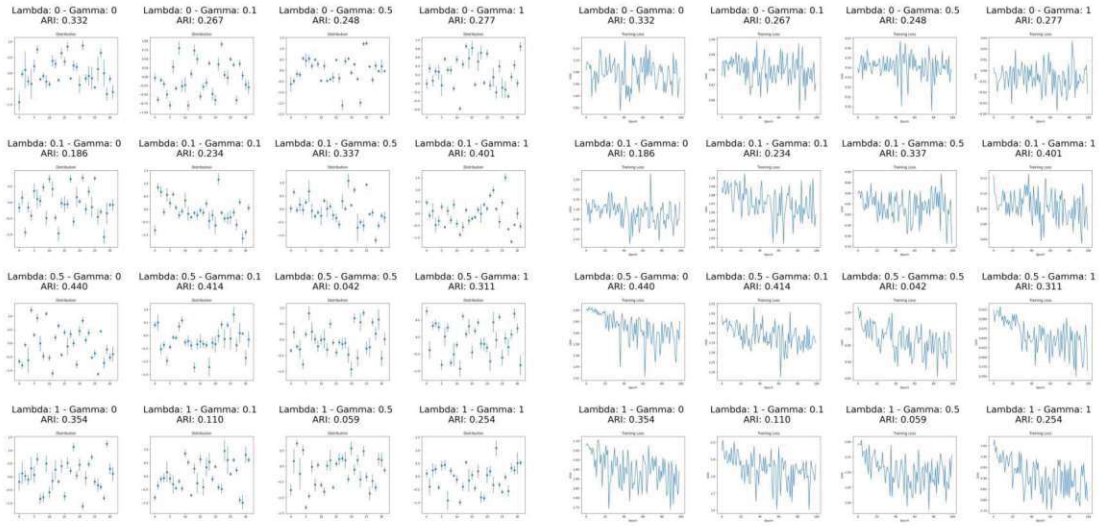


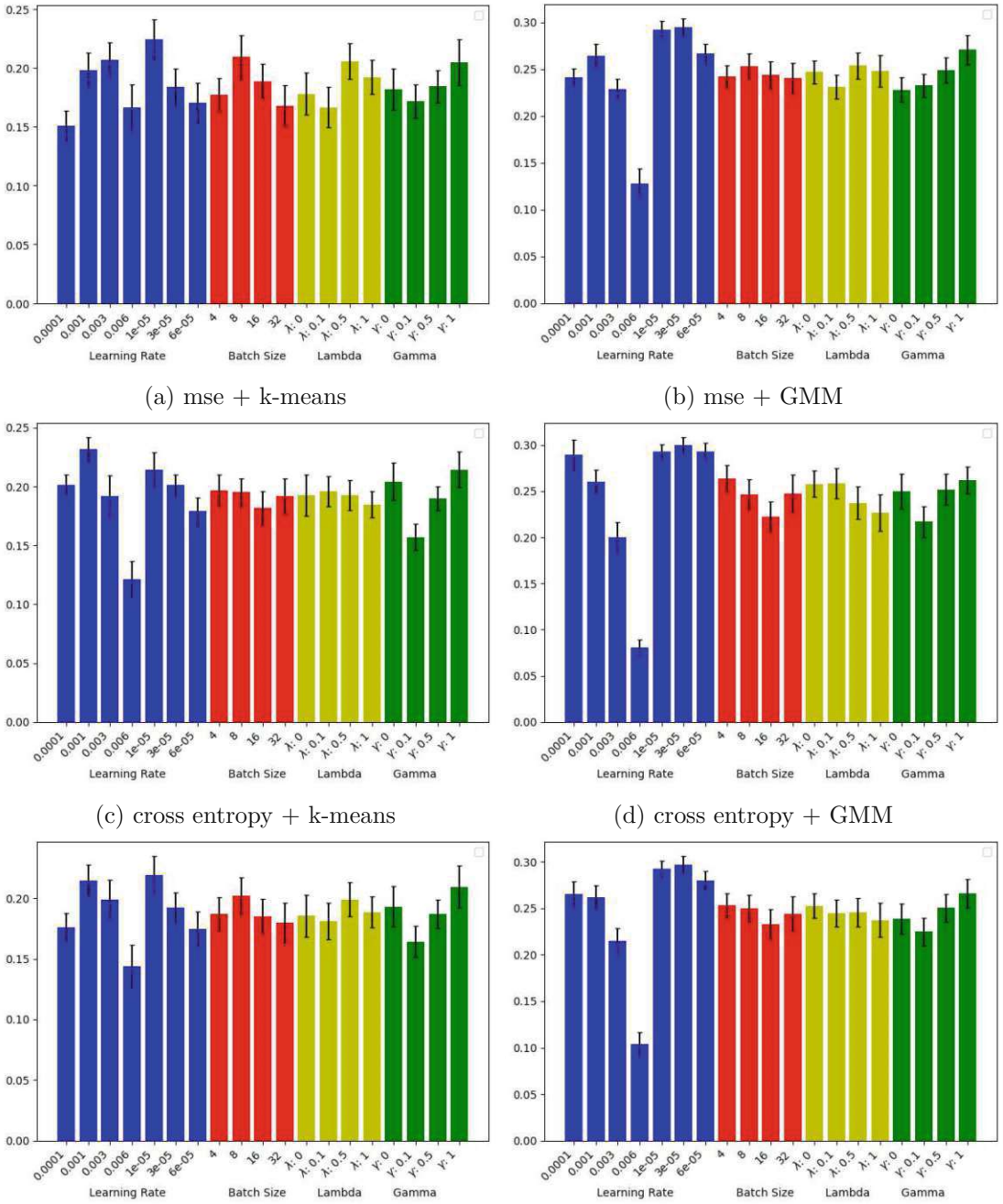
Figure 6.14: Fold 1 models trained with cross entropy, batch size=32 and **learning rate=1e-05**.

6.4 Hyperparameter Analysis

We analyze the influence of hyperparameter values. We compute the mean and variance for each hyperparameter value and show the results in Figure 6.15. The values are computed based on validation data. To compute the mean and variance values we always take into account all the models that are trained with a specific value for a specific hyperparameter, e.g., the learning rate, with all variations for the remaining hyperparameters.

We visually observe correlations between the distributions regarding the loss function and regarding the clustering algorithm. The models trained on mse, respectively cross entropy, have similar performance ordering for the values for each hyperparameter, i.e., $\lambda = 0.5$ is the best value on average for models trained on mse, $\lambda = 1$ is on second place, $\lambda = 0$ is third, and $\lambda = 0.1$ performs worst on average. Models clustered with GMM achieve better maximum results. The models evaluated according to Figure 6.15(b) and 6.15(d) have almost the same average performance, which is 0.245 and only differs in the following digits. We determine that 6.15(d) might produce more meaningful results, as cross entropy and GMMs are both based on data distributions. We also observe, that the choice of the learning rate has the biggest performance impact as the mean values vary most depending on the learning rate, most clearly visible for example in Figure 6.15(d).

In order to test, if the combination of best performing feature values leads to a strong model, we retrain a model based on cross entropy with the best performing values from Figure 6.15(d), i.e., with learning rate=3e-05, batch size=4, λ=0.1, γ=1. The combination of these hyperparameters also suffers from the learning incapability as the



(a) mse + k-means

(b) mse + GMM

(c) cross entropy + k-means

(d) cross entropy + GMM

(e) k-means

(f) GMM

Figure 6.15: Mean and variances analyses. Each bar visualizes the ARI distribution across the subset of the 896 models that contains the corresponding hyperparameter value, e.g., learning rate 0.0001 or batch size 4, etc.

loss curve does not go down during training. The average ARI scores of the reruns have a high variance and mediocre performance as shown in Table 5.3 for model M4.

6.5 Clustering

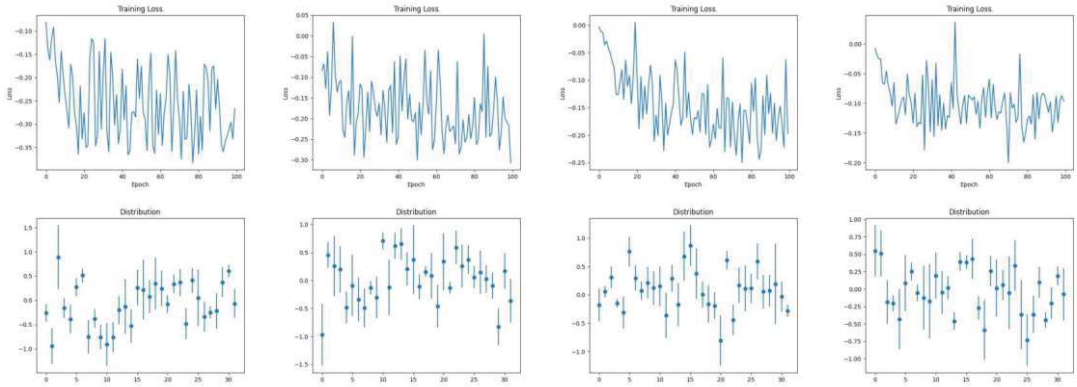
As discussed in chapter 5 and visible when comparing Figure 6.15(a)(c)(e) with 6.15(b)(d)(f), GMM clustering leads to better ARI scores than k-means clustering. These Figures each visualize the scores for models trained on cross entropy and on mse. We do not see significant differences in the ARI distributions of scores computed with GMM for models trained on cross entropy compared to models trained on mse. The same applies to ARI distribution scores computed with k-means. We therefore assume, that the clusters that are generated during the cross entropy training are not elongated, i.e., the clusters are circular in the embedding space, as k-means would not work otherwise, as depicted in Figure 2.2. Another explanation would be that the results are too random due to the small dataset size of ground truth labels.

The ARI scores computed with GMM and k-means vary depending on the initial random seed. For each model and fold 100 Gaussian mixture models are computed. Due to the stochastic calculation by averaging the results over 100 Gaussian mixture models, the results become stable. We depict this in Figure 5.21, which shows the ARI scores of 896 different models. Computing the ARI scores another time, again by averaging 100 GMMs, returns very similar results, as visible if comparing Figure 5.21(a) with Figure 5.21(b) and if comparing Figure 5.21(c) with Figure 5.21(d). We initialize the scikit-learn GMM clustering randomly and the scikit-learn k-means clustering with k-means++. If we use the parameter *random* instead of *k-means++* for the k-means initialization, the results are less stable.

6.6 NetDive Iterations

The results of the NetDive iterations to train lineage CM4 are depicted in Table 5.6. While the performance of the last iteration suggests a potential positive trend in performance, it is not possible to conclusively determine whether this represents a true trend or is the result of random variation. Figure 6.16 shows the loss curves and feature distributions of each iteration model. We see that all models have a variance in the feature distribution and the loss curves have downward trends.

Using NetDive to annotate support samples and to start the training seems intuitive. The tool yet needs to be tested by domain experts. In our case we had the ground truth for lineage CM4 available in NetDive and were able to leverage this using the feature to select clusters after applying the ground truth labels in NetDive. We then annotated single samples of the selected clusters for the retraining. We did this to be able to cross-check and evaluate against the manually assigned ground truth. Without having a corresponding ground truth, the user would apply the predicted colors to the data points and analyze the predicted clusters in order to assign labels.



(a) Self-Supervised (b) Iteration 1 (c) Iteration 2 (d) Iteration 3

Figure 6.16: Loss curves and feature distributions of the models trained on CM4 with increasing number of support samples, discussed in Sub-section 5.4.6.

CHAPTER 7

Conclusion and Future Work

This thesis addresses the problem of clustering graph data without initially having a ground truth for training whilst giving the user the possibility to guide the training process with low effort.

We developed this concept as the completely self-supervised training as implemented with GraphDINO [WHLE21] did not work well for the drosophila melanogaster larval graph representations. Originally we tried to steer the training by adjusting the augmentations of the contrastive learning, but we did not get the expected results and therefore looked into semi-supervised learning and visual analytics.

The reason that we did not receive satisfying results with GraphDINO probably lies in the heterogeneity of the drosophila melanogaster graph data in CATMAID. The shapes within and in-between lineages vary a lot and it is difficult to determine how the visual similarity that we use to define the ground truth reflects in the graph structure and topology. Besides that, Weis et al. [WHLE21] encoded compartment information for the BBP dataset, i.e., soma, axon, basal dendrite, apical dendrite, which is not widely available for the CATMAID drosophila melanogaster neurons. Another problem we faced was the limited amount of data we used for training. We could not use all available drosophila melanogaster neuron graphs as we can not evaluate the results without a ground truth. We hand selected visually distinguishable neuron graph shapes and annotated those ourselves.

We did achieve better results with GraphPAWS than with GraphDINO as depicted in Table 5.4. We trained 896 models for GraphPAWS, the ARI scores varied between approximately -0.2 and 0.627, with some of the models having no decrease in the loss curves and some of the models suffering from node collapsing. Our conclusion is, that GraphDINO needs to be tested on bigger datasets to receive more stable results, but we see that we can steer the training using GraphPAWS and have more control than with GraphDINO as we can add support samples.

As GraphDINO did not perform well, NetDive is a useful tool to incrementally improve the training by adding labeled data, as shown and discussed in sections 5.4.6 and 6.6. In order to execute user studies with domain experts we need to first improve GraphPAWS, so that we can reliably state that the results improve over the course of adding more support samples. The required time to annotate embeddings using NetDive depends on the quality of the clusters and the number of data points. The user can select the clusters they want to analyze and use the detail view to get an overview of the cluster content. Based on the findings the user can annotate prototypical and falsely clustered neuron graphs. NetDive is data agnostic and could be coupled with dimensionality reduced representation vectors of any input data type besides graphs.

We identified components that are interesting to elaborate in future work, regarding the NetDive UI, the deep learning model, and the evaluation. NetDive UI can be improved by adding simulations that visualize the cluster changes over time during training with color updates. It is also possible to add more characteristics of the neurons in the detail view, depicted in Figure 3.8(3). In future work we want to provide interaction techniques like brushing and linking for a feature space visualization of neurons to understand correlations between clusters and the cluster contents. We could extend the spatial representations and use the properties size and opacity of each data point to encode additional information besides the cluster label, e.g., the certainty of the cluster assignment in the opacity and the variance over a sequence of models in the size of the data point.

For the GraphPAWS training we could experiment with finetuning the model after adding new support samples, instead of training new randomly initialized models, and therefore reduce training times. We also discussed improving the preprocessing of the neuron graph data with alternative subsampling strategies. In the subsampling that we currently adopted from Weis et. al [WHLE21], nodes are randomly removed that have of maximum of two neighbors and are therefore no branching points. This still adds biases to the neuron graphs, as the nodes are not evenly distributed as a result. As future work it could be an option to (1) solely keep the branching and leaf points to work with a higher level abstraction of the neurons or to (2) perform equidistant subsampling.

It would be interesting to perform user studies with experts in the field of neuroscience to see how users outside the domain of deep learning can use visual analytics to refine pre-trained models and which features they are missing in the current NetDive setup. Our final conclusion is that NetDive, with our semi-supervised GraphPAWS in the backend, is a promising direction for clustering graph data, which should be explored further.

Bibliography

[AA15] Rubén Armañanzas and Giorgio A. Ascoli. Towards the automatic classification of neurons. *Trends in Neurosciences*, 38(5):307–318, May 2015.

[AB12] Ikpe Akpan and Roger Brooks. Users' perceptions of the relative costs and benefits of 2D and 3D visual displays in discrete-event simulation. *SIMULATION*, 88:464–480, April 2012.

[aba] Allen brain atlas. `http://alleninstitute.github.io/AllenSDK/cell_types.html`. Accessed: 2023-07-06.

[ACM+21] Mahmoud Assran, Mathilde Caron, Ishan Misra, Piotr Bojanowski, Armand Joulin, Nicolas Ballas, and Michael Rabbat. Semi-supervised learning of visual features by non-parametrically predicting view assignments with support samples. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, October 2021.

[All16] Allen cell types database technical white paper: Cell morphology and histology. *Allen Institute*, 2016.

[AZH+21] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1):53, March 2021.

[Bal11] Pierre Baldi. Autoencoders, unsupervised learning and deep architectures. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning workshop - Volume 27*, UTLW'11, pages 37–50, Washington, USA, July 2011. JMLR.org.

[BBL+17] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, July 2017.

[BCS22] Angie Boggust, Brandon Carter, and Arvind Satyanarayan. Embedding Comparator: Visualizing differences in global structure and Local neighborhoods via small multiples. *27th International Conference on Intelligent User Interfaces*, pages 746–766, March 2022.

[BF81] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume I.* Addison-Wesley (C), January 1981.

[BG14] Dibya Jyoti Bora and Anil Kumar Gupta. A comparative study between fuzzy clustering algorithm and hard clustering algorithm. *International Journal of Computer Trends and Technology*, 10(2):108–113, April 2014.

[BS10] Arindam Banerjee and Hanhuai Shan. Model-based clustering. In *Encyclopedia of Machine Learning*, pages 686–689. Springer US, Boston, MA, 2010.

[Cho] Tamal Chowdhury. How to set up a react project with vite. `https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-vite`. Accessed: 2023-04-12.

[CKNH20] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *ICML'20*, pages 1597–1607. JMLR.org, July 2020.

[CMM+20] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, pages 9912–9924, Red Hook, NY, USA, December 2020. Curran Associates Inc.

[CMO+16] Marta Costa, James D. Manton, Aaron D. Ostrovsky, Steffen Prohaska, and Gregory S.X.E. Jefferis. NBLAST: Rapid, sensitive comparison of neuronal structure and construction of neuron family databases. *Neuron*, 91(2):293–311, July 2016.

[COB22] Dexiong Chen, Leslie O'Bray, and Karsten M. Borgwardt. Structure-aware transformer for graph representation learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 3469–3489. PMLR, 2022.

[CTM+21] Mathilde Caron, Hugo Touvron, Ishan Misra, Herve Jegou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging Properties in Self-Supervised Vision Transformers. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9630–9640, October 2021.

[CZG+22] Yi Chen, Qinghui Zhang, Zeli Guan, Ying Zhao, and Wei Chen. GEMvis: a visual analysis method for the comparison and refinement of graph embedding models. *The Visual Computer*, 38(9-10):3449–3462, September 2022.

[DB20] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *AAAI 2021 Workshop on Deep Learning on Graphs: Methods and Applications*, 2020.

[DBK+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. OpenReview.net, October 2020.

[DKKAK11] Ozgur Demir-Kavuk, Mayumi Kamada, Tatsuya Akutsu, and Ernst-Walter Knapp. Prediction using step-wise L1, L2 regularization and feature selection for small data sets with large number of features. *BMC Bioinformatics*, 12:412, October 2011.

[DLCBP+13] Javier DeFelipe, Pedro L. López-Cruz, Ruth Benavides-Piccione, Concha Bielza, Pedro Larrañaga, Stewart Anderson, Andreas Burkhalter, Bruno Cauli, Alfonso Fairén, Dirk Feldmeyer, Gord Fishell, David Fitzpatrick, Tamás F. Freund, Guillermo González-Burgos, Shaul Hestrin, Sean Hill, Patrick R. Hof, Josh Huang, Edward G. Jones, Yasuo Kawaguchi, Zoltán Kisvárday, Yoshiyuki Kubota, David A. Lewis, Oscar Marín, Henry Markram, Chris J. McBain, Hanno S. Meyer, Hannah Monyer, Sacha B. Nelson, Kathleen Rockland, Jean Rossier, John L. R. Rubenstein, Bernardo Rudy, Massimo Scanziani, Gordon M. Shepherd, Chet C. Sherwood, Jochen F. Staiger, Gábor Tamás, Alex Thomson, Yun Wang, Rafael Yuste, and Giorgio A. Ascoli. New insights into the classification and nomenclature of cortical GABAergic interneurons. *Nature Reviews Neuroscience*, 14(3):202–216, February 2013.

[DW22] Mingjing Du and Fuyu Wu. Grid-based clustering using boundary detection. *Entropy*, 24(11):1606, November 2022.

[DXTL22] Kaize Ding, Zhe Xu, Hanghang Tong, and Huan Liu. Data augmentation for deep graph learning: A survey. *SIGKDD Explor.*, 24(2):61–77, 2022.

[Eds03] Robert M. Edsall. The parallel coordinate plot in action: design and use for geographic visualization. *Computational Statistics & Data Analysis*, 43(4):605–619, August 2003.

[EGLH22] Linus Ericsson, Henry Gouk, Chen Change Loy, and Timothy M. Hospedales. Self-supervised representation learning: Introduction, ad-

vances, and challenges. *IEEE Signal Processing Magazine*, 39(3):42–62, May 2022.

[FH18] Chaoran Fan and Helwig Hauser. Fast and accurate CNN-based brushing in scatterplots. *Computer Graphics Forum*, 37(3):111–120, 2018.

[FMFKG07] Miguel Angel Fernández-Moreno, Carol L. Farr, Laurie S. Kaguni, and Rafael Garesse. Drosophila melanogaster as a model system to study mitochondrial biology. In *Methods in Molecular Biology*, pages 33–49. Humana Press, 2007.

[for] Github: 3d-force-graph. `https://github.com/vasturiano/3d-force-graph`. Accessed: 2024-09-10.

[FS19] Pasi Fränti and Sami Sieranoja. How much can k-means be improved by using better initialization and repeats? *Pattern Recognition*, 93:95–112, September 2019.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GMP21] Atika Gupta, Priya Matta, and Bhasker Pant. Graph neural network: Current state of art, challenges and applications. *Materials Today: Proceedings*, 46:10927–10932, 2021.

[GSA+20] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, and Michal Valko. Bootstrap your own latent - a new approach to self-supervised learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, pages 21271–21284, Red Hook, NY, USA, December 2020. Curran Associates Inc.

[Guc17] Vladimir Guchev. Effective user interactions for visual analytics tools. In Paolo Bottoni, Cristina Gena, Andrea Giachetti, Samuel Aldo Iacolina, Fabio Sorrentino, and Lucio Davide Spano, editors, *Proceedings of the Doctoral Consortium, Posters and Demos at CHItaly 2017 co-located with 12th Biannual Conference of the Italian SIGCHI Chapter (CHItaly 2017), Cagliari, Italy, September 18-20, 2017*, volume 1910 of *CEUR Workshop Proceedings*, pages 90–101. CEUR-WS.org, 2017.

[Ham20] William L. Hamilton. *Graph Representation Learning*. Springer International Publishing, 2020.

[HB03] Mark Harrower and Cynthia Brewer. ColorBrewer.org: An online tool for selecting colour schemes for maps. *The Cartographic Journal*, 40:27–37, June 2003.

[HFW+20] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross B. Girshick. Momentum contrast for unsupervised visual representation learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 9726–9735. Computer Vision Foundation / IEEE, 2020.

[HKMG22] Florian Heimerl, Christoph Kralj, Torsten Möller, and Michael Gleicher. embcomp : Visual interactive comparison of vector embeddings. *IEEE Transactions on Visualization and Computer Graphics*, 28(8):2953–2969, August 2022.

[HKPC19] Fred Hohman, Minsuk Kahng, Robert Pienta, and Duen Horng Chau. Visual analytics in deep learning: An interrogative survey for the next frontiers. *IEEE Transactions on Visualization and Computer Graphics*, 25(8):2674–2693, August 2019.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 1025–1035, Red Hook, NY, USA, December 2017. Curran Associates Inc.

[IF09] Petra Isenberg and Danyel Fisher. Collaborative brushing and linking for co-located visual analytics of document collections. *Computer Graphics Forum - CGF*, 28:1031–1038, June 2009.

[IZJ18] Zainura Idrus, Hedzlin Zainuddin, and A.D.M. Ja'afar. Visual analytics: designing flexible filtering in parallel coordinate graph. *Journal of Fundamental and Applied Sciences*, 9:23, January 2018.

[JZH21] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electronic Markets*, 31(3):685–695, April 2021.

[KBB+10] Daniel A. Keim, Peter Bak, Enrico Bertini, Daniela Oelke, David Spretke, and Hartmut Ziegler. Advanced visual analytics interfaces. In *Proceedings of the International Conference on Advanced Visual Interfaces*. ACM, May 2010.

[KGH+21] Evangelos Karatzas, Maria Gkonta, Joana Hotova, Fotis A. Baltoumas, Panagiota I. Kontou, Christopher J. Bobotsis, Pantelis G. Bagos, and Georgios A. Pavlopoulos. VICTOR: A visual analytics web application for comparing cluster sets. *Computers in Biology and Medicine*, 135:104557, August 2021.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.

[KT13] Aleksi Kallio and Jarno Tuimala. Data mining. In *Encyclopedia of Systems Biology*, pages 525–528. Springer New York, 2013.

[KTC+19] Minsuk Kahng, Nikhil Thorat, Duen Horng (Polo) Chau, Fernanda B. Viégas, and Martin Wattenberg. GAN Lab: Understanding complex deep generative models using interactive visual experimentation. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):310–320, January 2019.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[LKHS20] Phuc Le-Khac, Graham Healy, and Alan Smeaton. Contrastive representation learning: A framework and review. *IEEE Access*, October 2020.

[LLH+22] Xu Liu, Yuxuan Liang, Chao Huang, Yu Zheng, Bryan Hooi, and Roger Zimmermann. When do contrastive learning signals help spatio-temporal graph forecasting? In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*. ACM, November 2022.

[LNH+18] Quan Li, Kristanto Sean Njotoprawiro, Hammad Haleem, Qiaoan Chen, Chris Yi, and Xiaojuan Ma. EmbeddingVis: A visual analytics approach to comparative network embedding inspection. *2018 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 48–59, October 2018.

[LNO+13] Jennifer Lovick, Kathy Ngo, Jaison Omoto, Darren Wong, Joseph Nguyen, and Volker Hartenstein. Postembryonic lineages of the drosophila brain: I. development of the lineage-associated fiber tracts. *Dev Biol.*, 2013.

[LWBM22] Zipeng Liu, Yang Wang, Jürgen Bernard, and Tamara Munzner. Visualizing graph neural networks with CorGIE: Corresponding a graph to its embedding. *IEEE Transactions on Visualization and Computer Graphics*, 2022.

[MCSM21] Grégoire Mialon, Dexiong Chen, Margot Selosse, and Julien Mairal. GraphiT: Encoding graph structure in transformers. *arXiv*, June 2021.

[mp] Max Planck: Warum erforschen wissenschaftler fliegen? `https://www.mpg.de/10885922/`. Accessed: 2023-07-04.

[MP69] Marvin Lee Minsky and Seymour Papert. *Perceptrons An Introduction to Computational Geometry*. Brand: MIT Press, 1969.

[MP19] Stephanie E. Mohr and Norbert Perrimon. Drosophila melanogaster: a simple system for understanding complexity. *Disease Models & Mechanisms*, 12(10), September 2019.

[mw:] Merriam-Webster neuroscience. `https://www.merriam-webster.com/dictionary/neuroscience`. Accessed: 2023-04-12.

[Nat89] National Research Council (US) Committee on Research Opportunities in Biology. *Opportunities in Biology*. National Academies Press (US), Washington (DC), 1989.

[NMA+20] Quang Vinh Nguyen, Natalie Miller, David Arness, Weidong Huang, Mao Lin Huang, and Simeon Simoff. Evaluation on interactive visualization data with scatterplots. *Visual Informatics*, 4(4):1–10, December 2020.

[npm] Npm trends: Vite vs create-react-app. `https://npmtrends.com/create-react-app-vs-vite`. Accessed: 2024-09-10.

[PHG+18] Nicola Pezzotti, Thomas Höllt, Jan Van Gemert, Boudewijn P.F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. DeepEyes: Progressive visual analytics for designing deep neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):98–108, January 2018.

[PXL+21] Hanchuan Peng, Peng Xie, Lijuan Liu, Xiuli Kuang, Yimin Wang, Lei Qu, Hui Gong, Shengdian Jiang, Anan Li, Zongcai Ruan, Liya Ding, Zizhen Yao, Chao Chen, Mengya Chen, Tanya L. Daigle, Rachel Dalley, Zhangcan Ding, Yanjun Duan, Aaron Feiner, Ping He, Chris Hill, Karla E. Hirokawa, Guodong Hong, Lei Huang, Sara Kebede, Hsien-Chi Kuo, Rachael Larsen, Phil Lesnar, Longfei Li, Qi Li, Xiangning Li, Yaoyao Li, Yuanyuan Li, An Liu, Donghuan Lu, Stephanie Mok, Lydia Ng, Thuc Nghi Nguyen, Qiang Ouyang, Jintao Pan, Elise Shen, Yuanyuan Song, Susan M. Sunkin, Bosiljka Tasic, Matthew B. Veldman, Wayne Wakeman, Wan Wan, Peng Wang, Quanxin Wang, Tao Wang, Yaping Wang, Feng Xiong, Wei Xiong, Wenjie Xu, Min Ye, Lulu Yin, Yang Yu, Jia Yuan, Jing Yuan, Zhixi Yun, Shaoqun Zeng, Shichen Zhang, Sujun Zhao, Zijun Zhao, Zhi Zhou, Z. Josh Huang, Luke Esposito, Michael J. Hawrylycz, Staci A. Sorensen, X. William Yang, Yefeng Zheng, Zhongze Gu, Wei Xie, Christof Koch, Qingming Luo, Julie A. Harris, Yun Wang, and Hongkui Zeng. Morphological diversity of single neurons in molecularly defined cell types. *Nature*, 598(7879):174–181, October 2021.

[RCA+15] Srikanth Ramaswamy, Jean-Denis Courcol, Marwan Abdellah, Stanislaw R. Adaszewski, Nicolas Antille, Selim Arsever, Guy Atenekeng, Ahmet Bilgili, Yury Brukau, Athanassia Chalimourda, Giuseppe Chindemi, Fabien Delalondre, Raphael Dumusc, Stefan Eilemann, Michael Emiel

Gevaert, Padraig Gleeson, Joe W. Graham, Juan B. Hernando, Lida Kanari, Yury Katkov, Daniel Keller, James G. King, Rajnish Ranjan, Michael W. Reimann, Christian Rössert, Ying Shi, Julian C. Shillcock, Martin Telefont, Werner Van Geit, Jafet Villafranca Diaz, Richard Walker, Yun Wang, Stefano M. Zaninetta, Javier DeFelipe, Sean L. Hill, Jeffrey Muller, Idan Segev, Felix Schürmann, Eilif B. Muller, and Henry Markram. The neocortical microcircuit collaboration portal: a resource for rat somatosensory cortex. *Frontiers in Neural Circuits*, 9:44, October 2015.

[rea] Semantic ui react. `https://react.semantic-ui.com/`. Accessed: 2024-09-10.

[RGD+22] Ladislav Rampášek, Mikhail Galkin, Vijay Dwivedi, Anh Luu, Guy Wolf, and Dominique Beaini. *Recipe for a General, Powerful, Scalable Graph Transformer*. Curran Associates Inc., May 2022.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[RSL+22] Agapi Rissaki, Bruno Scarone, David Liu, Aditeya Pandey, Brennan Klein, Tina Eliassi-Rad, and Michelle A. Borkin. BiaScope: Visual unfairness diagnosis for graph embeddings. *2022 IEEE Visualization in Data Science (VDS)*, pages 27–36, October 2022.

[SCHT09] Stephan Saalfeld, Albert Cardona, Volker Hartenstein, and Pavel Tomančák. CATMAID: collaborative annotation toolkit for massive amounts of image data. *Bioinformatics*, 25(15):1984–1986, 2009.

[scia] Clustering. `https://scikit-learn.org/stable/modules/clustering.html#`. Accessed: 2023-04-12.

[scib] Scikit ari. `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html`. Accessed: 2023-10-12.

[SDS21] Dominik Seidel, Yonten Dorji, and Bernhard Schuldt. New insights into tree architecture from mobile laser scanning and geometry analysis [dataset]. *Dryad*, 2021.

[SESM+20] Sharmishtaa Seshamani, Leila Elabbady, Casey Schneider-Mizell, Gayathri Mahalingam, Sven Dorkenwald, Agnes Bodor, Thomas Macrina, Daniel Bumbarger, Joann Buchanan, Marc Takeno, Wenjing Yin, Derrick Brittain, Russel Torres, Daniel Kapner, Kisuk Lee, Ran Lu, Jinpeng Wu, Nuno daCosta, Clay Reid, and Forrest Collman. *Automated Neuron Shape Analysis from Electron Microscopy*. May 2020.

[Shn96] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, September 1996. ISSN: 1049-2615.

[Shn99] Ben Shneiderman. Dynamic queries for visual information seeking. In *Readings in information visualization: using vision to think*, pages 236–243. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, January 1999.

[SK19] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, July 2019.

[SK23] Imrus Salehin and Dae-Ki Kang. A review on dropout regularization approaches for deep neural networks within the scholarly domain. *Electronics*, 12(14):3106, January 2023.

[SKB+21] Federico Scala, Dmitry Kobak, Matteo Bernabucci, Yves Bernaerts, Cathryn René Cadwell, Jesus Ramon Castro, Leonard Hartmanis, Xiaolong Jiang, Sophie Laturnus, Elanine Miranda, Shalaka Mulherkar, Zheng Huan Tan, Zizhen Yao, Hongkui Zeng, Rickard Sandberg, Philipp Berens, and Andreas S. Tolias. Phenotypic variation of transcriptomic cell types in mouse motor cortex. *Nature*, 598(7879):144–150, 2021.

[SMS09] Adnan Salman, Allen Malony, and Matthew Sottile. An open domain-extensible environment for simulation-based scientific investigation (ODESSI). In *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, volume 5544 of *Lecture Notes in Computer Science*, pages 23–32. Springer, May 2009.

[SPA08] 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.

[STN+16] Daniel Smilkov, Nikhil Thorat, Charles Nicholson, Emily Reif, Fernanda Viégas, and Martin Wattenberg. Embedding Projector: Interactive visualization and interpretation of embeddings. *NIPS 2016 Workshop on Interpretable Machine Learning in Complex Systems*, November 2016.

[SUV18] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana, 2018. Association for Computational Linguistics.

[swa] Swagger. `https://swagger.io/`. Accessed: 2024-09-10.

[swc] Swc documentation. https://neuroinformatics.nl/swcPlus/. Accessed: 2023-04-12.

[SWP22] Venkatesh Sivaraman, Yiwei Wu, and Adam Perer. Emblaze: Illuminating machine learning representations through interactive comparison of embedding spaces. *27th International Conference on Intelligent User Interfaces*, pages 418–432, March 2022.

[VCL+18] Petar Veličković, Arantxa Casanova, Pietro Liò, Guillem Cucurull, Adriana Romero, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR*. OpenReview.net, 2018.

[VG20] Anupriya Vysala and Joseph Gomes. Evaluating and validating cluster results. *9th International Conference on Data Mining Knowledge Management Process*, page 47, 2020.

[VMS+18] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. *Programmatically Interpretable Reinforcement Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5052–5061. PMLR, 2018.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, Vichy Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[WBF+19] Johan Winnubst, Erhan Bas, Tiago A. Ferreira, Zhuhao Wu, Michael N. Economo, Patrick Edson, Ben J. Arthur, Christopher Bruns, Konrad Rokicki, David Schauder, Donald J. Olbris, Sean D. Murphy, David G. Ackerman, Cameron Arshadi, Perry Baldwin, Regina Blake, Ahmad Elsayed, Mashtura Hasan, Daniel Ramirez, Bruno Dos Santos, Monet Weldon, Amina Zafar, Joshua T. Dudman, Charles R. Gerfen, Adam W. Hantman, Wyatt Korff, Scott M. Sternson, Nelson Spruston, Karel Svoboda, and Jayaram Chandrashekar. 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.e13, September 2019.

[WGYS18] Junpeng Wang, Liang Gou, Hao Yang, and Han-Wei Shen. GANViz: A visual analytics approach to understand the adversarial game. *IEEE Transactions on Visualization and Computer Graphics*, March 2018.

[WHLE21] Marissa Weis, Laura Hansel, Timo Lüddecke, and Alexander Ecker. Self-supervised representation learning of neuronal morphologies. *arXiv*, December 2021.

[WPB+23] Michael Winding, Benjamin D. Pedigo, Christopher L. Barnes, Heather G. Patsolic, Youngser Park, Tom Kazimiers, Akira Fushiki, Ingrid V. Andrade, Avinash Khandelwal, Javier Valdes-Aleman, Feng Li, Nadine Randel, Elizabeth Barsotti, Ana Correia, Richard D. Fetter, Volker Hartenstein, Carey E. Priebe, Joshua T. Vogelstein, Albert Cardona, and Marta Zlatic. The connectome of an insect brain. *Science*, 379(6636):eadd9330, March 2023.

[WPC+21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[WPLE23] Marissa A. Weis, Laura Pede, Timo Lüddecke, and Alexander S Ecker. Self-supervised graph representation learning for neuronal morphologies. *Transactions on Machine Learning Research*, 2023.

[WT04] Pak Chung Wong and Jim Thomas. Visual analytics. *IEEE Computer Graphics and Applications*, 24(5):20–21, September 2004.

[YCL+21] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 28877–28888, 2021.

[YCS+20] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[YJK+19] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. Graph transformer networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 1073, pages 11983–11993. Curran Associates Inc., Red Hook, NY, USA, December 2019.

[YXL+22] Yalong Yang, Wenyu Xia, Fritz Lekschas, Carolina Nobre, Robert Krüger, and Hanspeter Pfister. The pattern is in the details: An evaluation of

interaction techniques for locating, searching, and contextualizing details in multivariate matrix visualizations. In Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, David A. Shamma, Steven Mark Drucker, Julie R. Williamson, and Koji Yatani, editors, *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022*, pages 84:1–84:15. ACM, 2022.

[ZCAW17] Luisa M. Zintgraf, Taco S. Cohen, Tameem Adel, and Max Welling. *Visualizing Deep Neural Network Decisions: Prediction Difference Analysis*. 5th International Conference on Learning Representations, ICLR, 2017.

[ZCH+20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, January 2020.

[ZL22] Yiqun Zhang and Houbiao Li. A new distance measurement and its application in k-means algorithm. Technical report, June 2022. arXiv:2206.05215 [cs, math] type: article.

[ZLP+18] Zhihao Zheng, J. Scott Lauritzen, Eric Perlman, Camenzind G. Robinson, Matthew Nichols, Daniel Milkie, Omar Torrens, John Price, Corey B. Fisher, Nadiya Sharifi, Steven A. Calle-Schuler, Lucia Kmecova, Iqbal J. Ali, Bill Karsh, Eric T. Trautman, John A. Bogovic, Philipp Hanslovsky, Gregory S. X. E. Jefferis, Michael Kazhdan, Khaled Khairy, Stephan Saalfeld, Richard D. Fetter, and Davi D. Bock. A complete electron microscopy volume of the brain of adult drosophila melanogaster. *Cell*, 174(3):730–743.e22, July 2018.

[ZZSX20] Jiawei Zhang, Haopeng Zhang, Li Sun, and Congying Xia. Graph-Bert: Only attention is needed for learning graph representations. *arXiv*, January 2020.