ABSTRACT

Title of Dissertation: Efficient Machine Learning Techniques for Neural Decoding Systems

Xiaomin Wu

Dissertation Directed by: Professor Shuvra S. Bhattacharyya (Chair/Advisor)
Dept. of Electrical and Computer Engr., and
Institute for Advanced Computer Studies

Professor Rong Chen (Co-Chair/Co-Advisor)
Department of Diagnostic Radiology
and Nuclear Medicine,
University of Maryland School of Medicine

In this thesis, we explore efficient machine learning techniques for calcium imaging based neural decoding in two directions: first, techniques for pruning neural network models to reduce computational complexity and memory cost while retaining high accuracy; second, new techniques for converting graph-based input into low-dimensional vector form, which can be processed more efficiently by conventional neural network models.

Neural decoding is an important step in connecting brain activity to behavior — e.g., to predict movement based on acquired neural signals. Important application areas for neural decoding include brain-machine interfaces and neuromodulation. For application areas such as these, real-time processing of neural signals is important as well as high quality information extraction from the signals. Calcium imaging is a modality that is of increasing interest for studying brain activity. Miniature calcium imaging is a neuroimaging modality that can observe cells in behaving animals with high spatial and

temporal resolution, and with the capability to provide chronic imaging. Compared to alternative modalities, calcium imaging has potential to enable improved neural decoding accuracy. However, processing calcium images in real-time is a challenging task as it involves multiple time-consuming stages: neuron detection, motion correction, and signal extraction. Traditional neural decoding methods, such as those based on Wiener and Kalman filters, are fast; however, they are outperformed in terms of accuracy by recently-developed deep neural network (DNN) models. While DNNs provide improved accuracy, they involve high computational complexity, which exacerbates the challenge of real-time processing. Addressing the challenges of high-accuracy, real-time, DNN-based neural decoding is the central objective of this research.

As a first step in addressing these challenges, we have developed the NeuroGRS system. NeuroGRS is designed to explore design spaces for compact DNN models and optimize the computational complexity of the models subject to accuracy constraints. GRS, which stands for Greedy inter-layer order with Random Selection of intra-layer units, is an algorithm that we have developed for deriving compact DNN structures. We have demonstrated the effectiveness of GRS to transform DNN models into more compact forms that significantly reduce processing and storage complexity while retaining high accuracy.

While NeuroGRS provides useful new capabilities for deriving compact DNN models subject to accuracy constraints, the approach has a significant limitation in the context of neural decoding. This limitation is its lack of scalability to large DNNs. Large DNNs arise naturally in neural decoding applications when the brain model under investigation involves a large number of neurons. As the size of the input DNN increases,

NeuroGRS becomes prohibitively expensive in terms of computational time. To address this limitation, we have performed a detailed experimental analysis of how pruned solutions evolve as GRS operates, and we have used insights from this analysis to develop a new DNN pruning algorithm called Jump GRS (JGRS). JGRS maintains similar levels of model quality — in terms of predictive accuracy — as GRS while operating much more efficiently and being able to handle much larger DNNs under reasonable amounts of time and reasonable computational resources. Jump GRS incorporates a mechanism that bypasses ("jumps over") validation and retraining during carefully-selected iterations of the pruning process. We demonstrate the advantages and improved scalability of JGRS compared to GRS through extensive experiments in the context of DNNs for neural decoding.

We have also developed methods for raising the level of abstraction in the signal representation used for calcium imaging analysis. As a central part of this work, we invented the WGEVIA (Weighted Graph Embedding with Vertex Identity Awareness) algorithm, which enables DNN-based processing of neuron activity that is represented in the form of microcircuits. In contrast to traditional representations of neural signals, which involve spiking signals, a microcircuit representation is a graphical representation. Each vertex in a microcircuit corresponds to a neuron, and each edge carries a weight that captures information about firing relationships between the neurons associated with the vertices that are incident to the edge. Our experiments demonstrate that WGEVIA is effective at extracting information from microcircuits. Moreover, raising the level of abstraction to microcircuit analysis has the potential to enable more powerful signal extraction under limited processing time and resources.

Efficient Machine Learning Techniques for Neural Decoding Systems

by

Xiaomin Wu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Rong Chen, Co-Chair/Co-Advisor
Professor Manoj Franklin
Professor Behtash Babadi
Professor Aravind Srinivasan, Dean's Representative

Dedication

To my family. To my father Guanghai Wu, my mother Jiehong Liu, and my grandparents.
To my girlfriend Lingling Wang.
To professors at SBU. To Professor Wei Lin. To Professor Kenneth L. Short. To Professor Dmitri Donetski.
To professors at UMD. To Professor Rama Chellappa. To Professor Donald Yeung. To Professor Rajeev Barua. To Professor Bruce Jacob. To Professor Romel Gomez.
To Amazon ADS team. To my mentors John Malik, Qianli Chen, Hanbo Sun, Jian Gao. To my managers Zheng Du, Cheng Cao.
To my friend Shuoyang, Chaoze, Yicheng, Yang Zhu, Renke, Yuege, Po-Wei, Shin Choi, Roy, Angela, Layal, Zixuan, Haotian, Cong, Qian, Shuangqi, Pan, Zuyang and all other friends.
To my high school teachers: Ms. Wang, Ms. Qin, Ms. Yang, Mr. Huang, Mr. Weng, Mr. He, Mr. Yao, Mr. Zhu, Mr. Qiu

Acknowledgements

Many people provide kind support and help for the completion of this dissertation.

First, I would like to express my most profound appreciation to my research advisor Professor Shuvra Bhattacharyya, and my co-advisor Professor Rong Chen, for their support and guidance throughout the journey of my Ph.D. study.

I want to extend my sincere thanks to my dissertation committee members. I am very appreciative of Professor Behtash Babadi, Professor Manoj Franklin, and Professor Aravind Srinivasan for serving on my committee and providing encouragement, support, insightful questions, and valuable feedback on my dissertation.

I would also like to thank my lab mates Jing Xie, Yaesop Lee, Zhenjia Ding, and Dr. Eung-Joo Lee for their help and insightful discussions. And also great thank to Dr. Jiahao Wu, Dr. Kyunghun Lee, Dr. Jing Geng for guiding me during different periods of my Ph.D.

Last but not least, I thank my family for their support during my study.

This work was supported by the NIH NINDS (R01NS110421) and the BRAIN Initiative.

Table of Contents

Chapter 1

Introduction

This research thesis explores efficient machine learning techniques for neural decoding systems, including new machine learning algorithms, techniques for efficient real-time extraction of information from neural signals, and techniques for handling different levels of abstraction in the representation of neural signals.

Computational neuroscience is a multidisciplinary area where computer engineering and mathematical modeling contribute to the understanding of brain function. Neural decoding refers to the extraction of information from electrical signals in the brain. For example, neural decoding is used to predict movements based on activity in the motor cortex [1, 2], decisions based on activity in the prefrontal and parietal cortices [3, 4], and spatial locations based on activity in the hippocampus [5, 6].

Neural decoding is a central tool in neural engineering and for neural data analysis. Thus, improving the performance of neural decoding algorithms allows neuroscientists to better understand the information contained in neural activity and can help to advance engineering and biomedical applications, such as brain–machine interfaces and treatments for brain disorders. The specific context of neural decoding that we study in our experiments involves extracting information about an animal's intent to move (i.e., predicting its future motion). Although we study neural decoding in this specific context, we anticipate that many of the insights and techniques developed in the proposed research have broader

applicability and can be transferred or adapted to other neural decoding contexts.

Machine learning methods, such as deep learning, have been developing rapidly during recent years. Modern machine learning tools have the potential to significantly improve neural decoding performance compared to traditional neural decoding techniques [7]. Machine learning methods such as deep neural networks and ensemble models significantly outperform traditional approaches, such as Wiener and Kalman filters [8] on various signal and information processing applications. An important theme in this thesis is the development of novel machine learning methods for neural decoding.

A particular class of machine learning models that we focus on in this thesis is the class of neural network models. Neural network prediction models, especially deep neural network (DNN) models, are becoming increasingly popular in many application fields because of their potential for high accuracy, and their capability for exploiting large datasets when such data is available for model training. DNNs form an important class of machine learning methods that learn complex function approximations from input/output examples [9]. In recent years, DNNs have demonstrated accuracy levels that are close to or even higher than human-beings in fields such as computer vision [10], natural language processing [11], and speech recognition [12]. However, the model sizes associated with state-of-the-art DNN models have been increasing at a rapid pace. Large DNN models require very high computational power, which makes them unsuitable for real-time or resource-constrained applications of neural decoding, such as those associated with closed-loop neuromodulation systems and with experimental platforms that are designed to work with moderately-priced, commodity hardware. Additionally, the presence of redundant information in large DNN models may make it more difficult for scientists and biomedical

engineers to understand and interpret predictive models that utilize neural signals.

In this research, we focus on two important directions for improving the effectiveness of machine learning methods for neural decoding. First, we are interested in making machine learning for neural decoding more efficient so that more sophisticated algorithms can be applied in real-time and so that real-time neural decoding can be deployed in a wider variety of devices, including portable and low-cost devices. Second, we investigate methods for design and optimization of machine learning models so as to enhance the accuracy of information extraction from neural signals at different levels of abstraction. Our joint investigation of efficiency and accuracy aspects leads to significantly improved trade-offs in neural decoding implementations, such as trade-offs among accuracy, real-time performance, and memory requirements.

The capability for real-time neural decoding enables new applications that are not possible with offline neural decoding systems. For example, real-time neural decoding can give experimental neuroscientists better ability to focus analysis on the functional area of the brain that is most closely related to the observed activity behavior. Real-time neural decoding also enables more powerful and user-friendly brain-machine interfaces that can react in a timely manner in relation to the monitored brain state. However, real-time DNN-based neural decoding is challenging because modern DNN systems typically involve massive numbers of parameters and are computationally very intensive. The majority of the parameters typically come from the weight matrices of the models.

To help address the challenges of real-time, DNN-based system design for neural decoding, we apply dataflow-based design methodologies for signal and information processing. Dataflow modeling is widely used for design and implementation of signal

and information processing systems under complex constraints [13]. When dataflow is used as an abstraction for signal processing system design, applications are represented as directed graphs, called dataflow graphs [14]. Vertices in dataflow graphs, called actors, represent basic functional units, which often can be reused across a variety of application graphs, such as graphs for neuron detection, motion correction, and signal stabilization. Each edge in a dataflow graph represents a first-in, first-out (FIFO) buffer that stores data as it passes from the output of one actor to the input of another. Each unit of data within such a buffer is referred to as a token.

A dataflow actor executes as a sequence of *firings*, where each firing can be viewed as a discrete unit or quantum of the actor's computation. A firing can be executed when an actor has sufficient data, where this notion of sufficiency is defined precisely as part of the design of the actor. The actual time at which different actors execute is determined by an implementation subsystem called a *scheduler*, which is not part of the dataflow graph model. This separation of concerns between functional specification (provided by the dataflow graph) and dispatching of component executions (provided by the scheduler) is an important feature of dataflow-based design processes. For more details on the form of dataflow that we apply in this work, we refer the reader to [15, 13].

To improve the effectiveness of machine learning methods for neural decoding and to gain more insight into the neural decoding process, we explore different representations of neural signals. The input representation to machine learning classifiers is important and can impact how much information the classifier can extract from the input data, as well as the efficiency of the information extraction process. The representations that we study can be viewed as a hierarchy of increasing levels of abstraction: raw signals from calcium

imaging; neuron spiking signals, which are derived by processing calcium imaging signals to extract discrete neuron firings; and microcircuits, which capture firings of individual neurons as well as firing relationships among different neurons in the form of a graph.

The remainder of this thesis document is summarized as follows. In Chapter 2, we present a new algorithm, called Greedy inter-layer order with Random Selection (GRS), for pruning DNNs and demonstrate the effectiveness of GRS in significantly enhancing the efficiency of neural decoding with little or no degradation in information extraction accuracy. Here, by pruning a DNN we mean removing selected parameters from the network so that their costs in terms of storage and computation can be eliminated. The parameters to remove (prune) are selected carefully to ensure that the removals do not lead to any significant degradation in accuracy. GRS is a form of pruning called structured pruning, which involves the removal of regular patterns of network connections rather than removal of arbitrary collections of connections. Structured pruning approaches are useful because they can be exploited to achieve more efficient implementations without any need for specialized hardware support or software libraries. In addition to presenting the GRS algorithm, Chapter 2 presents NeuroGRS, which is a software tool for applying GRS with a high degree of automation. NeuroGRS provides a novel platform for experimentation with streamlined DNN implementations of behavior prediction based on neural activity data.

In Chapter 3, we extend the GRS approach to efficiently handle large neural networks, which arise naturally in the analysis of signals acquired from large collections of biological neurons. For example, recent advances in calcium imaging can observe up to a million neurons. Large DNN models are useful for effectively extracting information from large-

scale neural signals. Moreover, DNN pruning methods are important for such large DNN models when such models must be executed in real-time or in resource-constrained environments. While GRS is capable of deriving compact DNN models with low accuracy degradation, the algorithm requires intensive retraining and validation of intermediate DNNs throughout the pruning process. This characteristic makes it infeasible to apply GRS on large DNN models within a reasonable time budget on typical computational platforms that are used in laboratory environments. We have performed a careful experimental analysis of GRS to discover and understand distinct phases of operation as the pruning process progresses. We have applied insights from this analysis to develop a new algorithm, called Jump-GRS (JGRS), that is designed to retain key benefits of GRS while operating much more efficiently and being able to handle large DNN models. We have demonstrated the novel capabilities of JGRS on several relevant datasets related to neural decoding.

In Chapter 4, we address the problem of converting microcircuit representations of neural signals into low-dimensional vector representations that are more suitable for machine learning analysis than applying the analysis directly to the microcircuits. The general problem of converting graph representations into low-dimensional vector form is referred to as graph embedding. The main contribution presented in Chapter 4 is a novel graph embedding framework, called Weighted Graph Embedding with Vertex Identity Awareness (WGEVIA), and extensive experimental results that demonstrate the utility of WGEVIA for neural decoding. In this chapter, we also introduce a dataset, called the five vertices dataset, that helps in assessing how well graph embedding methods are suited to analysis using microcircuit methods.

Chapter 2

Generating Compact Neural Networks for Neural Decoding

2.1 Chapter Overview

In this chapter, we develop methods for efficient and accurate information extraction from calcium-imaging-based neural signals. The particular form of information extraction we investigate involves predicting behavior variables linked to animals from which the calcium imaging signals are acquired. More specifically, we develop algorithms to systematically generate compact deep neural network (DNN) models for accurate and efficient calcium-imaging-based predictive modeling. We also develop a software tool, called NeuroGRS, to apply the proposed methods for compact DNN derivation with a high degree of automation. GRS stands for Greedy inter-layer order with Random Selection of intra-layer units, which describes the central algorithm developed in this work for deriving compact DNN structures. Through extensive experiments using NeuroGRS and calcium imaging data, we demonstrate that our methods enable highly streamlined information extraction from calcium images of the brain with minimal loss in accuracy compared to much more computationally expensive approaches.

Material in this chapter was published in partial, preliminary form in [16].

2.2 Introduction

Predictive modeling based on calcium imaging data plays an important role in optic causality discovery based on optogenetics and calcium imaging. In this chapter, we are concerned with the efficient and accurate utilization of calcium images to predict behavior variables linked to animals. While the methods developed in the chapter are relevant to a wide variety of prediction scenarios, we demonstrate them concretely to predict the motion of a target mouse. The specific prediction problem involves predicting among the states of remaining stationary, moving relatively slowly (*fine motion*) or moving fast (*coarse motion*).

Real-time performance on resource-constrained hardware platforms is very important for such prediction problems. The capability to derive accurate predictions in real-time allows scientists to dynamically adjust experiment parameters based on observations during an experiment. Additionally, real-time performance is critical when prediction is used as part of a closed-loop (feedback) system, such as in a device for neuromodulation [17]. In such a system, real-time prediction is essential to ensure the timeliness of control functions, such as neural stimulation. In practical systems for real-time, calcium-image-based prediction, hardware resource constraints must often be considered as well. Tools for calcium-imaging-based experiments are more cost-effective and more easily customized when they operate on commodity computers, such as desktops or laptops, as opposed to high end servers. Connections to cloud-based servers involve large communication latencies, which interfere with real time performance. In scenarios where prediction is deployed in biomedical devices (as in the neuromodulation example above), size, cost, and

power consumption constraints may severely limit the computational resources that are available.

In this chapter, we develop new methods to transform large DNN models into effective *compact DNN models*, which require much less resources for real-time computation and much less memory compared to the corresponding original models, while achieving similar prediction accuracy. The transformation process that we investigate in this chapter — from large to compact DNN form — is referred to as *pruning*. In addition to facilitating real-time, resource-constrained prediction, the ability of compact (pruned) models to identify the most important connections between artificial neurons enhances scientists' understanding of brain function by more clearly showing relationships between neuron activities and behavior variables. Here and throughout the rest of this chapter, we use *artificial neuron* to refer to a neuron within a DNN and *neuron* (without the "artificial" qualifier) to refer to a neuron in the brain. Also in the context of pruning, we refer to the DNN that is input to the pruning process as the *original* network, and the resulting smaller network as the *pruned* network.

We develop a new pruning approach called GRS, which stands for Greedy inter-layer order with Random Selection of intra-layer units. GRS is a form of pruning called *structured pruning*, where connections are removed from the original network according to regular patterns rather than at arbitrary points in the network (e.g., see [18]). Structured pruning is easier to exploit for efficient implementation compared to unstructured pruning approaches, and unlike unstructured pruning, it requires no additional cost in terms of hardware or specialized libraries to efficiently exploit [18]. We show that our proposed GRS approach can be used in isolation or in combination with additional network transformation

stages that are based on unstructured pruning.

We refer to our combination of GRS with unstructured pruning, as described above, as GRS+TQ, where "TQ" represents the specific sequence of two unstructured pruning stages that we have applied in this work to post-process the network derived by GRS. Here "TQ" stands for Threshold and Quantize. The first unstructured pruning stage uses a thresholding approach to iteratively remove low-weight connections, while the second unstructured pruning stage performs quantization to reduce the number of distinct weight values that need to be maintained in memory.

This work is among the first to systematically generate compact DNN models for accurate and efficient calcium-imaging-based predictive modeling. The pruning approaches that we develop, GRS and GRS+TQ, are applicable across different types of DNN models, which further strengthens their utility in design space exploration for real-time, resource-constrained applications of calcium image analysis. We demonstrate this generality by evaluating GRS and GRS+TQ on two different types of DNN models — multilayer perceptron (MLP) models and convolutional neural network (CNN) models. Intuitively, GRS is well suited to implementation using off-the-shelf processors and neural network libraries, whereas GRS+TQ provides the potential for further improvement through specialized hardware/software for handling the irregular computations resulting form unstructured pruning.

Building on our proposed pruning methods, we develop a software tool to apply the methods with a high degree of automation. Our software tool, called *NeuroGRS*, provides a novel platform for experimentation with streamlined DNN implementations of behavior prediction based on neural activity data. Through extensive experiments using

NeuroGRS and calcium imaging data, we demonstrate that GRS and GRS+TQ lead to highly streamlined information extraction from calcium images of the brain with minimal loss in accuracy compared to much more computationally expensive approaches.

2.3 Related Work

Li et al. apply calcium images of the mouse brain to a DNN model to predict forelimb reach results [19]. Due to their direct use of calcium images as input to the DNN, they use a highly complex model, ResNet18 [20], which is composed of 18 layers. In contrast, we demonstrate in this chapter that by operating on neural activity signals, we can apply much more compact and efficient DNN models while maintaining accurate prediction of mouse motion. The activity signals are derived by preprocessing the input calcium image stream. The datasets that we experiment with in this work include the effect of such preprocessing.

Lee et al. [21] apply online learning with an incremental linear discriminant analysis (LDA) method to predict mouse motion based on neural activity signals that are extracted from calcium images. In this chapter, we apply DNNs as prediction models, which provide higher prediction accuracy, and more stable performance across different datasets, and can be easily used for online learning.

Liu et al. and Frankle/Carbin use carefully designed comparison experiments to demonstrate that for structured pruning methods, training the pruned models from scratch yields similar accuracy to that of the pruned models with the weights that are inherited from the original network [22, 23]. We use this insight to allow our proposed new pruning process to jointly optimize the weights and network architecture (rather than constraining

the network to use the weights of the original network).

Li et al. and Hu et al. show that significant loss of accuracy can result from pruning multiple filters or artificial neurons without retraining between the pruning operations [24, 25]. Motivated by these results, our proposed approach, GRS, applies retraining after each removal of a filter or artificial neuron throughout the pruning process. This approach increases computational cost of the pruning process, but helps to maximize the accuracy of the pruned solutions.

State of the art pruning methods mostly focus on identifying units that provide the least contribution based on pre-trained weights (e.g., see [26, 27, 24, 28, 29, 30, 31, 32]). GRS differs from these methods in that GRS does not restrict itself to the use of pre-trained weights. Instead, as described earlier in this section, GRS retrains the weights after every pruning operation. In our experiments, we include a comparison of GRS with a representative approach — the approach of Li et al. [24] — that is based on using pre-trained weights. Details of this comparison are discussed in Section 2.6.3.

The main novelty of our work is two-fold. First, we introduce a new structured pruning method that involves weight retraining throughout the pruning process. Second, we present the first study, to our knowledge, of DNN pruning for efficient behavior prediction from calcium-imaging based neural signals, and we demonstrate the utility of our proposed new structured pruning approach on this important problem in neural signal analysis. Additionally, we develop a software tool, called NeuroGRS, which combines the proposed structured pruning method with two unstructured pruning techniques. NeuroGRS also provides automated synthesis of C/C++ code for deployment of efficient behavior prediction implementations on resource-constrained platforms.

2.4 Methods

In this section, we present the methods developed in this work, including the NeuroGRS tool, which enables automated application of the proposed pruning methods.

2.4.1 NeuroGRS

Input to NeuroGRS includes a set of n alternative, overparameterized DNN models (*candidates*) $M_1, M_2, \ldots, M_n$. This input is provided by the neuromodulation system designer (user) based on his or her own previous experience with DNN models and off-the-shelf customized models that he or she has access to. Each M_i has an associated *model type* $mtype(M_i)$, which must be either MLP or CNN. Extension of NeuroGRS to support other model types is an interesting direction for future work.

NeuroGRS provides an automated framework that allows the designer to derive an optimal model from among the candidate models. This capability is provided while taking into account the optimized application of structured pruning and also (optionally) further model compression through unstructured pruning.

A dataflow representation of the computational process underlying NeuroGRS is shown in Fig. 2.1. To facilitate experimentation and adaptation to different application requirements, NeuroGRS is developed using modular interfaces and components for the different blocks illustrated in Fig. 2.1. Background on dataflow and details on dataflow modeling in NeuroGRS are discussed in Section 2.4.2 and Section 2.4.3.

NeuroGRS evaluates each candidate M_i individually by training it, applying pruning (GRS or GRS+TQ) to the trained model, and then evaluating the resulting pruned model

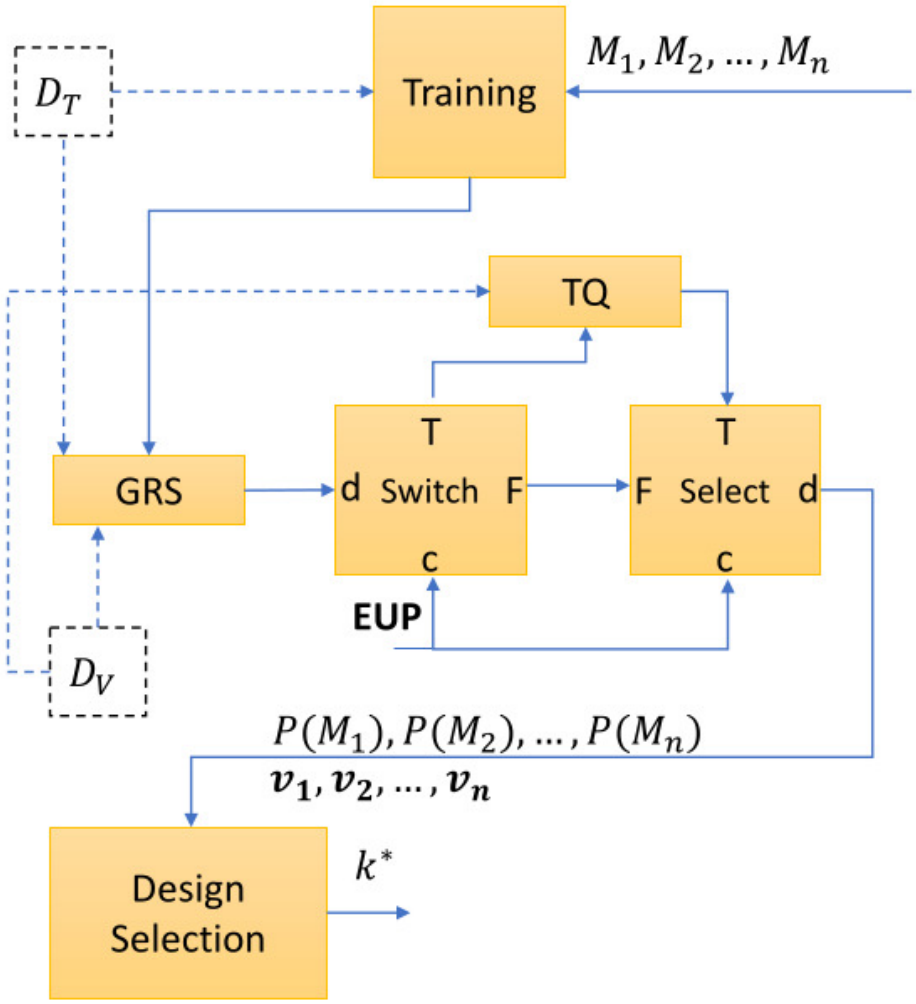

Figure 2.1: A dataflow representation of the computational process underlying NeuroGRS

$P(M_i)$. The training subsystem involved in this process is represented by the block labeled Training in Fig. 2.1, and the pruning subsystem encompasses the blocks labeled TQ, GRS, Switch and Select. The interaction among these four blocks is described in Section 2.4.3 and details on the GRS and TQ pruning processes are presented in Section 2.4.4 and Section 2.4.6, respectively.

As shown in Fig. 2.1, the output from the pruning process is a set of pruned models $P(M_1), P(M_2), \ldots, P(M_n)$, and a set of corresponding vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, where each $P(M_i)$ is the pruned version of M_i and each $\mathbf{v}_i$ encapsulates selected design evaluation

metrics for the pruned model $P(M_i)$. In particular, $\mathbf{v}_i$ is a 3-element vector that gives the measured accuracy, FLOP count, and number of remaining (unpruned) weights in $P(M_i)$.

The blocks with dashed borders in Fig. 2.1 (D_T and D_V) represent datasets that are used for training and pruning, respectively. We refer to these two datasets as the Training Dataset and Validation Dataset, respectively. More details on these datasets and their usage in NeuroGRS are discussed in Section 2.4.4, Section 2.4.6, and Section 2.5.

After all candidates have been pruned and evaluated, an optimized design is selected based on application-specific criteria along with the results $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$. This selection process is represented by the block in Fig. 2.1 labeled Design Selection. The application-specific selection criteria can be configured flexibly by the user — for example, by defining constraints that must be satisfied for a subset of the metrics, and defining optimization priorities for the remaining metrics. Examples of this kind of selection criteria are (a) a constraint on accuracy together with an objective of optimizing (minimizing) a weighted sum of FLOPs and memory cost (model size), and (b) constraints on FLOPs and memory cost together with an objective of optimizing accuracy. When constraints coexist with optimization objectives, the selection process optimizes the specified objectives subject to the given constraints. The flexibility and modularity provided for customizing the design selection processes is important for design of neuromodulation systems, which must satisfy stringent implementation constraints, as described in Section 2.2, in addition to their goal of providing high accuracy prediction capabilities.

To generate the vectors $\{\mathbf{v}_i\}$ at the output of the pruning process, each $P(M_i)$ is executed on each instance in the validation dataset D_V, and the results from these executions are aggregated to derive the corresponding result vector $\mathbf{v}_i$. In the current

version of NeuroGRS, the aggregation process involves simply averaging across all dataset instances. However, this aggregation process can easily be adapted to incorporate more elaborate statistics (e.g., minimum, maximum, and average), and similarly, the Design Selection block can easily be extended adapted to take into account more elaborate design evaluation results.

2.4.2 Dataflow Modeling in NeuroGRS

As mentioned in Section 2.4.1, the system model illustrated in Fig. 2.1 is a dataflow representation. Dataflow is a useful modeling format for signal and information processing systems because it provides well-defined interfaces between functional components, and exposes important forms of high-level application structure that are useful for reliable and efficient implementation in hardware or software [13]. In the form of dataflow that is commonly applied for signal processing system design, an application is represented as a directed graph. Vertices in the graph are called *actors* and represent functional modules, such as digital filters or machine learning classifiers, and edges represent first-in, first-out communication channels that buffer data as it passes from the output of one actor to the input of another. Each unit of data that passes through a dataflow edge is called a *token*. Tokens can have arbitrary types associated with them, such as integers, floating point values, images or video streams.

A dataflow actor executes as a sequence of *firings*, where each firing can be viewed as a discrete unit or quantum of the actor's computation. A firing can be executed when an actor has sufficient data, where this notion of sufficiency is defined precisely as part of

the design of the actor. The actual time at which different actors execute is determined by a subsystem called a *scheduler*, which is not part of the dataflow graph model. This separation of concerns between functional specification (provided by the dataflow graph) and dispatching of component executions (provided by the scheduler) is an important feature of dataflow-based design processes. For more details on the form of dataflow that we apply in this work, we refer the reader to [15, 13].

2.4.3 Pruning Modes

The Switch and Select actors shown in Fig. 2.1 provide conditional execution of the TQ (Threshold and Quantize) actor based on the Boolean-valued system input EUP, which stands for "Enable Unstructured Pruning." This conditional execution provides two alternative modes of pruning, GRS and GRS+TQ, where the mode used can be configured using the EUP input.

The Switch actor has two inputs — labeled c (control) and d (data), and two outputs, labeled T (true) and F (false). The actor consumes a Boolean-valued token on its control input. The Boolean value of the token consumed from the control input indicates which output (T or F) the actor should produce its next output token onto. Upon reading the next token τ from the data input, the token τ is copied onto the output that is indicated by the control input. This process is repeated for each pair of corresponding tokens that arrive at the control and data inputs. Similarly, the Select actor reads a token from its T or F input based on the value of the token arriving on its control input. The token read from T or F is copied onto the data output. More background about dataflow graphs involving Switch

and Select actors can be found in [33].

Since the EUP signal is applied simultaneously as the control input for both the Switch and Select actors in Fig. 2.1, the subgraph involving GRS, TQ, Switch and Select applies the GRS+TQ algorithm only if the EUP signal is true-valued and otherwise, if EUP is false-valued, the subgraph applies only GRS.

2.4.4 Structured Pruning with GRS

GRS is an iterative algorithm that selects and removes a computational unit from the input model on each iteration. The selection process applies a greedy strategy. Here, by a *unit*, we mean a single artificial neuron within a dense layer or a single filter within a convolutional layer. The iterative removal of units is carried out until the accuracy reaches a pre-defined threshold on the maximum acceptable accuracy loss or the size of each network layer reaches a pre-defined threshold on the minimum number of units in the layer.

Algorithm 1 gives a pseudocode description of the GRS algorithm. The algorithm takes as input a DNN model M_{GRS}, training dataset D_T, validation dataset D_V, minimum layer size constraint vector $MinStruct$, and accuracy drop tolerance $\mathcal{T}$. The output is a pruned version $\mathcal{M}_{GRS}$ of M_{GRS} along with the accuracy Acc determined from evaluating the model on D_V; the number of FLOPs in $\mathcal{M}_{GRS}$; and the number of parameters (model size) in $\mathcal{M}_{GRS}$. The inputs $\mathcal{T}$ and $MinStruct$ have been omitted in Fig. 2.1 to avoid excessive clutter in the diagram.

The vector $MinStruct$ has positive-integer-valued elements and is indexed by the

Algorithm 1: A pseudocode sketch of the GRS algorithm.

Input : M_{GRS}: input DNN model.
D_T: training dataset.
D_V: validation dataset.
$MinStruct$: minimum number units to retain in any given layer.
$\mathcal{T}$: tolerance of accuracy drop.

Output : $\mathcal{M}_{GRS}$: pruned DNN model that approximates M_{GRS}.
Acc: accuracy of $\mathcal{M}_{GRS}$.
N_{FLOPs}: the number of FLOPs in $\mathcal{M}_{GRS}$.
N_{params}: the number of parameters in $\mathcal{M}_{GRS}$.

```
OriValAcc ← validation(M_GRS,D_V)
pruningM ← M_GRS
lastValAcc_GRS ← OriValAcc
maxValAcc ← OriValAcc
while maxValAcc >= T × OriValAcc do
    ValAccs ← []
    IntermediateStructures ← []
    foreach λ in pruningM.hiddenLayers do
        if λ.UnitCount > MinStruct[λ] then
            IMS ← randomCutOneUnit(pruningM,λ)
            fineTuning(IMS,D_T)
            ValAccs.append(validation(IMS,D_V))
            IntermediateStructures.append(IMS)
    end
    if ValAccs == [] then
        maxValAcc ← 0
    else
        maxValAcc = max(ValAccs)
    if maxValAcc >= T * OriValAcc then
        lastValAcc_GRS ← maxValAcc
        index ← argmax(ValAccs)
        pruningM ← IntermediateStructures[index]
end
M_GRS ← pruningM
N_FLOPs ← M_GRS.FLOPCount()
N_params ← M_GRS.ParamCount()
return M_GRS, Acc, N_FLOPs, N_params
```

hidden layers in M_{GRS}. Each vector element $MinStruct[i]$ gives the minimum number of units to retain in hidden layer i of the model throughout the pruning process. The $MinStruct$ input therefore gives the user a means for controlling the maximum amount to

which each layer can be pruned. The accuracy drop tolerance $\mathcal{T}$ is a real value in $(0, 1]$. The pruning process in GRS is constrained so that the accuracy is not allowed to fall below $\mathcal{T} \times OriValAcc$, where $OriValAcc$ is the accuracy of the input (unpruned) model M_{GRS} when evaluated on D_V. Hold-out validation is used to assess the accuracy of the intermediate model in each iteration of the pruning process. We use hold-out validation instead of k-fold cross-validation to reduce computational cost since the validation process must be performed repeatedly, one or more times per iteration.

The symbol [] represents an empty list. For a given hidden layer λ within a given model M, $\lambda.UnitCount$ represents the number of units in λ. Given a nonempty list L of real numbers, $argmax(L)$ gives the index of an element of the list that has maximum value (ties are broken arbitrarily).

Function $validation$ takes a DNN model and a dataset as arguments, evaluates the model with the dataset, and returns the average accuracy of the model across the dataset. The function $randomCutOneUnit$ takes a DNN model and layer of the model as its arguments, randomly selects one unit from the layer, removes the selected unit, and returns the modified (smaller) model. Function $fineTuning$ takes a DNN model and a dataset as arguments, and retrains the model using the dataset.

GRS finishes when either (a) the minimum unit counts imposed by $MinStruct$ are reached for all hidden layers or (b) all candidate pruning operations reduce the accuracy below $\mathcal{T} \times OriValAcc$. In our experiments, we use $\mathcal{T} = 0.985$.

2.4.5 Baseline Models for GRS

To help demonstrate the effectiveness of GRS, we compare GRS with two baseline methods in Section 2.6.3. The first baseline method was presented by Li et al. [24], while the second method is developed for experimentation purposes as part of our work on NeuroGRS. More specifically, the second method is a variation of GRS that replaces the greedy inter-layer traversal order with a random inter-layer traversal. We refer to these methods, respectively, as NWM (Natural inter-layer order and Weight Magnitude based selection of intra-layer unit to prune), and RRS (Random inter-layer order and Random Selection of intra-layer unit).

NWM was introduced as a pruning method that focuses on filters in convolutional layers. In our experiments, we used an extended version of NWM that also prunes artificial nodes in dense layers. NWM removes a preset number of units with relatively small sum of absolute weight magnitudes from each layer. The removal is performed by traversing the network layer by layer, starting at the input side of the network and traversing towards the output side. We refer to this as the *natural* inter-layer traversal order. In the remainder of this proposal document, when we write "NWM", we refer to our extended version of the method by Li et al. [24], unless otherwise stated.

We use RRS as a baseline in our experiments to help validate and quantify the utility of optimizing the inter-layer traversal order during the structured pruning process of GRS. Liu et al. show that the pruned structure is of special importance [22]. Since removal of units in different layers results in different intermediate structures during the pruning process, the greedy inter-layer order selection of GRS is motivated by the objective of

deriving more favorable intermediate structures. For example, unsuitable intermediate structures may cause an early stop of the overall structured pruning process, especially when the accuracy drop tolerance $\mathcal{T}$ is strict.

2.4.6 Unstructured Pruning Extensions with TQ

Pruning Stage T applies the concept of pruning weights that have relatively low magnitudes. This concept has been presented previously in the literature (e.g., see [31]), and our contribution here is to integrate the concept into the NeuroGRS framework as an optional follow-on to the GRS Pruning Stage, and with a subsequent refinement that provides further optimization through weight quantization (Stage Q) [32].

Pruning Stage T in NeuroGRS seeks to reduce the FLOP count and weight count of a model by removing weights that have relatively small absolute values, and that contribute relatively little to the forward propagation computation of the given DNN model. We sometimes refer to this stage more concisely as *Stage T*. Like GRS, Stage T is an iterative algorithm. At each iteration all weights whose magnitudes fall below a threshold $Thresh$ are temporarily removed. If the model resulting from the removals has sufficient accuracy, then the temporary removals are made permanent, and the threshold $Thresh$ for the next iteration is increased. The iteration continues until the temporary weight removals result in an accuracy level that falls below the minimum accuracy constraint. As with GRS, hold-out validation is used to assess the accuracy of the intermediate model in each iteration of the pruning process.

Algorithm 2 gives a pseudocode description of Stage T. Input to Stage T consists

of an input DNN model M_T, validation dataset D_V, accuracy drop tolerance $\mathcal{T}$, initial setting for $Thresh$, and constant amount $ThreshStep$ by which to increase $Thresh$ before transitioning to a new algorithm iteration. As in Algorithm 1, $\mathcal{T}$ is a real value in $(0, 1]$, and the pruning process in Stage T is constrained so that the accuracy is not allowed to fall below a factor of $\mathcal{T}$ times the accuracy of the input model (M_T). The $\mathcal{T}$ value provided to Stage T need not be the same as that provided to Algorithm GRS.

Algorithm 2: A pseudocode sketch of Pruning Stage T.

Input : M_T: input DNN model.
D_V: validation dataset.
$\mathcal{T}$: tolerance of accuracy drop.
$ThreshInit$: initial threshold for cutting weights.
$ThreshStep$: amount of threshold increase at each step.

Output : $\mathcal{M}_T$: pruned DNN model that approximates M_T.
Acc: accuracy of $\mathcal{M}_T$.
N_{FLOPs}: the number of FLOPs in $\mathcal{M}_T$.
N_{params}: the number of parameters in $\mathcal{M}_T$.

```
ValAcc ← validation(M_T, D_V)
Acc ← ValAcc
minAcc ← T × ValAcc
M_T ← M_T
Thresh ← ThreshInit
while ValAcc >= minAcc do
    cutWeights(M_T, Thres)
    ValAcc ← validation(M_T, D_V)
    if ValAcc >= minAcc then
        Acc ← ValAcc
        Thres ← Thres + ThreshStep
    else
        undoCuts(M_T)
end
N_FLOPs ← M_T.FLOPCount()
N_params ← M_T.ParamCount()
return M_T, Acc, N_FLOPs, N_params
```

Function $validation$ operates as specified in Section 2.4.4. Function $cutWeights$ takes a DNN model and a threshold value as arguments. The function removes all weights

from the model whose magnitudes are smaller than the threshold. The function keeps a record of the weights that it removes in a given call to the function while discarding any weight removal records associated with the previous call. Function $undoCuts$ reinserts the weights that were removed in the most recent call to $cutWeights$.

Stage T calls function $undoCuts$ only if it is found from Function $validation$ that the most recent weight removals have caused the accuracy to become unacceptably low. The weight removal process (iteration) of Pruning Stage T terminates after the first call to $undoCuts$ completes.

In our experiments with Stage T, we use $\mathcal{T} = 0.995$, $Thresh = 0.001$, and $ThreshStep = 0.001$.

Pruning Stage Q applies the concept of quantizing weights, which can result in smaller memory requirements and faster operations when specialized software libraries or hardware is used [32]. As illustrated in Fig. 2.1, Pruning Stage Q is applied after Pruning Stage T whenever the EUP (Enable Unstructured Pruning) input to NeuroGRS is true-valued.

Stage Q takes as input a DNN model M_Q, a validation dataset D_V, an accuracy drop tolerance $\mathcal{T}$, and an initial maximum number of decimal places n_d to control the rounding. Stage Q starts by rounding all weights in M_Q to n_d decimal places. If the resulting, weight-rounded model has sufficient accuracy (determined using D_V), then the number of decimal places to retain is decreased by 1 (to $(n_d - 1)$). This process is iteratively repeated until the minimum accuracy constraint is crossed, at which point the model from the previous iteration is output as the final weight-rounded model. For brevity, we omit a pseudocode sketch of Stage Q. In our experiments with Stage Q, we use $n_d = 4$

and $\mathcal{T} = 0.990$.

Exploiting the unstructured pruning result of TQ in general requires a specialized software library or specialized hardware that is capable of exploiting irregular, sparse DNN computations. The model output by the TQ block can be applied to platforms that are equipped with such specialized software or hardware. However, experimentation using such specialized hardware/software is beyond the scope of this proposal. Instead, we assess the cost (in accuracy) and potential benefit (in computation and memory savings) resulting from TQ by measuring the accuracy drop, reduction in FLOPs, and reduction in DNN parameters, respectively.

2.4.7 Synthesis of Real-time Implementations

From the optimized model produced by the Design Selection block in Fig. 2.1, NeuroGRS automatically generates a C/C++ implementation that implements the model. Such synthesized implementations are useful for experimenting with or deploying the models in real-time, resource-constrained operational scenarios, where the efficiency of the inference code is critical. The automated C/C++ code synthesis feature of NeuroGRS is developed by applying capabilities of the DSPCAD Framework, which provides tools for prototyping and experimenting with dataflow-based design flows for signal and information processing systems [34]. Here, DSPCAD stands (in reverse order) for computer-aided design (CAD) for digital signal processing (DSP) systems. For details on the DSPCAD Framework, which provides an important foundation for NeuroGRS's code synthesis capability, we refer the reader to [34].

2.5 Datasets

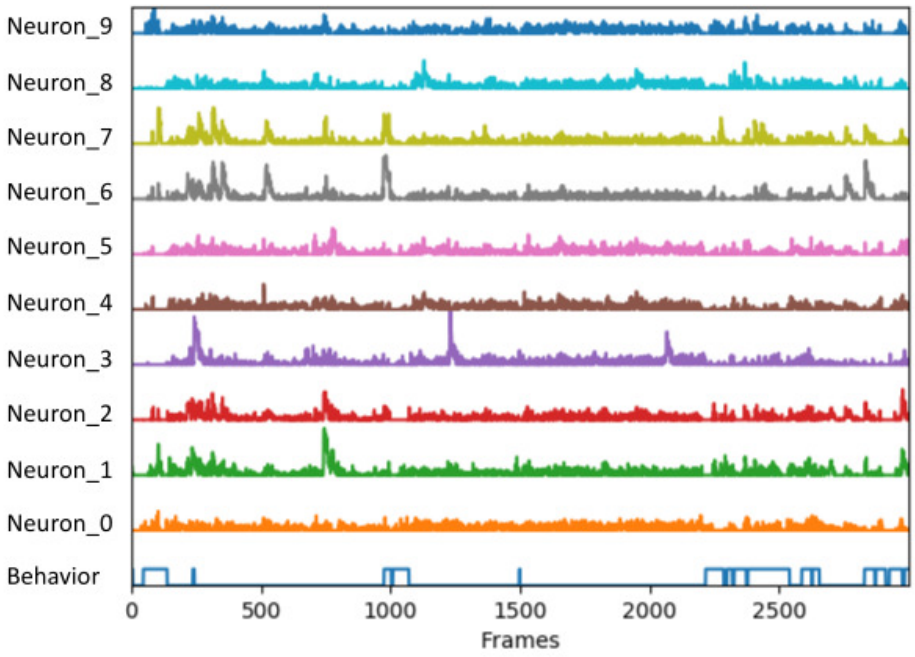

Figure 2.2: Example of input neural signals and behavior labels.

NeuroGRS involves calcium imaging data that was generated in an open-field experiment. The experiment was focused on studying general locomotion activity levels of a group of mice [35]. NeuroGRS takes extracted neural signals from calcium imaging as inputs. Examples of input neural signals (neurons 0 to 9) and behavior labels are shown in Fig. 2.2. Neuron signals were extracted using the method described in [35]. Each dataset corresponds to a single calcium imaging session. In this work, we used 9 datasets from the first three imaging sessions (e1, e2, e3) on three different mice, identified as mouse 04 (m04), mouse 05 (m05) and mouse 06 (m06). The datasets associated with these three mice have calcium imaging traces involving 273, 140, and 114 neurons, respectively.

Each dataset includes 3000 samples, where each sample corresponds to a given

instant in time t during the experiment, and has the form of a vector v_t that is indexed by the neurons extracted from the corresponding calcium imaging trace. For a given neuron n, $v_t[n]$ gives the measured signal value associated with n at time t. The label associated with a sample v_t indicates whether or not the mouse is engaged in fine motion at time t. In this context, fine motion is defined to mean that the mouse is moving at a speed $s(t)$, where $0.2\mathrm{cm/s} < s(t) < 2\mathrm{cm/s}$). The label value "1" indicates that the mouse is engaged in fine motion, whereas "0" indicates that the mouse does not exhibit fine motion. The sampling interval used across the 3000 samples in each dataset is 100 milliseconds (10 Hz sample rate).

In our experiments, we apply NeuroGRS independently to each of the 9 datasets described above. To apply NeuroGRS to a given dataset, we first partition the dataset into three independent portions, which results in three smaller datasets. These smaller datasets are referred to as D_S, D_V, and D_E. The dataset D_S is used as a starting point for constructing the training dataset D_T, and the dataset D_V is used for validation during pruning, as described in Section 2.4.4 and Section 2.4.6. The third dataset D_E (the subscript "E" stands for "evaluation") is used for testing. The dataset D_E is not used in the process of pruning and design selection; it is only used to evaluate performance for the pruned models that are derived by NeuroGRS. The size ratio used when partitioning each dataset is $|D_S| : |D_V| : |D_E| = 8 : 1 : 1$, where $|d|$ denotes the number of samples in dataset d.

The original datasets involving mouse locomotion are imbalanced in that the fine motion class (label 1) is represented less frequently compared to class 0: the minimum, maximum, and average percentage of label 1 among the 9 datasets are 13%, 45.3%, and

26.1%, respectively. Bias caused by imbalanced datasets during training can lead machine learning algorithms to ignore or excessively suppress the minority class. To minimize such bias, balanced data is extracted from the D_S set associated with each dataset when deriving the associated training dataset D_T. We evaluate and test model performance using the original distribution of a given dataset; that is, we do not modify the imbalance ratios of D_V and D_E.

A random undersampling method is used to select a proper of subset $D_0 \subset D_S$ of the original samples from D_S having label 0 such that $|D_0| = |D_1|$, where D_1 is the set of samples in D_S having label 1. The training dataset D_T, as represented in Fig. 2.1, is then derived as $D_T = D_0 \cup D_1$. The datasets D_V and D_E are taken from the original datasets without application of undersampling.

As mentioned in Section 2.4.1, NeuroGRS can operate on MLP models as well as CNN models. Vectors of neuron signals are applied directly as inputs to MLP models in NeuroGRS. On the other hand, for CNN models, neuron signals are first rearranged into a 2D square matrix Γ using a spatial arrangement algorithm. One objective of the rearrangement is to keep the input small to reduce computational requirements. Another objective is to associate spatial relationships in the neural signals according to neuron position information that is available in the datasets.

In the input rearrangement algorithm for CNN models, neurons are first sorted in increasing order according to the sum $x(\mu) + y(\mu)$, where $(x(\mu), y(\mu)$ gives the spatial coordinates associated with a given neuron μ. Γ is an $n \times n$ matrix, where $n = 1$ initially, and n is increased as the algorithm operates until all of the neurons have been inserted into Γ. Each neuron occupies a distinct matrix element within Γ. At the end of the insertion

process, any vacant elements in Γ have the value 0.

Algorithm 3 sketches the procedure used to transfer neurons from the sorted list L of $(x(\mu) + y(\mu))$ values into Γ, while progressively increasing the size of Γ as needed. The function $createMatrix$ creates a 1×1 matrix containing a single, zero-valued element. The function pop removes and returns the first element from a given list. The function $extendMatrix$ takes as arguments the matrix Γ along with the current matrix size n. The function enlarges Γ to be an $(n+1) \times (n+1)$ matrix by adding a row of zeros as the new topmost row, and a column of zeros as the new rightmost column. The data in the lower left $n \times n$ submatrix of the enlarged version of Γ is unchanged. Indexing of matrix rows and columns in Algorithm 3 starts at 0.

Algorithm 3: The algorithm used for rearranging CNN input in NeuroGRS.

Input : L: sorted list of neurons (increasing order).
Output : Γ: rearranged matrix.

```
Γ ← createMatrix()
n ← 1
while L.isNotEmpty() do
    for i in range(0 : n) do
        if L.isNotEmpty() then
            Γ[0][i] ← L.pop()
        if i != n − 1 then
            if L.isNotEmpty() then
                Γ[n − 1 − i][n − 1] ← L.pop()
    end
    Γ ← extendMatrix(Γ, n)
    n ← n + 1
end
return Γ
```

2.6 Experiments and Results

We experiment with four different overparameterized models as the inputs $\{M_i\}$ to NeuroGRS. These include two MLP models nn1 and nn2, and two CNN models cnn1 and cnn2. The structure of each initial model is summarized in Fig. 2.3. The difference between nn1 and nn2 or cnn1 and cnn2 involves the number of layers. We studied different numbers of layers in this context to explore how the number of layers impacts the pruning results derived by NeuroGRS, and to demonstrate that NeuroGRS can effectively prune models with different numbers of layers. In Section 2.6.2, we report on pruning experiments that demonstrate the effectiveness of NeuroGRS in deriving compact DNN models for calcium image based prediction. In Section 2.6.3, we perform comparison experiments between Algorithm GRS, the structured pruning stage of NeuroGRS, and the two baseline methods, NWM and RRS, discussed in Section 4.4.1.

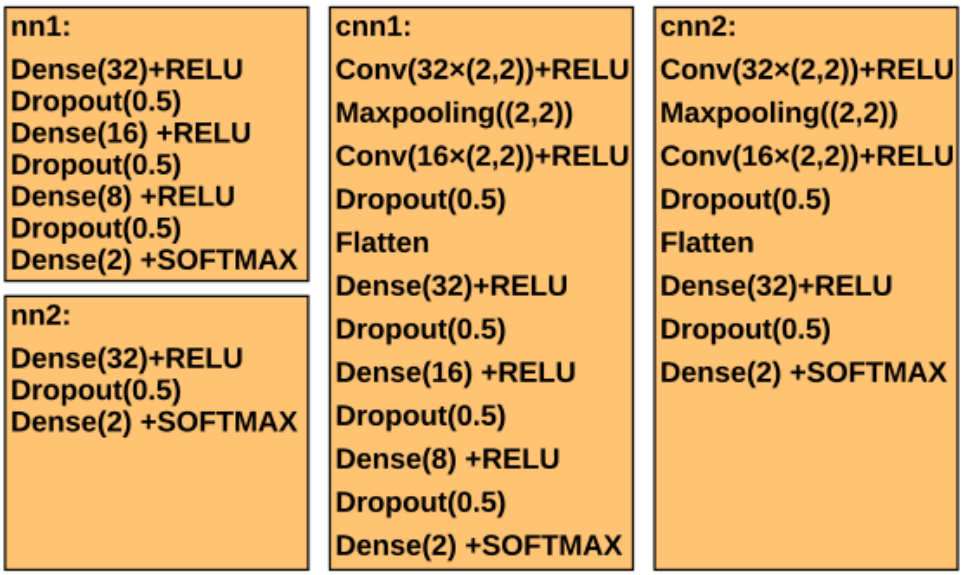

Figure 2.3: Structure of input DNN models.

The models $\{M_i\}$ can be viewed as models that are representative of what neuro-

modulation system designers would apply as input to a tool such as NeuroGRS. We design the models $\{M_i\}$ for the experiments conducted in this research because, to the best of our knowledge, there are no publicly available, baseline DNN models that are suitable for real-time calcium-imaging-based behavior prediction on resource-constrained platforms. Popular, state-of-the-art DNN models in the literature, such as ResNet or VGGNet, focus on very different kinds of classification problems on inputs of much larger dimensionality. In our neural signal processing context, these models do not match well with our objective of deriving streamlined models from calcium imaging signals.

Thus, to investigate how different structures are handled by NeuroGRS, we develop four moderately-sized MLP models and CNN models as our initial models. These initial models, $\{M_i\}$ were carefully designed to reach acceptable prediction accuracy on our datasets without excessive model complexity. These models are suitable for evaluating the effectiveness of NeuroGRS since they are designed from the beginning with compactness as a design objective. Further improvements in compactness provided by NeuroGRS are achieved on top of compactness properties that are inherent through the careful design of the initial models.

In Section 2.6.5, we apply the inference model synthesis capabilities described in Section 2.4.7, and we experiment with the synthesized models on two resource-constrained platforms, including a Raspberry Pi platform. Whereas the inference experiments in Section 2.6.2 and Section 2.6.3 are performed using Keras [36], the experiments in Section 2.6.5 are performed using models derived from the synthesis capabilities of NeuroGRS.

2.6.1 Common Experiment Parameters

This section briefly summarizes settings that are common to the experiments presented in Section 2.6.2 and Section 2.6.3. All of the training and retraining used in NeuroGRS uses the Adam optimizer [37]. For training, the batch size is set to 32, and the number of epochs is 150 for each of the four input models nn1, nn2, cnn1, and cnn2. The number of epochs is set to 50 when retraining intermediate models during the pruning process. A decay ratio of 0.95 is applied to the initial dropout ratio of 0.5 at each retraining iteration as the model is shrinking. The results presented for each data set / model combination are averaged across 10 independent trials.

As mentioned in Section 2.4, accuracy drop tolerances (ADTs) of GRS, Pruning Stage T, and Stage Q are set to $\mathcal{T}_{GRS} = 0.985$, $\mathcal{T}_T = 0.995$, and $\mathcal{T}_Q = 0.990$, respectively. These settings limit the overall accuracy drop in NeuroGRS to $\mathcal{T}_{GRS} \times \mathcal{T}_T \times \mathcal{T}_Q = 0.970$ of the original accuracy $ACC_{original}$. We have empirically tuned $\mathcal{T}_{GRS}, \mathcal{T}_T$, and $\mathcal{T}_Q$, and found that GRS is more sensitive to the ADT than Stage T and Stage Q. Additionally, we found that Stage Q is more sensitive to the ADT than Stage T. In other words, if we allocate slightly more ADT to GRS than to Stage T and Stage Q, then the result (in terms of the compactness of the pruned model) is typically better than if we decompose the overall ADT equally across all three stages. Decomposing an overall ADT level Z equally in this context means assigning an ADT of $Z^{1/3}$ to each of the three stages.

Based on the trends that we observed empirically, we provided slightly more ADT to GRS compared to Stage Q, and slightly more ADT to Stage Q compared to Stage T. For our application, we estimated that an overall ADT of 0.97 is acceptable, meaning that the

accuracy of the pruned model should have an accuracy no less than $0.97 \times ACC_{original}$. For other applications, the three ADT parameters can be tuned based on specific runtime and accuracy requirements. As described above, the tuning process was performed empirically in our study — i.e., we tuned the ADTs by iterating through a series of experiments. Systematic optimization of the parameters $\mathcal{T}_{GRS}$, $\mathcal{T}_T$, and $\mathcal{T}_Q$ is an interesting direction for future work.

2.6.2 Pruning Experiments

Pruning experiments with the GRS Algorithm and GRS+TQ combination are performed on nn1, nn2, cnn1, and cnn2 to obtain compact versions of these models while maintaining the prediction accuracy within a specified minimum value. As mentioned in Section 2.4.4, the accuracy tolerances for Stages GRS, T, and Q were set to $\mathcal{T} = 0.985, 0.995, 0.990$, respectively. The results for nn1, nn2, cnn1 and cnn2, are summarized in Table 2.1, Table 2.2, Table 2.3, and Table 2.4, respectively.

For each of the four input models, results are provided for the 9 datasets described in Section 2.5. The columns labeled Acc_S, FLOPs_S, and Params_S provide the accuracy, FLOP count and number of parameters after the GRS stage of NeuroGRS. Here, the "_S" suffix stands for "Structured pruning". The parenthetic term "(loss %)" gives the loss in accuracy compared to the original, unpruned model. This accuracy loss is measured as $(\alpha_o - \alpha_p)/\alpha_o$, where α_o and α_p represent the accuracy levels of the original and pruned models, respectively. Similarly, the parenthetic term "(% of initial)" gives a measure of the FLOP count or parameter count of the pruned model relative to the original model. For

Table 2.1: Results of pruning experiments with nn1 as the input model.

Dataset	Acc_S (loss%)	FLOPs_S (% of initial)	Params_S (% of initial)
m04e1	0.967 (0.55%)	6425 (34.19%)	3234 (34.21%)
m04e2	0.949 (0.17%)	5068 (26.97%)	2554 (27.01%)
m04e3	0.915 (2.23%)	2743 (14.6%)	1384 (14.64%)
m05e1	0.934 (2.1%)	3392 (32.99%)	1716 (33.01%)
m05e2	0.901 (0.4%)	4461 (43.39%)	2255 (43.37%)
m05e3	0.897 (-0.06%)	5635 (54.8%)	2849 (54.78%)
m06e1	0.937 (1.4%)	3981 (46.19%)	2017 (46.17%)
m06e2	0.907 (2.0%)	3823 (44.36%)	1938 (44.36%)
m06e3	0.938 (0.49%)	3237 (37.56%)	1642 (37.6%)

Dataset	Acc_U (loss%)	FLOPs_U (% of initial)	Params_U (% of initial)
m04e1	0.966 (0.68%)	2803 (14.91%)	202 (2.14%)
m04e2	0.95 (-0.01%)	1968 (10.47%)	393 (4.16%)
m04e3	0.911 (2.61%)	1211 (6.45%)	326 (3.45%)
m05e1	0.929 (2.62%)	2325 (22.61%)	235 (4.53%)
m05e2	0.897 (0.92%)	3152 (30.65%)	484 (9.33%)
m05e3	0.891 (0.58%)	4392 (42.72%)	214 (4.12%)
m06e1	0.928 (2.32%)	3164 (36.72%)	344 (7.9%)
m06e2	0.904 (2.4%)	2918 (33.86%)	396 (9.09%)
m06e3	0.933 (0.95%)	2055 (23.84%)	295 (6.77%)

Table 2.2: Results of pruning experiments with nn2 as the input model.

Dataset	Acc_S (loss%)	FLOPs_S (% of initial)	Params_S (% of initial)
m04e1	0.974 (0.24%)	5894 (33.47%)	2962 (33.51%)
m04e2	0.953 (0.21%)	6224 (35.34%)	3127 (35.38%)
m04e3	0.921 (2.09%)	7049 (40.03%)	3541 (40.06%)
m05e1	0.94 (0.83%)	6086 (66.91%)	3069 (66.94%)
m05e2	0.906 (0.47%)	6370 (70.03%)	3212 (70.06%)
m05e3	0.894 (0.33%)	7023 (77.21%)	3541 (77.23%)
m06e1	0.941 (0.38%)	4881 (65.67%)	2466 (65.71%)
m06e2	0.919 (0.54%)	5414 (72.85%)	2735 (72.88%)
m06e3	0.93 (0.46%)	3813 (51.31%)	1927 (51.36%)

Dataset	Acc_U (loss%)	FLOPs_U (% of initial)	Params_U (% of initial)
m04e1	0.972 (0.44%)	3529 (20.04%)	28 (0.32%)
m04e2	0.952 (0.28%)	2796 (15.88%)	43 (0.49%)
m04e3	0.918 (2.41%)	5025 (28.54%)	55 (0.63%)
m05e1	0.94 (0.83%)	4128 (45.38%)	60 (1.33%)
m05e2	0.903 (0.76%)	4474 (49.18%)	270 (5.91%)
m05e3	0.89 (0.74%)	5399 (59.35%)	276 (6.03%)
m06e1	0.939 (0.56%)	3878 (52.17%)	277 (7.39%)
m06e2	0.918 (0.69%)	4136 (55.65%)	108 (2.89%)
m06e3	0.931 (0.36%)	2647 (35.62%)	66 (1.78%)

Table 2.3: Results of pruning experiments with cnn1 as the input model.

Dataset	Acc_S (loss%)	FLOPs_S (% of initial)	Params_S (% of initial)
m04e1	0.969 (1.09%)	24907 (44.59%)	12504 (44.6%)
m04e2	0.952 (-0.33%)	12863 (23.02%)	6463 (23.06%)
m04e3	0.913 (2.63%)	14041 (25.14%)	7051 (25.15%)
m05e1	0.881 (4.03%)	1677 (7.6%)	862 (7.74%)
m05e2	0.854 (0.62%)	3029 (13.73%)	1545 (13.88%)
m05e3	0.851 (1.58%)	5989 (27.14%)	3034 (27.25%)
m06e1	0.898 (2.65%)	2061 (9.34%)	1057 (9.49%)
m06e2	0.867 (-4.29%)	8318 (37.69%)	4204 (37.75%)
m06e3	0.916 (0.88%)	3149 (14.27%)	1601 (14.38%)

Dataset	Acc_U (loss%)	FLOPs_U (% of initial)	Params_U (% of initial)
m04e1	0.961 (1.97%)	23304 (41.72%)	872 (3.11%)
m04e2	0.945 (0.33%)	11987 (21.46%)	1507 (5.38%)
m04e3	0.906 (3.34%)	13197 (23.62%)	2034 (7.26%)
m05e1	0.887 (3.36%)	1541 (6.98%)	370 (3.33%)
m05e2	0.853 (0.75%)	2533 (11.48%)	505 (4.54%)
m05e3	0.85 (1.73%)	5469 (24.78%)	1480 (13.3%)
m06e1	0.885 (4.0%)	1847 (8.37%)	335 (3.01%)
m06e2	0.858 (-3.26%)	7706 (34.91%)	570 (5.12%)
m06e3	0.913 (1.17%)	2870 (13.01%)	360 (3.24%)

Table 2.4: Results of pruning experiments with cnn2 as the input model.

Dataset	Acc_S (loss%)	FLOPs_S (% of initial)	Params_S (% of initial)
m04e1	0.972 (0.88%)	12154 (22.23%)	6107 (22.28%)
m04e2	0.952 (0.35%)	23644 (43.25%)	11865 (43.28%)
m04e3	0.93 (1.76%)	6567 (12.01%)	3308 (12.07%)
m05e1	0.93 (1.24%)	8283 (39.67%)	4183 (39.76%)
m05e2	0.881 (0.52%)	6231 (29.85%)	3154 (29.98%)
m05e3	0.862 (1.89%)	7743 (37.09%)	3915 (37.21%)
m06e1	0.917 (3.13%)	5035 (24.12%)	2551 (24.25%)
m06e2	0.871 (3.04%)	7453 (35.7%)	3767 (35.8%)
m06e3	0.936 (0.7%)	7938 (38.02%)	4010 (38.12%)

Dataset	Acc_U (loss%)	FLOPs_U (% of initial)	Params_U (% of initial)
m04e1	0.97 (1.05%)	11721 (21.44%)	1375 (5.02%)
m04e2	0.953 (0.22%)	23162 (42.37%)	8572 (31.28%)
m04e3	0.93 (1.78%)	5918 (10.82%)	1310 (4.78%)
m05e1	0.925 (1.79%)	6965 (33.36%)	1748 (16.63%)
m05e2	0.882 (0.44%)	5988 (28.68%)	2341 (22.27%)
m05e3	0.859 (2.27%)	6835 (32.74%)	1105 (10.51%)
m06e1	0.918 (3.05%)	4736 (22.68%)	982 (9.34%)
m06e2	0.866 (3.56%)	6514 (31.2%)	1249 (11.88%)
m06e3	0.937 (0.55%)	7616 (36.48%)	3103 (29.52%)

example, the "% of initial" value of 44.6% for the FLOP count of model nn1 on dataset m04e1 means that the GRS reduced the FLOP count by 55.4% for this model/dataset combination.

Similarly, the columns labeled Acc_U, FLOPs_U, and Params_U provide the accuracy, FLOP count and number of parameters after GRS+TQ; the "_U" suffix indicates that the effects of "Unstructured pruning" are included in these results.

Measurement of FLOP counts and parameter counts are based on the TensorFlow library [38]. Each trial includes an independent execution through the entire dataflow of NeuroGRS, starting with the training of the input model, with the exception that the Design Selection block is disabled. Instead of applying Design Selection, we collect and report the results for all four pairs $(P(Mi), \mathbf{v_i})$ (see Fig. 2.1).

The results in Table 2.1 through Table 2.4 show that NeuroGRS can prune all of the four input models into very compact DNN models without much loss in testing accuracy. Some of the results even show that higher accuracy is achieved by the pruned models compared to the corresponding original models (e.g., see the Acc_S result for Dataset m05e3 in Table 2.1). Such increases in accuracy are indicated by negative "loss %" values in the tables. It has been argued that pruning can sometimes increase accuracy in this way because pruned structures are simpler and help to reduce overfitting [31].

To summarize and compare the effectiveness of NeuroGRS on all four input DNN models on calcium imaging data, the results from Table 2.1 through Table 2.4 are aggregated in Table 2.5, which shows average results for each of the four input models across all 9 datasets. As shown in Table 2.5, all of the pruned models maintain most of the accuracy provided by the corresponding original models. Additionally, the MLP

models (nn1 and nn2) are seen to suit our datasets better with higher testing accuracy, lower parameter counts, and lower FLOP counts. These trends are seen in Table 2.5 for both structured-only (GRS) and structured+unstructured (GRS+TQ) pruning modes. Overall, the results demonstrate the effectiveness of NeuroGRS in consistently providing optimization capability across a diverse set of input models and datasets.

Table 2.5: Summary of pruning experiments over all four input models.

Model	Acc_S (loss%)	FLOPs_S (% of initial)	Params_S (% of initial)
nn1	0.927 (1.03%)	4307 (37.23%)	2177 (37.24%)
nn2	0.931 (0.62%)	5861 (56.98%)	2953 (57.01%)
cnn1	0.9 (0.98%)	8448 (22.5%)	4258 (22.59%)
cnn2	0.917 (1.5%)	9450 (31.33%)	4762 (31.42%)

Model	Acc_U (loss%)	FLOPs_U (% of initial)	Params_U (% of initial)
nn1	0.923 (1.45%)	2665 (24.69%)	321 (5.72%)
nn2	0.929 (0.79%)	4001 (40.2%)	131 (2.97%)
cnn1	0.895 (1.49%)	7828 (20.7%)	893 (5.37%)
cnn2	0.915 (1.63%)	8828 (28.86%)	2421 (15.69%)

2.6.3 Comparison Among GRS, NWM, and RRS

In this section, we present an experimental comparison of GRS with the NWM and RRS methods for structured pruning, which were described as baseline methods in Section 4.4.1. The comparison experiments are performed on all four initial models $\{M_i\}$.

Table 2.6 shows aggregated results for each of the four input models across all 9 datasets. More detailed experimental results for the four input models across these datasets are provided in Section 2.6.4. The abbreviations AL, FCI, and PCI stand, respectively, for

Table 2.6: Summary of comparison experiments over all four input models.

	GRS vs. NWM								
	AL difference			FCI difference			PCI difference		
Model	min	max	avg	min	max	avg	min	max	avg
nn1	-0.10%	2.04%	0.78%	8.13%	60.59%	24.18%	8.33%	60.52%	24.22%
nn2	-0.98%	0.50%	0.00%	-11.54%	15.93%	3.03%	-11.54%	15.91%	3.04%
cnn1	-3.48%	1.62%	-0.25%	3.19%	48.33%	21.91%	3.23%	48.27%	21.91%
cnn2	0.00%	3.51%	1.14%	19.13%	50.99%	34.54%	19.08%	50.89%	34.46%

	GRS vs. RRS								
	AL difference			FCI difference			PCI difference		
Model	min	max	avg	min	max	avg	min	max	avg
nn1	-0.32%	1.93%	0.87%	19.96%	67.49%	41.45%	19.91%	67.43%	41.41%
nn2	-0.50%	0.46%	-0.12%	-5.93%	11.25%	2.39%	-5.92%	11.23%	2.40%
cnn1	-2.83%	2.18%	0.74%	47.72%	79.25%	62.92%	47.72%	79.17%	62.87%
cnn2	0.31%	3.51%	1.39%	43.30%	63.75%	56.87%	43.29%	63.63%	56.82%

Accuracy Loss, FLOP Count Improvement, and Parameter Count Improvement, where the losses and improvements are interpreted with respect to the original (unpruned) model. For example, the AL difference of GRS vs. NWM for a given input model M_i is computed from the nine values defined by:

$$\begin{aligned} &\frac{(\mathrm{Acc\_O}(d)-\mathrm{Acc\_G}(d))}{\mathrm{Acc\_O}(d)} - \frac{(\mathrm{Acc\_O}(d)-\mathrm{Acc\_N}(d))}{\mathrm{Acc\_O}(d)} \\ &= \frac{(\mathrm{Acc\_N}(d)-\mathrm{Acc\_G}(d))}{\mathrm{Acc\_O}(d)} \end{aligned} \tag{2.1}$$

for $d \in D$, where $D = \mathrm{m04e1}, \mathrm{m04e2}, \ldots \mathrm{m06e3}$ denotes our set of nine datasets, and $\mathrm{Acc\_X}(d)$ represents the accuracy measured in our experiments by model X on dataset d for the given input model M_i. Here, $\mathrm{X} = \mathrm{O}$ denotes the original model (without any pruning), while $\mathrm{X} = \mathrm{G}$, $\mathrm{X} = \mathrm{N}$ and $\mathrm{X} = \mathrm{R}$ for the pruned models derived by applying GRS, NWM and RRS, respectively, to the original model. The AL difference values are reported as the maximum, minimum and average of the nine values

$$\left\{ \frac{\mathrm{Acc\_N}(d) - \mathrm{Acc\_G}(d)}{\mathrm{Acc\_O}(d)} \Big| d \in D \right\}. \tag{2.2}$$

The FCI difference and PCI difference values reported in Table 2.6 are calculated by taking Equation 2.2 and replacing each $\mathrm{Acc\_X}(d)$ term with $\mathrm{FLOPs\_X}(d)$ or $\mathrm{Params\_X}(d)$, respectively, for the corresponding model specifier X. For example, the FCI difference statistics for the GRS vs. RRS comparison are computed from the nine values

$$\left\{ \frac{\mathrm{FLOPs\_R}(d) - \mathrm{FLOPs\_G}(d)}{\mathrm{FLOPs\_O}(d)} \Big| d \in D \right\}. \tag{2.3}$$

Note that from the formulation of AL, FIC and PCI differences, as described above, a positive value for the AL difference in Table 2.6 means that GRS performs worse compared to the associated baseline model, while positive values for FCI and PCI indicate that GRS performs better than the associated baseline model.

The results show that for models nn1, cnn1, and cnn2, GRS provides large advantages on average in reducing FLOP count and parameter count compared to both NWM and RRS. One case (GRS vs. NWM for nn2) has no accuracy change and another (GRS vs. NWM for cnn1) shows a slight accuracy improvement with GRS; in all other cases, the average accuracy for GRS is worse than that of NWM or GRS by a small percentage (0.74%–1.39%).

For nn2, the trend is different in that while there is still a reduction in average FCI and PCI, the magnitude of the reduction is small. This is because nn2 contains only one hidden layer, whereas the advantage of GRS stems from its ability to flexibly apply

different inter-layer orderings in the pruning process.

By looking at the two "min" columns for FIC difference and the two "min" columns for PIC difference in Table 2.6, we see that GRS provides highly consistent improvements in FLOP count and Parameter count for models nn1, cnn1, and cnn2 — for each of these three models, improvements in FLOP and parameter counts are delivered by GRS on all 9 datasets.

2.6.4 Additional Experiment Results on Separate Datasets

The results for nn1, nn2, cnn1 and cnn2, are summarized in Table 2.7, Table 2.8, Table 2.9, and Table 2.10, respectively. In each of these tables, the columns labeled Acc_X, FLOPs_X, and Params_X provide the accuracy, FLOP count and number of parameters for the model represented by X. Here, $\mathrm{X} = \mathrm{O}$ to denote the original model (without any pruning), while $\mathrm{X} = \mathrm{G}$, $\mathrm{X} = \mathrm{N}$ and $\mathrm{X} = \mathrm{R}$ for the pruned models derived by applying GRS, NWM and RRS, respectively, to the original model.

2.6.5 Resource-Constrained Inference

To demonstrate the utility of NeuroGRS's structured pruning capabilities for neural signal processing on resource constrained-platforms, we present experiments in this section using a moderate-complexity laptop computer platform and a low-complexity Raspberry Pi platform. Both platforms are used in isolation, without any cloud computing support. These experiments also demonstrate the code synthesis capabilities presented in Section 2.4.7. Our experiments in this section do not employ specialized hardware/software subsystems

Table 2.7: Results of comparison experiments with nn1 as the input model.

Dataset	ACC_O	FLOPs_O	Params_O
m04e1	0.976	9457	18795
m04e2	0.951	9457	18795
m04e3	0.945	9457	18795
m05e1	0.947	5201	10283
m05e2	0.910	5201	10283
m05e3	0.903	5201	10283
m06e1	0.955	4369	8619
m06e2	0.931	4369	8619
m06e3	0.944	4369	8619
Dataset	ACC_G	FLOPs_G	Params_G
m04e1	0.967	6425	3234
m04e2	0.949	5068	2554
m04e3	0.915	2743	1384
m05e1	0.934	3392	1716
m05e2	0.901	4461	2255
m05e3	0.897	5635	2849
m06e1	0.937	3981	2017
m06e2	0.907	3823	1938
m06e3	0.938	3237	1642
Dataset	ACC_N	FLOPs_N	Params_N
m04e1	0.973	10740	5406
m04e2	0.956	16455	8277
m04e3	0.928	5805	2930
m05e1	0.944	6944	3513
m05e2	0.905	6240	3159
m05e3	0.905	7235	3659
m06e1	0.936	4682	2381
m06e2	0.926	5773	2930
m06e3	0.938	4929	2498
Dataset	ACC_R	FLOPs_R	Params_R
m04e1	0.976	15345	7718
m04e2	0.955	17752	8931
m04e3	0.930	11540	5804
m05e1	0.944	8434	4264
m05e2	0.908	7876	3982
m05e3	0.906	7930	4009
m06e1	0.939	5701	2887
m06e2	0.925	7995	4051
m06e3	0.935	6546	3318

Table 2.8: Results of comparison experiments with nn2 as the input model.

Dataset	ACC_O	FLOPs_O	Params_O
m04e1	0.979	17609	8841
m04e2	0.956	17609	8841
m04e3	0.942	17609	8841
m05e1	0.948	9097	4585
m05e2	0.906	9097	4585
m05e3	0.884	9097	4585
m06e1	0.950	7433	3753
m06e2	0.920	7433	3753
m06e3	0.934	7433	3753
Dataset	ACC_G	FLOPs_G	Params_G
m04e1	0.974	5894	2962
m04e2	0.953	6224	3127
m04e3	0.921	7049	3541
m05e1	0.94	6086	3069
m05e2	0.906	6370	3212
m05e3	0.894	7023	3541
m06e1	0.941	4881	2466
m06e2	0.919	5414	2735
m06e3	0.93	3813	1927
Dataset	ACC_N	FLOPs_N	Params_N
m04e1	0.976	8369	4204
m04e2	0.953	7109	3584
m04e3	0.923	6004	3017
m05e1	0.943	7535	3799
m05e2	0.901	5320	2683
m05e3	0.885	7109	3584
m06e1	0.940	4904	2478
m06e2	0.923	5020	2536
m06e3	0.935	4835	2443
Dataset	ACC_R	FLOPs_R	Params_R
m04e1	0.971	6169	3100
m04e2	0.957	6598	3327
m04e3	0.916	7764	3901
m05e1	0.944	7109	3584
m05e2	0.903	5831	2941
m05e3	0.893	7166	3613
m06e1	0.938	4904	2478
m06e2	0.921	5159	2606
m06e3	0.925	4556	2302

Table 2.9: Results of comparison experiments with cnn1 as the input model.

Dataset	ACC_O	FLOPs_O	Params_O
m04e1	0.979	55863	28033
m04e2	0.951	55863	28033
m04e3	0.938	55863	28033
m05e1	0.919	22071	11137
m05e2	0.863	22071	11137
m05e3	0.868	22071	11137
m06e1	0.920	22071	11137
m06e2	0.836	22071	11137
m06e3	0.928	22071	11137
Dataset	ACC_G	FLOPs_G	Params_G
m04e1	0.969	24907	12504
m04e2	0.952	12863	6463
m04e3	0.913	14041	7051
m05e1	0.881	1677	862
m05e2	0.854	3029	1545
m05e3	0.851	5989	3034
m06e1	0.898	2061	1057
m06e2	0.867	8318	4204
m06e3	0.916	3149	1601
Dataset	ACC_N	FLOPs_N	Params_N
m04e1	0.970	44280	22216
m04e2	0.948	31398	15762
m04e3	0.910	35429	17770
m05e1	0.891	3131	1601
m05e2	0.858	5545	2817
m05e3	0.848	8888	4502
m06e1	0.866	3918	1994
m06e2	0.859	9023	4564
m06e3	0.931	13815	6976
Dataset	ACC_R	FLOPs_R	Params_R
m04e1	0.975	51567	25881
m04e2	0.949	50041	25112
m04e3	0.933	43015	21586
m05e1	0.901	13265	6712
m05e2	0.869	19643	9914
m05e3	0.863	21385	10792
m06e1	0.900	17561	8867
m06e2	0.843	20048	10122
m06e3	0.930	20641	10418

Table 2.10: Results of comparison experiments with cnn2 as the input model.

Dataset	ACC_O	FLOPs_O	Params_O
m04e1	0.981	27417	54671
m04e2	0.958	27417	54671
m04e3	0.951	27417	54671
m05e1	0.941	10521	20879
m05e2	0.881	10521	20879
m05e3	0.885	10521	20879
m06e1	0.945	10521	20879
m06e2	0.911	10521	20879
m06e3	0.944	10521	20879
Dataset	ACC_G	FLOPs_G	Params_G
m04e1	0.972	12154	6107
m04e2	0.952	23644	11865
m04e3	0.93	6567	3308
m05e1	0.93	8283	4183
m05e2	0.881	6231	3154
m05e3	0.862	7743	3915
m06e1	0.917	5035	2551
m06e2	0.871	7453	3767
m06e3	0.936	7938	4010
Dataset	ACC_N	FLOPs_N	Params_N
m04e1	0.973	23613	11846
m04e2	0.952	43549	21836
m04e3	0.940	29147	14615
m05e1	0.949	13251	6682
m05e2	0.888	14642	7375
m05e3	0.870	16381	8250
m06e1	0.926	9029	4558
m06e2	0.903	18100	9121
m06e3	0.945	15590	7857
Dataset	ACC_R	FLOPs_R	Params_R
m04e1	0.975	46777	23465
m04e2	0.956	47316	23735
m04e3	0.938	40842	20494
m05e1	0.937	18103	9127
m05e2	0.894	19541	9849
m05e3	0.881	19635	9896
m06e1	0.937	18275	9213
m06e2	0.903	19933	10046
m06e3	0.945	18716	9436

that are needed to exploit unstructured pruning. Therefore, we focus only on applying GRS Pruning in these experiments and disable the TQ part of the NeuroGRS dataflow.

First, we employ a laptop computer platform, based on a single, moderately-priced device — the Intel Core i7 7700HQ CPU. The experiment does not apply cloud computing or high-end servers, which are often not appropriate platforms for real-time neural signal processing systems, as discussed in Section 2.2. On the targeted CPU platform, we execute pruned model implementations that are generated using NeuroGRS together with its code synthesis capabilities. For comparison, we also implement the corresponding original model for execution on the targeted device. This implementation is generated with NeuroGRS's code synthesis capability using an option that bypasses the entire pruning process, and just generates code for the given input model.

The execution time results reported in this section only pertain to the inference code that is synthesized by NeuroGRS, not to the execution of the pruning algorithms and code synthesis within NeuroGRS. The implementations that are evaluated in this section are single-threaded since our focus is to isolate the benefit provided by the proposed pruning methods. Application of the proposed methods to multi-threaded implementations is an interesting direction for further study.

We report only the runtime associated with inference computations, and omit startup overhead that is required to initialize the model. In particular, we give results on the average runtime required to perform a prediction operation on a single input image frame once the system has been initialized to process an arbitrary number of frames. The resulting runtime measurements can be used to assess the runtime cost of prediction in relation to the frame rate of 10Hz, which is associated with our collection of datasets (see Section 2.5).

The results on runtime evaluation are shown in Table 2.11. The results are tabulated for the original model and the pruned model that results from GRS. Each value represents the average measured runtime to process a single input image frame, as described above. Each experiment is executed on all nine datasets. For each dataset, GRS is executed 10 times to derive 10 pruned models. Recall that randomization is applied within the GRS algorithm, which means that the solution derived by GRS is not unique. To account for this variation, we average across 10 different pruning solutions for each dataset. Runtime is measured by executing each pruned model on each dataset 100 times, and averaging the resulting $9 \times 10 \times 100 = 9000$ measurements to derive the corresponding value shown in Table 2.11. For the original model, we average across the same number of executions (9000) for consistency, even though the original model has no difference in structure among the 10 independent executions for each dataset.

Table 2.11: Results of inference experiments over all four input models for laptop computer platform.

	Inference runtime		
Model	Original model (ms)	Pruned model (ms)	Improvement
nn1	0.048	0.027	43.36%
nn2	0.044	0.029	34.28%
cnn1	0.608	0.183	69.81%
cnn2	0.656	0.209	68.08%

Note that the magnitudes of the improvements in Table 2.11 are generally different from the improvements in FLOPs that are reported in Section 2.6.2 and Section 2.6.3. This is because the runtime includes overhead in executing the model, such as communication overhead between the DNN layers, which is not captured by the FLOP count assessments

in Section 2.6.2 and Section 2.6.3. Nevertheless, the results in Table 2.11 represent large improvements delivered by GRS.

Next, we employ a Raspberry Pi platform (Raspberry Pi Zero W V1.1). The experimental setup is identical to that used for the laptop-targeted experiments, as described above, except that the target platform is different. For example, we again average across 9000 measurements to derive each reported value. The results are shown in Table 2.12. As with the laptop computer platform, the results show that NeuroGRS provides large improvements. The improvements for the Raspberry Pi are slightly to moderately larger than those reported for the laptop platform, depending on the input DNN model.

With a 10Hz frame rate, the runtime of the original models on the laptop are already well within the frame interval (0.1 sec), and the absolute reductions provided by pruning are negligible in comparison to the frame interval, although they are significant in a relative sense. The impact of runtime reduction is more significant on the Raspberry Pi, as shown in Table 2.12, especially for larger DNNs, such as cnn1 and cnn2. For these two DNNs, we see improvements of 11.631 ms and 11.108 ms, respectively. The runtime improvement percentages (relative improvement levels) on the laptop and Raspberry Pi are similar.

The runtime improvement provided by NeuroGRS can be significant and useful under any of the following three scenarios: (1) when the application requires higher frame rate; (2) when the application is deployed on a device with lower computational power, such as the Raspberry Pi device that we have experimented with; and (3) when the inference model is much more complex than the models that we have studied in our experiments. Also, considering other neural signal preprocessing steps that are needed for end-to-end application processing, the time reduction of the pruned model derived by

NeuroGRS provides more time for other processing steps, which facilitates the operation of complex end-to-end applications in real-time.

Table 2.12: Results of inference experiments over all four input models for Raspberry Pi platform.

	Inference runtime		
Model	Original model (ms)	Pruned model (ms)	Improvement
nn1	1.203	0.572	52.44%
nn2	1.074	0.666	37.97%
cnn1	16.277	4.646	71.46%
cnn2	16.139	5.031	68.83%

2.7 Chapter Summary

In this chapter, we have introduced a new structured pruning method called Greedy inter-layer order with Random Selection of intra-layer units (GRS) and combined it with two state-of-the-art unstructured pruning methods, which we refer to collectively as TQ. Additionally, we have developed a novel software tool, called NeuroGRS, which neural signal processing system designers can use to apply GRS or GRS+TQ with a high degree of automation. NeuroGRS includes capabilities not only for deriving compact DNN models but also for synthesizing efficient code for deploying those models on resource-constrained platforms. We have demonstrated the effectiveness of GRS, GRS+TQ, and NeuroGRS through extensive experiments involving nine calcium imaging datasets acquired from animal models. Useful directions for future work include extending NeuroGRS to support model types other than convolutional neural networks and multilayer perceptrons, and to exploit graphics processing unit and multicore CPU acceleration capabilities in the

inference target platform.

Chapter 3

Jump-GRS: A Multi-Phase Approach to Structural Pruning of Neural Networks for Neural Decoding

3.1 Chapter Overview

Neural decoding, an important area of neural engineering, helps to link neuron activity to behavior in the outside world. Deep Neural Networks (DNNs), which are becoming increasingly popular in many application fields of machine learning, show promising performance in neural decoding compared to traditional neural decoding methods. Various neural decoding applications, such as Brain Computer Interface (BCI) applications, require both high decoding accuracy and real-time decoding speed. Pruning methods are used to produce compact DNN models for faster computational speed. Greedy inter-layer order with Random Selection (GRS) is a recently-designed structured pruning method that derives compact DNN models for calcium-imaging-based neural decoding. Although GRS has advantages in terms of detailed structure analysis and consideration of both learned information and model structure during the pruning process, the method is very computationally intensive, and is not feasible when large-scale DNN models need to be pruned within typical constraints on time and computational resources. Large-scale DNN models arise in neural decoding when large numbers of neurons are involved. In this chapter, we build on GRS to develop a new structured pruning algorithm called Jump GRS

(JGRS) that is designed to efficiently compress large-scale DNN models. We compare the pruning performance and speed of JGRS and GRS with extensive experiments in the context of neural decoding. Our results demonstrate that JGRS provides significantly faster pruning speed compared to GRS, and at the same time, JGRS provides pruned models that are similarly compact as those generated by GRS.

3.2 Introduction

Deep learning [39] is a powerful tool to learn valuable features from input data, and can be used in a great variety of machine learning tasks. Deep learning models are widely applied in different application areas, including computer vision, natural language processing, and neuroscience.

Multi-layer Perceptrons (MLPs) [40] and Convolutional Neural Networks (CNNs) [41] are two common types of Deep Neural Network (DNN) models. MLPs and CNNs are widely used in classification and prediction tasks with vector- and matrix-format data. The size and structure of MLPs and CNNs generally have large influence on the performance of machine learning tasks on datasets containing massive amounts of information. Models with large sizes and complicated structures often require a long time to train and can overfit the training data. Here, by model size we mean the number of weight parameters of the given network. In many cases, models with large sizes eventually converge and give high-quality predictions on validation data [42]. Models with small sizes and simple structures take less time for training and run faster during inference. However, such models may have trouble extracting large portions of useful information from the given datasets

and therefore may limit prediction performance.

It is common to construct and use an overparameterized model when the model accuracy is the only consideration. However, when a DNN is applied in a time- or resource constrained application, such as a real-time or Internet-of-things (IoT) application, it is often important to derive a compact DNN model, where the model size is streamlined while maintaining sufficient prediction accuracy [43].

Pruning is a general approach to deriving compact models. Pruning involves taking an overparameterized or otherwise less compact model and removing selected weights that are considered to not have significant contribution to the predictive capability of the enclosing network [44]. In this chapter, we develop a novel pruning approach that has important applications in neural decoding and in other neural engineering applications that involve DNN-based analysis of large collections of neurons.

DNNs have grown in popularity in the neural decoding field due in part to their potential for high-accuracy decoding [45, 46]. In traditional DNN application areas, such as computer vision and natural language processing, DNN models for large-scale inputs and high-accuracy performance often have very large size [47].

Some works involving DNNs for neural decoding focus on decoding accuracy (e.g., see [48, 7]). However, for neural decoding applications, the speed of inference is also important — for example, a compact DNN model can help to perform real-time neural decoding in the context of closed-loop neuromodulation [49]. Many methods for pruning DNN models have been introduced in the literature (e.g., see [50, 51, 52, 53, 16]). These methods aim to reduce DNN model size and increase inference speed while minimizing any degradation in inference accuracy. Among previously-developed pruning methods,

NeuroGRS provides an approach that is demonstrated to be particularly well-suited to neural decoding [16]. In the name NeuroGRS, the acronym GRS stands for Greedy inter-layer order with Random Selection, which is the name of the novel pruning algorithm that is introduced as part of NeuroGRS.

Compact DNNs are useful for real-time neural decoding tasks, such as brain computer interfaces [54] and precise neuromodulation [55]. Compact models can reduce the time used for model inference and possibly allow additional runtime to be devoted to other important preprocessing tasks, such as neuron detection and and motion correction [56]. In other words, the efficiency of compact models can help to achieve real-time performance if execution speed is a critical factor, or to improve the effectiveness of other system-tasks if there is some slack in achieving real-time performance.

Recent advances in calcium imaging can observe up to a million neurons. Large DNN models are needed to effectively extract information from large-scale neural signals. Inference of such large-size DNN models is very challenging to perform in real-time without significant reduction in model size. Thus, compact DNN models provide the potential to make better use of state-of-the-art calcium imaging technology for the acquisition of neural signals.

NeuroGRS is a software tool that is geared towards pruning DNN models, such as MLP or CNN models, for neural decoding applications [16]. DNN models often contain parameters that have little or no significance, and without which a DNN model can still achieve similar accuracy performance. A pruned DNN model generally allows faster training and inference compared to the original overparameterized neural network. A pruned DNN can also be significantly more interpretable than its original form. This higher

degree of interpretability can help neuroscientists analyze relationships between specific neurons and behavior prediction decisions produced by neural decoding [57].

In this chapter, we focus on a particular form of pruning called *structured pruning*, which involves the removal of regular patterns of connections from the given network instead of allowing removal from arbitrary points in the network. For more background on structured pruning, we refer the reader to [18]. Results from structured pruning can be exploited in DNN implementations without any need for specialized hardware or software libraries. Without specialized hardware or software support, unstructured pruning approaches may increase the computational complexity of the network, thereby having the opposite of the effect that is usually desired from pruning.

NeuroGRS [16] introduces a new structured pruning algorithm, called Greedy inter-layer order with Random Selection (GRS), and combines GRS with two state-of-the-art unstructured pruning methods [31, 32] together into a unified software tool. The unstructured pruning methods within NeuroGRS can be applied optionally if appropriate hardware support or software libraries are available to exploit the corresponding pruning results. NeuroGRS can prune an overparameterized DNN model into a compact form. The resulting compact form, compared to the original model, has faster training and inference speed and a smaller model size, while providing comparable classification performance in terms of accuracy.

GRS is different from other structured pruning methods in its joint consideration of both trained weights and model structure when generating compact models. However, one limitation is its computationally intensive nature, which makes it unsuitable for large DNNs, such as those arising in the analysis of neural signals from large collections of

neurons. In this chapter, we build on the methods in GRS for joint weight/structure consideration, while developing new techniques to reduce time complexity, and allow for scaling up to large DNNs. We refer to our proposed new pruning algorithm as *Jump GRS* (*JGRS*). JGRS also incorporates parameterization to allow the neural decoding system designer to systematically explore trade-offs among computational efficiency, accuracy impact, and reduction in network complexity (pruning effectiveness) in the pruning process. For example, during system prototyping, the designer may prefer a pruning process that favors computational efficiency (fast pruning), whereas when the system is being finalized for deployment, more emphasis may be placed on reducing the complexity of the pruned network.

JGRS was developed through careful analysis of the operation of GRS. Through this analysis, which included experimenting with GRS and studying intermediate pruning structures produced by the algorithm, we observed that GRS often operates through two distinct phases of operation, which we refer to as the *far phase* and the *near phase*. By an *intermediate structure* during the pruning process of GRS, we mean the network that results from the first i channel removals performed by GRS, where i is some integer that is less than the total number of channel removals performed on the given network. In the *far phase*, intermediate structures are relatively far (different) from the final compact form derived by the algorithm; instead, the intermediate structures are closer to the overparameterized initial network. On the other hand, in the near phase, intermediate structures are closer to the the final form.

In our analysis of GRS operation, we also observed that the model being pruned is generally more sensitive to structural changes in the *near phase* compared to the *far phase*.

Here, by *more sensitive*, we mean that model performance in terms of accuracy tends to be affected more heavily by changes in model structure. We introduce a technique, called the *jump* mechanism in JGRS, during the far phase that exploits the relative insensitivity during this phase to structural changes. Intuitively, the jump mechanism avoids repeated retraining and validation iterations, thereby allowing successive pruning operations to be carried out much more quickly than in the original GRS algorithm. More details on the jump mechanism are presented in Section 3.3.

JGRS can be viewed as a modified version GRS, where the pruning process is decomposed into far and near phases, as described above, and the jump mechanism is integrated into the far phase to significantly accelerate the overall operation of the pruning process. We perform extensive experiments to demonstrate that JGRS provides large reductions in pruning time compared to GRS without significant impact on the prediction accuracy of the final, compact networks that are produced. In addition to presenting the JGRS algorithm and its extensive experimental evaluation, this chapter studies the two-phase phenomena associated with the near and far phases, and provides useful insight that has more general implications to network pruning (beyond development of the JGRS algorithm).

3.3 Methods

In this section, we first analyze important characteristics involved in the operation of the GRS algorithm. Next, based on this analysis, we present a significant modification to GRS that helps to retain the key advantages of GRS while addressing its key weakness, which is

its long runtime, and an associated lack of scalability to large DNN models.

3.3.1 Analysis of the GRS Algorithm

NeuroGRS and the underlying GRS algorithm are developed to generate compact neural network models that accelerate inference speed by eliminating selected weight parameters while maintaining model accuracy within an acceptable range. GRS exhaustively examines intermediate sub-structures of the input over-parameterized neural network. Units are selected as candidates in a randomized manner, and selection among candidates is performed using a greedy strategy. The exhaustive assessment evaluates all possible pruning results in the current step and uses a greedy strategy to select the best result of the current step to carry over into the next iteration. More details about how the GRS algorithm works can be found in [16].

An advantage of such a retraining-intensive approach in GRS is that the approach can accurately assess the accuracy impact of each pruning candidate with a diverse range of candidates considered at each pruning iteration. However, there is a disadvantage in that retraining in each iteration costs a large amount of computational power. As demonstrated in [16], such an investment of computational effort can be beneficial for certain types of information extraction from neural signals where relatively small neural networks are involved. However, the scalability of the GRS technique to large networks is limited.

In our extensive experimentation with GRS, we have observed an interesting pattern in which the pruning process often progresses through two distinct phases of operation. We refer to the first of the regimes as the *far phase* because during this phase, the intermediate

pruned models are relatively far from their final compact form. Similarly, we refer to the second phase, where the model is approaching its final form, as the *near phase*. This phenomenon is demonstrated concretely in Section 3.4.3. Based on these experimental observations, which we have observed consistently, we hypothesize that optimizing the accuracy/efficiency trade-off of pruning will benefit from significantly different approaches in these two phases.

To intuitively describe the hypothesized multi-phase phenomenon in the pruning process, we consider a multi-layer-perceptron (MLP) network as an example. Suppose an MLP model has more than enough representation power with respect to the given accuracy requirement — that is, the model has sufficient connections between successive layers. Intuitively, in that case, a learned informative forward path P can be transferred to other nodes in the model through a sufficient amount of retraining. However, if an MLP model is in a relatively compact form, then the informative path P may be impossible to retain if some critical node is removed. The reduced sensitivity to specific pruning operations in the higher-connectivity case (far phase) opens up the possibility for more computationally efficient strategies. More intensive computational effort can then be devoted to the near phase, a heightened-sensitivity phase, where the effort can be utilized more productively towards the ultimate goal of accuracy-constrained compactness optimization.

3.3.2 JGRS

We formulate the JGRS algorithm as a structured pruning algorithm that can be applied to CNNs or MLPs. Extension to other classes of deep learning models is an interesting

direction for future work. As a structured pruning approach, JGRS removes relatively coarse-grained components from the input DNN (CNN or MLP) as its basic unit of removal. In particular, it removes filters and nodes for CNNs and MLPs, respectively, as its basic unit of removal. Each removal operation in the algorithm involves either a whole filter or a whole node. Henceforth, we refer to these basic units of removal as *removal units* in our discussion of JGRS.

As the JGRS algorithm operates on an input DNN model M, it generates a sequence $I_0, I_1, I_2, \ldots, I_n$ of progressively more compact DNNs, where $I_0 = M$, I_n is the final compact DNN produced by the algorithm, and for each $i = 1, 2, \ldots, n$, I_i is derived from I_{i-1} by removing one or more removal units from I_{i-1}. We refer to I_0 as the input to JGRS, $I_1, I_2, \ldots, I_{n-1}$ as the intermediate models generated by JGRS when applied to I_0, and I_n as the output produced by JGRS corresponding to the input I_0. Since JGRS employs randomization techniques, the output I_n is not unique for a given I_0; the output in general depends on the mechanism used to guide randomized decisions, and how the mechanism is initialized (seeded).

We have designed the JGRS algorithm such that it retains GRS as a core pruning approach, but incorporates along with the approach to exploit the insights described in Section 3.3.1 about far and near phases in the operation of GRS. The design of the algorithm is broken down into far and near phases of operation, where different strategies are used in each phase. The definitions of the far and near phases for the JGRS algorithm are intended to capture the corresponding phases of GRS algorithm operation that are described in Section 3.3.1.

We define the *compression rate*, denoted CR, as the number of removal units that

are eliminated from the network between successive retraining and validation operations. Intuitively, a higher CR leads to faster pruning but has more risk of reducing model accuracy. Based on the discussion in Section 3.3.1, this risk is much lower during the far phase compared to the near phase.

In addition to the model M that is to be pruned, another important input to JGRS is the *minimum unit-count vector* (*MUCV*), denoted μ. The vector μ is a positive-integer vector that is indexed by the hidden layers in M. For each hidden layer λ, $\mu[\lambda]$ specifies a constraint on the minimum number of units (removal units) in λ that must be retained by the pruned solution computed by JGRS. The vector μ thus gives the neural decoding system designer a mechanism to control the maximum degree of pruning that will occur in each layer. Moreover, experimentation with different settings of μ gives the system designer a means for exploring accuracy/compactness trade-offs associated with a given prediction framework for neural decoding. Investigation into automated mechanisms for configuring μ is an interesting direction for future work.

Given a DNN M, an MUCV μ, and a hidden layer λ in M, we define the *pruning midpoint*, denoted $midpoint(\lambda)$, by:

$$midpoint(\lambda) = max(floor(\frac{unitCount(M,\lambda)-\mu(\lambda)}{2}), 1), \qquad (3.1)$$

where $unitCount(M, \lambda)$ denotes the number of removal units contained in hidden layer λ within M.

The far phase of JGRS is decomposed into two sub-phases, called *far subphase 1*

and *far subphase 2*. In far subphase 1, $midpoint(\lambda)$ removal units are removed from each hidden layer before retraining and validation are invoked. Thus, the CR in far phase 1 can be expressed as:

$$\sum_{\lambda=1}^{N_H} midpoint(\lambda), \tag{3.2}$$

where N_H denotes the number of hidden layers in the given DNN.

On the other hand, in far subphase 2, retraining and validation are invoked after each hidden layer λ is processed, where the processing in this context involves removing $midpoint(\lambda)$ removal units. Thus, the CR in far subphase 2 depends on the hidden layer that is being processed, and can be expressed simply as $midpoint(\lambda)$.

In contrast to the two far subphases, where retraining and validation steps are interleaved with batches of pruning operations, the near phase performs retraining and validation after elimination of each selected removal unit. The CR is therefore reduced to 1 in the near phase. This is because of the significantly heightened sensitivity of accuracy to pruning that is characteristic of the near phase.

Table 3.1 summarizes the compression rates in the successive stages through which JGRS operates.

As JGRS progresses through the pruning process, it progressively decreases the CR — that is, it progressively increases the frequency of retraining and validation steps. The bypassing of retraining and validation accelerates operation of the pruning process. The reduced sensitivity in the far phase of accuracy to pruning operations allows us to perform this bypassing without significantly degrading the model's predictive capability.

Table 3.1: Compression rates in different stages of JGRS.

Algorithm stage	CR
Far phase 1	$\sum_{\lambda=1}^{N_H} midpoint(\lambda)$
Far phase 2	$midpoint(\lambda)$
Near phase	1

The jump mechanism, which was introduced in Section 3.2 and is the basis for the name "JGRS", is based on this bypassing of retraining and validation. In the far phase, JGRS effectively "jumps across" groups of removal unit eliminations (pruning operations) before it re-invokes retraining and validation. On the other hand, there is no jumping in the near phase, where retraining and validation are interleaved with individual pruning operations.

Algorithm 4 provides a pseudocode sketch of the portion of JGRS that operates during far phase 1. We refer to this portion of JGRS as the $far\_subphase\_1$ function. This function takes five arguments: the input DNN model M that is to be pruned, a training dataset D_T, a validation dataset D_V, an MUCV μ, and a constraint on the validation accuracy $ValAcc\_min$. The argument $ValAcc\_min$ gives the minimum accuracy, as determined by evaluation on D_V, that is acceptable by a solution produced by $far\_subphase\_1$.

The function $validation(m, d)$ in Algorithm 4 evaluates the input model m on the input dataset d and returns the resulting prediction accuracy. The function $randomCutUnits(m, l, num\_units)$ takes as argument a DNN model m, a layer l within m, and a positive integer num_units. The function randomly removes num_units removal units from the layer l of the model m and returns the modified DNN model. The function $fineTuning(m, d)$ retrains the model m with the input dataset d. The functions

$FLOPCount(m)$ and $ParamCount(m)$ return the number of floating-point operations (FLOPs) and parameters, respectively, in the given model m. In Algorithm 4, IMS stands for "intermediate model structure",

Algorithm 5 provides a pseudocode sketch of the $far\_subphase\_2$ Function, which is the portion of JGRS that operates during far phase 2. The $validation$, $randomCutUnits$, and $fineTuning$ functions are the same as the corresponding functions that are called in Algorithm 4.

The overall JGRS algorithm is shown in Algorithm 6. In addition to the original DNN model M, the training and validation datasets D_T and D_V, the MUCV μ, and an accuracy drop tolerance $\mathcal{T}$, the JGRS algorithm takes three inputs that control the different algorithm stages: $attempt\_stage\_1$, $attempt\_stage\_2$, and $attempt\_stage\_3$. These latter three input arguments control how many attempts are carried out for each pruning stage, where the three successive stages correspond to far phase 1, far phase 2, and the near phase, respectively. By having multiple attempts during each stage, we exploit the randomization inherent in the stage to examine different sets of removal units to eliminate. Using multiple attempts in this manner also helps to reduce bias caused by the underlying randomness.

The accuracy drop tolerance $\mathcal{T}$ can be expressed as

$$\mathcal{T} = \frac{ValAcc_{min}}{OriValAcc}, \tag{3.3}$$

where $ValAcc_{min}$ is the minimum tolerable accuracy after pruning and $OriValAcc$ is the accuracy of the input model M. The accuracy values in this context are interpreted in relation to the validation dataset D_V.

Algorithm 4: *far_subphase_1* Function: A pseudocode sketch of the portion of JGRS that operates during far subphase 1.

Input : M: input DNN model.
D_T: training dataset.
D_V: validation dataset.
μ: minimum unit-count vector.
$ValAcc_{min}$: minimum validation accuracy.

Output : M_{FS1}: pruned DNN model that approximates M.
Acc: accuracy of M_{FS1}.
N_{FLOPs}: the number of FLOPs in M_{FS1}.
N_{params}: the number of parameters in M_{FS1}.

$OriValAcc \leftarrow validation(M, D_V)$
$M_{start}, IMS \leftarrow M$
$lastValAcc \leftarrow OriValAcc$
$ValAcc \leftarrow OriValAcc$
while $ValAcc >= ValAcc_{min}$ **do**
  $minStructReached \leftarrow True$
  foreach λ *in* $M_{start}.hiddenLayers$ **do**
    if $\lambda.UnitCount > \mu[\lambda]$ **then**
      $minStructReached \leftarrow False$
      $unitDistance \leftarrow max(floor((\lambda.UnitCount - \mu[\lambda])/2), 1)$
      $IMS \leftarrow$ randomCutUnits(IMS,λ, $unitDistance$)
  end
  if $minStructReached$ **then**
    $ValAcc \leftarrow 0$
  else
    fineTuning(IMS,D_T)
    $ValAcc \leftarrow$validation(IMS,D_V)
  if $ValAcc >= ValAcc_{min}$ **then**
    $lastValAcc \leftarrow ValAcc$
    $M_{start} \leftarrow IMS$
end
$M_{FS1} \leftarrow M_{start}$
$Acc \leftarrow lastValAcc$
$N_{FLOPs} \leftarrow FLOPCount(M_{FS1})$
$N_{params} \leftarrow ParamCount(M_{FS1})$
return $M_{FS1}, Acc, N_{FLOPs}, N_{params}$

Algorithm 5: *far_subphase_2* Function: A pseudocode sketch of the portion of JGRS that operates during far subphase 2.

Input : M: input DNN model.
D_T: training dataset.
D_V: validation dataset.
μ: minimum unit-count vector.
$ValAcc_{min}$: minimum validation accuracy.

Output : M_{FS2}: pruned DNN model that approximates M.
Acc: accuracy of M_{FS2}.
N_{FLOPs}: the number of FLOPs in M_{FS2}.
N_{params}: the number of parameters in M_{FS2}.

$OriValAcc \leftarrow$ validation(M,D_V)
$M_{start}, IMS \leftarrow M_{GRS}$
$lastValAcc \leftarrow OriValAcc$
$maxValAcc \leftarrow OriValAcc$
while $maxValAcc >= ValAcc_{min}$ **do**
  $ValAccs \leftarrow []$
  $IntermediateStructures \leftarrow []$
  foreach λ *in* $M_{start}.hiddenLayers$ **do**
    if λ.*UnitCount* $> \mu[\lambda]$ **then**
      $unitDistance \leftarrow max(floor((\lambda.UnitCount - \mu[\lambda])/2), 1)$
      $IMS \leftarrow$ randomCutUnits(M_{start},λ, $unitDistance$)
      fineTuning(IMS,D_T)
      $ValAccs$.append(validation(IMS,D_V))
      $IntermediateStructures$.append(IMS)
  end
  if $ValAccs$ == [] **then**
    $maxValAcc \leftarrow 0$
  else
    $maxValAcc$ = max($ValAccs$)
  if $maxValAcc >= ValAcc_{min}$ **then**
    $lastValAcc \leftarrow maxValAcc$
    $index \leftarrow$ argmax($ValAccs$)
    $M_{start} \leftarrow IntermediateStructures[index]$
end
$M_{FS2} \leftarrow M_{start}$
$Acc \leftarrow lastValAcc$
$N_{FLOPs} \leftarrow FLOPCount(M_{FS2})$
$N_{params} \leftarrow ParamCount(M_{FS2})$
return $M_{FS2}, Acc, N_{FLOPs}, N_{params}$

Algorithm 6: A pseudocode sketch of the $JGRS$ algorithm.

Input : M: input DNN model.
D_T: training dataset.
D_V: validation dataset.
μ: minimum unit-count vector.
$\mathcal{T}$: tolerance of accuracy drop.
$attempt\_stage\_1$: the amount of attempts for $far\_subphase\_1$.
$attempt\_stage\_2$: the amount of attempts for $far\_subphase\_2$.
$attempt\_stage\_3$: the amount of attempts for the GRS pruning.

Output : M_{JGRS}: pruned DNN model that approximates M.
Acc: accuracy of M_{JGRS}.
N_{FLOPs}: the number of FLOPs in M_{JGRS}.
N_{params}: the number of parameters in M_{JGRS}.

$Acc \leftarrow validation(M, D_V)$
$ValAcc_{min} \leftarrow Acc \times \mathcal{T}$
$N_{FLOPs} \leftarrow FLOPCount(M)$
$N_{params} \leftarrow ParamCount(M)$
$M_{JGRS} \leftarrow M$
while $attempt\_stage\_1 > 0$ **do**
  $attempt\_stage\_1 -= 1$
  $M_{JGRS}, Acc, N_{FLOPs}, N_{params} = far\_subphase\_1$
   $(M_{JGRS}, D_T, D_V, \mu, ValAcc_{min})$
end
while $attempt\_stage\_2 > 0$ **do**
  $attempt\_stage\_2 -= 1$
  $M_{JGRS}, Acc, N_{FLOPs}, N_{params} = far\_subphase\_2$
   $(M_{JGRS}, D_T, D_V, \mu, ValAcc_{min})$
end
while $attempt\_stage\_3 > 0$ **do**
  $attempt\_stage\_3 -= 1$
  $M_{JGRS}, Acc, N_{FLOPs}, N_{params} = GRS(M_{JGRS}, D_T, D_V, \mu, ValAcc_{min})$
end
return $M_{JGRS}, Acc, N_{FLOPs}, N_{params}$

Table 3.2: Summary of pruning experiments over all four input models using JGRS on the MSN datasets.

model	TestAcc_S(lost%)	ValAcc_S(lost%)	FLOPs_S(% of initial)	Paras_S(% of initial)	prune_time
nn1	0.886(2.10%)	0.910(0.71%)	2410(27.80%)	1223(27.84%)	157.0
nn2	0.887(2.26%)	0.910(0.98%)	2821(36.41%)	1426(36.49%)	63.2
cnn1	0.865(2.37%)	0.877(0.67%)	3648(15.98%)	1855(16.12%)	1157.0
cnn2	0.868(2.24%)	0.892(0.71%)	3594(17.28%)	1826(17.44%)	532.2

In our experiments, we set

$$attempt\_stage\_1 = attempt\_stage\_2 = attempt\_stage\_3 = 3. \tag{3.4}$$

The *validation* function in Algorithm 6 is the same as that called in Algorithm 4 and Algorithm 5.

3.4 Experimental Results

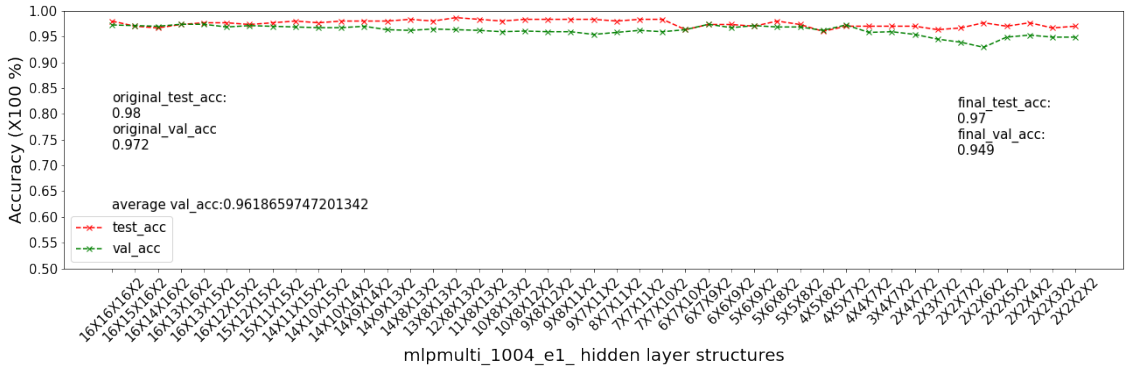

Figure 3.1: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 04e1.

In this section, we present an extensive experimental evaluation, which concretely

Table 3.3: Summary of pruning experiments over all four input models using GRS on the MSN datasets.

model	TestAcc_S(lost%)	ValAcc_S(lost%)	FLOPs_S(% of initial)	Paras_S(% of initial)	prune_time
nn1	0.889(1.82%)	0.912(0.65%)	3801(40.86%)	1925(40.87%)	922.9
nn2	0.893(1.61%)	0.911(0.86%)	4361(56.16%)	2203(56.21%)	115.7
cnn1	0.865(2.38%)	0.88(0.46%)	7208(27.34%)	3644(27.47%)	5933.6
cnn2	0.868(2.24%)	0.893(0.55%)	5514(26.06%)	2792(26.2%)	2329.1

Table 3.4: Summary of comparison experiments of JGRS and GRS over all four input models with the MSN datasets.

	JGRS vs GRS											
	AL_difference			FCI_difference			PCI_difference			Time_ratio		
model	min	max	avg	min	max	avg	min	max	avg	min	max	avg
nn1	-8.35%	11.13%	0.28%	-45.47%	93.58%	13.05%	-45.10%	93.51%	13.03%	0.64	23.61	7.95
nn2	-11.87%	9.22%	0.65%	-40.58%	87.39%	19.75%	-40.55%	87.29%	19.72%	0.33	8.11	2.09
cnn1	-14.35%	11.65%	0.00%	-31.40%	93.04%	11.37%	-31.24%	92.98%	11.35%	0.35	18.92	5.58
cnn2	-14.46%	10.87%	0.00%	-50.23%	84.09%	8.78%	-50.20%	83.76%	8.76%	0.3	10.6	4.84

Table 3.5: Summary of pruning experiments over all four scaled input models using JGRS on the MSN datasets.

model	TestAcc_S(lost%)	ValAcc_S(lost%)	FLOPs_S(% of initial)	Paras_S(% of initial)	prune_time
$nn1\_scaled$	0.904(1.94%)	0.927(0.73%)	5237(1.18%)	2651(1.19%)	346.8
$nn2\_scaled$	0.904(1.99%)	0.926(0.87%)	5665(5.16%)	2863(5.17%)	138.8
$cnn1\_scaled$	0.883(2.90%)	0.907(0.84%)	54926(1.32%)	27579(1.33%)	3064.9
$cnn2\_scaled$	0.887(2.39%)	0.911(0.80%)	33275(1.54%)	16726(1.55%)	1016.8

Table 3.6: Summary of pruning experiments over all four scaled input models using JGRS on the WGEVIA graph embedding dataset.

model	TestAcc_S(lost%)	ValAcc_S(lost%)	FLOPs_S(% of initial)	Paras_S(% of initial)	prune_time
$nn1\_scaled$	0.944(1.95%)	0.968(1.20%)	765(0.19%)	394(0.19%)	161.4
$nn2\_scaled$	0.940(2.53%)	0.966(1.21%)	1632(1.94%)	830(1.95%)	94.6
$cnn1\_scaled$	0.954(1.03%)	0.968(0.96%)	2841(0.08%)	1452(0.08%)	1005.5
$cnn2\_scaled$	0.948(1.87%)	0.963(1.25%)	600(0.04%)	316(0.04%)	336.9

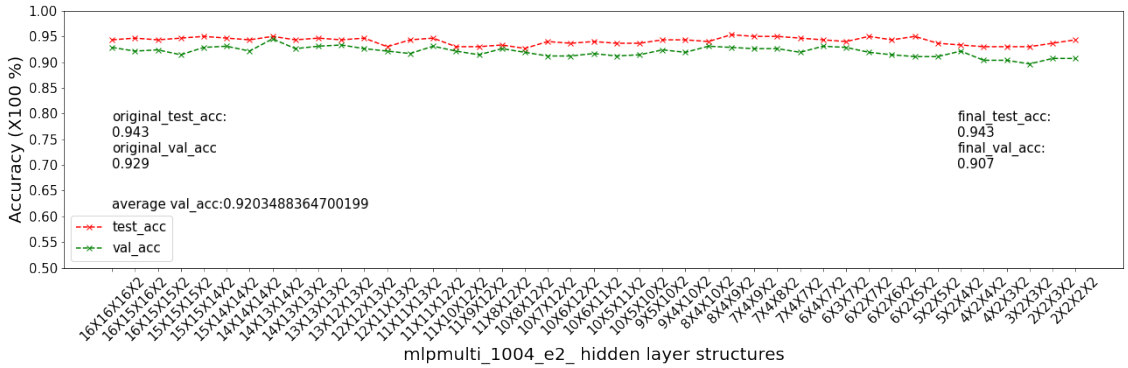

Figure 3.2: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 04e2.

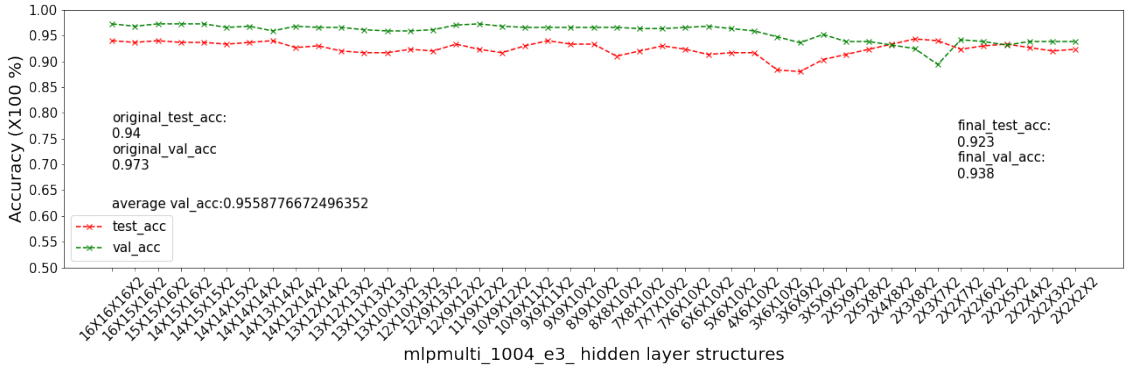

Figure 3.3: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 04e3.

demonstrates the effectiveness of JGRS in the context of calcium-imaging-based neural decoding.

3.4.1 Dataset

Our experiments involve a collection of datasets called MSN (medium spiny neurons). MSN is a collection of datasets involving calcium imaging sessions with mouse models; details about this dataset collection can be found in [58]. In our experiments, we use

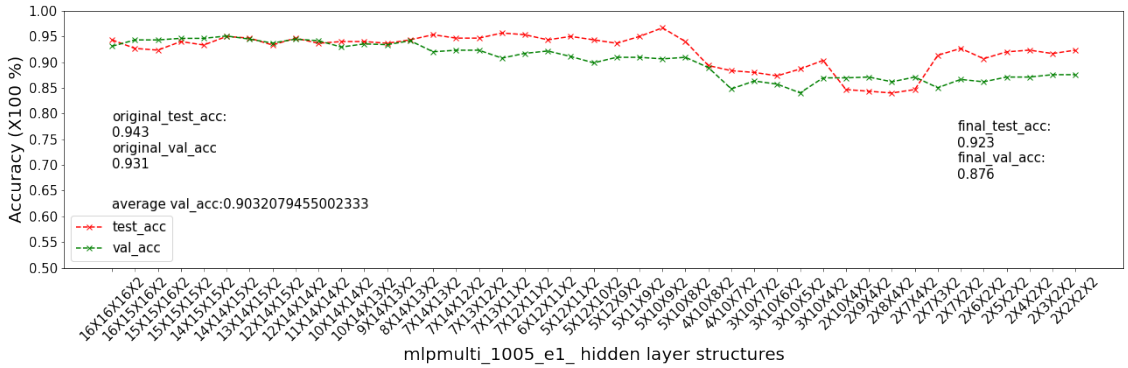

Figure 3.4: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 05e1.

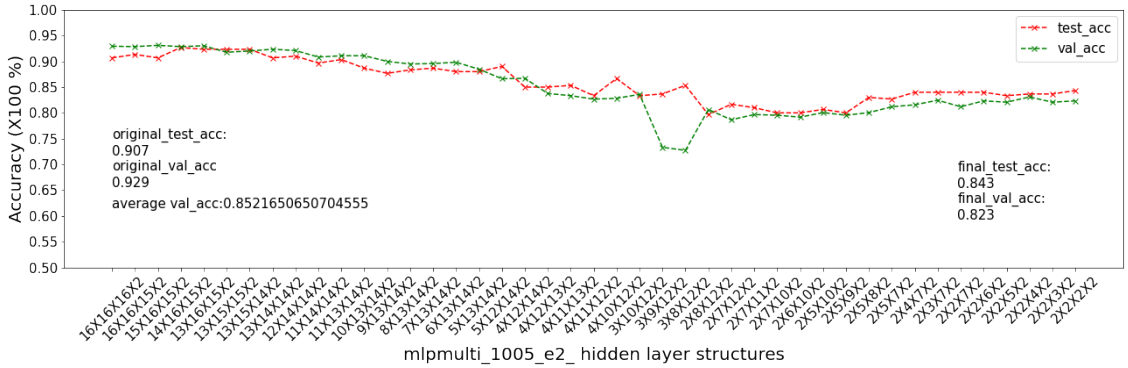

Figure 3.5: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 05e2.

the 9 datasets in MSN from the first three imaging sessions (e1, e2, e3) of mouse targets $04, 05, 06$; these are the same datasets that GRS has been evaluated on in [16], and therefore they provide a natural testbed for evaluating JGRS. Furthermore, to provide more extensive evaluation, we use 9 additional datasets from MSN — these are from the first three imaging sessions (e1, e2, e3) with three different mouse targets, $07, 08, 09$.

Each of the 18 datasets that we worked with contains a $3000 \times \beta(T)$ matrix $Z(T)$ of floating point values for each mouse model T, and a 3000-element vector $L(T)$ of binary

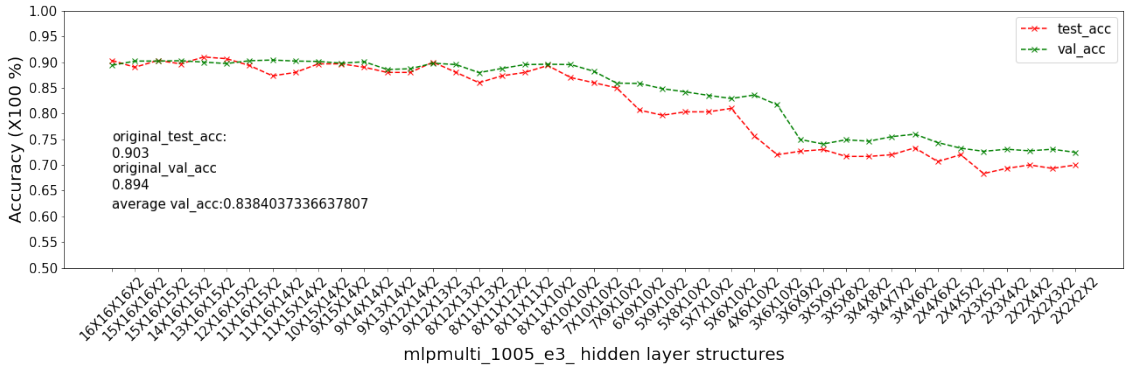

Figure 3.6: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 05e3.

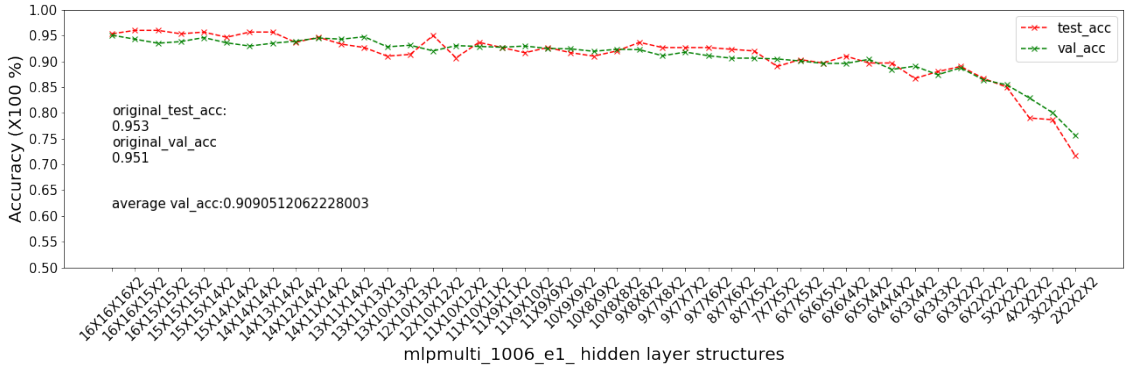

Figure 3.7: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 06e1.

values. Here, $\beta(T)$ denotes the number of detected neurons associated with T. The rows of $Z(T)$ and elements of $L(T)$ correspond to distinct sampling intervals, which are spaced 0.1 seconds apart (due to the 10Hz sample rate) and collectively cover 5 minutes of imaging activity. Each column of $Z(T)$ corresponds to a distinct neuron. Each matrix element $Z(T)[i, j]$ gives the value of the measured calcium imaging sample for neuron j at sample index i. Similarly, each element $L(T)[i]$ gives binary a label indicating whether or not the mouse was engaged in fine motion at the time instant associated with sample index i. Fine

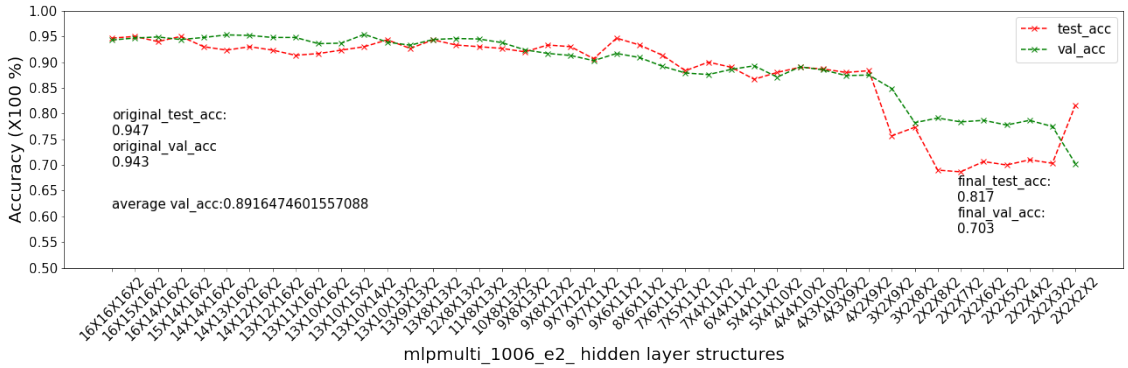

Figure 3.8: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 06e2.

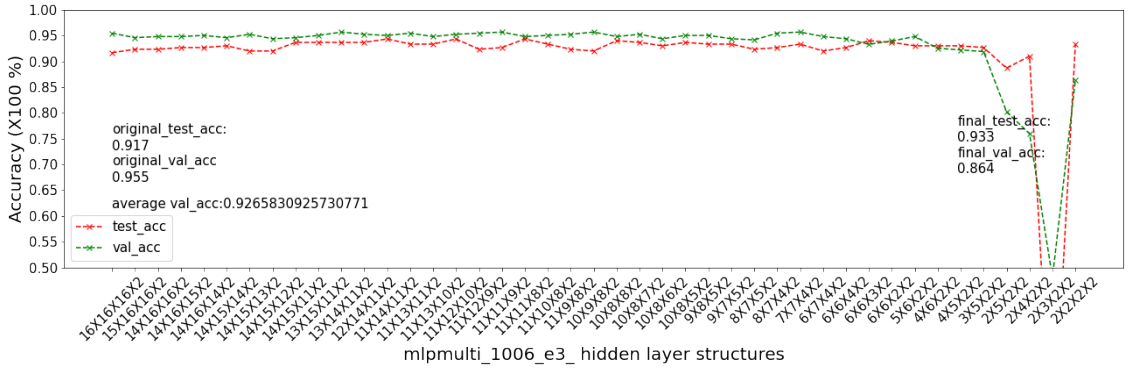

Figure 3.9: Trends of validation accuracy and test accuracy when pruning an over-parameterized MLP model with hidden structure 16X16X16 to the preset minimum hidden structure 2X2X2 on dataset 06e3.

motion at a given time instant t means that the mouse is moving at a speed $s(t)$ such that $0.2\text{cm/s} < s(t) < 2\text{cm/s})$. The mouse models whose data we used in our experiments, models $04, 05, 06, 07, 08, 09$, have $\beta(T) = 273, 140, 114, 154, 53, 96$ neurons, respectively. More details about the MSN dataset collection can be found in [58].

Another dataset that we used in our experiments consists of embedding vectors of functional microcircuits generated by the Weighted Graph Embedding with Vertex Identity Awareness (WGEVIA) algorithm [59]. These embedding vectors were generated

when WGEVIA was applied to a functional microcircuit dataset called REAL [60]. We refer to this dataset as the *WGEVIA-REAL Embedding Vector Dataset* or simply the *WGEVIA-REAL* dataset for short.

Functional microcircuits are collections of neurons that are spatially close to one another and exhibit some form of synchrony in their associated neural activities. Functional microcircuit data can be generated from calcium imaging. Functional microcircuits can be represented by graphs in which vertices correspond to neurons and edges represent pairs of neurons that are related in terms of spatial proximity and synchrony. WGEVIA is an algorithm for transforming graph models, such as those used for microcircuits, into low-dimensional vector representations, called *embedding vectors*, that are effective for processing by DNNs. Each embedding vector in the WGEVIA-REAL dataset is an 80 dimensional vector associated with a binary label indicating one of two categories of the original functional microcircuit. The dataset contains 1600 embedding vectors evenly labeled with '1' and '0' labels. More details about the WGEVIA-REAL dataset can be found in [59].

3.4.2 Input Models

Our experiments involve several overparameterized models that are used as input for pruning. These models are summarized in Fig. 3.10 and Fig. 3.11. Two different types of DNNs are included in Fig. 3.10: two of the models summarized are MLP models — nn1 and nn2 — and other two models, cnn1 and cnn2, are CNNs. The models summarized in Fig. 3.10 are the same overparameterized models that are used the experiments reported

in [16] for the GRS algorithm. We use the models nn1, nn2, cnn1 and cnn2 mainly to compare the pruning performance of JGRS with GRS.

The four scaled models shown in Fig. 3.11 hold the same numbers of layers compared to corresponding models in Fig. 3.10 but contain 16 times more artificial nodes in each hidden layer. These four scaled models are used to experiment with the pruning performance when applying JGRS on large input models. The purpose of having initial models with different network types, numbers of layers, and numbers of hidden nodes is to test the ability of JGRS to handle different types of DNN structures.

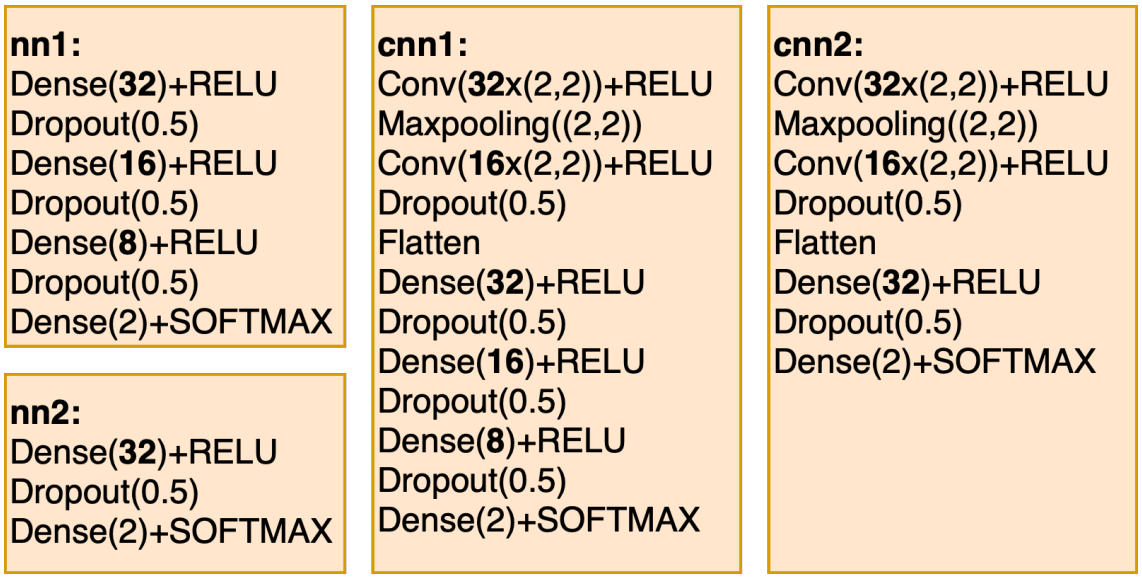

Figure 3.10: Overparameterized DNN models that are used as input for pruning in our experiments.

3.4.3 Multi-phase Behavior in GRS

In this experiment, we show that as removal units are removed from a DNN, the model being pruned becomes increasingly sensitive to the elimination of removal units as the

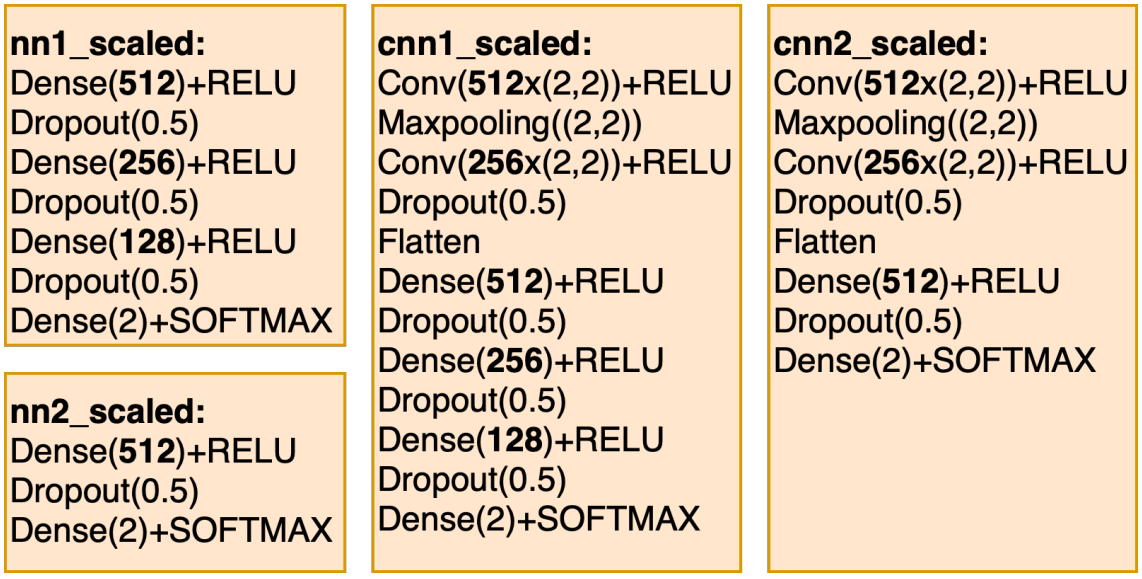

Figure 3.11: Scaled DNN models that are used to exercise the ability of JGRS to handle different kinds of DNN structures.

model becomes more compact. This experiment uses the 9 MSN datasets related to mouse models $04, 05, 06$.

For this experiment, we use the Random inter-layer order and Random Selection of intra-layer unit (RRS) algorithm [16], which selects intermediate structures for pruning randomly rather than using the greedy strategy of GRS. We use RRS in this experiment because our objective is to demonstrate the sensitivity of accuracy to pruning as a model becomes more compact rather than to demonstrate specific characteristics of GRS or its underlying greedy strategy. The accuracy-drop tolerance $\mathcal{T}$ in RRS is set to 0.5 to help ensure that RRS pruning reaches the preset MUCV μ.

As input, we take a $16X16X16$ multi-layer MLP model $mlpmulti$ with 16 nodes in each hidden layer. One purpose of selecting this hidden structure is to avoid unbalanced substructures, where only one layer does not get pruned to the number of artificial nodes defined in μ. The MUCV μ is set as $2X2X2$. We run RRS with the $mlpmulti$ model on

the 9 MSN datasets and observe the pruning trend. We plot the average of the validation accuracy levels of all of the intermediate structures derived from removing a single artificial node from each layer. The RRS iterates with the derived intermediate structures, each having progressively smaller size, until the MUCV μ is achieved. For each intermediate structure evaluated through validation as the RRS algorithm progresses, we averaged across three evaluations to reduce bias due to randomness in the retraining process.

We show the results of all 9 datasets in Fig. 3.1, Fig. 3.2, Fig. 3.3, Fig. 3.4, Fig. 3.5, Fig. 3.6, Fig. 3.7, Fig. 3.8, and Fig. 3.9. In each of these figures, we see that the average validation accuracy across all substructures, plotted in green color, is stable when the model is close to the over-parameterized side — i.e., the left side of each plot. However, the average validation accuracy drops significantly when the model is close to structures that match the MUCV — i.e., the right side of each plot. With GRS, the compact model will generally be found between an intermediate structure at which the average validation accuracy starts to drop significantly and a structure corresponding to the MUCV.

The experimental results presented in Fig. 3.1, Fig. 3.2, Fig. 3.3, Fig. 3.4, Fig. 3.5, Fig. 3.6, Fig. 3.7, Fig. 3.8, and Fig. 3.9 illustrate in detail and with great consistency the phenomena of far and near phases, which was introduced in Section 3.2 and applied in the development of the JGRS algorithm. These results demonstrate more concretely the motivation for using JGRS and why the $far\_subphase\_1$ and $far\_subphase\_2$ functions can be applied before the original GRS pruning method to derive a much more efficient structured pruning algorithm compared to applying GRS uniformly during both phases.

3.4.4 Comparison Between JGRS and GRS

In this section, we experiment with JGRS and GRS on the same MSN datasets to compare their pruning performance, including pruning speed, final compact model size, and test accuracy drop from pruning.

We applied GRS and JGRS to 4 DNN models with different structures and nine datasets related to mouse models 04,05,06. Furthermore, we extended the experiment to 9 new MSN datasets (beyond the datasets used in [16]). These datasets are related to mouse targets 07,08,09.

To maintain a fair comparison with JGRS, the GRS method used in this experiment is also averaged across 3 attempts. This averaging is performed to reduce bias from randomness associated with evaluating different randomly-cut nodes. The accuracy drop tolerance is set as $\mathcal{T} = 0.985$ for both JGRS and GRS.

For measurement of pruning speed, we measure the wall time of both methods in an environment with 6 experiment tasks running in parallel. Each experiment task repeats 10 trials on datasets associated with a single mouse model — for example, datasets the $m04e1$, $m04e2$, and $m04e3$ datasets associated with mouse model $m04$. Results are averaged across the 10 trials. Four input DNN models of different types and structures are pruned in each trial with GRS or JGRS.

We applied the metrics defined in [16] for comparing pruning methods: AL, FCI, and PCI, which stand, respectively, for Accuracy Loss, FLOP Count Improvement, and Parameter Count Improvement. Differences in AL are reported as the maximum, minimum, and average of Equation 3.5:

$$\left\{ \frac{\mathrm{Acc\_G}(d) - \mathrm{Acc\_J}(d)}{\mathrm{Acc\_O}(d)} . \middle| d \in D \right\}. \tag{3.5}$$

Here, $d \in D$, where $D = \{\mathrm{m04e1}, \mathrm{m04e2}, \ldots \mathrm{m09e3}\}$ denotes our set of eighteen MSN datasets, and $\mathrm{Acc\_X}(d)$ represents the accuracy measured in our experiments by method X on dataset d for the given input model M. Here, $\mathrm{X} = \mathrm{O}$ denotes the original model (without any pruning), while $\mathrm{X} = \mathrm{G}$ and $\mathrm{X} = \mathrm{J}$ denote the pruned models derived by applying GRS and JGRS, respectively, to the original model. The initial models before pruning for both JGRS and GRS are trained separately for different datasets.

The FCI difference and PCI difference values reported in Table 3.4 are calculated by taking Equation 3.5 and replacing each $\mathrm{Acc\_X}(d)$ term with $\mathrm{FLOPs\_X}(d)$ or $\mathrm{Params\_X}(d)$, respectively, for the corresponding model specifier X.

Intuitively, from the formulation of AL, FIC, and PCI differences, as described above, a positive value for the AL difference in Table 3.4 means that JGRS performs worse than GRS, while positive values for FCI and PCI indicate that JGRS performs better than GRS.

As shown in Equation 3.6, the $Time\_ratio$ reports the maximum, minimum, and average runtime comparison between J : JGRS and G : GRS for each dataset d. A value higher than 1 indicates that JGRS is faster than GRS, and a value lower than 1 indicates that JGRS is slower than GRS.

$$\left\{ \frac{\mathrm{Runtime\_G}(d)}{\mathrm{Runtime\_J}(d)} \middle| d \in D \right\}. \tag{3.6}$$

From the results shown in Table 3.4, JGRS can reach compact models with around

10% better FCI and PCI than GRS with negligible AL difference. We anticipate that the improved pruning results of JGRS are because JGRS jumps through less important pruning decisions during the *far phase* and examines more critical pruning decisions by evaluating different candidate-sets of nodes for removal. In addition to improved FCI and PCI, JGRS achieves significantly lower pruning time than GRS when processing models with the same over-parameterized structures.

We further consider the nn1 model as a detailed case study and configure different hidden structures $16X8X4$, $32X16X8$, $64X32X16$, $128X64X32$, $256X128X64$, $512X256X128$ to it. We configure GRS to have 3 attempts, and JGRS to have $attempt\_stage\_1 = attempt\_stage\_2 = attempt\_stage\_3 = 3$. We apply JGRS and GRS on the WGEVIA-REAL dataset with 10 repeated trials on each initial model structure. The average runtimes of JGRS and GRS are plotted in Fig. 3.12. As the WGEVIA-REAL dataset contains well-featured embedding vectors of microcircuits, the final compact model size is small and the pruning process goes through a relatively long far phase when GRS is applied. We can see that JGRS effectively jumps across the far phase, taking a relatively small amount of time in this phase, and derives the final compact model with a small number of iterations of GRS pruning. In comparison, GRS pruning time keeps increasing with increasing model size, as GRS spends more and more time in the far phase before it reaches the final compact model.

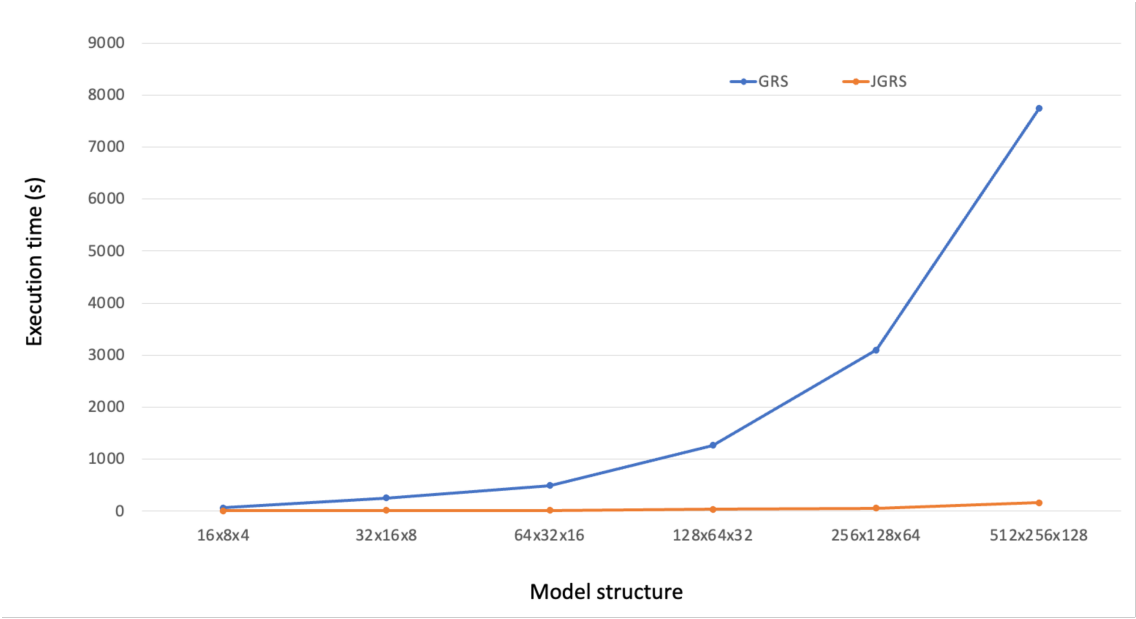

Figure 3.12: Trends of JGRS and GRS pruning speed with different sizes of MLP models on the WGEVIA-REAL dataset.

3.4.5 JGRS for Large-Scale Models

Since JGRS operates very efficiently when structured pruning proceeds under the far phase, JGRS can prune much larger-scale DNN models than those that can be processed by GRS using similar amounts of time and computational resources. In this section, we demonstrate the ability of JGRS to fully process large-scale input models that GRS cannot finish pruning with a reasonable time budget on the computer platform that we used for our experiments. The platform is a equipped with a core i7-2600K CPU and a GeForce GTX 1080 GPU. We refer to the large-scale initial models used in these experiments as *large models*, and the previous input models (discussed in Section 3.4.4) as *small models*. Each large model is derived by scaling up a corresponding small model.

In these experiments, the accuracy drop tolerance is set as $\mathcal{T} = 0.985$.

Comparing the results in Table 3.5, Table 3.3 and Table 3.2, we see that the test

accuracy and validation accuracy of the pruned models resulting from large models are higher than those from the corresponding small models. The final sizes of the compact models derived from the large models are larger too. These results are because $\mathcal{T}$ is the same across the experiments, and the large models generally have higher initial accuracy.

From the results in Table 3.5, Table 3.3 and Table 3.2, we see also that the runtime of pruning large models with JGRS is shorter than pruning corresponding small models with GRS, which shows that JGRS effectively increases the pruning speed. GRS is not suitable for pruning large-scale models because GRS interleaves retraining and validation with each removal unit elimination. In contrast, JGRS implements the jump mechanism, which breaks the rapid runtime increase (in terms of the number of DNN nodes) during the far phase regime.

We also examined the pruning speed of JGRS on the WGEVIA-REAL dataset. The input models used in this experiment are the large-scale models shown in Fig. 3.11. The experiment results are averaged over 10 independent trials. According to the results shown in Table 3.6, we observe significant model size reduction through pruning within a 3% test accuracy drop. We also observe that JGRS derives much smaller compact models with faster pruning speed on the WGEVIA-REAL dataset compared to the MSN datasets. We anticipate that there are two main aspects to this more favorable performance on the WGEVIA-REAL dataset. First, the WGEVIA-REAL dataset contains more informative features that allow the JGRS pruning process to jump across a longer *far phase*. Second, the WGEVIA-REAL dataset has lower dimensionality compared to the MSN datasets, which facilitates much faster retraining.

3.5 Chapter Summary

In this chapter, we developed new methods for scalable pruning of deep neural networks (DNN). These new methods have important applications in neural engineering when analysis of large collections of neurons is involved. We first analyzed Greedy inter-layer order with Random Selection (GRS), which is a previously-developed state-of-the-art algorithm for DNN pruning. We performed a detailed experimental analysis of how the GRS algorithm operates, and identified two distinct regimes in the operation of the algorithm. Based on insights derived from this analysis, we proposed a significantly improved structured pruning method called Jump GRS (JGRS). JGRS builds upon the advantage of GRS, and introduces a concept called the the jump mechanism, which exploits the multiple regimes of operation in GRS, and greatly improves the time-efficiency of the pruning process. Through extensive experiments on calcium-imaging-based neural signals, we demonstrated that JGRS has comparable pruning ability as GRS in terms of the accuracy of the generated compact models. At the same, we also demonstrated that JGRS provides much faster pruning speed.

Chapter 4

WGEVIA: Graph Level Embedding for Microcircuit Data

4.1 Chapter Overview

Functional microcircuits are useful for studying interactions among neural dynamics of neighboring neurons during cognition and emotion. A functional microcircuit is a group of neurons that are spatially close, and that exhibit synchronized neural activities. For computational analysis, functional microcircuits are represented by graphs, which pose special challenges when applied as input to machine learning algorithms. Graph embedding, which involves the conversion of graph data into low dimensional vector spaces, is a general method for addressing these challenges. In this chapter, we discuss limitations of conventional graph embedding methods that make them ill-suited to the study of functional microcircuits. We then develop a novel graph embedding framework, called Weighted Graph Embedding with Vertex Identity Awareness (WGEVIA), that overcomes these limitations. Additionally, we introduce a dataset, called the five vertices dataset, that helps in assessing how well graph embedding methods are suited to functional microcircuit analysis. We demonstrate the utility of WGEVIA through extensive experiments involving real and simulated microcircuit data.

Material in this chapter was published in partial, preliminary form in [59].

4.2 Introduction

Graph-related data is widely used in real world applications, including social network graphs and customers' interest graphs in consumer applications [61], molecule and protein networks in biology and chemistry [62], and brain networks [63] for neuroscience and biomedical engineering. Graph embedding is a general approach that is used to reduce the computational and storage complexity of graph analytics, and to facilitate processing of graphs by well-known machine learning methods. Graph embedding involves the conversion of graph data into low-dimensional spaces, and allows graphs to be represented in compact vector form (input format preserving relevant network properties) [64]. Since vector format is one of the most widely used input formats in machine learning algorithms, graph embedding enables downstream analysis with a wide variety of machine learning methods, including methods for clustering and classification. Most graph embedding algorithms are unsupervised learning algorithms with no requirement for labels (or annotations) in the associated graph datasets. This eliminates the time consuming and error prone process of labeling graphs. These characteristics of graph embedding (compact vector representation and unsupervised learning) provide opportunities for developing new applications such as neural decoding [65, 66].

In this work, we focus on graph embedding for graphical models of functional microcircuits [67], which are studied in neuroscience and biomedical engineering. A set of neurons forms a functional microcircuit if (a) they are spatially close to one other (locality condition), and (b) their neural activities are synchronized [68] (synchrony condition). Various experimental and computational research works have reported on interactions

among neural dynamics of neighboring neurons during cognition and expression of emotion [69, 68, 70]. Many mechanisms can contribute to the observed synchrony in functional microcircuits. For example, functional microcircuits may reflect synaptically coupled subnetworks. Ko et al. studied somatic calcium signals of nearby layer 2/3 pyramidal neurons in mouse visual cortex in vivo, and synaptic connectivity of these neurons in vitro. They found that bidirectional synaptic connections were more frequent between neuronal pairs with strongly correlated visual response [71]. A functional microcircuit can be examined by using calcium imaging [65].

For computational analysis, functional microcircuits are represented by graphs, which we refer to as *microcircuit models*. Microcircuit models can be in the form of weighted or unweighted graphs. Vertices in these models correspond to neurons, and edges connect pairs of neurons that satisfy or are estimated to satisfy the locality and synchrony conditions described above. The graphs may be directed or undirected, and can consist of several hundreds of vertices. In a widely used framework to derive microcircuit models, the connectivity (edge weight) between a pair of neurons is quantified by a correlation coefficient (with respect to some functional behavior) [72]. Vertices in a functional microcircuit represent the biological cells, the so-called neurons. The vertices are, therefore, generally not interchangeable. In the context of graph embedding, this means that the identities of the vertices must be taken into account in the embedding process.

In this work, we develop a novel graph embedding method for microcircuit models that is based on deep learning (DL). DL has achieved excellent performance levels in a great variety of research fields, such as computer vision and natural language processing.

This work represents the first work, to our knowledge, in applying and optimizing DL for the problem of graph embedding for microcircuit models.

Two general classes of graph embedding methods are vertex-level embedding methods, which generate a separate feature vector for each vertex, and graph-level embedding methods, which generate a single feature vector for the whole graph. In this work, we are interested only in graph-level embedding methods due to their importance in analyzing microcircuit models, where the primary interest is on understanding the microcircuit as a whole rather than on finer grained understanding of individual neurons. In the remainder of this proposal, when we refer to "graph embedding", we specifically mean "graph-level embedding" unless otherwise specified.

Representative methods in the literature for graph embedding include DeepWalk [73], node2vec [74], graph2vec [75], and PowerGNN [76]. Among these, graph2vec is especially effective. graph2vec is an unsupervised learning method that employs random walks together with Doc2Vec [77], which is a popular DL-based method for natural language processing. In graph2vec, a feature extractor is used to generate a corpus of graphs, and then Doc2Vec converts this corpus into a vector representation. On various benchmark datasets, graph2vec has been shown to provide better performance compared to alternative methods, including node2vec [74], sub2vec [78], Weisfeiler-Lehman graph kernels [79], and deep Weisfeiler-Lehman graph kernels [80].

Existing graph embedding methods are not well suited to generating graph embeddings for microcircuit models. First, many graph embedding methods focus on unweighted graphs [75, 76, 81], whereas, as mentioned previously, both unweighted and weighted graphs are generally relevant in the analysis of microcircuit models. The inability to handle

weighted graphs therefore makes a graph embedding approach too restrictive for the objectives of our work, which is to develop a general framework that provides efficient graph embedding for microcircuit models. Similarly, many graph embedding methods, including the popular graph2vec and PowerGNN methods, do not take the identities of vertices into account. To the best of our knowledge, the graph embedding approach presented in this chapter is the first to simultaneously support weighted graphs, and take into account the identities of vertices. This makes it uniquely well suited for application to microcircuit models.

To address the limitations of existing graph embedding algorithms and leverage the capabilities of DL, we aim to develop a new DL-based, graph-level embedding algorithm that is effective for microcircuit models. To this end, we develop in this chapter a new unsupervised learning algorithm called Weighted Graph Embedding with Vertex Identity Awareness (WGEVIA). WGEVIA can generate embeddings for both weighted and unweighted graphs, and can also account for vertex identities. Through extensive experiments, we show that the graph embeddings generated by WGEVIA are more effective than previously developed graph embedding methods for analysis of microcircuit models — specifically, for the problem of behavior classification from calcium-imaging-based microcircuit data.

4.3 Methods

In this work, we are interested specifically in an important class of microcircuit models called *coherence-based* models [82, 83]. In this context, coherence is a directionless

association between pairwise neurons. The coherence association models the synchrony between neuron pairs with respect to the enclosing functional microcircuit. The degree of coherence or synchrony between a pair of neurons can be quantified by a metric such as the correlation, partial correlation, mutual information or maximal information coefficient. In the remainder of this chapter, when we use the term "microcircuit model", we mean a coherence-based model, unless otherwise stated.

Although this work is focused on coherence based microcircuit models, we envision that it provides a valuable foundation that can be extended to other types of microcircuit models, including those that involve directional associations between neurons. Such extensions represent an interesting direction for future work.

4.3.1 Objective

As described in Section 4.2, a microcircuit model is a graphical model of a functional microcircuit. Coherence-based models use a form of graph called an *undirected graph*. An undirected graph is an ordered pair $G = (V, E)$, where V is a finite set whose elements are referred to as *vertices*, and each element of E is a pair $\{x, y\}$ of vertices ($x, y \in V$). Each element of E is called an *edge*. In the remainder of this chapter, when we use the term "graph", we mean an undirected graph, unless otherwise stated.

A *microcircuit model* is an ordered pair $M = (G, W)$, where G is a graph, and $W : E \to \mathbb{R}$ is a function that maps the set E of edges (in G) into the set $\mathbb{R}$ of real numbers. We refer to G and W as the *graph and weight function associated with* M. The vertex set and edge set of G are denoted as $vertices(M)$ and $edges(M)$, respectively. Also, given an

edge $e = \{x, y\}$, we say that the vertices x and y are *incident* to e, and we say that x and y are *neighbors* (x is a neighbor of y and vice versa). The number of edges that are incident to a given vertex x is called the *degree* of the vertex, and is denoted as $deg(x)$.

In our experiments, the weight of a microcircuit edge is taken to be the Spearman correlation coefficient [84] between the neurons that are incident to the edge. However the proposed methods are not specific to use of the Spearman correlation coefficient, and alternatively, arbitrary methods for assigning edge weights can be used, such as the partial correlation, mutual information or maximal information coefficient.

We say that a *weighted graph* is a graph with a real-valued quantity called the *weight* (or *edge weight*) associated with each edge in the graph. Thus, a microcircuit model (G, W) can be viewed as a weighted graph with the weight of each edge e defined to be $W(e)$.

A *microcircuit dataset* D is a set $D = \{M[1], M[2], \ldots, M[D_{n_G}]\}$, where $M[1], M[2], \ldots, M[D_{n_G}]$ are microcircuit models with a common vertex set. We denote the common vertex set of D as $Vertices(D)$; thus, $vertices(M[i]) = Vertices(D)$ for all i.

The core contribution of this work is a novel algorithm, called WGEVIA (Weighted Graph Embedding with Vertex Identity Awareness), that takes as input a microcircuit dataset D and a positive integer m, and outputs a mapping $\theta_m : D \rightarrow \mathbb{R}^m$. The mapping is derived in such a way so that machine learning tasks (*downstream analysis tasks*) can operate efficiently and accurately using the *embedded dataset* $\{\theta_m(d) \mid d \in D\}$. A mapping of this form (from a microcircuit D into $\mathbb{R}^m$ for some m) is called a *microcircuit embedding*. More generally, it can also be viewed as an embedding of a set of weighted

graphs; however, the motivation and experiments in this chapter are developed in the specific context of microcircuit models.

WGEVIA applies a novel algorithm, called UGEVIA (Unweighted Graph Embedding with Vertex Identity Awareness), which we develop to compute embeddings of unweighted graphs. An embedding θ_u for unweighted graphs, which we refer to as a *graph embedding*, maps a set of graphs $U = \{G[1], G[2], \ldots, G[n_U]\}$ into $\mathbb{R}^m$ for some positive integer m — $\theta_u : U \to \mathbb{R}^m$.

The remainder of this section is summarized as follows. We first discuss the problem of retaining vertex identities when performing graph embedding, and we introduce a novel dataset called the five vertices problem dataset. Next, in Section 4.3.3 through Section 4.3.5, we introduce in detail the overall UGEVIA algorithm, the feature extraction algorithm used in the UGEVIA algorithm, and the WGEVIA algorithm, respectively. Then in Section 4.3.6, we summarize our approach for evaluating the effectiveness of the proposed new embedding methods for microcircuit models.

4.3.2 Identities of Graph Vertices

An important characteristic of microcircuit models is that each vertex represents a unique neuron, and in general, the identities of the neurons and not just the connectivity patterns among them are relevant to microcircuit analysis methods.

The *five vertices problem* is a simplified benchmark problem that we have designed to help study the interaction between graph embedding and vertex identity awareness. In our experiments, we use this benchmark to assess how effective a graph embedding

approach is in terms of taking vertex identities into account.

The five vertices problem is a binary classification problem involving a dataset Δ of 3000 unweighted graphs. The dataset, called the *five vertices dataset*, is a union of six subsets $\Delta = \delta_1 \cup \delta_2 \cup \ldots \cup \delta_6$, where each subset δ_i consists of 500 identical unweighted graphs. We incorporate a large number of graphs, even though they are identical, because the downstream classifier needs a sufficiently large number of instances for proper training. At the same time, in the design of this dataset, we would like to make the two graph classes for each classification problem as simple as possible — this helps to isolate the ability of a given algorithm to take into account vertex identities (without introducing other complicating factors to the classification problem). All 3000 unweighted graphs in Δ have a common vertex set $V_\Delta = \{v_0, v_1, v_2, v_3, v_4\}$. The structures of the graphs in $\delta_1, \delta_2, \ldots, \delta_6$ are illustrated in Fig. 4.1. Each graph $g \in \Delta$ is assigned a binary label $L(g)$, where $L(g) = 0$ for $g \in (\delta_1 \cup \delta_3 \cup \delta_5)$, and $L(g) = 1$ for $g \in (\delta_2 \cup \delta_4 \cup \delta_6)$.

Now consider a graph embedding generator γ', which operates on elements of the dataset Δ, and provides input to a classifier C, where C is designed to learn L based on some partitioning of Δ into training and testing subsets d_{tr} and d_{te}, respectively.

Intuitively, it is easy for C to distinguish between graphs in δ_1 and δ_2 even if γ' does not take vertex identities into account. On the other hand, without the use of vertex-identity awareness in γ', it is impossible for C to distinguish between graphs in δ_3 and δ_4 and between graphs in δ_5 and δ_6. This is because of the isomorphic relationship between graphs in the subset pairs δ_3/δ_4 and δ_5/δ_6.

Now consider the accuracy of the (γ', C) combination in classifying graphs in d_{te} that belong to $\delta' = (\delta_3 \cup \delta_4 \cup \delta_5 \cup \delta_6)$. This accuracy measurement can be considered

as an assessment of the effectiveness of γ' in taking vertex identities into account in the embedding process. In particular, if γ_1 and γ_2 are alternative graph embedding generators, then comparing the accuracy of (γ_1, C) and (γ_2, C) on δ' provides a useful assessment of the relative effectiveness between γ_1 and γ_2 in taking vertex identities into account.

At the same time, the subsets δ_1 and δ_2 help to provide a comparison between γ_1 and γ_2 for scenarios in which vertex identity information is not critical.

In summary, the five vertex dataset is a novel dataset for assessing graph embedding algorithms. The primary emphasis in this dataset is to help assess the effectiveness of an embedding technique in taking vertex identity information into account. The dataset is of special utility in computational neuroscience because neurons that are represented in microcircuit models often need to be distinguished in analysis that is performed on these models.

4.3.3 UGEVIA: Identity-Aware Embedding for Unweighted Graphs

The proposed UGEVIA algorithm can be viewed as a modification to the algorithm used by graph2vec. Our developed modification is aimed at incorporating vertex-identity awareness, and further improving embedding quality.

The graph2vec [75] framework for graph embedding is designed for unweighted graphs. The framework is based on Weisfeiler-Lehman Graph Kernels [79], and it also applies Doc2Vec [77], as described in Section 4.2. In graph2vec, a feature extractor based on Weisfeiler-Lehman Graph Kernels is used to generate a collection of text strings (strings) from the given input graphs to be embedded. Such a collection of strings is

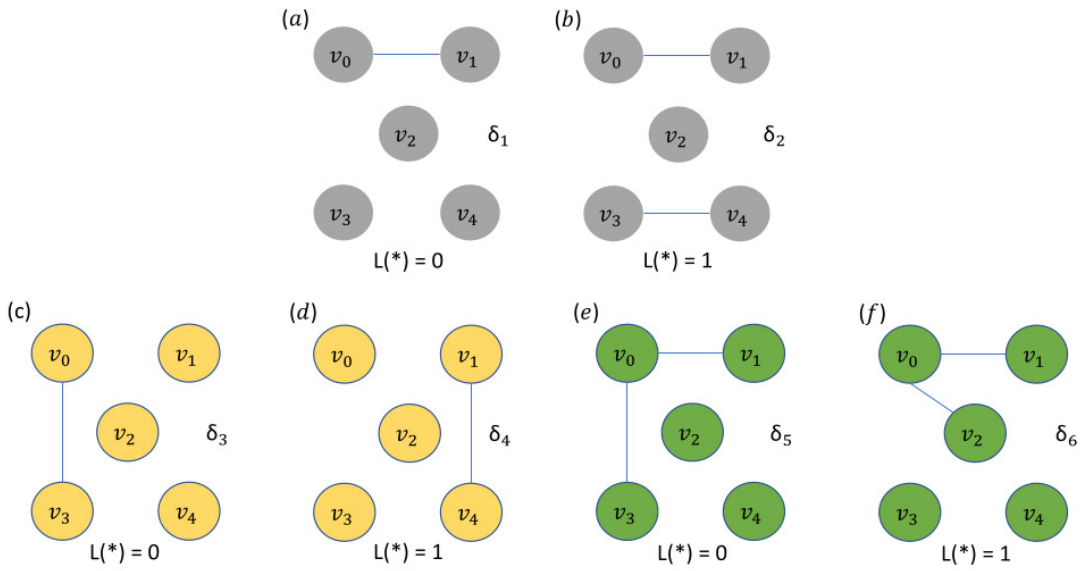

Figure 4.1: The five vertices problem is a binary classification problem involving a dataset Δ of 3000 unweighted graphs. All graphs in Δ have a common vertex set $V_{\Delta} = \{v_0, v_1, v_2, v_3, v_4\}$. The structures of graphs with label 0 are depicted in (a)(c)(e). The structures of graphs with label 1 are depicted in (b)(d)(f).

referred to as a *corpus*. Doc2Vec is then applied to convert the generated corpus into vector representations for the input graphs.

UGEVIA takes as input an indexed set $G^u = \{g_1^u, g_2^u, \ldots, g_{n_G}^u\}$ of unweighted graphs which share a common indexed vertex set, and a positive integer m, which gives the dimension for the output embedding that is to be computed. The common indexed vertex set for the input graphs is denoted as, employing a minor abuse of notation, $Vertices(G^u) = \{v_1, v_2, \ldots, v_k\}$, and the index of a given vertex v_i is referred to as its identifier (ID) and denoted by $ID(v_i)$ (i.e., $ID(v_i) = i$). UGEVIA also takes as input a positive integer $featureGenIters$, which specifies the number of iterations with which to execute the feature extractor on each input graph; this input is discussed further in Section 4.3.4.

UGEVIA produces as output an indexed set $\Omega^u = \{\nu_1^u, \nu_2^u, \ldots, \nu_{n_G}^u\}$ of vectors in

$\mathbb{R}^m$, where each $\nu_i^u \in \mathbb{R}^m$ is the constructed embedding for g_i^u.

As in graph2vec, all of the vertices in all of the input graphs are initially labeled with the associated vertex degrees. The major differences between UGEVIA and graph2vec, which are designed to make the approach more suitable for microcircuit models, are summarized as follows.

- The input to UGEVIA includes unique label IDs, and the feature extraction process utilizes these IDs so that they are taken into account in the generated embeddings.
- Non-zero-degree vertices are labeled with the index of the vertex, while zero-degree vertices are labeled with the character $'Z'$ along with the index of the vertex.
- Within each iteration of feature extraction, features of non-zero-degree vertices are updated with features of their connected neighbors. Unique features of the input graph will be generated based on the relative indexing of zero-degree vertices.

The special treatment of zero-degree vertices in UGEVIA preserves the identities of zero-degree vertex subsets, and avoids problems with random-walk approaches when zero-degree vertices are encountered — in particular, random-walk based approaches typically get stuck at zero-degree vertices (if the walks start at such a vertex) or can never reach such a vertex (if they start at a vertex with positive degree). UGEVIA's careful handling of zero-degree vertices is important in computational neuroscience applications because microcircuit models are often sparse, and therefore contain a large proportion of zero-degree vertices. Although many vertices have zero degree, the associated neuron identities must be taken into account for accurate analysis of microcircuit models.

Algorithm 7 presents a pseudocode representation of the UGEVIA algorithm. Here, $featureDoc$ is a list of ordered pairs having the form (g, β), where g is a graph, and β is a list of strings that provide a feature representation for g. A feature set, represented as a text string, is associated with each vertex in each g_i^u.

The function $setFeatures(v, s)$ sets the features of vertex v to be the string s. The feature set associated with a vertex, as set by function $setFeatures$, can be retrieved or overwritten by the UGEVIA feature extractor (see Section 4.3.4) using the $getFeatures$ or $setFeatures$ functions, respectively.

The function $concat(s_1, s_2, delim)$ returns the concatenation of string s_1 followed by string $delim$ (called the delimiter string), and then followed by string s_2. For example, $concat($"27", "33", "_"$)$ returns "27_33", which is a delimited concatenation of the string representations for two integers. The function $toString(x)$ returns the string representation of the integer x.

The function $featureExtractor(g, I)$ returns a list of strings as features for the graph g. The feature extraction process is parameterized with a number of iterations; the argument I provides the setting for the iteration count parameter. More details on the algorithm underlying function $featureExtractor$ are discussed in Section 4.3.4.

The function $append(\kappa, elem)$ in Algorithm 7 adds the new list element $elem$ (an ordered pair in this particular use of the function) to the end of the list κ.

The function $doc2vec$ in Algorithm 7 calls the off-the-shelf Doc2Vec utility, which has been described above. The $doc2vec$ wrapper utilizes certain configuration parameters for Doc2Vec; settings for these parameters in our experiments are discussed in Section 4.4.

Algorithm 7: A pseudocode representation of the UGEVIA algorithm.

Input : $G^u = \{g_1^u, g_2^u, \ldots, g_{n_G}^u\}$: an indexed set of graphs with a common indexed vertex set $\{v_1, v_2, \ldots, v_k\}$.
$featureGenIters$: the number of iterations to use within the feature extractor.
m: the dimension for the output vectors.
Output : $\Omega^u = \{\nu_1^u, \nu_2^u, \ldots, \nu_{n_G}^u\}$: an indexed set of vectors (each $\nu_i^u \in \mathbb{R}^m$), which respectively provide embeddings for the graphs $g_1^u, g_2^u, \ldots, g_{n_G}^u$.

$featureDoc \leftarrow emptyList$

for $i = 1$ to (n_G) **do**
  for $j = 1$ to k **do**
    if $deg(v_j) == 0$ **then**
      $setFeatures(v_j, concat(\text{“Z”}, toString(j), \text{“”}))$
    else
      $setFeatures(v_j, toString(j))$
  end
  $featureLib \leftarrow featureExtractor(g_i^u, featureGenIters)$
  $append(featureDoc, (g_i^u, featureLib))$
end
$\Omega^u \leftarrow doc2vec(featureDoc, m)$
return Ω^u

4.3.4 UGEVIA Feature Extractor

The feature extractor for UGEVIA takes as input a graph g^u with an ordered vertex set $\{v_1, v_2, \ldots, v_k\}$. It is assumed that each vertex has an initial feature set, which is accessible using the $getFeatures$ function (see Section 4.3.3). The algorithm also takes as input a positive integer $featureGenIters$, which as mentioned previously, controls the number of iterations to employ in the feature extraction process. The feature extractor produces as output a list of strings $featureLib$, called a *feature library*, that provides a feature set for representation of the input graph g^u.

Use of a larger value of $featureGenIters$ results in longer computation time and more storage required for feature extraction, but it may also lead to higher quality feature sets, which improve the accuracy of downstream classification tasks. In our current

approach to applying UGEVIA, we determine the value of $featureGenIters$ empirically so as to optimize accuracy up to a point of diminishing returns — where additional iterations increase computational time with negligible accuracy improvement. In our experiments, we used the value $featureGenIters = 4$, which is found by Bayesian optimization. With the value $featureGenIters = 4$, the extracted features are effective for all of the datasets involved in our experiments.

During each feature extraction iteration, the feature extractor updates features for each vertex v_i, and appends new features to the feature library. If a vertex has non-zero degree, its features are updated using the features of its neighbors in a given feature extraction iteration. On the other hand, if a vertex has zero degree, its features are updated with the features of the vertices that immediately precede and succeed it within the ordering $v_1, v_2, \ldots, v_k$. This is an additional way in which the identities of vertices are taken into account in the graph embedding process — different pairs of preceding and succeeding vertices have distinct pairs of vertex IDs. The determination of preceding and succeeding vertices is performed in a wrap-around sense — that is, using the interpretation that v_k immediately precedes v_1, and equivalently, that v_1 immediately succeeds v_k.

The method described above is tolerant to rearrangement of the vertex sequence in the common vertex set. The algorithm is tolerant to such rearrangement because the algorithm requires only that each vertex has a unique label, and that the vertex labels are consistent across all graphs. The consistency of the labels allows the algorithm to extract features associated with graph structure.

A pseudocode representation of the UGEVIA feature extractor, encapsulated by the function $featureExtractor$, is shown in Algorithm 8. Some functions, such as $concat$ and

$getFeatures$, have been discussed already in Section 4.3.3; in the remainder of this section, we define the functions used in Algorithm 8 that have not already been discussed.

The function $edgeCount(g)$ returns the number of edges in the graph g. The function $getNeighbors(g, v)$ returns the set of neighbors of a given vertex v in a given graph g. The function $card(S)$ returns the cardinality of the set S. The function $sort(list)$ modifies the list of strings $list$ by sorting its elements.

The function $\lambda(s)$ first employs a hash function to convert a string s of arbitrary length into a 128-bit representation. The function then converts the 128-bit representation into a compact, fixed-length string representation of 32 characters. The hash function used is the MD5 hash function [85]. The 32-character string representation is derived as a string representation of the hexadecimal number for the 128-bit output of the hash function. The function λ is used to avoid excessive storage size for vertex and graph features.

4.3.5 WGEVIA: A Multi-channel Approach for Microcircuit Datasets

UGEVIA is not suitable for microcircuit datasets because although it takes vertex identities into account, the algorithm does not take into account graph weights. To simultaneously take into account vertex identities and graph weights, we develop the WGEVIA algorithm.

A simplified description of the inputs to WGEVIA was given in Section 4.3.1. The full input list along with an output specification, and a pseudocode representation is shown in Algorithm 9. Since a microcircuit dataset is defined as an indexed set of weighted graphs, WGEVIA can be applied to many other application areas that employ weighted graphs; the underlying method is not limited to applications in neural signal processing

Algorithm 8: Feature extraction algorithm for UGEVIA.

Input : $g^u = (V^u, E^u)$: a graph with an ordered set of vertices $\{v_1, v_2, \ldots, v_k\}$.
$featureGenIters$: number of feature extraction iterations to carry out.
Output : $featureLib$: a list of features to represent g^u.

```
featureLib ← emptyList
for i = 1 to k do
    append(featureLib, getFeatures(v_i))
end

if edgeCount(g^u) = 0 then
    return emptyList
else
    for iterNum = 0 to featureGenIters do
        nextIterFeatures ← emptyList
        for i = 1 to k do
            neighbors ← getNeighbors(g^u, v_i)
            if card(neighbors) > 0 then
                φ_n ← emptyList
                foreach neighbor ∈ neighbors do
                    newFeatures = getFeatures(neighbor)
                    append(φ_n, newFeatures)
                end
                sort(φ_n)
                φ_v ← concat(getFeatures(v_i), φ_n, "_")
            else
                # v_i has zero degree
                if 1 < i < k then
                    φ_v ← concat(getFeatures(v_{i-1}), getFeatures(v_{i+1}), "_")
                else if i = 1 then
                    φ_v ← concat(getFeatures(v_k), getFeatures(v_2), "_")
                else
                    # i = k
                    φ_v ← concat(getFeatures(v_{k-1}), getFeatures(v_1), "_")
            encodedFeature = λ(φ_v)
            append(nextIterFeatures, encodedFeature)
            append(featureLib, encodedFeature)
        end
        for i = 1 to k do
            # Replace features of v_i with the newly-generated features
            setFeatures(v_i, nextIterFeatures[i − 1])
        end
    end
    return featureLib
```

and computational neuroscience.

Algorithm 9: A pseudocode representation of the WGEVIA algorithm.

Input : A microcircuit dataset $D = \{M[1], M[2], \ldots, M[D_{n_G}]\}$ consisting of graphs $\{{g_1}^w, {g_2}^w, \ldots, g_{n_G}^w\}$, respectively, with the common indexed vertex set $\{v_1, v_2, \ldots, v_k\}$.

n_c: the number of channels used for graph embedding.

μ_c: the dimension of the embedding vector for a single channel.

$featureGenIters$: the number of iterations to use within the UGEVIA feature extractor.

Output : $\Theta^w = \{\nu_1^w, \nu_2^w, \ldots, \nu_{n_G}^w\}$: an indexed set of vectors (each $\nu_i^w \in \mathbb{R}^{\mu_c \times n_c}$), which respectively provide embeddings for the models $M[1], M[2], \ldots, M[D_{n_G}]$.

$\omega_{max} \leftarrow max(getWeights(D))$
$\phi_C \leftarrow emptyList$
$\Gamma^u = disconnectedGraph(\{v_1, v_2, \ldots, v_k\})$
for $t = 0$ to $(n_c - 1)$ **do**
    $T \leftarrow \frac{\omega_{max}}{2 \times n_c} + \frac{\omega_{max} \times t}{n_c}$
    for $i = 1$ to n_G **do**
        # Initialize g_i^u with a copy of Γ^u
        $g_i^u = \Gamma^u$
        $edges = getEdges(g_i^w)$
        for $j = 0$ to $(card(edges) - 1)$ **do**
            if $getWeight(edges[j]) > T$ **then**
                $addUnweightedEdge(g_i^u, edges[j])$
            end
        end
    end
    $G^u = \{g_1^u, g_2^u, \ldots, g_{n_G}^u\}$
    $subFeatures = UGEVIA(G^u, featureGenIters, \mu_c)$
    $appendList(\phi_C, subFeatures)$
end
$\Theta^w \leftarrow \phi_C$
return Θ^w

WGEVIA repeatedly applies UGEVIA on sets of unweighted graphs, called *channels*, that are derived from the input microcircuit dataset. The use of channels in WGEVIA is inspired by the common use of channelization in convolutional neural networks (e.g., channelization of networks to process red, blue and green components of color images).

Intuitively, channels in WGEVIA are derived by applying a threshold to the edge weights and retaining only those edges in a channel whose weights exceed the threshold. As shown in Algorithm 9, the threshold starts from $\frac{w_{max}}{2\times n_c}$, increments by $\frac{w_{max}}{n_c}$, and stops at $\frac{w_{max}}{2\times n_c} + w_{max} \times \frac{n_c - 1}{n_c}$, where n_c is the number of channels. For example, if the edge weights are uniformly distributed in $[0, 1]$, and the number of channels is 10, then according to method of how to set the threshold in Algorithm 9, the first channel will have 95% of edges remaining, the second channel will have 85% of edges remaining, and the tenth channel will have 5% of edges remaining.

Note that the dimension of the output embedding vectors for WGEVIA is not specified explicitly; instead, it is derived as ($\mu_c \times n_c$), where n_c is the number of channels to be employed in WGEVIA, and μ_c is the embedding vector dimension used on each channel when invoking UGEVIA.

We tune the WGEVIA parameters μ_c and n_c experimentally. Through empirical analysis together with Bayesian optimization [86], we derived the values $n_c = 10$ and $\mu_c = 8$ for use in our experiments.

In Algorithm 9, the function $max(S)$ returns the maximum value from a given set S of real numbers. The function $getWeights(H)$ returns the set of all of the unique edge weight values (i.e., a set of real numbers) across all of the weighted graphs within the given microcircuit dataset H. The function $disconnectedGraph(V)$ takes as argument a set V of vertices and returns an unweighted graph whose vertex set is V and whose edge set is empty. The function $getEdges(g)$ returns the set of edges in the weighted or unweighted graph g. The function $getWeight(e)$ returns the weight of the given edge e. The function $addUnweightedEdge(g, e)$ takes as argument an unweighted graph g and a weighted edge

e. The function modifies g by adding an unweighted version of e to the edge set of g. An unweighted version of e is simply an edge that is incident to the same pair of vertices, but has no associated weight. The function $appendList(L_1, L_2)$ modifies the list L_1 by appending to it all of the elements in L_2. The resulting L_1 is the concatenation of the original L_1 with L_2.

The approach in WGEVIA for deriving thresholds can be modified in various ways. For example, instead of setting the minimum threshold to be $\frac{\omega_{max}}{2\times n_c}$, one can generate thresholds within the range $(\omega_{min}, \omega_{max})$, where ω_{min} is the minimum weight across all edges in the microcircuit dataset. Similarly, instead of using uniformly-spaced thresholds, one can distribute the thresholds nonuniformly using more sophisticated computations to determine the inter-threshold spacings. Investigation into strategies for adapting the threshold derivation in WGEVIA is beyond the scope of the chapter. This is another interesting direction for future work.

4.3.6 Graph Classification

We demonstrate the effectiveness of WGEVIA by applying it to two different microcircuit classification problems and studying its performance when it is connected to four different classifiers for each classification problem.

Both of the classification problems that we study are binary classification problems, where the objective involves discriminating between 0- and 1-valued labels for each microcircuit model. The first classification problem, which is based on data generated from a real-world calcium imaging study, involves a dataset that we refer to as the REAL

dataset. The second classification problem involves a dataset generated using simulation techniques; we refer to it as the SIMU dataset. Section 4.4.4 provides more details on these microcircuit classification problems, and the associated datasets.

The four different types of classifiers used in our experiments are:

- single-mlp: a multilayer perceptron (MLP) [40] with one hidden layer.
- multi-mlp: an MLP with two hidden layers.
- SVM-rbf: a support vector machine (SVM) [87] with Radial Basis Function (RBF) kernel.
- LDA: a Linear Discriminant Analysis (LDA) classifier [88].

The classifiers are used in the experiments as downstream analysis tasks that operate on the graph embeddings generated by WGEVIA. The experiments with these classifiers demonstrate the improvements in classification accuracy enabled by WGEVIA when applied to our two different microcircuit classification problems. The use of two different classification problems and a variety of classifiers in the experiments helps to demonstrate the general utility of WGEVIA in enhancing the performance of different types of downstream analysis subsystems.

4.4 Experimental Results

In this section, we present experiments that demonstrate the effectiveness of our proposed graph embedding framework for microcircuit models. The section is organized as follows. First, Section 4.4.1 describes two baseline methods that we use for comparison purposes in our experiments. Next, Section 4.4.2 describes four different classifiers that we use

as downstream analysis tasks to demonstrate the application of WGEVIA to a variety of different analysis subsystems. Then Section 4.4.3 describes common settings of the doc2vec utility that we employ in our experiments (doc2vec is used within graph2vec and WGEVIA). Section 4.4.4 describes microcircuit datasets used in our experiments. Section 4.4.5 evaluates graph2vec, PowerGNN and UGEVIA on the five vertices problem described in Section 4.3.2. Recall from Section 4.2 that graph2vec and PowerGNN are two state-of-the-art methods for graph embedding. In Section 4.4.6, we present a quantitative evaluation comparing WGEVIA with graph2vec and PowerGNN. In Section 4.4.7, comparison experiments are presented to assess the impact of selected design decisions in Section 4.3. Section 4.4.8 shows how visualization can be used to gain intuitive insight into the results produced by WGEVIA. Section 4.4.9 examines the runtime of the proposed algorithms and studies hyperparameter tuning. Our algorithms are implemented with Python 3.

4.4.1 Baseline Methods

We apply graph2vec and PowerGNN, which were introduced in Section 4.2, as two baseline graph embedding methods to compare with the proposed UGEVIA and WGEVIA methods. Both graph2vec and PowerGNN are state-of-the-art graph embedding methods for unweighted graphs. Recall that graph2vec is also a foundation on which the UGEVIA method builds. For our comparisons with graph2vec, we use the popular graph2vec implementation by Rozemberczki et al. [89].

PowerGNN is a specialized graph embedding method for graph classification ap-

plications. Unlike graph2vec, UGEVIA and WGEVIA, a classifier is integrated within PowerGNN. The fully connected layers in a PowerGNN model can be considered as its classifier. The fully connected layers are similar to a multi-layer perceptron (MLP) introduced in Section 4.4.2. Although PowerGNN uses a graph embedding technique internally, it outputs only the classification results, not the intermediate embedding results. Thus, in our experiments, we apply PowerGNN as a combined embedding+classification method without adding any downstream classifier to it. We use a Bayesian optimization technique to tune the hyperparameters of PowerGNN.

4.4.2 Downstream Classifiers

All reported accuracy scores are from classification tasks on input graphs with embedding vectors generated by different graph embedding methods. The embedding vectors are the input to classification tasks. The embedding vector is a representation of the whole graph. Each element of the vector is a floating point value. Graph embedding quality is assessed in terms of the accuracy of classifiers that operate on the generated embeddings. To ensure that our measurement of graph embedding quality is not specific to a particular type of downstream classifier, we used four different downstream classifier models: single-mlp, multi-mlp, SVM-rbf and LDA, as described in Section 4.3.6.

The single-mlp model has one hidden layer with 128 nodes with a RELU activation function. The output layer has two nodes with a Softmax activation function. The multi-mlp model has two hidden layers. The first hidden layer has 128 nodes with a RELU activation function. The second hidden layer has 64 nodes, also with a RELU activation

function. The output layer is the same as that of the single-mlp classifier.

The Adam optimizer is used to train the single-mlp and multi-mlp models [90]. Both models are constructed with TensorFlow [38], and trained for 100 epochs each. The SVM-rbf classifier has two hyperparameters, C and γ; we refer readers to [91] for more details about these two hyperparameters in this classifier. We have tuned the SVM-rbf model using grid search to determine the optimized hyperparameter values $C = 100$ and $\gamma = 0.1$. For the LDA classifier, we used the same grid search strategy to tune the tolerance tol; the value we derived for this hyperparameter is $tol = 0.0001$. For each of the four classifiers, we fixed the structure and hyperparameters across all experiments to help derive fair comparisons across different graph embedding methods.

All models are trained with a Tesla K40c GPU with 12 GB memory and an Intel Xeon E5-2620 v4 CPU with 128 GB memory. All reported accuracy results are based on 10-fold cross-validation [92]. For each round of cross validation, the input dataset is randomly divided into training and test sets. The random division is performed in such a way that there is no overlap between the test sets of different rounds.

4.4.3 Doc2Vec Settings

As described in Section 4.3, the Doc2Vec algorithm is used both in graph2vec and WGEVIA. Doc2Vec involves many hyperparameters. We carefully tuned the hyperparameters and used the same hyperparameters for the Doc2Vec model used inside both graph2vec and WGEVIA. Specifically, we set the learning rate to 0.025, downsampling rate to 0.0001, minimum count to 1, and number of epochs to be 100. The minimum

count provides a threshold that Doc2Vec uses to determine which words to ignore. All words with total frequency below this threshold are ignored. The downsampling rate is a threshold that Doc2Vec uses to downsample high-frequency words. We refer the reader to [77] for more details about the hyperparameters of Doc2Vec.

4.4.4 Microcircuit Datasets

We assessed the proposed algorithm on two microcircuit datasets: REAL and SIMU, which were briefly introduced in Section 4.3.6. In this section, we provide more details about these datasets. As mentioned in Section 4.3.6, REAL is generated from a real-world calcium imaging study [93], while SIMU is generated by simulation. In both datasets, each microcircuit is associated with a binary behavioral variable as its label.

The neuron model used in SIMU is the integrate-and-fire model with additive noise [94]. The model can be represented by

$$\frac{d\rho}{dt} = \frac{\rho_{rest} - \rho}{\tau} + \sigma \times \tau \times (-0.5) \times \varepsilon \tag{4.1}$$

where ρ is the membrane potential, ε is a Gaussian random variable with mean 0 and standard deviation 1, σ is a parameter that controls the noise, τ is the membrane time constant, and ρ_{rest} is the resting potential. The membrane potential ρ changes with spikes that are received through the synapses. A neuron generates a spike if ρ is greater than a threshold. The neuron model has refractoriness — i.e., a brief time period is needed between two spikes of a neuron. Such a neuron model can represent postsynaptic potentials described in the literature [95].

Our simulation to generate the SIMU dataset included a simulation model consisting of 100 neurons. Neurons in the simulation were divided into two groups: groups A and B. Neurons in group A (50 neurons) had no parent neurons. They were activated by a stimulus. Neurons in group B (50 neurons) were activated by neurons in group A. We simulated two experimental conditions. The simulated data were binary. In condition 1, neurons in group B were activated by one or two neurons in group A. In condition 2, neurons in group B were activated by three or four neurons in group A. Based on the generated ensemble neural activities, we calculated the Spearman correlation coefficient of two neurons' activities within a time window. The Spearman correlation coefficient between a neuron pair is calculated based on the signals of the two neurons in a time window. SIMU has 580 graphs labeled as '0' and 583 microcircuit models labeled as '1', where these labels correspond to conditions 1 and 2, respectively. Details about the microcircuit generation process employed to generate SIMU are described in [65].

The REAL dataset was generated from the reward zone study in [93]. This dataset (id: jz121, 2015-02-21-16h06m) was originally used in [93]. We reanalyzed this dataset. The experimental procedures conducted in that study involving animals were approved by the Columbia University Institutional Animal Care and Ethics Committee. A mouse licked to receive water rewards when entering a fixed reward zone on a treadmill belt. The context considered involved the environment and a set of features during the experiment, including the cues, fabric belts, and nonspatial odor. The animal was trained to learn the reward zone location during context presentation. A reward zone was a 20-cm region in a 2-m long treadmill belt. Two-photon imaging was used to image the CA1 pyramidal layer. There were 21 trials. Details about the animal, virus, surgical procedure, behavioral

training, stimulus presentation, and in vivo two-photon imaging were discussed in [93]. A microcircuit model in this dataset has 420 vertices. The Spearman correlation coefficient between a neuron pair is calculated based on the signals of the two neurons in a time window of 40 frames. The edges are quantified by the Spearman correlation coefficient. The label of a microcircuit model in REAL is a binary behavioral variable that indicates whether or not the mouse is in the reward zone — labels of 1 and 0 correspond to being and not being in the reward zone, respectively. For behavior assessment, only location (whether or not the mouse is in the reward zone) is used and no other behavior variables, such as the animal's velocity, are used. REAL has 1836 graphs labeled as '1' and 16036 graphs labeled as '0'. Because REAL is a highly imbalanced dataset, we report the balanced accuracy for experiments involving REAL. The balanced accuracy in our context is the average recall across both classes (label 1 and label 0).

Microcircuits are generated based on a stream of neuron signals which are generated by the neuron detection algorithm. The neuron detection algorithm analyzes the whole data stream and guarantees that the microcircuit graphs will have a common fixed vertices set.

Examples of microcircuits from both label '1' and label '0' classes in SIMU and REAL are plotted in Fig. 4.2 and Fig. 4.3. The weighted edges are represented as colored connections between neurons with weight values mapped to colors, as shown in the color bars.

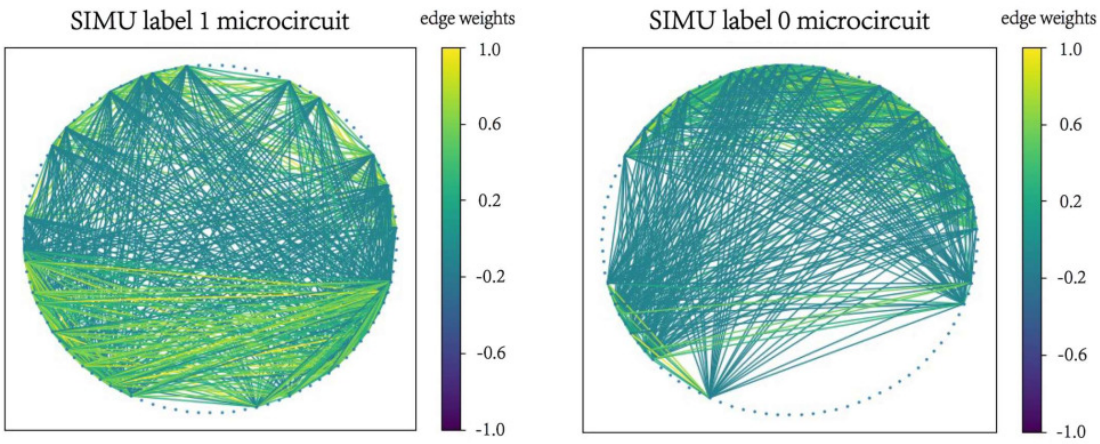

Figure 4.2: Examples of microcircuits from both label '1' and label '0' classes in the SIMU dataset.

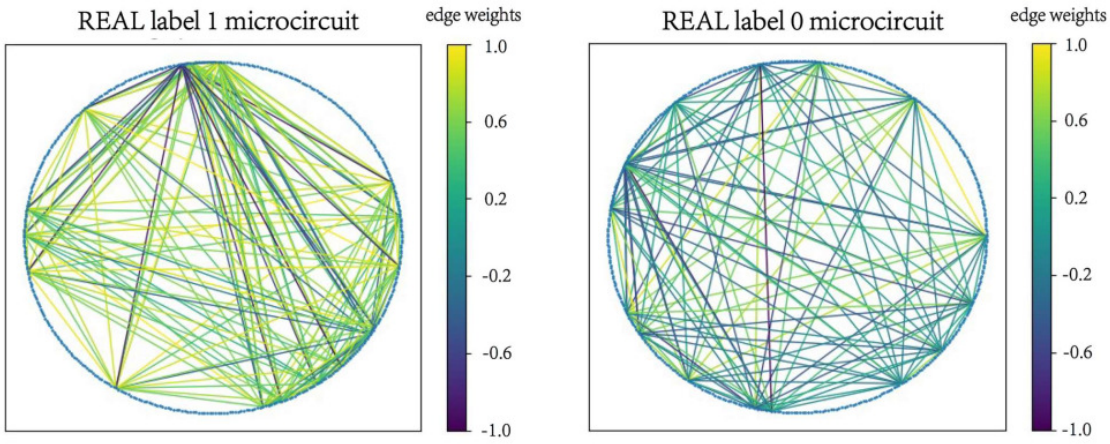

Figure 4.3: Examples of microcircuits from both label '1' and label '0' classes in the REAL dataset.

4.4.5 Experiments Using the Five Vertices Dataset

We perform experiments using the five vertices dataset (see Section 4.3.2) to evaluate the ability of different graph embedding methods to take vertex identities into account. UGEVIA, graph2vec, and PowerGNN are applied in three comparison experiments involving different subsets of the five vertices dataset Δ. As described in Section 4.3.2, each panel $\delta_1, \delta_2, \ldots, \delta_6$ in the five vertices dataset Δ contains 500 identical unweighted graphs

with structure shown in Fig. 4.1. In the first experiment, δ_1 and δ_2 are combined into a set with 1000 graphs. The first experiment aims to test whether a given graph embedding algorithm can effectively distinguish graphs holding unique vertex identities and different numbers of edges. In the second experiment, δ_3 and δ_4 are combined into a set with 1000 graphs. In the third experiment, δ_5 and δ_6 are combined into a set with 1000 graphs. The goal of the second and third experiments is to test whether a graph embedding algorithm can effectively distinguish graphs with identical numbers of edges but with the edges located between vertices having different identities.

Throughout the remainder of this section, results on classification accuracy are reported with the mean and standard deviation from 10-fold cross-validation. The results are tabulated in the format "$mean \pm standard\ deviation$".

The results for the first, second and third experiments are summarized in Table 4.1, Table 4.2, and Table 4.3, respectively. In Table 4.1, Table 4.2, and Table 4.3, we use "$*$" to mark a significant difference between the reference method and the comparison (p-value < 0.05, pairwise t-test, two-tailed). UGEVIA is the reference method. Note that only one value is reported in each row associated with PowerGNN because PowerGNN uses its own internal classifier instead of the downstream classifier that we use with graph2vec and UGEVIA (see Section 4.4.1). According to the results of the first experiment, graph2vec is not able to fully distinguish between graphs from δ_1 and δ_2, whereas UGEVIA and PowerGNN can fully distinguish graphs from these two subsets. From the results of the second and third experiments, we see that graph2vec and PowerGNN are unable to distinguish between graphs from δ_3 and δ_4, nor between δ_5 and δ_6, respectively. On the other hand, the UGEVIA algorithm can fully distinguish between those pairs of

graph subsets. This is because the graphs from δ_3 and δ_4 are indistinguishable without a mechanism for vertex identity awareness, and so are the graphs from δ_5 and δ_6. The graph2vec and PowerGNN methods lack any such mechanism. For graphs from δ_1 and δ_2, although the two groups of graphs contain different numbers of edges, graph2vec treats edges in subsets (a) and (b) similarly, and extracts the same features for all edges. The identical features extracted by graph2vec degrade the final classification performance.

Table 4.1: Results from the first experiment using the five vertices dataset. Performance results for 10-fold cross-validation are reported in the form $mean \pm standard\ deviation$.

	Test accuracy (Graph set 1)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.726±0.043*	0.73±0.038*	0.481±0.093*	0.761±0.047*
UGEVIA	1±0	1±0	1±0	1±0
PowerGNN			1±0	

	Train accuracy (Graph set 1)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	1±0	1±0	1±0	0.791±0.008*
UGEVIA	1±0	1±0	1±0	1±0
PowerGNN			1±0	

4.4.6 Graph Classification with Microcircuit Data

We apply WGEVIA, graph2vec and PowerGNN to the SIMU and REAL datasets. Since PowerGNN and graph2vec were designed for unweighted graphs, we introduce a threshold-based method to convert weighted graphs to unweighted graphs before inputting microcircuit models to these two methods. The conversion method uses a hyperparameter χ, which is a threshold for determining whether or not a weighted edge is removed from the graph

Table 4.2: Results from the second experiment using the five vertices dataset.

Algorithm	Test accuracy (Graph set 2) single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.482±0.057*	0.516±0.063*	0.446±0.037*	0.507±0.052*
UGEVIA	1±0	1±0	1±0	1±0
PowerGNN	0.5±0*			

Algorithm	Train accuracy (Graph set 2) single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	1±0	1±0	1±0	0.625±0.005*
UGEVIA	1±0	1±0	1±0	1±0
PowerGNN	0.5±0*			

Table 4.3: Results from the third experiment using the five vertices dataset.

Algorithm	Test accuracy (Graph set 3) single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.489±0.029*	0.514±0.059*	0.453±0.024*	0.51±0.028*
UGEVIA	1±0	1±0	1±0	1±0
PowerGNN	0.5±0*			

Algorithm	Train accuracy (Graph set 3) single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	1±0	1±0	1±0	0.621±0.011*
UGEVIA	1±0	1±0	1±0	1±0
PowerGNN	0.5±0*			

during the conversion process. In particular, if $W(e)$ represents the weight of an edge e in a microcircuit model, then our weighted-to-unweighted conversion process removes e from the graph if $W(e) \leq \chi$; otherwise, the edge e is retained in the converted graph. In either case, the weight information ($W(e)$) is discarded in the conversion process since the objective is to derive an unweighted graph. We tune the hyperparameter χ to maximize accuracy using Bayesian optimization [86]. The hyperparameter value in our experiments that results from this tuning process is $\chi = 0.196$. Hyperparameter settings for WGEVIA that are used in this experiment are: $n_c = 10$ and $\mu_c = 8$.

The results of this experiment using microcircuit data are summarized in Table 4.4. In Table 4.4, we again use "$*$" to mark a significant difference between the reference method and the comparison (p-value < 0.05, pairwise t-test, two-tailed). Our method WGEVIA is the reference method. From the results, we see that WGEVIA consistently outperforms graph2vec and PowerGNN on both datasets SIMU and REAL. WGEVIA outperforms the other two methods by significant margins for each dataset. This is perhaps not surprising since edge weights are critical to microcircuit analysis, and we expect that we would need to deeply take weight values into account to achieve high accuracy. The results in this experiment help to quantify this intuition, and to validate the effectiveness of WGEVIA in taking edge weights into account.

4.4.7 Evaluation on Design Decisions

As described in Section 4.3, there are three main improvements incorporated on top of graph2vec in our design of the WGEVIA algorithm: (1) adding vertex identity awareness,

Table 4.4: Results of WGEVIA, graph2vec, and PowerGNN for graph classification on microcircuit data.

	Test accuracy (SIMU)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.943±0.018*	0.955±0.014*	0.896±0.027*	0.769±0.035*
WGEVIA	1±0	1±0	1±0	0.997±0.004
PowerGNN	0.788±0.033*			

	Train accuracy (SIMU)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.999±0	0.999±0	1±0	0.83±0.008*
WGEVIA	1±0	1±0	1±0	0.998±0
PowerGNN	0.819±0.021*			

	Test accuracy (REAL)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.677±0.011*	0.686±0.008*	0.55±0.008*	0.513±0.007*
WGEVIA	0.991±0.001	0.993±0.001	0.993±0.001	0.913±0.003
PowerGNN	0.509±0.002*			

	Train accuracy (REAL)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.789±0.006*	0.822±0.024*	0.644±0.003*	0.498±0.001*
WGEVIA	0.998±0.001	0.998±0.001	1±0	0.909±0.002
PowerGNN	0.522±0.002*			

(2) designing special features for zero-degree vertices, and (3) utilizing a multi-channel approach, where a weighted graph is processed as multiple unweighted graphs, and each of the unweighted graphs is referred to as a *channel*. In this section, we aim to evaluate the impact of each of these improvements. The algorithms applied in this evaluation are:

- graph2vec, which is the original graph2vec algorithm without any modification except for weighted-to-unweighted graph conversion being applied to its input, as described in Section 4.4.6;
- Multi-Channel graph2vec (MC-graph2vec), which incorporates Modification 3 described above while not incorporating Modifications 1 and 2;
- Multi-Channel Index Labeled graph2vec, which incorporates Modifications 1 and 3, but not Modification 2;
- WGEVIA, which incorporates all three modifications.

Both the SIMU and REAL datasets are used for the evaluation presented in this section. We perform the experiments with two sets of hyperparameters for the multi-channel approaches: (a) $n_c = 10$ and $\mu_c = 8$, and (b) $n_c = 4$ and $\mu_c = 8$. The first set of hyperparameters is the common set that we used for all other experiments involving multi-channel approaches. The second set of hyperparameters makes the generated embedding vectors less informative. The results with the second set help to provide insight into the robustness of the different methods to variations in hyperparameter settings.

The results from the experiments presented in this section are summarized in Table 4.5 and Table 4.6. For both sets of hyperparameters, we see that Modification 3 alone (MC-graph2vec), Modifications 1+3, and Modifications 1+2+3 (WGEVIA) provide progressively better (monotonically increasing) accuracy compared to the original graph2vec

approach. Moreover, except for the case of the LDA classifier with the REAL dataset, WGEVIA provides very high accuracy even with the lower-quality hyperparameter configuration in the second hyperparameter set. Even for the LDA classifier and the second hyperparameter set, the results provided by WGEVIA on REAL are significantly better than the original graph2vec method.

In Table 4.5 and Table 4.6, we continue using "$*$" to mark a result if it indicates a significant difference (p-value < 0.05, pairwise t-test, two-tailed) between two designs. A given row R is only compared with rows below it, and methods in all rows below R can be regarded as reference methods for the method in the row R. For example, we compare graph2vec to MC-graph2vec with MC-graph2vec being the reference method.

4.4.8 Result Visualization

A distinguishing characteristic of WGEVIA is its multi-channel approach, where individual channels correspond to sets of unweighted graphs that are derived from the given microcircuit dataset D using thresholds that are applied to the edge weights. Each channel contains all of the unweighted graphs that result from a given threshold setting T (see Algorithm 9) when applied to D. A core part of WGEVIA is the iterative application of UGEVIA to each channel.

In this section, we show how visualizations can be used to gain intuitive insight into the contributions of different channels to the overall process in WGEVIA of deriving embedding vectors. For this purpose, we use t-Distributed Stochastic Neighbor Embedding (t-SNE) [96], which enables the visualization of high dimensional data by mapping each

Table 4.5: Results from evaluating the impact of modifications incorporated on top of graph2vec in our design of the WGEVIA algorithm — results for $n_c = 10$ and $\mu_c = 8$.

	Test accuracy (SIMU)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.943±0.018*	0.955±0.014	0.896±0.027*	0.769±0.035*
MC-graph2vec	0.973±0.015*	0.97±0.02*	0.938±0.02*	0.851±0.046*
MC-graph2vec (Index Labeled)	0.998±0.003*	0.998±0.003*	0.997±0.008	0.987±0.002*
WGEVIA	1±0	1±0	1±0	0.997±0.004

	Train accuracy (SIMU)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.999±0	0.999±0	1±0	0.83±0.008*
MC-graph2vec	1±0	1±0	1±0	0.864±0.005*
MC-graph2vec (Index Labeled)	1±0	1±0	1±0	0.993±0.004*
WGEVIA	1±0	1±0	1±0	0.998±0

	Test accuracy (REAL)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.677±0.011*	0.686±0.008*	0.55±0.008*	0.513±0.007*
MC-graph2vec	0.753±0.007*	0.794±0.008*	0.726±0.006*	0.569±0.006*
MC-graph2vec (Index Labeled)	0.97±0.015*	0.986±0.002*	0.97±0.003*	0.647±0.006*
WGEVIA	0.991±0.001	0.993±0.001	0.993±0.001	0.913±0.003

	Train accuracy (REAL)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.789±0.006*	0.822±0.024*	0.644±0.003*	0.498±0.001*
MC-graph2vec	0.998±0.001	0.996±0.004	0.761±0.006*	0.574±0.002*
MC-graph2vec (Index Labeled)	0.998±0.001	0.998±0.001	1±0	0.649±0.007*
WGEVIA	0.998±0.001	0.998±0.001	1±0	0.909±0.002

Table 4.6: Results from evaluating the impact of modifications incorporated on top of graph2vec in our design of the WGEVIA algorithm — results for $n_c = 4$ and $\mu_c = 8$.

	Test accuracy (SIMU)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.943±0.018*	0.955±0.014*	0.896±0.027*	0.769±0.035*
MC-graph2vec	0.961±0.02*	0.963±0.01*	0.924±0.02*	0.798±0.02*
MC-graph2vec (Index Labeled)	0.995±0.006*	0.992±0.008*	0.993±0.005*	0.869±0.02*
WGEVIA	1±0	1±0	1±0	0.988±0.009
	Test accuracy (REAL)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.677±0.011*	0.686±0.008*	0.55±0.008*	0.513±0.007*
MC-graph2vec	0.733±0.01*	0.783±0.006*	0.713±0.003*	0.561±0.006*
MC-graph2vec (Index Labeled)	0.96±0.002*	0.984±0.002*	0.962±0.001*	0.621±0.007*
WGEVIA	0.986±0.002	0.99±0.002	0.986±0.002	0.857±0.005

	Train accuracy (SIMU)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.999±0	0.999±0	1±0	0.83±0.008*
MC-graph2vec	1±0	1±0	1±0	0.787±0.03*
MC-graph2vec (Index Labeled)	1±0	1±0	1±0	0.877±0.06*
WGEVIA	1±0	1±0	1±0	0.998±0.009
	Train accuracy (REAL)			
Algorithm	single-mlp	multi-mlp	SVM-rbf	LDA
graph2vec	0.789±0.006*	0.822±0.024*	0.644±0.003*	0.498±0.001*
MC-graph2vec	0.977±0.007*	0.993±0.005*	0.734±0.004*	0.531±0.002*
MC-graph2vec (Index Labeled)	0.996±0.002	0.997±0.002	0.997±0	0.646±0.005*
WGEVIA	0.996±0.001	0.997±0.001	0.997±0.001	0.868±0.004

data point into a point in a two-dimensional space.

Fig. 4.4 shows visualizations derived using t-SNE for all channels when WGEVIA is applied to the REAL dataset with $n_c = 10$ and $\mu_c = 8$. Plots are shown for each of the 10 channels, where higher channel indices (1–10) correspond to higher threshold values (T values) in Algorithm 9.

For a given plot in Fig. 4.4, each point is derived from the embedding vector generated by UGEVIA for a specific unweighted graph within the channel associated with the plot. The t-SNE approach is used to project the μ_c-dimensional embedding vector associated with each point into the two-dimensional space illustrated in the plot. A point is colored in blue if the corresponding unweighted graph is derived from a 0-labeled element of the REAL dataset, while brown-colored points correspond to 1-labeled elements.

Intuitively, we expect that a channel contributes more useful information to the output of WGEVIA if its embedding vectors are clustered together more tightly for the two different input labels (0 or 1). For example in Fig. 4.4, we see that Channels 1 and 4 have most of the 1-labeled (brown) points clustered together, whereas the 1-labeled points are highly scattered in the plots for Channels 9 and 10. The application of the t-SNE visualization approach, as illustrated in Fig. 4.4, therefore is useful in gaining insight into how different channels contribute to the output embedding vectors produced by WGEVIA, and the utility of the channels in helping a downstream classification task distinguish between different classes in the associated classification problem.

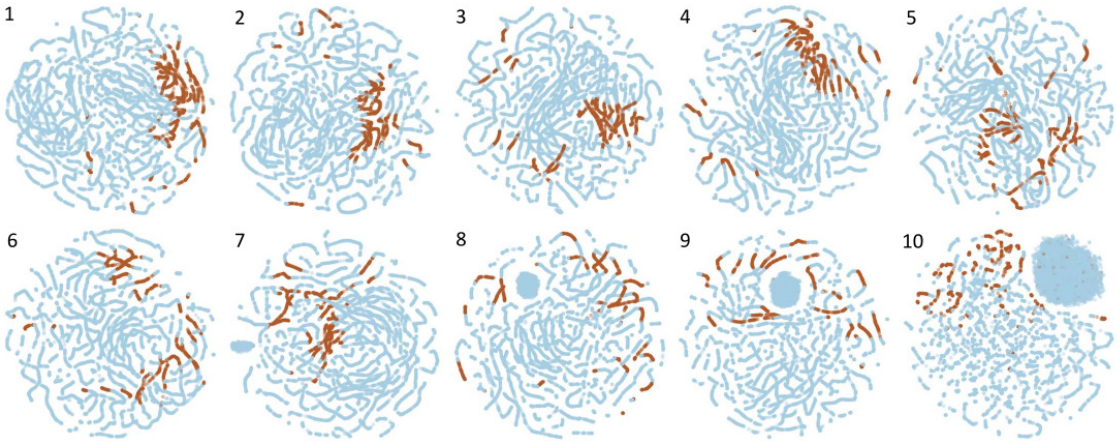

Figure 4.4: Visualizations derived using t-SNE for all channels when WGEVIA is applied to the REAL dataset with $n_c = 10$ and $\mu_c = 8$.

4.4.9 Runtime and Hyperparameter Tuning

As shown in Fig. 4.5 and Table 4.7, we measured the runtimes $T_{featureExtractor}$, $T_{doc2vec}$, and T_{WGEVIA} of the UGEVIA feature extractor `featureExtractor`, doc2vec, and WGEVIA, respectively. The runtime measurements were performed with $featureGenIters = [1, 2, 3, 4, 5, 6]$, $n_c = 10$, and $\mu_c = 8$ on the REAL dataset ($n_G = 17872$, $k = 420$). We found that the product $n_G \times T_{featureExtractor}$ is far less (over 100 times less) than the runtime $T_{doc2vec}$. As $featureGenIters$ varies, T_{WGEVIA} remains almost linearly related to $T_{doc2vec}$: $T_{WGEVIA} \approx \alpha \times T_{doc2vec}$, where $\alpha = 2.4$ based on our experimental device and parallelism settings; we anticipate that the factor α arises mainly due to the overhead of executing UGEVIA in parallel.

Based on Fig. 4.5, the choice of $featureGenIters$ does not significantly impact the embedding vector quality. This is because for the REAL dataset, smaller values of

$featureGenIters$ are sufficient to extract most of the information. For this reason, we are able to get high quality results with a setting of $featureGenIters = 4$ in our experiments.

Fig. 4.6 depicts experimental results with $n_c = 10$, $featureGenIters = 4$, and $\mu_c = [1, 2, \ldots, 10]$. In these results, we see that the classification accuracy increases as μ_c increases and $\mu_c = 8$ is sufficient for WGEVIA to produce high quality results for the REAL dataset. By comparing with Fig. 4.7, we see that the parameter μ_c has less impact on the runtime of WGEVIA than n_c.

Fig. 4.7 depicts experimental results with $\mu_c = 8$, $featureGenIters = 4$, and $n_c = [1, 2, \ldots, 11]$. In these results, the classification accuracy increases as n_c increases. This demonstrates that the proposed multi-channel approach extracts more information when more channels are employed. The setting $n_c = 10$ is sufficient for WGEVIA to produce high quality results for the REAL dataset. To further assess the impact of the n_c parameter, we run experiments with $\mu_c = 4$ and the corresponding results are shown in Fig. 4.8. Here, n_c and μ_c work together to impact on the embedding quality of WGEVIA. In these results, we see that with a smaller value of μ_c, the impact of n_c becomes more significant.

The runtime of the downstream classifiers depends on the number and dimension of the input embedding vectors and the structure of classifier models. In our experiments, the output embedding vector from WGEVIA has a dimension of $n_c \times \mu_c = 80$ since $n_c = 10$ and $\mu_c = 8$. There are $n_G = 17872$ graphs in the REAL dataset and we use 90 percent of them for training. The MLP model has 128 nodes in the first hidden layer ($h_1 = 128$), and 64 nodes in the second hidden layer ($h_2 = 64$). The output layer has 2 nodes corresponding to the two binary classes ($o = 2$). The measured run time results for embedding vectors

generated from the REAL dataset are shown in Table 4.8. All of the runtime results here are averaged over 10 runs.

Table 4.7: Measured runtime of the UGEVIA featureExtractor, doc2vec, and WGEVIA with increasing values of the *featureGenIters* parameter.

featureGenIter	1	2	3	4	5	6
featureExtractor runtime (s)	0.00177	0.00368	0.00503	0.00683	0.00859	0.01011
doc2vec runtime(s)	1047.93	2528.76	5350.58	8791.51	12829.42	16861.66
WGEVIA runtime(s)	2894.96	6381.24	12600.51	20962.63	30241.33	40034.20

Table 4.8: Runtime results for downstream classifiers.

	single-mlp	multi-mlp	SVM	LDA
Train time (s)	130.481	140.642	32.371	0.224
Inference time (ms)	0.0355	0.0361	2.1	0.000547

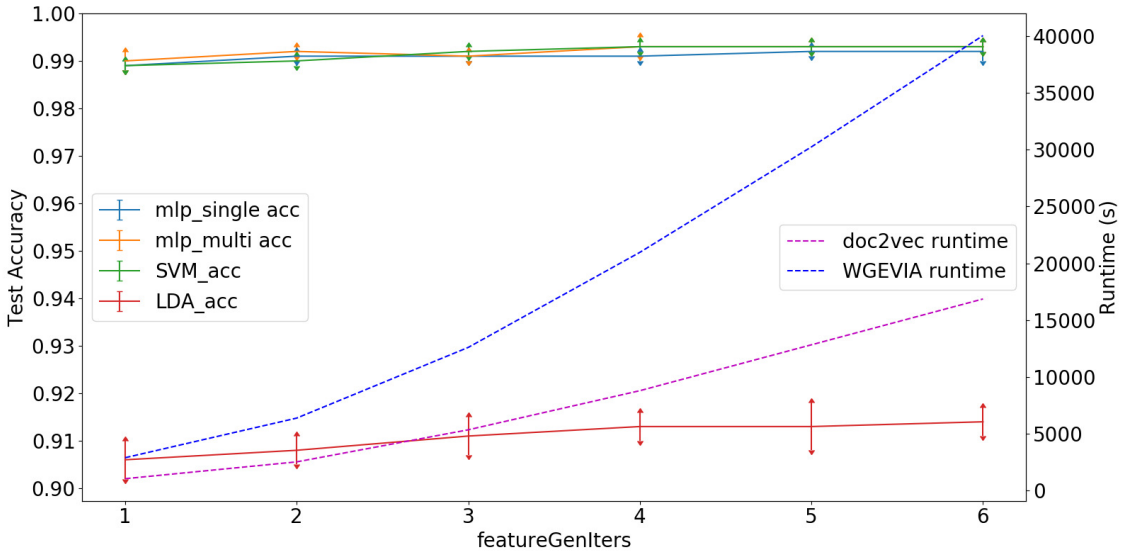

Figure 4.5: Test Accuracy and runtime for different settings of the *featureGenIters* parameter when WGEVIA is applied to the REAL dataset with $n_c = 10$ and $\mu_c = 8$.

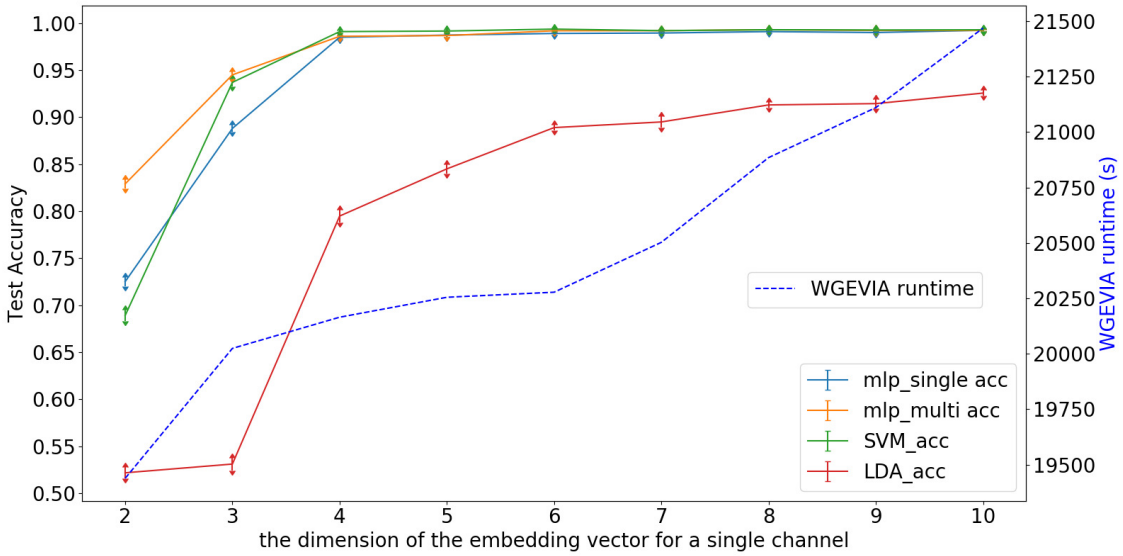

Figure 4.6: Test Accuracy and runtime for different settings of the μ_c parameter when WGEVIA is applied to the REAL dataset with $n_c = 10$ and $featureGenIters = 4$.

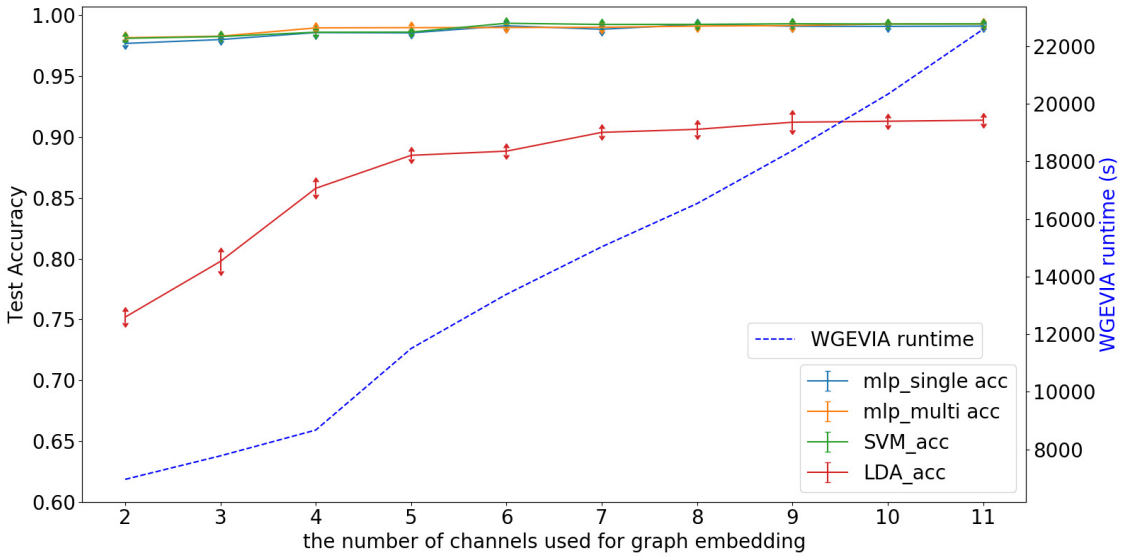

Figure 4.7: Test Accuracy and runtime for different values of the n_c parameter when WGEVIA is applied to the REAL dataset with $\mu_c = 8$ and $featureGenIters = 4$.

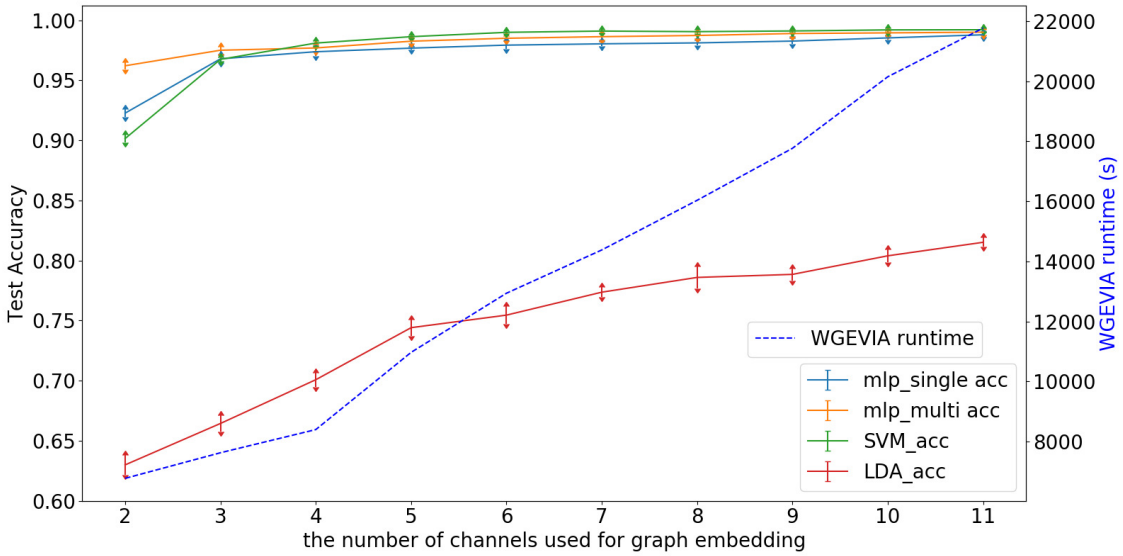

Figure 4.8: Test Accuracy and runtime for different values of the n_c parameter when WGEVIA is applied to the REAL dataset with $\mu_c = 4$ and $featureGenIters = 4$.

4.4.10 WGEVIA Runtime analysis

In this section, we analyze the time complexity of WGEVIA.

From Algorithm 9, the runtime T_{WGEVIA} of WGEVIA is $O(n_c \times T_{UGEVIA})$, where T_{UGEVIA} is the runtime of UGEVIA. On a machine with sufficient resources, the channels can be processed in parallel so that the actual runtime does not necessarily scale linearly with n_c; however, for the analysis in the remainder of this discussion, we do not take into account the potential for acceleration by exploiting parallelism.

From Algorithm 7, the runtime T_{UGEVIA} is $O(n_G \times T_{featureExtractor} + T_{doc2vec})$, where n_G is the set of input graphs, $T_{featureExtractor}$ is the complexity of the UGEVIA feature extractor, and $T_{doc2vec}$ is the complexity of doc2vec. The authors have been unable to find

a complexity analysis of doc2vec in the literature so the expression $T_{doc2vec}$ is left as as a "black box" in the remainder of our analysis.

Finally, from Algorithm 8, the runtime $T_{featureExtractor}$ can be expressed as $O(featureGenIters \times k^2)$, where k is the number of vertices in a given input graph. The term k^2 can be replaced by k for sparse graphs where the number of neighbors for a given vertex is bounded by a constant.

Putting the three pieces discussed above together yields the following time complexity expression for WGEVIA:

$$O(n_c \times ((n_G \times featureGenIters \times k^2) + T_{doc2vec})).$$

4.5 Chapter Summary

In this chapter, we have developed a novel algorithm, called Weighted Graph Embedding with Vertex Identity Awareness (WGEVIA), for embedding graph models of functional microcircuits. Distinguishing characteristics of WGEVIA that make it well-suited for functional microcircuit analysis include its ability to take graph weights and vertex identities into account. We also introduce a novel dataset, called the five vertices dataset, which helps to experiment with and evaluate how effective graph embedding algorithms are at taking vertex identities into account. WGEVIA introduces a novel concept of analyzing functional microcircuits in terms of well-defined collections of simplified (unweighted) graph models. These collections, called channels, can be visualized to gain insight into

how different thresholds on inter-neuron synchrony lead to different levels of behavior discrimination. Through extensive experiments using real and simulated microcircuit data, we demonstrate the effectiveness of the proposed new models and methods for graph embedding, and we show that the proposed methods outperform state-of-the-art graph embedding methods when applied to microcircuit data.

Chapter 5

Conclusion and Future Work

This thesis has explored new machine learning techniques for calcium-imaging-based neural decoding tasks. We have focused on methods for analysis of neural signals using deep neural networks (DNNs), which form an important class of machine learning models that are of increasing interest in neuroscience and neural engineering. We introduced the Greedy inter-layer order with Random Selection (GRS) method for structured pruning of deep neural networks (DNNs), and an accelerated extension of GRS, called Jump-GRS or JGRS, which can efficiently prune large DNNs such as those that are used to analyze collections of numerous biological neurons. The pruning algorithms GRS and JGRS are designed to produce compact DNNs, which facilitate neural decoding for resource-constrained or real-time applications. We demonstrated the effectiveness of GRS and JGRS in extensive experiments on many datasets that are relevant to neural decoding.

In addition to developing methods to make neural decoding more efficient, we have explored algorithms that help to better understand brain activity. To this end, we have introduced Weighted Graph Embedding with Vertex Identity Awareness (WGEVIA), which is a framework for generating vector embeddings of graph-theoretic models for functional microcircuits. Functional microcircuits are of increasing interest in the study of neural dynamics. The embeddings generated by WGEVIA are produced to enable application of important general-purpose machine learning techniques, such as support

vector machines (SVM), multi-layer perceptrons (MLPs), convolutional neural networks (CNNs) and attention models, to interpret microcircuits.

In the remainder of this chapter, we summarize interesting directions for future work to go beyond the developments of this thesis.

5.1 Integration with Synthesis Techniques

Development of synthesis techniques that automatically derive implementations of pruned networks is a useful direction of our ongoing and planned future work that builds upon the NeuroGRS tool. To this end, we have developed the first version of a software tool called NGSynth, which is short for NeuroGRS Synthesis tool. NGSynth takes as input a neural network model implemented in TensorFlow [97], applies NeuroGRS to reduce the size of the input model, and then automatically synthesizes C code that implements the pruned model in the form of a dataflow graph. The generated dataflow graph is constructed using the Lightweight Dataflow Environment (LIDE) [34]. Synthesis as a LIDE implementation is useful because it facilitates application of many types of dataflow-based optimizations — for example, to enhance energy efficiency or derive strategic trade-offs between latency and throughput. A LIDE-based design is also efficient to work with when retargeting an implementation across different platforms.

The current version of NGSynth utilizes a single CPU core and optionally also a single GPU for inference using the synthesized network implementation. Despite using only a single CPU/GPU pair, the run-time of synthesized implementations from NGSynth is sufficiently low to enable end-to-end neural decoding systems to meet the real-time

requirement of 10 Hz on desktop and laptop platforms that we have experimented with. Here, 10Hz refers to the frame rate associated with the datasets that we employed in our experiments (see Chapter 2). However, state-of-the-art devices for acquiring calcium images commonly support higher maximum frequencies, such as $30Hz$ [98]. Incorporating optimizations in the synthesis process of NGSynth to help meet such tighter real-time constraints on moderately-priced hardware platforms is an interesting direction for future work. For example, extending NGSynth to leverage multiple CPU cores in a useful topic under this direction.

5.2 Node-Level Embedding for Microcircuits

WGEVIA, summarized earlier in this chapter, is an unsupervised graph embedding method that generates effective graph-level embeddings for functional microcircuits. WGEVIA can generate embeddings with compact size to represent microcircuit graphs. However, WGEVIA does not retain node-level information from graphs that it embeds; only information about the overall graph is retained. Thus, information related to the contribution of specific neurons to specific analysis tasks cannot be traced back to the neurons. The ability to trace-back neuron-level contributions enables analyses such as studying the relative importance of different neurons during neural decoding.

A useful direction for future work is the extension of WGEVIA to generate microcircuit embeddings that contain neuron-level information. One possible approach to addressing this problem is to form graph-level embeddings with node-level embeddings as the basic units. Downstream model structures, such as self-attention [99] and Hierarchical

Graph Pooling with Structure Learning (HGPSL) [100], can be used to utilize neuron-level information and retain the information in the generated embeddings. In this line of investigation, it may also be useful to apply the graph transformer model (GTM) [101] to microcircuit classification tasks and compare the results to classification using graph embeddings.

Node2vec [74] is a state-of-the-art node-level embedding method. Node2vec regards graph nodes as words and applies word2vec [102] to generate embeddings for nodes. Node2vec is especially useful for node analysis within a single massive graph, such as knowledge graphs and social network graphs. Exploring the joint application of node2vec and WGEVIA is an interesting direction for future work.

5.3 Chatterjee Coefficient of Correlation

The recently-introduced Chatterjee correlation coefficient [103] requires only simple and lightweight computation. The coefficient value lies in the range $[0, 1]$. A value close to 1 means that the two variables are highly related, while values close to 0 indicate a high degree of independence. Thus, the coefficient captures the strength of the relationship between variables. The coefficient also supports meaningful application under small sample sizes, such as 20 samples. These features make the Chatterjee correlation coefficient a good candidate for use as edge weights in microcircuit models. A promising direction for future work is the application of the Chatterjee correlation coefficient to real-time inference from microcircuit models.

Bibliography

[1] Mijail D Serruya, Nicholas G Hatsopoulos, Liam Paninski, Matthew R Fellows, and John P Donoghue, "Instant neural control of a movement signal," *Nature*, vol. 416, no. 6877, pp. 141–142, 2002.

[2] Christian Ethier, Emily R Oby, Matthew J Bauman, and Lee E Miller, "Restoration of grasp following paralysis through brain-controlled stimulation of muscles," *Nature*, vol. 485, no. 7398, pp. 368–371, 2012.

[3] EH Baeg, YB Kim, K Huh, I Mook-Jung, HT Kim, and MW Jung, "Dynamics of population code for working memory in the prefrontal cortex," *Neuron*, vol. 40, no. 1, pp. 177–188, 2003.

[4] Guilhem Ibos and David J Freedman, "Sequential sensory and decision processing in posterior parietal cortex," *Elife*, vol. 6, pp. e23743, 2017.

[5] Kechen Zhang, Iris Ginzburg, Bruce L McNaughton, and Terrence J Sejnowski, "Interpreting neuronal population activity by reconstruction: unified framework with application to hippocampal place cells," *Journal of neurophysiology*, vol. 79, no. 2, pp. 1017–1044, 1998.

[6] Thomas J Davidson, Fabian Kloosterman, and Matthew A Wilson, "Hippocampal replay of extended experience," *Neuron*, vol. 63, no. 4, pp. 497–507, 2009.

[7] Joshua I Glaser, Ari S Benjamin, Raeed H Chowdhury, Matthew G Perich, Lee E Miller, and Konrad P Kording, "Machine learning for neural decoding," *Eneuro*, vol. 7, no. 4, 2020.

[8] AJ Berkhout and PR Zaanen, "Comparison between wiener filtering, kalman filtering, and deterministic least squares estimation," *Geophys. Prospect.;(Netherlands)*, vol. 24, no. 1, 1976.

[9] Kurt Hornik, Maxwell Stinchcombe, and Halbert White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 1106–1114. 2012.

[11] Ronan Collobert and Jason Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[12] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al., "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[13] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, third edition, 2019.

[14] Edward A Lee and Thomas M Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[15] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, 1995.

[16] Xiaomin Wu, Da-Ting Lin, Rong Chen, and Shuvra S Bhattacharyya, "Learning compact dnn models for behavior prediction from neural activity of calcium imaging," *Journal of Signal Processing Systems*, pp. 1–18, 2021.

[17] R. J. Andrews, "Neuromodulation: Advances in the next decade," *Annals of the New York Academy of Sciences*, pp. 212–220, June 2010.

[18] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, pp. 1–18, 2017.

[19] Chunyue Li, Danny CW Chan, Xiaofeng Yang, Ya Ke, and Wing-Ho Yung, "Prediction of forelimb reach results from motor cortex activities based on calcium imaging and deep learning," *Frontiers in cellular neuroscience*, vol. 13, pp. 88, 2019.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[21] Yaesop Lee, Sreenuj Chellath Madayambath, Yanzhou Liu, Da-Ting Lin, Rong Chen, and Shuvra S Bhattacharyya, "Online learning in neural decoding using incremental linear discriminant analysis," in *2017 IEEE International Conference on Cyborg and Bionic Systems (CBS)*. IEEE, 2017, pp. 173–177.

[22] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.

[23] Jonathan Frankle and Michael Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[24] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf, "Pruning filters for efficient ConvNets," *arXiv preprint arXiv:1608.08710*, 2016.

[25] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[26] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[27] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[28] Yihui He, Xiangyu Zhang, and Jian Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1389–1397.

[29] Xavier Suau, Luca Zappella, Vinay Palakkode, and Nicholas Apostoloff, "Principal filter analysis for guided network compression," *arXiv preprint arXiv:1807.10585*, 2018.

[30] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2736–2744.

[31] Song Han, Jeff Pool, John Tran, and William Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[32] Song Han, Huizi Mao, and William J Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[33] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[34] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD framework for modeling and synthesis of signal processing systems," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds., pp. 1–35. Springer, 2017.

[35] Giovanni Barbera, Bo Liang, Lifeng Zhang, Charles R Gerfen, Eugenio Culurciello, Rong Chen, Yun Li, and Da-Ting Lin, "Spatially compact neural clusters in the dorsal striatum encode locomotion relevant information," *Neuron*, vol. 92, no. 1, pp. 202–213, 2016.

[36] "Keras," `https://keras.io/`, 2020.

[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, arXiv:1412.6980 [cs.LG].

[38] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, arXiv:1603.04467v2 [cs.DC].

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[40] Matt W Gardner and SR Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[41] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*. Ieee, 2017, pp. 1–6.

[42] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever, "Deep double descent: Where bigger models and more data hurt," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2021, no. 12, pp. 124003, 2021.

[43] Michael Chan, Daniel Scarafoni, Ronald Duarte, Jason Thornton, and Luke Skelly, "Learning network architectures of deep cnns under resource constraints," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 1703–1710.

[44] Rahul Mishra, Hari Prabhat Gupta, and Tanima Dutta, "A survey on deep neural network compression: Challenges, overview, and solutions," *arXiv preprint arXiv:2010.03954*, 2020.

[45] Fangyu Liu, Saber Meamardoost, Rudiyanto Gunawan, Takaki Komiyama, Claudia Mewes, Ying Zhang, EunJung Hwang, and Linbing Wang, "Deep learning for neural decoding in motor cortex," *Journal of Neural Engineering*, vol. 19, no. 5, pp. 056021, 2022.

[46] Jesse A Livezey and Joshua I Glaser, "Deep learning approaches for neural decoding across architectures and recording modalities," *Briefings in bioinformatics*, vol. 22, no. 2, pp. 1577–1591, 2021.

[47] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel, "Wider or deeper: Revisiting the resnet model for visual recognition," *Pattern Recognition*, vol. 90, pp. 119–133, 2019.

[48] Yihan Jiang, Hyeji Kim, Himanshu Asnani, and Sreeram Kannan, "Mind: Model independent neural decoder," in *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE, 2019, pp. 1–5.

[49] Hyun-Chool Shin, Vikram Aggarwal, Soumyadipta Acharya, Marc H Schieber, and Nitish V Thakor, "Neural decoding of finger movements using skellam-based maximum-likelihood decoding," *IEEE Transactions on Biomedical Engineering*, vol. 57, no. 3, pp. 754–760, 2009.

[50] Feifei Zhao and Yi Zeng, "Dynamically optimizing network structure based on synaptic pruning in the brain," *Frontiers in Systems Neuroscience*, vol. 15, pp. 620558, 2021.

[51] Jing Chang and Jin Sha, "Prune deep neural networks with the modified $l\_\{1/2\}$ penalty," *IEEE Access*, vol. 7, pp. 2273–2280, 2018.

[52] Christos Louizos, Karen Ullrich, and Max Welling, "Bayesian compression for deep learning," *Advances in neural information processing systems*, vol. 30, 2017.

[53] Song Han, Huizi Mao, and William J Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[54] Jonathan R Wolpaw, Niels Birbaumer, William J Heetderks, Dennis J McFarland, P Hunter Peckham, Gerwin Schalk, Emanuel Donchin, Louis A Quatrano, Charles J Robinson, Theresa M Vaughan, et al., "Brain-computer interface technology: a review of the first international meeting," *IEEE transactions on rehabilitation engineering*, vol. 8, no. 2, pp. 164–173, 2000.

[55] Philip M Lewis, Richard H Thomson, Jeffrey V Rosenfeld, and Paul B Fitzgerald, "Brain neuromodulation techniques: a review," *The neuroscientist*, vol. 22, no. 4, pp. 406–421, 2016.

[56] Yaesop Lee, Jing Xie, Eungjoo Lee, Srijesh Sudarsanan, Da-Ting Lin, Rong Chen, and Shuvra S Bhattacharyya, "Real-time neuron detection and neural signal extraction platform for miniature calcium imaging," *Frontiers in computational neuroscience*, vol. 14, pp. 43, 2020.

[57] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller, "Methods for interpreting and understanding deep neural networks," *Digital signal processing*, vol. 73, pp. 1–15, 2018.

[58] G. Barbera, B. Liang, L. Zhang, C. R. Gerfen, E. Culurciello, R. Chen, Y. Li, and D.-T. Lin, "Spatially compact neural clusters in the dorsal striatum encode locomotion relevant information," *Neuron*, vol. 92, no. 1, pp. 202–213, 2016.

[59] Xiaomin Wu, Shuvra S Bhattacharyya, and Rong Chen, "Wgevia: A graph level embedding method for microcircuit data," *Frontiers in Computational Neuroscience*, p. 115, 2021.

[60] J. D. Zaremba et al., "Impaired hippocampal place cell dynamics in a mouse model of the 22q11.2 deletion," *Nature Neuroscience*, vol. 20, pp. 1612–1623, 2017.

[61] Z. Huang, W. Chung, and H. Chen, "A graph model for e-commerce recommender systems," *Journal of the American Society for Information Science and Technology*, February 2004.

[62] X. Yue et al., "Graph embedding on biomedical networks: Methods, applications, and evaluations," *Bioinformatics*, vol. 36, no. 4, pp. 1241–1251, 2020.

[63] R. Chen, Y. Zheng, E. Nixon, and E. H. Herskovits, "Dynamic network model with continuous valued nodes for longitudinal brain morphometry," *NeuroImage*, vol. 155, pp. 605–611, 2017.

[64] H. Cai, V. W. Zheng, and K. C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, 2018.

[65] R. Chen and D.-T. Lin, "Decoding brain states based on microcircuits," in *Proceedings of the IEEE International Conference on Cyborg and Bionic Systems*, 2018, pp. 397–400.

[66] K. Lee, Y. Lee, D.-T. Lin, S. S. Bhattacharyya, and R. Chen, "Real-time calcium imaging based neural decoding with a support vector machine," in *Proceedings of the IEEE Biomedical Circuits and Systems Conference*, 2019, pp. 1–4.

[67] Sarah Feldt, Paolo Bonifazi, and Rosa Cossart, "Dissecting functional connectivity of neuronal microcircuits: experimental and theoretical insights," *Trends in neurosciences*, vol. 34, no. 5, pp. 225–236, 2011.

[68] H. Ko, L. Cossell, C. Baragli, J. Antolik, C. Clopath, S. B. Hofer, and T. D. Mrsic-Flogel, "The emergence of functional microcircuits in visual cortex," *Nature*, vol. 496, pp. 96–100, 2013.

[69] Shigeyoshi Fujisawa, Asohan Amarasingham, Matthew T Harrison, and György Buzsáki, "Behavior-dependent short-term assembly dynamics in the medial prefrontal cortex," *Nature neuroscience*, vol. 11, no. 7, pp. 823, 2008.

[70] Giovanni Barbera, Bo Liang, Lifeng Zhang, Charles R Gerfen, Eugenio Culurciello, Rong Chen, Yun Li, and Da-Ting Lin, "Spatially compact neural clusters in the dorsal striatum encode locomotion relevant information," *Neuron*, vol. 92, no. 1, pp. 202–213, 2016.

[71] Ho Ko, Sonja B Hofer, Bruno Pichler, Katherine A Buchanan, P Jesper Sjöström, and Thomas D Mrsic-Flogel, "Functional specificity of local synaptic connections in neocortical networks," *Nature*, vol. 473, no. 7345, pp. 87–91, 2011.

[72] Bruno B Averbeck, Peter E Latham, and Alexandre Pouget, "Neural correlations, population coding and computation," *Nature reviews neuroscience*, vol. 7, no. 5, pp. 358–366, 2006.

[73] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: online learning of social representations," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014, pp. 701–710.

[74] Aditya Grover and Jure Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[75] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[76] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[77] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the International Conference on Machine Learning*, 2014, pp. 1188–1196.

[78] Bijaya Adhikari, Yao Zhang, Naren Ramakrishnan, and B Aditya Prakash, "Sub2vec: Feature learning for subgraphs," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2018, pp. 170–182.

[79] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, no. Sep, pp. 2539–2561, 2011.

[80] Pinar Yanardag and SVN Vishwanathan, "Deep graph kernels," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1365–1374.

[81] Leonardo Gutiérrez-Gómez and Jean-Charles Delvenne, "Unsupervised network embeddings with node identity awareness," *Applied Network Science*, vol. 4, no. 1, pp. 82, 2019.

[82] B. B. Averbeck, P. E. Latham, and A. Pouget, "Neural correlations, population coding and computation," *Nature Reviews Neuroscience*, vol. 7, pp. 358–366, 2006.

[83] E. Zohary, M. N. Shadlen, and W. T. Newsome, "Correlated neuronal discharge rate and its implications for psychophysical performance," *Nature*, vol. 370, pp. 140–143, 1994.

[84] J. H. McDonald, *Handbook of Biological Statistics*, Sparky House Publishing, third edition, 2014.

[85] R. Rivest, "The MD5 message-digest algorithm," Tech. Rep., MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.

[86] M. Pelikan, D. E. Goldberg, and E. E. Cantú-Paz, "BOA: the Bayesian optimization algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999, pp. 525–532.

[87] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory*, 1992, pp. 144–152.

[88] Suresh Balakrishnama and Aravind Ganapathiraju, "Linear discriminant analysis-a brief tutorial," in *Institute for Signal and information Processing*, 1998, vol. 18, pp. 1–8.

[89] B. Rozemberczki, O. Kiss, and R. Sarkar, "Karate Club: An API oriented open-source Python framework for unsupervised learning on graphs," arXiv:2003.04819v3 [cs.LG], 2020.

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

[91] S. Han, C. Qubo, and H. Meng, "Parameter selection in SVM with RBF kernel function," in *Proceedings of the World Automation Congress*, 2012.

[92] Payam Refaeilzadeh, Lei Tang, and Huan Liu, *Cross-Validation*, pp. 532–538, Springer US, Boston, MA, 2009.

[93] Jeffrey D Zaremba, Anastasia Diamantopoulou, Nathan B Danielson, Andres D Grosmark, Patrick W Kaifosh, John C Bowler, Zhenrui Liao, Fraser T Sparks, Joseph A Gogos, and Attila Losonczy, "Impaired hippocampal place cell dynamics in a mouse model of the 22q11. 2 deletion," *Nature neuroscience*, vol. 20, no. 11, pp. 1612, 2017.

[94] Robert Gütig and Haim Sompolinsky, "The tempotron: a neuron that learns spike timing–based decisions," *Nature neuroscience*, vol. 9, no. 3, pp. 420–428, 2006.

[95] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris, et al., "Simulation of networks of spiking neurons: a review of tools and strategies," *Journal of computational neuroscience*, vol. 23, no. 3, pp. 349–398, 2007.

[96] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[97] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[98] Andrea Giovannucci, Johannes Friedrich, Matt Kaufman, Anne Churchland, Dmitri Chklovskii, Liam Paninski, and Eftychios A Pnevmatikakis, "Onacid: Online analysis of calcium imaging data in real time," *Biorxiv*, p. 193383, 2017.

[99] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[100] Zhen Zhang, Jiajun Bu, Martin Ester, Jianfeng Zhang, Chengwei Yao, Zhi Yu, and Can Wang, "Hierarchical graph pooling with structure learning," *arXiv preprint arXiv:1911.05954*, 2019.

[101] Vijay Prakash Dwivedi and Xavier Bresson, "A generalization of transformer networks to graphs," *arXiv preprint arXiv:2012.09699*, 2020.

[102] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[103] Sourav Chatterjee, "A new coefficient of correlation," *Journal of the American Statistical Association*, vol. 116, no. 536, pp. 2009–2022, 2021.