Sparse deep neural networks for modeling physical and biological systems

Olivia T. Zahn

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2023

Reading Committee:
J. Nathan Kutz, Chair
Tom Daniel
Paul A. Wiggins

Program Authorized to Offer Degree:
Physics

University of Washington

Abstract

Sparse deep neural networks for modeling physical and biological systems

Olivia T. Zahn

Chair of the Supervisory Committee:
J. Nathan Kutz
Department of Applied Mathematics,
Department of Electrical and Computer Engineering

Characterizing the relationship between network performance and its parameters is an active area of investigation within the fields of deep learning and complex network science. In this thesis, sparse deep neural networks (DNNs) are explored as tools for modeling physical and biological systems and the functional complexity of trained, sparse DNNs is characterized using tools from complex network theory, namely network motif theory. Neural network pruning is used to find a sparse computational model for controlling a biological motor task. Using a sequential, magnitude-based pruning algorithm, as many as 93% of network parameters can be removed from a DNN without compromising performance. Through quantifying the distribution of network motifs in the remaining sparse network, we visualize the change in network complexity throughout the pruning process and across networks. We find that, despite the random initialization of network parameters before training, enforced sparsity causes DNNs to converge to similar non-random connectivity patterns as characterized by their network motif significance. Furthermore, we find that network motifs become more significant through the sparsification process. In summary, we find that through using a simple selection process to determine parameter importance, the number of parameters in a trained DNN model can be reduced dramatically and results in a non-trivial network topology.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

DEEP NEURAL NETWORK: (DNN) a machine learning model characterized by many parameters connected via simple matrix operations.

TRAINING: an optimization procedure that finds the best set of parameters that minimize the difference between the ground truth of the data set and the model predictions.

PERCEPTRON MODEL: a neural network with one input, one output, and two parameters defining their functional relationship, the weight, w and the bias, b.

BATCH: a subset of the training data over which a single iteration of gradient descent and backpropagation are implemented.

GRADIENT DESCENT: iterative algorithm for finding a minimum of a high-dimensional function; one of the core algorithms in deep learning.

BACKPROPAGATION: algorithm for propagating the error of a DNN back through the network which enables the parameter updates in gradient descent.

OVER-FITTING: occurs when a machine learning model too closely learns the features of a data set and consequently lacks generalization capabilities.

GENERALIZATION: refers to a machine learning model's ability to correctly make predictions from data outside its training domain.

INFERENCE: or "forward-pass;" input is passed through model to get a prediction.

SPARSE NEURAL NETWORK: a DNN where many parameters are zero.

PRUNING: a DNN sparsification method where parameters are systematically removed after training based on some predefined criterion.

NETWORK MOTIF: a statistically significant sub-graph within a larger graph.

ACKNOWLEDGMENTS

During my time here, I have had many mentors that have helped me get to the finish line. Thank you to my advisors, Nathan Kutz and Tom Daniel. Your support and guidance throughout this program helped me get to where I am today. Thank you to the rest of my supervisory committee, Paul Wiggins, Shih-Chieh Hsu, and Marine Denolle. Thank you to Daniel Shea and Michael J. Henry for your mentoring during my internships. I am so grateful to have been given the opportunity to learn from you all. I would also like to thank the NISC MURI team for the opportunity to contribute to and learn from a diverse group of scientists.

To my parents, your love, your support, and your values, have lead me to where I am today. I inherited your shared curiosity and affinity for wonders big and small. Your example and unrelenting encouragement has inspired me to always strive after I want and to work to build the life that I want to live.

To Dani, in a field and program in which I've always felt a bit out of place, I found a friendship that I will cherish for the rest of my life. You are inspiring, encouraging, and courageous. I have never had a peer that I trust enough to make mistakes with. Thank you for the trust, confidence, and friendship I gained through knowing you.

To Leah, if you had not paved the way first, I would not be where I am today. I will always be grateful for how you built my confidence when I was a young girl and made me believe that I could do anything I wanted in life. I admire you so much and I know that for the rest of my life I will always look to you for inspiration, guidance, and unconditional love.

To Kyle, these years have been an adventure for the both of us. Through the hardship and loss, you have always supported this pursuit. Your faith in me has always stood in stark

contrast to all of my self-doubt. Your love helped me celebrate big moments and overcome the dark times. Thank you for always building me up. Thank you for always being there for me when I needed you. I could not have done it without you.

DEDICATION

for me

Chapter 1
INTRODUCTION

1.1 Motivation

Deep neural networks are becoming popular tools for modeling in the physical and biological sciences. Additionally, they are being deployed in wide-ranging engineering applications. The success of deep neural networks is due primarily their size (i.e., number of parameters) which provides them with a high capacity for learning complex relationships between inputs, parameters, and outputs. Enabled by modern parallel computing, deep neural networks can model once intractable problems ranging from high-dimensional fluid flow to human language recognition and generation.

While the scope of the problems solvable by deep learning is exciting, there are trade-offs to using such tools. Deep neural networks are costly in terms of dollars, time, materials, and environmental and human impact. There are also less tangible costs. The computational complexity of deep neural networks gives them the capacity to model high-dimensional and non-linear systems, but as models, they are often difficult to interpret. Modern deep learning architectures contain hundreds of billions of parameters [79]. In these and in more modestly sized networks, knowing how each parameter contributes to the global function of the network is impossible. Reducing the size of a deep neural network may make it more interpretable, but doing so reduces its capability to model complex problems.

Additionally, there is the question of whether such models need all of those parameters in the first place. Models in most all other scientific disciplines, but particularly in physics, have very few parameters and are wildly successful at simulating a large variety of dynamics. In most models in physics, the parameters are clearly interpretable or at the very least have a name. More importantly, physics models are distinct from deep learning models in their

application. Physicists model by reducing the complexity of a system to the point at which it can be modeled with analytical or numerical techniques. In contrast, practitioners of deep learning increase the complexity of their model until it is flexible enough to simulate the desired system.

The advantage of the deep learning approach is that one can model a complex system without necessarily understanding its underlying dynamics. From fluid dynamics to vision, there are many systems for which it is impossible to find a closed-form solution. Consequently, deep learning has found its success in these and in other systems. The deep learning literature is dominated by applications of novel and time-tested deep learning architectures to increasingly complex problems. There are few studies relating the function of deep neural networks to their individual components or connectivity structure (sparse or otherwise).

One approach to the problems of complexity, function, and interpretability in deep learning is to treat them as complex networks. Complex networks are networks with a non-trivial topology. Fully-connected deep neural networks do indeed have a trivial network topology (i.e., all nodes are connected in a feed-forward fashion). However, the connectivity of deep neural networks can be made non-trivial through through model sparsification. Sparse deep neural networks can achieve comparable performance to fully-connected networks and may have a more interpretable architecture. In this thesis, sparse deep neural networks are explored as tools for modeling physical and biological systems. Concepts from complex network theory, namely network motifs, are then used to explore the connectivity structure of sparse deep learning models.

1.2 Background

Deep neural networks (DNNs) are a class of machine learning models with a network structure that was originally inspired by Hubel and Wiesel's study of the visual cortex [50, 88]. The networked structure of DNNs allows for complex interactions between input variables and makes them powerful modeling tools. DNNs have been successful in modeling a variety of complex tasks from computer vision to natural language processing. They are particularly

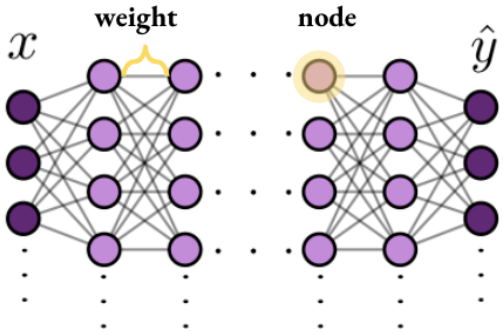


Figure 1.1: Diagram of simple, fully-connected, feed-forward deep neural network.

useful in a context in which the mechanism of decision making is unknown or too complex to write a deterministic algorithm for. For example, in computer vision, the relationship between the visual features of an image and the determination of what is in that image is either too complex or unknown entirely. Not only is the relationship between image feature and decision complex, but there can also be multiple features that affect a DNN's output. The capacity of a DNN to learn a set of features is determined by its architecture (i.e., number of layers, number of nodes in each layer, and type of layer). It is generally accepted that the more layers a DNN has, the higher its capacity for learning sets of complex features.

DNNs have been used to model a multitude of systems. Thanks to their architectural flexibility, size, and modern computing capabilities, DNNs can model a variety of complex problems including multi-variate regression, classification, segmentation, object detection, and language generation [61, 33, 70, 10].

DNNs are capable of modeling complex systems with many interdependent relationships between inputs, parameters, and outputs, but mathematically they have a simple network structure. At their core, DNNs are a series of matrix operations (mainly matrix multiplication) with non-linearity introduced via non-linear activation functions.

$$\hat{y} = \sigma_n(\mathbf{A_n}...(\sigma_1(\mathbf{A_1}x)) \tag{1.1}$$

Deep neural networks are often depicted as directed graphs with nodes arranged in layers. The edges of the graph are the weights or parameters of the model and the nodes are the activated outputs of each layer. Equation 1.1 is the mathematical representation of a simple, n-layer, feed-forward DNN (as depicted in Figure 1.1). In Figure 1.1 and Equation 1.1, x is the input to the network (e.g., multidimensional vector, image, encoded text, etc.) and y is the output or prediction. $A_1, ..., A_n$ are the weight matrices of the network and $\sigma_1, ..., \sigma_n$ are the non-linear activation functions (e.g., hyperbolic tangent, ReLU, etc.). Each layer of a DNN also often contains a bias term, which is a constant (not determined by the previous layer) that is added to the output of a layer before activation. Mathematically, the bias term can be absorbed into the weight matrix, A_i.

The modularity of DNN components means one can construct large and complex architectures with these simple building blocks. The number of layers and nodes per layer are limited by compute resources and the algorithms used to train the network. *Training* a DNN is an optimization procedure that finds the best set of parameters that minimize the difference between the ground truth of the data set and the model predictions. During training, the parameters of the network (weights and biases) are updated based on its performance according to some predefined loss function. Training is typically achieved by way of two algorithms: a variation of gradient descent and the backpropagation algorithm [84]. These two algorithms make deep learning possible and a simple example of their derivation will be shown in Section 1.2.1.

1.2.1 Deriving gradient descent and backpropagation for a simple linear model

When the loss landscape of a model is low-dimensional and convex, the global minimum of the loss function can be found analytically. Strictly convex functions (such as in the following example) only have one minimum (as opposed to ones with multiple local minima) which

makes finding the set of parameters to minimize the function simple. However, DNNs are have a high-dimensional loss landscape and are non-convex. Additionally, finding the global minimizer will cause the DNN to overfit to the data set [41]. Over-fitting will be discussed in more detail in Section 1.2.2. Therefore, to fit a DNN to a data set, DNNs are trained using optimization techniques and the backpropagation algorithm [84].

In summary, the model parameters are adjusted by the gradient of the loss function with respect to the model parameters. Thanks to their layer-wise structure and simple mathematical operations (i.e., matrix multiplication and non-linear activation function) the gradients are easily found using the chain rule. This means that updates for each parameter can be calculated locally as opposed to globally in the case of analytical minimization. This feature reduces computing costs and speeds up the training of DNNs. In this section, gradient descent and backpropagation will be explained using a simple linear regression as an example.

The choice of loss function is determined by the type of model and problem. In classification problems, where the prediction of the model is binary or multi-class, a cross-entropy loss function is used to minimize the cross-entropy between the predicted and true distributions (i.e., the model predictions and the true values). In regression problems, we often calculate the L^1 or L^2 distance between the prediction and ground truth. For example, in a simple linear regression problem, we can use the mean-squared error function as our loss function.

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1.2}$$

In Equation 1.2, y_i are the true values and $\hat{y}_i$ are the predictions of the model. With this loss function, we are minimizing the L^2 distance between the predictions and the true values. While this loss function can be used for many systems of varying complexity, here we will consider a simple linear model to define the relationship between input and output. A very simple neural network that can model this relationship is the *perceptron model*. Conceptually, the perceptron is a neural network with one input, one output, and two parameters defining their functional relationship, the weight, w and the bias, b. In the case with no non-linear

activation, this neural network and the following optimization scheme represents a simple linear regression. Mathematically, our model is

$$\hat{y}_i = wx_i + b \tag{1.3}$$

We don't need a neural network to solve this problem (we can solve it analytically by minimizing the loss function). However, if we were to use optimization, the best linear model can be found through an iterative process of updating the weight and bias using the derivative of the loss with respect to w and b.

$$w_{new} \leftarrow w_{old} - \eta \frac{\partial L}{\partial w_{old}} \tag{1.4a}$$

$$b_{new} \leftarrow b_{old} - \eta \frac{\partial L}{\partial b_{old}} \tag{1.4b}$$

In Equations 1.4a and 1.4b, η is the learning rate (a constant that controls how much the parameters are updated in a given step). The derivative of our loss function with respect to the slope and intercept can be found using the chain rule with Equations 1.2 and 1.3.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} \tag{1.5a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} \tag{1.5b}$$

We can find these derivatives using our equations for the loss function and the model function (Equation 1.3).

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{\partial}{\partial \hat{y}_i} \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \frac{-2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) \tag{1.6}$$

The derivative of $\hat{y}_i$ with respect to w is

$$\frac{\partial \hat{y}_i}{\partial w} = \frac{\partial}{\partial w}(wx_i + b) = x_i. \tag{1.7}$$

Therefore, the derivative of $\frac{\partial L}{\partial w}$ is

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = \frac{-2x_i}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i). \tag{1.8}$$

The derivative with respect to the bias can be found in the same way. The derivative of $\hat{y}_i$ with respect to b is

$$\frac{\partial \hat{y}_i}{\partial b} = \frac{\partial}{\partial w}(wx_i + b) = 1. \tag{1.9}$$

Therefore,

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = \frac{-2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i). \tag{1.10}$$

On each update iteration, these derivatives are found and the parameters of the model are updated according to Equations 1.5a and 1.5b. Equations 1.5a and 1.5b are known as the backpropagation equations. This simple example is for the perceptron model but we can extend the same steps to a n-layer neural network.

DNN optimization is an iterative process. The above example is one where every update made to the model parameters is based off of the model's performance on a single sample. However, in practice, updates are usually made after a *batch* of data is evaluated. A batch is a subset of the total training data. After we have iterated over all of the batches that make up the entire training set (known as an epoch), the process repeats. DNNs are optimized in this fashion until the training loss is minimized. However, the model can become overfit if the training loss is minimized too extremely, a phenomenon described in the next section, and mitigated by validation and regularization (Section 1.2.4).

1.2.2 Bias-Variance Trade-off

Local and parallelizable parameter updates, enabled by the backpropagation algorithm, is one of the main reasons why DNNs have seen such success as modeling tools. Limited only by computing capabilities, DNNs have grown enormous, with the largest DNNs able to model complex systems such as language and image generation. However, DNNs are

statistical models and are limited by statistical principles. One such principle is the bias-variance trade-off which explains the relationship between model complexity and predictive and generalization capability [41]. The bias-variance trade-off is the central problem of supervised learning where one wants a model that captures features within the training set, but is also able to generalize to data points outside of the training set.

Mathematically, the bias-variance trade-off can by understood by the bias-variance decomposition of the expected model error on an unseen sample, x_0.

$$\mathbb{E}[L(y_0, f(x_0)|x_0] = \sigma^2 + \text{Bias}^2(f(x_0)) + \text{Var}(f(x_0)) \tag{1.11}$$

In Equation 1.11, $f(x_0)$ is the model evaluated with an unseen data point x_0 and y_0 is the ground truth output. The expectation of the loss evaluated at this point can be decomposed into three components, the natural variance, the bias term, and the variance term. The natural variance or noise, σ^2, is the variance inherent to the system and cannot be mitigated during training (i.e., it does not depend on the model). The bias and variance terms are dependent on the learned model. The bias term is the square of the expectation value of the difference between the ground truth or true mean and the predicted value. The bias of a model encapsulates how well the model learned the features of the training data. The bias error can typically be reduced by further training or expanding the expressivity of the model by adding more parameters. The variance term is the expected value of the variance between the prediction and its expectation value or mean. The variance term encapsulates how closely a model will predict a value to its mean.

The complexity of a DNN model is determined in part by the number of parameters (weights and biases) that make up the layers of the network. It is also determined by the size and complexity of the data set used to train the network as well as the duration it is trained. The trade-off in relation to model complexity is depicted in Figure 1.2. As model complexity increases, the bias error of the model decreases, because the model is more able to closely match the features in the training data. However, the variance error increases as the model complexity increases because the model becomes more over-fit to the training data.

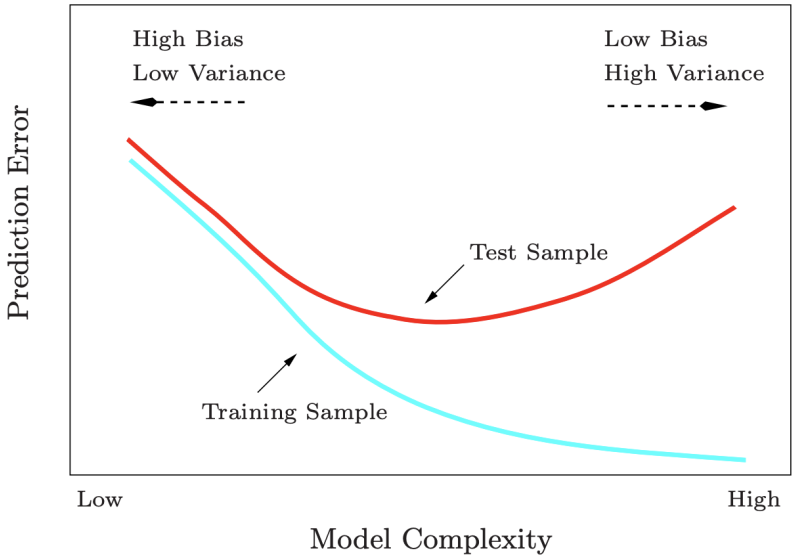


Figure 1.2: The bias-variance trade-off from "The Elements of Statistical Learning" by Hastie, Tibshirani, and Friedman [41].

Over-fitting to the training data means a model's complexity is so high that while it captures features of the training data well, it is unable to generalize to never-before-seen data. Due primarily to their size and number of parameters, DNNs are low bias, high variance models and are prone to over-fitting.

1.2.3 Supervised and unsupervised learning

The above linear regression problem is an example of supervised learning. There are two broad classes of machine learning algorithms: *supervised* and *unsupervised learning*. Supervised learning requires every data point, or input to the model, to have a true label or output. For example, in the above regression problem, each data point x_i has an associated output y_i. The model predicts $\hat{y}_i$ and is trained to minimize the error between $\hat{y}_i$ and y_i. Any data set with true labels, outputs, or annotations can be trained in a supervised manner.

Unsupervised models train without a true label or output and rely only on the statistics

of the data set to learn a good model. An example of unsupervised learning is clustering. In clustering algorithms, decisions about a data point are made based off of which cluster they most closely belong to. For example, one could use the *k-means* clustering algorithm (with a k value of 2) to cluster images of cats and dogs. The model would make decisions about the data set based on a data point's proximity to a cluster mean (i.e., images either belong to the "cat" or "dog" cluster).

One class of machine learning algorithm is not better than the other. Rather, the class of algorithm is chosen depending on the nature of the problem. There are many examples of systems where the data has no labels or annotations. There are also applications where labels exist for some data and not for others (especially common in medical applications). Additionally, the goal of the researchers and engineers may determine whether a supervised or unsupervised approach is appropriate (e.g., choosing between tasks such as classification or synthetic data generation). Furthermore, there are many deep learning algorithms that do not fall into one of these categories specifically. Some examples of other classes include semi-supervised, self-supervised, and reinforcement learning. Detailed discussion of these algorithm classes is beyond the scope of this thesis.

1.2.4 Validation, testing, and regularization

Whether supervised or unsupervised, machine learning (including deep learning) is a data-driven modeling approach. In fact, these techniques require very large amounts of data to be successful in modeling complex systems in all domains. It is common practice to divide a data set into three different groups: train, validation, and test. In general, each of these data sets are kept distinct from one another (some techniques such as $k-means$ cross-validation or ensemble modeling require intentional mixing of train and validation sets). The train data are used to train the model. In other words, the train data set is used in the optimization process (see example in Section 1.2.1). The validation data is also used while training the model, but it is not used in optimization. After each training epoch, *inference* is run on the validation data set. Inference is running the data through the model (forward-pass),

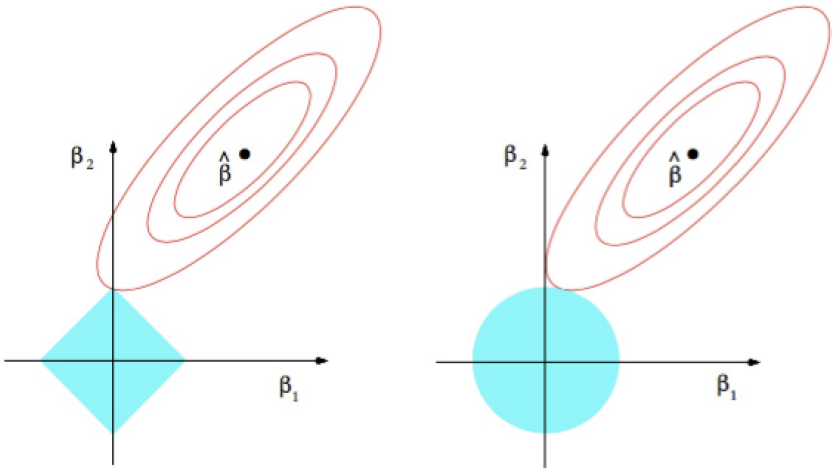


Figure 1.3: Geometric interpretation of L1 and L2 regularization from "The Elements of Statistical Learning" by Hastie, Tibshirani, and Friedman [41].

evaluating performance with the loss function, but not updating the model parameters. Validation data provides a way to monitor the performance of the model during training and helps avoid over-fitting. The test data is withheld from the training process entirely. The test data is used to evaluate the performance of the model after training. Using the test data during training or validation will reduce the generalization capabilities of the model.

Regularization is another tool for reducing over-fitting and increasing a model's ability to generalize. Regularization takes the form of a criterion on the model that adjusts the parameters of the model by some metric other than simply best fit (e.g., small weight magnitudes, sparse connectivity). Regularization can be accomplished by adding terms to the loss function or through a separate operation during optimization.

Two common regularization terms are the L^2 and L^1 terms. L^2 regression is commonly referred to as *ridge* regression and L^1 regression is referred to as *lasso* regression. Mathematically, the L^2 and L^1 regularization terms are added to the loss function. The optimization functions in both ridge and lasso regression are written in Equations 1.12 and 1.13 [41].

$$\hat{\beta}^{\text{ridge}} = \underset{\beta}{\text{argmin}} \left\{ \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\} \tag{1.12}$$

$$\hat{\beta}^{\text{lasso}} = \underset{\beta}{\text{argmin}} \left\{ \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \|\beta_j\| \right\} \tag{1.13}$$

In Equations 1.12 and 1.13, $\hat{y}_i = \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j$ where β_j and β_0 are the weights and bias of the model, respectively. p is the number of weights in the model. By finding the optimal parameters using these regularization terms, one can train a DNN to optimal performance while enforcing the desired criterion.

In the case of ridge regression, minimizing Equation 1.12 encourages the parameters of the model to be near zero. This is depicted geometrically in Figure 1.3. In lasso regression, the regularization term encourages the parameters *to be* zero. Therefore, lasso regression is a strategy for enforcing sparsity in the model.

DNNs can also be regularized through sparsification techniques such as neural network pruning. Network sparsification techniques will be discussed more deeply in Chapter 2. There may seem like a lot of choices when building and training a DNN, but they can be broadly categorized into architecture choice and optimization scheme. Both are dependent on the task the scientist wants to model.

1.3 Introduction to this Thesis

Trained sparse deep neural networks can achieve comparable performance to fully-conneceted networks. Additionally sparse DNNs may have a more interpretable architecture. In the following chapters, sparse deep neural networks are explored as tools for modeling physical and biological systems. Chapter 2 explores the history of sparsification in deep neural networks with a particular focus on neural network pruning methods. Chapter 3 describes pruning and how it is implemented in this thesis, and describes two pruning experiments. In Chapter 4, pruning is used to discover the most sparse optimal DNN capable of modeling a biomechanical control task. Chapter 5 explores the structure of the networks from Chapter 4

using tools from network theory. Specifically, the connectivity structure of trained sparse, DNNs is characterized by the distribution of network motifs across the graph. Finally, Chapter 6 provides conclusions and proposals for future research.

Chapter 2
LITERARY REVIEW

In the following sections, the literature on deep neural network sparsification, with a particular focus on neural network pruning, will be discussed. Following that is a discussion of the study of DNNs using tools from complex network theory, namely network motif theory.

2.1 Deep neural network sparsification

Deep neural networks are usually constructed and trained with a dense architecture which gives them the flexibility to approximate a diversity of complex functions. Modern parallel computing and the backpropagation algorithm enables the efficient training of such networks [84]. The sparsification of such DNNs has been motivated by the effect over-fitting has on generalization performance [41]. The literature discussing the relationship between network size, sparsity, and generalization capability is vast; beginning with the conception of machine learning, it continues to be an active field of investigation to this day [45, 8, 7, 107].

To a lesser extent, DNN sparsification has also been motivated by the desire to reduce their high computational and memory footprint (i.e., the desire to implement DNNs on small portable devices such as smart phones and laptops). Reducing computational costs is better achieved through more efficient architectures (such as MobileNet for computer vision tasks [83]) thanks to the efficiency of parallel computing [84]. However, many papers on DNN sparsification and pruning also report results on common DNN efficiency metrics [11]. To quantify network efficiency, authors use the floating-point operations per second (FLOPs) required for inference. Nearly all pruning papers use the number of parameters pruned as a metric for compression [60, 11, 28, 68, 100, 36, 54, 67]. However, FLOPs is a poor representation real-world performance, memory footprint, and power consumption

and most performance metrics are evaluated on simple deep learning tasks such as image classification [11]. The effect of neural network pruning and sparsification, in general, has on DNN efficiency in more complex problems (such as object detection, segmentation, etc.) has yet to be explored.

DNN sparsification techniques can be split into two general categories: sparsification via regularization and sparsification via pruning. Sparsification via regularization involves encouraging sparsity through the error minimization process and is usually achieved by adding a term to the loss function. Sparsification via pruning is not achieved during optimization, but rather parameters are retroactively determined to be unimportant and are then removed from the network.

2.1.1 Dropout as a means for regularization

The sparsification of DNNs has typically been motivated by the goal to make DNNs less prone to over-fitting and better able to generalize. Dropout was one of the early versions of DNN sparsification that allowed for greater generalization capabilities and reduce over-fitting [33, 32, 89].

Dropout methods enforce *temporary* sparsification during training. During optimization, some random subset of the parameters are chosen to be zeroed out, but regain their values during inference. This is achieved using dropout layers, which are binary masking layers that gate the flow of information during a forward pass. These layers are non-trainable, but restrict the flow of information such that parameters chosen to be dropped-out are not updated during the current training iteration. Dropout layers are not used during inference, meaning all parameters of the network pass input to the subsequent layer and the network as a whole remains densely connected. This is in contrast to permanent sparsification methods (such as neural network pruning), which hold sparsification throughout the training and inference processes.

2.1.2 Pruning

Another method for reducing the parameters of a DNN is through neural network pruning. Neural network pruning is inspired by synaptic pruning, a biological process the occurs between childhood and adolescence in which synaptic connections between neurons sharply decreases [17]. Synaptic pruning is not entirely understood, but it is as a mechanism for learning in which the synaptic connections are maintained or removed depending on the environment [19]. The biological mechanisms that underlie synaptic pruning are often activity dependent and include a range of processes (variety of semaphorins, increased GABAergic signaling, changes in dendritic spine density, and neuro-immune interactions) [27]. Across taxa and species, synaptic pruning plays a significant role in refining task-specific biological networks, the result of which is a more sparsely connected network that can still perform complex tasks.

The main finding across synaptic pruning literature is that synaptic pruning plays a major role in the refinement of neural connectivity [97]. Through the overgrowth of synapses and their subsequent pruning, biological neural systems are made more optimal for a task and more efficient through sparse connectivity. DNNs are often considered as mathematical proxies for biological neural processing. As such, bio-inspired mechanisms for their sparsification and refinement have been explored in the literature.

The concept of neural network pruning was first introduced in late 1980s as a method for reducing the size of a network and improving its ability to generalize [51, 75, 52]. In [51], the author introduces iterative magnitude-based weight removal and refinement through retraining. They show that through the iterative process, the number of parameters in a trained network can be greatly reduced while sustaining network performance. In magnitude-based pruning, the pruning criterion is based on the magnitude of the parameter. The n parameters with the lowest magnitudes are removed from a trained network by setting their magnitude to zero and the remaining parameters are then retrained, while keeping the pruned parameters set to zero [51]. Magnitude-based pruning works well [51, 104], but the

idea that parameters with low magnitudes are unimportant to network performance is a strong assumption that is dependent on network task and structural choices (such as choice of activation function).

In [75], the authors use the sensitivity of a network to parameter removal as a metric for parameter importance. Ideally, the importance, ρ_i of parameter i is

$$\rho = E_{without\,i} - E_{with\,i}. \tag{2.1}$$

Where $E_{without\,i}$ is the error calculated without parameter i and $E_{with\,i}$ is the error calculated with parameter i. This calculation is np hard in a n-parameter network with p possible patterns of parameter removal. The authors therefore introduce an approximation of Equation 2.1 that uses a binary gating coefficient α_i. This binary coefficient ($\alpha_i \in \{0, 1\}$) controls the flow of information between input, x_i, and output, x_j.

$$x_j = \sigma(\sum_i w_{ji}\alpha_i x_i) \tag{2.2}$$

When $\alpha_i = 0$, parameter w_{ji} has no effect on the output. The sensitivity metric defined in Equation 2.1 can be rewritten as

$$\rho_i = E_{\alpha_i=0} - E_{\alpha_i=1}. \tag{2.3}$$

Equation 2.3 can be estimated using the derivative of the error with respect to the gating coefficient.

$$\lim_{\gamma\to 1} \frac{E_{\alpha_i=\gamma} - E_{\alpha_i=1}}{\gamma - 1} = \frac{\partial E}{\partial \alpha_i}\Big|_{\alpha_i=1} \tag{2.4}$$

When $\gamma = 0$, the authors assume that the equality in Equation 2.4 holds.

$$\frac{E_{\alpha_i=0} - E_{\alpha_i=1}}{-1} \approx \frac{\partial E}{\partial \alpha_i}\Big|_{\alpha_i=1} \tag{2.5}$$

The parameter importance, ρ_i can therefore be approximated by Equation 2.6.

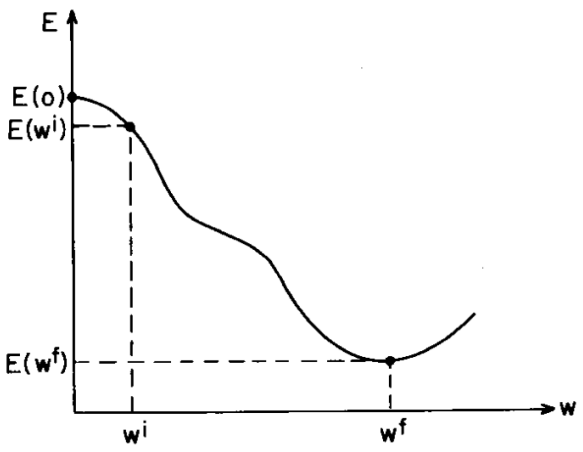


Figure 2.1: One dimensional error curve of parameters **w** where i denotes its initial state and f denotes the final state. These two points are used to estimate the error when **w** is zero [52].

$$\rho_i \approx -\frac{\partial E}{\partial \alpha_i}\Big|_{\alpha_i=1} \tag{2.6}$$

While this derivative can be calculated in the same manner as backpropagation, the authors found that the L1 error must be used as the mean-squared error provides a poor estimate of parameter importance if the output of the network is closely modeling the target (in other words, when the network is performant). Unfortunately, this means two backpropagation passes are required, one using the minimization error (usually L2 error) and one using the parameter importance error.

In [52], the author corrects this inefficiency by developing a similar approach that does not require changing the error function. The method in this paper takes advantage of the initialization of the parameters and assumes that the error when a parameter is set to zero, its associated error is close to the error of the initialized parameter.

When DNN layers are initialized, they are usually initialized with $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{infeatures}$ and $\mathcal{U}$ denotes the uniform distribution. The number of input features to a

layer is equivalent to the number of nodes in the preceding layer. Therefore, each parameter is initialized with a small, randomly chosen value. Dropping the gating coefficient, the authors rewrite the sensitivity equation in Equation 2.3 and make the following assumption.

$$\rho_i = -\frac{E(w_f) - E(0)}{w_f - 0} w_f \approx -\frac{E(w_f) - E(w_i)}{w_f - w_i} w_f \tag{2.7}$$

In Equation 2.7, w_i is the initial parameter value before training and w_f is its final value after training. The author assumes that the error without the parameter ($E(0)$) is approximately equivalent to the error when the parameter is randomly initialized. They visualize this approximation in Figure 2.1. The initial parameter, w_i is assumed to be not far off $w = 0$. As the network trains, the parameter changes (though backpropagation updates) until a minimum error is achieved and the parameter reaches its final value, w_f. The error is plotted on the vertical axis of Figure 2.1. However, the plotted error in Figure 2.1 is one-dimensional and assumes that the error is dependent only on the value of one parameter w. This is not generally true, and the authors use a two-dimensional model to explain their sensitivity calculation.

In Figure 2.2, the error landscape of two parameters w and u is drawn. I is the initial value of w and u and F is the final value. A is the point at which $w = 0$ and $u = u_f$ and B is the point at which $w = w_f$ and $u = 0$. We are looking for the error at point A, where all parameters in the model (just u in this simple example) are in their final state, while $w = 0$. The author rewrites the sensitivity equation in the form of the following integral.

$$E(w = w_f) - E(w = 0) = \int_A^F \frac{\partial E(u_f, w)}{\partial w} dw \tag{2.8}$$

However, the value of the error at point A is unknown, so the integral in Equation 2.8 is approximated by using the values at point I.

$$E(w = w_f) - E(w = 0) \approx \int_I^F \frac{\partial E(u, w)}{\partial w} dw \tag{2.9}$$

To further approximate and calculate during model training, the integral in Equation

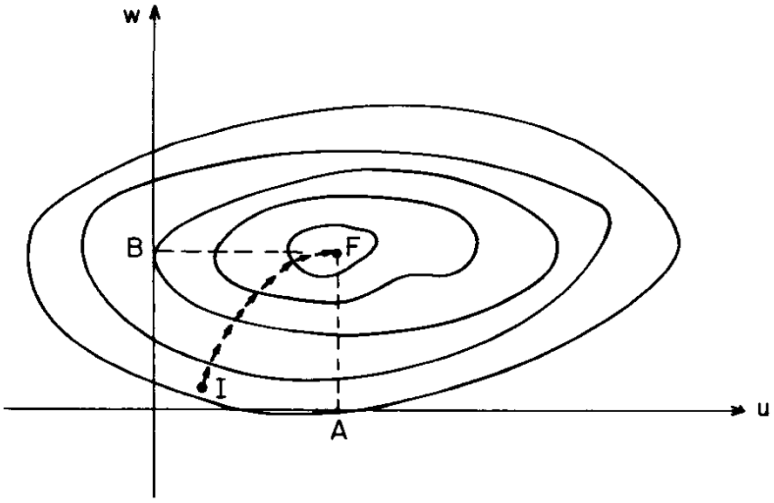


Figure 2.2: Two-dimensional version of Figure 2.1. The loss landscape of parameters **w** and **u** at their initial and final states are used to estimate the value of the loss when one parameter is zero [52].

2.9 is written as a summation taken over the discrete steps that the network takes during training. The final estimation of the sensitivity to the removal of parameter w is written in Equation 2.10 where N is the total number of training steps and n denotes a step.

$$S = -\sum_{n=0}^{N-1} \frac{\partial E}{\partial w(n)} \Delta w(n) \frac{w_f}{w_f - w_i} \tag{2.10}$$

The terms in Equation 2.10 are independent of the choice of loss function and are readily available during the optimization process. However, the assumption that the initial error is a good approximation of the error when all parameters are trained and $w = 0$ is not well supported.

In a 1989 paper called "Optimal Brain Damage", the authors aimed to move beyond the assumptions and algorithmic deficiencies proposed in previous papers [62]. The authors predict that the growing unwieldy size of DNNs will result in computationally expensive

models with little generalization capability. They introduce the technique called Optimal Brain Damage (OBD) to reduce the size of DNNs by removing or deleting parameters based on a novel saliency (or sensitivity) metric. Their metric uses the second derivative of the loss function with respect to the network parameters to compute the saliency of each parameter. The pruning algorithm involves: 1. training a model until a reasonable solution is obtained, 2. compute the second derivatives h_{kk} for every parameter, 3. compute the saliencies for each parameter $s_k = h_{kk}u_k^2/2$, 5. delete low-saliency parameters and retrain. The authors approximate a perturbation to the loss function (δE) caused by a perturbation in the parameter vector (δU) (i.e., by removing a parameter) with a Taylor series and make several approximations to reduce the expansion to a simpler form.

$$\delta E = \sum_i g_i \delta u_i + \frac{1}{2}\sum_i h_{ii}\delta u_i^2 + \frac{1}{2}\sum_{i \neq j} h_{ij}\delta u_i \delta u_j + O(\|\delta U\|^3) \tag{2.11}$$

In Equation 2.11, δu_i are elements of the parameter perturbation vector δU, while g_i are the elements of the gradient of the error function with respect to U. The h_{ij}s are the elements of the Hessian matrix of the error function with respect to U.

$$g_i = \frac{\partial E}{\partial u_i} \quad \text{and} \quad h_{ij} = \frac{\partial^2 E}{\partial u_i \partial u_j} \tag{2.12}$$

The perturbation to the error in Equation 2.11 is reduced by the following approximations. The cross terms can be neglected (third term in the right hand side of Equation 2.11) because the authors assume that the change in the error caused by deleting several parameters is equivalent to the sum of change in error by deleting each parameter individually. The contribution to the change in the error made by the gradient of the error with respect to the parameters (first term in the right hand side of 2.11) can also be ignored because parameter pruning will occur after the network has converged to a local minimum. Changes in parameters will not change the gradient at this point. Finally, the higher order terms in the Taylor expansion are ignored because the loss function is assumed to be nearly quadratic. The change in the error can be reduced to a single term.

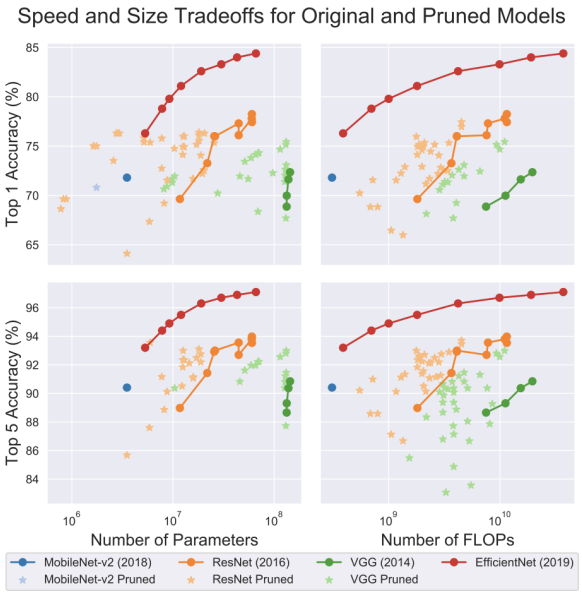


Figure 2.3: Performance and efficiency trade-offs in pruned and fully-connected models of various architectures [11].

$$\delta E = \frac{1}{2} \sum_i h_{ii} \delta u_i^2 \tag{2.13}$$

In this pruning paradigm, the saliency or sensitivity of the network to the removal of each parameter is determined entirely by the second-derivative of the error with respect to each parameter. This can be calculated using a form of backpropagation.

In another classical paper titled, "Optimal Brain Surgeon," the authors argue the the diagonal Hessian assumption made in [62] is not well supported [40]. They claim that removing parameters based off of the diagonal Hessian assumption will result in important parameters being removed. The authors of [40] follow the general paradigm laid out in [62], but make no restrictive assumptions about the Hessian of the network and dub their method Optimal Brain Surgeon (OBS).

Since these original classical papers, many papers about pruning modern architectures,

comparing results, and changing pruning paradigms have been published. The survey article "What it the State of Neural Network Pruning?" by Blalock, et. al summarizes the state of the field nicely [11]. The survey consolidates results over 81 papers and finds that the field suffers from a lack of standardized benchmarks and evaluations (a common problem in the broader field of deep learning). However, they state that their most significant finding is that neural network pruning works. Specifically, whether magnitude-based or saliency/sensitivity-based, pruning results in a sparse network that can perform comparably to a fully connected network. In fact, for small amounts of pruning, network performance can increase above the original trained network [104, 37, 93].

Other broad conclusions can be drawn from the literature. The first is that a structured pruning method (e.g., magnitude-based, or saliency/sensitivity-based) outperforms random pruning [11, 31, 102, 29, 71, 42]. Additionally, pruned networks are capable of outperforming dense networks and sometimes even larger dense networks, but rarely outperform networks of a better architecture [11].

Figure 2.3 shows the performance results of various pruned and dense convolutional networks [11]. The vertical axes show the top 1% and top 5% accuracy and the horizontal axes show the number of parameters and number of FLOPs. At the time of publication, MobileNet-V2 [85] and EfficientNet [94] were the state-of-the-art convolutional networks. We can see that pruned networks (see ResNet and VGG results) can outperform even larger counterparts of the same architecture. However, a better architecture (see MobileNet-V2 and EfficientNet results) outperform both pruned and dense lesser architectures.

2.1.3 Lottery Ticket Hypothesis

Another interesting and impactful study in the field of neural network pruning is "The Lottery Ticket Hypothesis" by Jonathan Frankle and Michael Carbin [29]. In this study, the authors propose the lottery ticket hypothesis: dense, randomly initialized feed-forward networks contain sub-networks that when trained in isolation, achieve comparable performance to the original dense network. They find that these sub-networks can be uncovered using

iterative magnitude-based neural network pruning. After discovering the sparse architecture via pruning, the parameters of the "lottery ticket" sub-network are re-initialized with their original weights. A key finding of this study is that lottery ticket sub-networks require re-initialized with their original weights to obtain comparable performance when trained in isolation. With this re-initialization, the sparse sub-network, when trained in isolation can reach test accuracy comparable to the original dense network [29].

This study is one of the first of its kind to attempt to draw a relationship between deep neural network performance and structure, but the authors emphasize that structure alone cannot explain the performance of lottery ticket sub-networks. When the sub-networks are randomly re-initialized, they fail to reach the original network performance with retraining. Both the structure and the original weight initialization are critical for sparse sub-network performance. From the results of [29], it can be concluded that high-magnitude weights are the weights critical for performance in the experiments the authors performed. This further verifies the assumption that magnitude-based pruning is an effective way to sparsify DNNs.

In this thesis, the structure of sparse, sub-networks, found by iterative magnitude-based pruning will be explored using tools from network theory, namely network motifs. Section 2.2 will broadly describe network theory tools applied to DNNs, and Chapter 5 will describe our contribution.

2.2 Deep neural networks and network theory

One approach to elucidate the relationship between DNN structure and function, as well as identify excess and redundant parameters, is to approach them as complex networks, and use the tools of the field to study them. Complex network theory is a well established field that spans disciplines from discrete mathematics to the social sciences. Historically the field has focused on natural, physical, and real-world networks and only recently have the tools been applied to large computational networks such as DNNs Unlike random and lattice networks, complex networks are characterized by a non-trivial network topology that enables complex collective dynamics through simple nodal connections. While it is generally accepted that

the function of a complex network is intrinsically related to its connectivity structure, there is little known about whether small-scale, local connectivity patterns affect the global behavior of a network. [92, 77, 3].

There are two well known topological properties of complex networks that relate local connectivity patterns to global function: the small-world property and the scale-free property. small-world networks are ones in which a node is connected to many or most other nodes by way of neighboring connections [98]. Mathematically, a small-world network is one in which the average distance between two nodes is proportional to the logarithm of the total number of nodes in the network. The small world property allows almost all nodes in a sparsely connected network to transmit information to almost all other nodes in the network. An example of a small world network is the gene co-expression network in yeast, *Saccharomyces cerevisiae* [95]. This network also exhibits the scale-free property of complex networks. A scale-free network is a network in which the distribution of nodal degrees (the number of connections of a single node) follows a power law [6]. These properties support the assumption that the local connectivity patterns within complex networks govern global network dynamics.

Another local connectivity property of complex networks are network motifs. Network motifs are statistically significant sub-graphs within a larger network [4]. A sub-graph is a small group of interconnected node. The order of a sub-graph is defined by the number of nodes within it (e.g., 2^{nd}-order sub-graphs contain 3 nodes, 3^{rd}-order sub-graphs contain 4 nodes, etc.) A sub-graph is considered to be a network motif if it occurs with high significance in the larger network when compared to its occurrence in a random network. Network motifs have been discovered in many natural networks and were originally discovered in ecological networks [4, 72, 91]. Network motifs in deep neural networks will be explored in detail in Chapter 5.

As stated earlier, the large networked structure of DNNs allows for complex interactions between input variables, parameters, and output variables. This quality makes DNNs powerful tools for modeling many tasks. Researchers have approached DNN topology from a few

different angles. One study found that the topological complexity of a binary classification data set is reduced as the data passed through the network during training [76]. In [34], the authors create novel DNN architectures that are built from empirical data on the structure of animal neural networks.

In [105], the authors study the emergence of network motifs in deep neural networks with initial connectivity defined as a stack of fully-connected bipartite graphs. Modeling DNNs as bipartite graphs is natural due to the layer-wise constraint of feed-forward networks. Their experiments show that network topology post training is shaped both by the training scheme and network parameter initialization. Specifically, non-trivial parameter initialization can make training more effective through promoting the emergence of network motifs. The authors find that both orthogonal and Glorot parameter initialization schemes initialize the network with more useful motifs, such as the bi-fan and bi-parallel motifs. These initializations make training more efficient because they initialize the network with the desired network motifs.

Chapter 3

DEEP NEURAL NETWORK SPARSIFICATION WITH PRUNING

3.1 Pruning DNNs

At a high level, computational DNN pruning is accomplished by choosing a network parameter to prune (based off some criteria) and setting it equal to zero. There are many algorithmic choices to specialize pruning to the specific network or desired outcome. For example, in convolutional neural networks (CNNs), one might wish to reduce the number of computational filters or channels of the network, thereby reducing its size and computational cost and reducing the number of redundant filters. Alternatively, in a network designed to model optimal sensor placement the input nodes could be targeted to reveal optimal sensor placement [99]. Much like DNN architecture design, pruning has a diversity of hyperparameters that will affect the quality and efficiency of the pruned network.

In the following section, the algorithmic basics of neural network pruning will be described. Future chapters and sections will provide several different applications of pruning and the differences from the base algorithm will be described.

Algorithm 1 describes the basic pruning algorithm utilized both in most of the literature and in this thesis. The first step is to train a dense or fully-connected DNN until the loss is minimized. Next, n parameters are chosen to be pruned based off of some pruning criteria. The parameters could be the weights and biases, entire nodes, or entire CNNs filters/channels. As described in Section 2.1.3, magnitude-based pruning is a reasonable pruning criteria and has been shown to work [29, 104]. In this thesis, the magnitude-based pruning criteria is used to quickly and efficiently prune. Finally, the network is retrained until the loss is once again minimized. The process can be repeated at step 2 of Algorithm

Algorithm 1: Pruning and Fine-tuning

Train fully-connected model until loss is minimized;

Set and fix n parameters to zero;

Retrain remaining parameters;

1 in an iterative process to reduce the number of parameters further.

While some pruning algorithms do not require a fine-tuning step [40], most algorithms involve retraining the remaining weights after pruning. However the parameters that were pruned need to be locked at zero (to accomplish the goal of pruning), so simply retraining the network will cause these previously pruned parameters to lift back off of zero. Using deep learning libraries (e.g., PyTorch) it is possible to lock individual parameters such that they are not updated during backpropagation. However, this is tedious and expensive. The more common and efficient way to prune a dense network is by way of binary masking matrices. These masking matrices achieve sparsity similar to the gating coefficients in [75]. Mathematically, they are binary matrices inserted in between every parameter matrix in the network. If our network is described by Equation 1.1, when we include binary masking layers, it takes the following form.

$$\hat{y} = \mathbf{M_4} \circ \sigma_4(\mathbf{A_4}...(\mathbf{M_1} \circ (\sigma_1(\mathbf{A_1}x))) \tag{3.1}$$

In Equation 3.1, the binary masking matrices, $\mathbf{M_i}$, are multiplied element-wise to the weight matrices ($\circ$ denotes the element-wise Hadamard product). The sparsity of each layer is controlled by separate masking matrices which can allow for different levels of sparsity in each layer. Initially, all elements of $\mathbf{M_i}$ are set to 1. During pruning, the n desired parameters are located and the corresponding elements of the the $\mathbf{M_i}$ are set to zero, thereby gating the flow of downstream information from the pruned parameter.

The specific pruning algorithms for experiments done in this thesis will be shown in Sections 3.2.1, 3.2.2, and 4.2.4.

3.2 Applications

Two pruning applications are explored in the remainder of this chapter. In Section 3.2.1, the maximum achievable sparsity of convolutional neural networks via pruning is found and the relationship between task complexity and sparsity is explored. In Section 3.2.2, magnitude-based pruning is used to locate optimal sensors in an encoder-decoder architecture. In this application magnitude-based pruning results in poor sensor placement compared to other methods and the potential reasons why are discussed.

3.2.1 Pruning computer vision models

Computer vision is one of the areas where deep learning models have shown great success. Convolutional neural networks (CNNs) are neural networks where the operation transforming input to output is a convolution, rather than a simple matrix multiplication as in feed-forward DNNs. This allows CNNs to learn spatial correlations between areas within an image. Pruning in CNNs has been explored in many different studies [11, 74, 101, 67, 36, 37].

The reduction of parameters in CNNs has been motivated by the push to use such models on mobile devices. In the case of CNN filter pruning, networks can be made more compact through the removal of entire convolutional filters (as opposed to individual parameters within the filter). In [74], the authors extend on the DNN pruning algorithms developed in [16, 62] and use the first and second order Taylor expansions to approximate a CNN filter's contribution to the output of a network. Using this method, the authors see a 40% reduction in FLOPS by removing 30% of the parameters, with a loss of 0.02% in the top-1 accuracy on the ImageNet data set.

The structure of CNNs and the task of computer vision (e.g., object detection, image classification, etc.) make them somewhat more interpretable than feed-forward DNNs. The parameters of a convolutional layer make up a convolutional filter and are learned via gradient descent and backpropagation. Each layer of a CNN contains some number of convolutional filters defined by its channel number. Each filter is independently updated via gradient

descent and backpropagation, meaning each filter can learn a different feature representation of the image. Therefore, a CNN is made up of many interrelated convolutional filters that represent different spatial features for the given computer vision task. However, much like other architectures, CNNs are overparameterized and may contain redundant filters. Some filters learned by a CNN may also be unimportant to prediction. The sub-field of explainable AI (XAI) provides a tool set for explaining the relationship between model parameters and prediction. In [101], the authors connect the fields of explainable AI (XAI) and model compression and develop a novel criterion for pruning based on filter relevance scores found using XAI tools. Their novel pruning criterion is based on Layer-wise Relevance Propagation (LRP) [5]. In this method, every parameter in the model is assigned a relevance score, which is determined by how a single pixel (or feature) from an input affects the prediction of the model. Filters are then pruned based on low relevance scores. The authors show that by using this method of pruning, the size of networks can be reduced without compromising accuracy. This method is distinct from many of the others discussed in this thesis because it is a data-based method. Parameter importance is dependent on a particular input from the data set. The authors show that this important quality of the algorithm improves predictive performance in toy-classification problems.

Here, we use magnitude-based neural network pruning to reduce the number of filters in ResNet-18 convolutional neural networks, trained to perform image classification on three toy data sets and explore the relationship between pruning and task complexity. For each of the experiments performed in this study, we apply the same sequential pruning algorithm to sparsify the networks. This algorithm utilizes a PyTorch pruning library [81] and features a sequential pruning paradigm that sparsifies the deepest layers of a network before pruning more shallow layers. This reverse, layer-wise pruning allows each layer to be maximally sparsified while maintaining an accuracy comparable to the fully-connected network. Every layer of the network is sparsified via magnitude-based channel pruning in which the channels with the lowest L^2 norm are removed.

Algorithm 2: Reverse-layer, magnitude-based pruning and fine-tuning

```
Train fully-connected model until loss is minimized;
Set baseline accuracy to accuracy of fully-connected network;
for each layer in reversed order do
    Define n as the initial number of channels to be pruned.;
    Calculate L^2 norm of each channel;
    Prune n channels with minimum L^2 norm and evaluate accuracy;
    if accuracy < baseline accuracy −1 then
        Retrain for one epoch, evaluate accuracy, and reduce n;
        if accuracy > baseline accuracy then
            baseline accuracy ← accuracy;
        else if accuracy < baseline accuracy −1 then
            Retrain for one epoch and evaluate accuracy;
            if accuracy > baseline accuracy then
                baseline accuracy ← accuracy;
            else if accuracy < baseline accuracy −0.5 then
                Stop pruning and fine-tuning this layer;
            else
                Pass;
            end
        else
            Pass;
        end
        else
            Skip retraining and proceed with pruning;
        end
end
```

The sequential pruning algorithm used in this work is outlined in detail in Algorithm 2. Each layer is pruned independently and in reverse order, starting with the final layer. Pruning the layers in reverse order allows us to maximally sparsify the deeper layers of the network, which capture the higher-order features, before pruning the shallow layers, which capture lower-order features [106].

Following Algorithm 2, first, a network is trained to convergence. Then, beginning with the final layer (and with subsequent layers in reverse order), n channels with minimum L^2 norm are pruned. n is decided arbitrarily, but here we start with $n = \frac{N}{8}$ where N is the total number of channels in the given layer. Through some trial and error, we found this to generally be a good starting point so to not prune too much before a retraining epoch. For example, if we are starting by pruning the final layer in ResNet-18, the initial value for n is $512/8 = 64$. After pruning, the network is evaluated and if the validation accuracy drops below a specified threshold, the network undergoes a single training epoch and the value of n is decreased by a factor of two. If retraining brings the accuracy back up to the baseline accuracy (defined as the initial accuracy before pruning), the next set of n channels are pruned. If the retraining does not result in a regained baseline accuracy, the network is retrained once more over a single epoch. If the second retraining brings the accuracy back up to the baseline accuracy, the pruning process continues and if the second retraining does not improve performance, pruning and retraining stops for this layer. The number of channels removed during each prune, n, is adjusted post retraining as described above and it is also updated preemptively to avoid pruning more channels than exist in the layer, i.e. if the number of remaining channels in a layer exceed n, n will be reduced by a factor of 2 until $n = 1$. The prune, retrain, repeat cycle described here allows us to maximally sparsify a layer while maintaining the accuracy of the fully-connected network. In other words, pruning is stopped and maximum sparsity is achieved when we can no longer remove a channel and regain original accuracy with two retraining epochs.

Each layer of the network is sparsified via magnitude-based channel pruning. We utilizes PyTorch pruning packages, specifically `torch.nn.utils.prune.ln_structured`, to prune

Table 3.1: **ResNet-18 Experimental Results** Caption text.

data set	**Global Sparsity**
MNIST	99.51%
CIFAR-10	85.91%
CIFAR-100	35.55%

entire channels from the convolutional layers based on their L^2 norms [81]. Three experiments were performed with three different toy data sets: MNIST, CIFAR-10, and CIFAR-100 [21, 56]. The data were split into training, validation, and test sets. T he train-val-test spilt for each data set was 50,000, 10,000, 10,000 for MNIST; 40,000, 10,000, 10,000 for CIFAR-10; and 40,000, 10,000, 10,000 for CIFAR-100. Each data set was transformed the same way with a normalization factor of 0.5 in all channels. The experiments were performed with a batch size of 128, a learning rate of 5^{-4}, 3 epochs of training prior to pruning, and optimized with the Adam optimizer. We trained and pruned 100 networks for each data set to compare average levels of sparsity across networks with different random initialization.

Tables 3.1 and 3.2 show the results of our CNN pruning experiments. The average (across all 100 trained network) global sparsity (across all layers of ResNet-18) is shown in Table 3.1. Note that these sparsity levels are the maximum sparsity achieved via pruning without compromising network performance. As the complexity of the computer vision task increases (from classifying MNIST digits, to CIFAR-10 and CIFAR-100 images), the maximum sparsity decreases. This may indicate a relationship between filter redundancy and task complexity. MNIST classification is known to be a much easier task to model with deep learning than natural image classification (as in CIFAR-10 or CIFAR-100). It is even possible to accurately classify MNIST digits with feed-forward architectures. Our results indicate that the maximum amount a network can be sparsified is limited by task complexity.

Table 3.2: **ResNet-18 Convolutional Layer Sparsity** Caption text.

Layer Name	MNIST	CIFAR-10	CIFAR-100
conv1	33.68%	18.13%	14.53%
layer1.0.conv1	48.33%	14.06%	12.90%
layer1.0.conv2	91.77%	17.13%	13.41%
layer1.1.conv1	98.44%	14.88%	12.90%
layer1.1.conv2	98.38%	20.85%	16.10%
layer2.0.conv1	99.22%	14.55%	13.15%
layer2.0.conv2	99.22%	17.31%	13.09%
layer2.1.conv1	99.22%	23.77%	17.17%
layer2.1.conv2	99.22%	36.56%	22.46%
layer3.0.conv1	99.61%	26.26%	13.75%
layer3.0.conv2	99.61%	51.97%	14.84%
layer3.1.conv1	99.61%	88.15%	61.67%
layer3.1.conv2	99.61%	88.03%	52.25%
layer4.0.conv1	99.80%	92.45%	19.47%
layer4.0.conv2	99.80%	93.91%	25.91%
layer4.1.conv1	99.80%	97.58%	21.57%
layer4.1.conv2	99.80%	95.89%	69.91%

We also look closer at the average sparsity within the layers of ResNet-18. Table 3.2 shows the final average sparsity of each layer of ResNet-18 after the pruning process described in 2. Recall that pruning of each layer is performed in reverse order, starting with the deepest

layers and moving to the shallower layers. Each layer is pruned until maximum sparsity is achieved without compromising performance. The rows of 3.2 show the layers of ResNet-18 and the columns are the results for each denoted data set. For the MNIST data set, all layers downstream of the first two convolutional layers can be reduced by as much as 90%. This indicates that the features necessary for accurate MNIST digit classification are almost entirely captured by the most shallow layers. The equivalent results for CIFAR-10 and CIFAR-100 data sets are shown in the rest of the table. The level of sparsity achieved in each layer of the CIFAR-10 trained network increases with layer depth. The deepest layers of the ResNet-18 trained on CIFAR-10 can be pruned to over 90% sparsity. As you move to shallower layers, the maximum achievable sparsity is reduced. A similar pattern can be seen in the results for the CIFAR-100 trained ResNet-18s, but the depth-sparsity relationship extends to the residual blocks of ResNet-18. For example, in the final residual block of ResNet-18 (denoted by `layer4`) deeper layers are more sparse than shallow layers within the block.

This work indicates a potential relationship between the maximum achievable sparsity via pruning and the complexity of a computer vision task. The results in Table 3.1 indicate that sparsity is inversely related to task complexity, with networks trained to classify the simplest data set, MNIST digits, being pruned to the highest levels of sparsity without compromising performance. The results in 3.2 show that features critical to classification are captured in the shallow layers of ResNet-18 or in the shallow layers of each residual block. Extending this comparison to different CNNs could lead to insights into the optimal architecture for a given task based on the complexity of the task.

3.2.2 Data-driven sensor placement via pruning

In this section, we explore neural network pruning as a method for finding optimal sensor placement for full-state reconstruction. In the context of network-based reconstructions, optimal sensor placement can be viewed as identifying optimal input nodes in a *shallow decoder network* (SDN) that maps full-states to full-states, depicted in Figure 3.1. The

network pictured here has an auto-encoder structure, where inputs are mapped to a low-dimensional space and then reconstructed back to the original state-space. Each input node can be thought of as a point in the state-space and the goal of the SDN is to find the optimal low-dimensional mapping to reconstruct the state-space accurately. However, in many applications the full state-space is unknown and we would like to reconstruct the state-space from a small set of measurements. In this formulation, limiting measurements corresponds to removing entire nodes (and all their downstream connections) from the input layer of the SDN.

Here, we compare the reconstruction performance of three sparse SDNs with (a) randomly selected input nodes (R-SDN), QR (linear) selected nodes (Q-SDN), and pruned (non-linear selected) nodes (P-SDN). QR selection is selecting nodes based on their rank determined via QR decomposition. Performance is compared in application to the reconstruction of sea surface temperature [1]. As discussed previously, pruned networks have been shown to retain comparable accuracy to larger models. Although pruning can be applied to either weights or nodes, here we consider the removal of nodes, specifically the input nodes of the SDN. We use a form of magnitude-based pruning which identifies nodes for removal by computing the root mean square of the trained weights corresponding to each node. The nodes associated with the smallest root mean square weights are removed. Pruning a node from the input layer of an SDN mapping full-states to full-states has the physical interpretation of removing a sensor measurement.

When applied to the input layer of a network, pruning can identify important input features by removing the inputs that contribute minimally to the prediction [104]. Here we use pruning as a method for discovering a set of sparse sensors by pruning only the input layer the SDN. The method begins by training an auto-encoder to map full state sensor measurements back to the original high-dimensional state space. The first layer of the network is the encoder network which maps the full state sensor measurements to a low-dimensional representation. Following the method in [104], we prune this layer iteratively. Specifically, once the training error plateaus, we prune a small subset of input nodes with

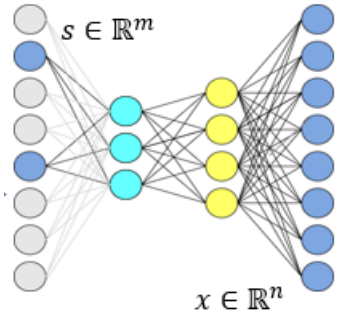


Figure 3.1: Shallow-decoder network diagram mapping sensor measurements, $s \in \mathbb{R}^m$ to $x \in \mathbb{R}^n$. The gray nodes represent elements of the full-state x that are not measured and the blue nodes represent measurements.

corresponding weights that have the smallest root mean square. The pruned network is then retrained and this process repeats until only a few sensor measurements remain. This results in a pruned version of the encoder, while the decoder remains fully connected. The pruned SDN (P-SDN) assumes that the nodes most critical to accurate reconstruction are those that impart the highest activation to the latent space. Additionally, the P-SDN simultaneously learns sensor locations and reconstructions, which is quite different than Q-SDN which learns these in serial.

Here, we follow the iterative pruning protocol in [104]. First, we train a fully-connected SDN to convergence. We define a target sparsity (i.e. 10%, 20%,..., 90%) and use root mean square pruning in the input layer to sparsify the layer to the target sparsity. Sparsity is achieved by using a binary masking layer which can be represented mathematically the following way:

$$\mathcal{F}(;\mathbf{W}) := R(\mathbf{W}^k R(\mathbf{W}^{k-1} \cdots \mathbf{M^1} \circ R(\mathbf{W^1 s}))), \tag{3.2}$$

where $\mathbf{M^1}$ is a binary masking matrix that multiplies element-wise ($\circ$ represents the Hadamard product) to the input layer of the SDN. The matrix $\mathbf{M^1}$ is initialized with ones.

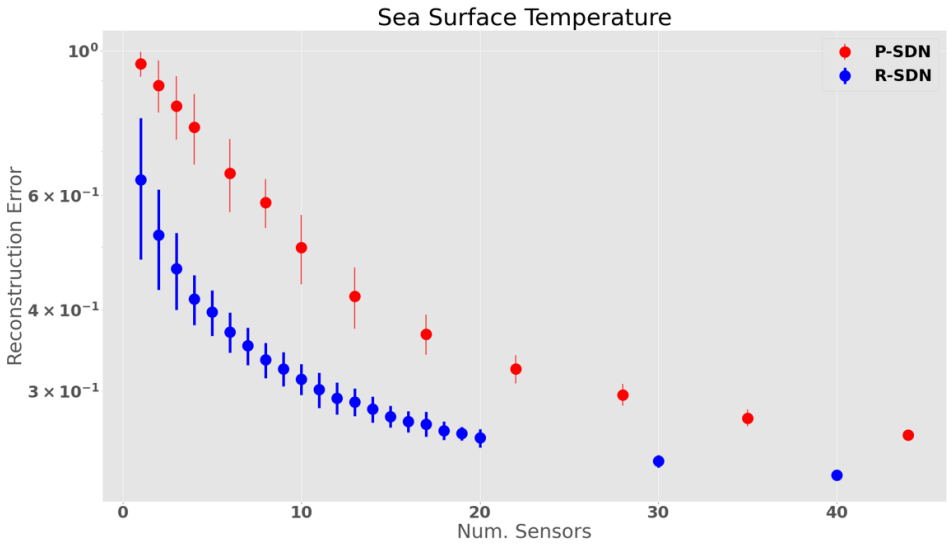


Figure 3.2: Reconstruction performance of R-SDN and P-SDN for the example data set of sea surface temperature.

During pruning, zeroing-out a node corresponds to making an entire column of $\mathbf{M^1}$ zeros. During each prune, we zero-out the number of columns necessary to reach the target sparsity. The pruned network is then retrained and the process repeats, incrementally increasing the target sparsity until very few inputs remain. To avoid over-fitting to the training data, the remaining weights in the network are reinitialized after each prune. The pruned inputs nodes have no effect on the prediction of the model because they remained zeroed-out throughout the entire process. This is achieved by making the masking matrix non-trainable, meaning it cannot be updated during backpropagation.

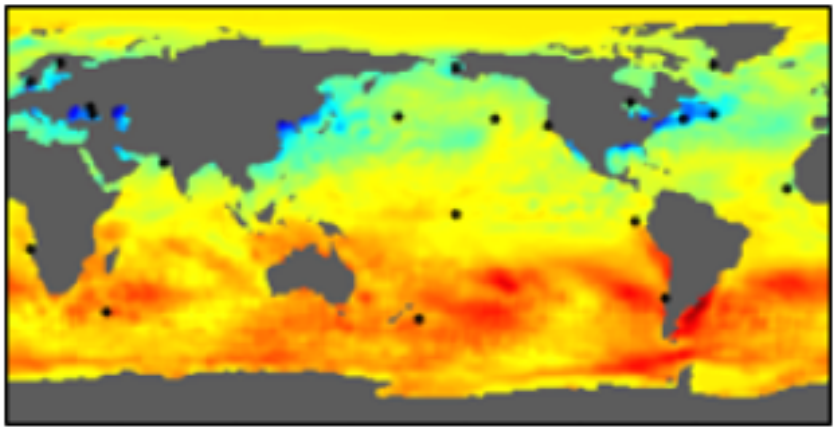
Since the input nodes represent sensor measurements, we consider magnitude-based pruning of input nodes as a nonlinear method for sensor selection. In contrast with QR placement, P-SDN, relies exclusively on SDNs to learn measurement selection and reconstruction in parallel. The reconstruction error (RE) is measured by

$$\texttt{RE} = \frac{1}{N}\sum_{k=1}^{N}\frac{\|\hat{x}_k - x_k\|}{\|x_k\|} \tag{3.3}$$

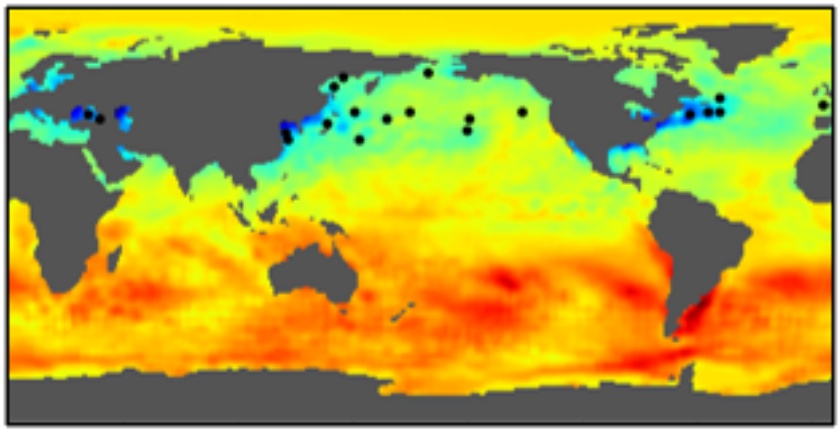
where x is the mean subtracted full state, $\hat{x}$ is the mean subtracted reconstructed state, and k indexes the samples withheld during training. The mean subtracted reconstructions are considered because in data sets where the mean state accounts for much of the spatial structure, the relative deviation provides a more sensitive measure of reconstructive performance [26]. We emphasize the importance of performance in the "low sensor" limit (i.e., when increasing the number of sensors significantly improves reconstructive performance). With larger numbers of sensors, the question of sensor placement becomes less important because even randomly chosen measurements can yield accurate reconstructions. We find that Q-SDN consistently outperforms all other considered techniques. Surprisingly, we show P-SDN is significantly outperformed by SDNs with randomly chosen measurements (R-SDN). R-SDN serves as an important benchmark for the performance of both Q-SDN and P-SDN as a successful placement scheme should outperform random placement.

We begin by considering the performance of P-SDN in comparison to that of R-SDN in an example data set consisting of 1,400 samples of a a 180×360 grid representing sea surface temperature (44,219 points represent sea surface temperature, the rest landmass). We initialize the P-SDN network with an input layer of 2,000 randomly chosen measurements, a first hidden layer of 350 nodes, a second hidden layer of 400 nodes, and an output layer of size 44,219. The ADAM optimizer with learning rate of 0.001 is used to train all networks and an early stopping criteria with a patience of 5 is used. This early stopping criteria determines the intervals at which the network is pruned in our implementation of P-SDN. 20% of the remaining nodes are pruned at each iteration and the resulting networks performances are compared to R-SDN with comparable numbers of sensors. In all cases, 1,000 samples are randomly selected to serve as training data, while the 400 withheld samples are used to evaluate performance. We consider 32 instances of R-SDN and P-SDN to determine a mean and standard deviation of reconstruction error, as defined in Eq. (3.3). The results are

displayed in Figure 3.2.



(a) QR Placement (20)



(b) Pruning Placement (20)

Figure 3.3: Placement of 20 sensors obtained by a QR decomposition (a) and iterative, magnitude based pruning (b). QR placement scatters sensors throughout the field, while the placement obtained by pruning clusters sensors in confined geographic regions.

Surprisingly, R-SDN outperforms P-SDN at all numbers of sensors we consider; rather than learning ideal sensor locations, P-SDN as implemented here identifies poor locations. Figure 3.3 illustrates why this might be the case. While P-SDN identifies locations similar

to that of QR, it restricts its placement to only a small subset of physical locations. This appears to indicate that magnitude based pruning alone is not successful at identifying ideal placement, although we conjecture other pruning protocols might have greater success. Nodes associated with collectively (root mean square) small weights were pruned. During training, input nodes with very low temperatures gained large connections to the latent space through training so that the network can reconstruct colder regions, as well as warmer ones. This is a case where the pruning methods based on parameter importance (described in detail in Section 2.1.2 may result in more ideal sensor locations.

3.2.3 Sparse, bio-inspired non-linear controller

The final application of neural network pruning discussed in this thesis explores bio-inspired sparsification of a DNN trained to model a bio-mechanical system. Additionally, the structure of sparse networks trained to model this system will be approached using tools from complex network theory. Chapters 4 and 5 will describe a body of work focused on pruning deep neural network controllers for a motor control task. In Chapter 4, the methods for discovering the optimal sparse model for the bio-mechanical system will be described. Optimal sparsity will be compared across many networks with different random initialization. In Chapter 5, the structure of the sparse DNN model will be explored using network motif theory.

Chapter 4

PRUNING DEEP NEURAL NETWORKS GENERATES A SPARSE, BIO-INSPIRED NONLINEAR CONTROLLER FOR INSECT FLIGHT

Originally inspired by biological nervous systems, deep neural networks (DNNs) are powerful computational tools for modeling complex systems. DNNs are used in a diversity of domains and have helped solve some of the most intractable problems in physics, biology, and computer science. Despite their prevalence, the use of DNNs as a modeling tool comes with some major downsides. DNNs are highly over-parameterized, which often results in them being difficult to generalize and interpret, as well as being incredibly computationally expensive. Unlike DNNs, which are often trained until they reach the highest accuracy possible, biological networks have to balance performance with robustness to a noisy and dynamic environment. Biological neural systems use a variety of mechanisms to promote specialized and efficient pathways capable of performing complex tasks in the presence of noise. One such mechanism, synaptic pruning, plays a significant role in refining task-specific behaviors. Synaptic pruning results in a more sparsely connected network that can still perform complex cognitive and motor tasks. Here, we draw inspiration from biology and use DNNs and the method of neural network pruning to find a sparse computational model for controlling a biological motor task.

4.1 Introduction

Between childhood and adolescence, the number of synaptic connections between neurons sharply decreases through a process called synaptic pruning [17]. Depending on the neural system, synaptic pruning can improve the brain's efficiency and affect cognitive function.

In fact, synaptic pruning is seen as a mechanism for learning, in which the environment affects which neural connections are maintained and which are removed [19]. Refinement of neural connections via pruning occurs in wide ranging taxa, from humans to *Drosophila* and in systems ranging from sensory input to motor control [97, 53]. For example, during metamorphosis of the hawkmoth, *Manduca Sexta*, synapses are pruned and reconnected to enable adult-specific behaviors [65]. The biological mechanisms that underlie synaptic pruning (often activity dependent) have a range of processes including a variety of semaphorins, increased GABAergic signaling, changes in dendritic spine density (with some enigmatic mechanisms), and even neuro-immune interactions [27]. Synaptic pruning plays a significant role in refining task-specific behaviors, the result of which is a more sparsely connected network that can still perform complex cognitive and motor tasks.

There is a rich basis of literature in biological synaptic pruning. However, the main finding across these numerous studies is that synaptic pruning plays a major role in the refinement of neural connectivity [97]. Through the overgrowth of synapses and their subsequent pruning, biological neural systems are made both optimal for a specific task and more efficient for having more sparse connectivity. Deep neural networks (DNNs), which were originally motivated by the visual cortex of cats and the pioneering work of Hubel and Wiesel [50, 88], are often considered as mathematical proxies for biological neural processing. The universal approximation properties of DNNs [47] make them ideal for modeling high-dimensional, complex, nonlinear mappings for a large diversity of problems. From image and speech recognition [61, 33] to fluid flow control [70, 10], DNNs learn input-to-output mappings by combining gradient descent with the backpropagation algorithm. Like biological pruning, DNNs have an extensive literature dedicated to improving the generalization capabilities (i.e. performance on unseen data) and computational efficiency of DNNs through the mechanism of pruning.

The sparsification of such DNNs has typically been motivated by the pernicious effects of over-fitting data, and to a lesser extent, the DNNs computational and memory footprint, i.e. the need to be implemented on small portable devices such as smart phones. Dropout, for

instance, was one of the early versions of sparsification that allowed for greater generalization capabilities [33, 32, 89]. However, standard dropout methods typically only enforce *temporary* sparsification since the algorithm often allows for nodes to again re-train their weights from their zeroed-out state. Thus most DNNs typically remain highly over-parameterized and their layers are fully-connected. For example, the natural language processing model, GPT-3, is the largest DNN ever built with 175 billion parameters [13], and successful models with millions of parameters are not uncommon. There are many different methods to make DNNs more sparse, ranging from regularization during training [87] to specifying sparse architectures [73].

Biologically inspired neural network pruning has also been shown to be an effective method for sparsifying a DNN without compromising performance [63, 40, 69, 68, 59]. In neural network pruning, the connectivity of a DNN is made more sparse by forcing select weights between the layers to zero and then retraining, resulting in a more sparse network that is capable of performing comparably to the fully-connected network up to a certain limit. Pruning has been used to prevent network over-fitting and to reduce overall model size. Pruned DNNs have the advantage of (i) having a small memory footprint, (ii) providing improved generalization, and (iii) being more efficient for generating input-output computations. Thus they have important practical advantages over their fully-connected counterparts. They are also more representative of biological neural systems, in which neural pathways are sparsely and specifically connected for task performance. In fact, a diversity of sparse networks exist across species. For example, the respiratory rhythm patterns of mammals are generated by sparsely connected networks [35]. In the olfactory system of *Drosophila*, high-dimensional odor signals are sparsely encoded via the mushroom body [46, 20]. Neural network pruning enables the exploration of biologically-inspired, sparse learning and the strengths of the resultant sparse networks.

The inverse problem of insect flight is a highly nonlinear dynamical system, in part due to the unsteady mechanisms of flapping flight [23, 86] and the noisy environment through which insects maneuver. As such, the inverse problem of insect flight serves as an exemplar

to study whether a DNN can solve a biological motion control problem while maintaining a sparse connectivity pattern. In an inverse problem, the initial and final conditions of a dynamical system are known and used to find the parameters necessary to control the system. In other words, the DNN in this study is trained to predict the controls required to move the simulated insect from one state space to another. Solving the inverse problem of insect flight has been previously simulated using a genetic algorithm wedded with a simplex optimizer for hawkmoth level forward flight and hovering [43]. Another study linearized the dynamical system of simulated hawkmoth flight and found the system to operate on the edge of stability [24]. Recently, a study developed an inertial dynamics model of *M. sexta* flight as it tracked a vertically oscillating signal, which modeled the control inputs using Monte Carlo methods in a model-inspired by model predictive control (MPC) [14].

In this work, we use the inertial dynamics model in [14] to simulate examples of *M. sexta* hovering flight. Fig 4.1 shows the physical parameters of the simulated moth and the inertial dynamics model. These data are used to train a DNN to learn the controllers for hovering. Drawing inspiration from pruning in biological neural systems, we sparsify the network using neural network pruning. Here, we prune weights based simply on their magnitudes, removing those weights closest to zero. Importantly, the pruned weights remain zeroed out throughout the sparsification process. This bio-inspired approach to sparsity allows us to find the optimally sparse network for completing flight tasks. Insects must maneuver through high noise environments to accomplish controlled flight. It is often assumed that there is a trade-off between perfect flight control and robustness to noise and that the sensory data may be limited by the signal-to-noise ratio. Thus the network need not train for the most accurate model since in practice noise prevents high-fidelity models from exhibiting their underlying accuracy. Rather, we seek to find the sparsest model capable of performing the task given the noisy environment. We employed two methods for neural network pruning: either through manually setting weights to zero or by utilizing binary masking layers. Furthermore, the DNN is pruned sequentially, meaning groups of weights are removed slowly from the network, with retraining in-between successive prunes, until a target sparsity is reached. Monte

Carlo simulations are also used to quantify the statistical distribution of network weights during pruning given random initialization of network weights. This work shows that sparse DNNs are capable of predicting the controls required for a simulated hawkmoth to move from one state-space to another, or through a sequence of control actions. Specifically, for a given signal-to-noise level the pruned network can perform at the level of the fully-connected network while requiring only a fraction of the memory footprint and computational power.

4.2 Methods

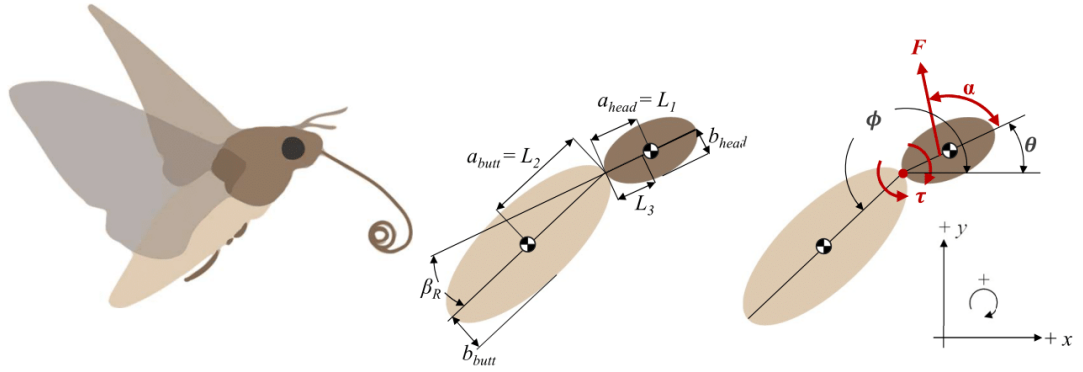
All code associated with the simulations and the DNNs is available on Github [103].

4.2.1 Moth model

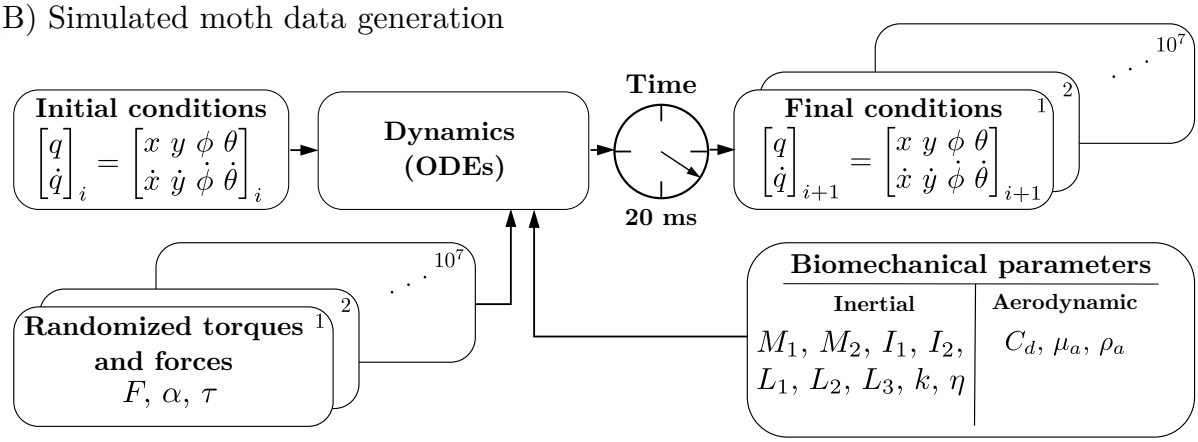
The simulated insect uses an inertial dynamics model developed in Bustamante et al., 2021 [14] and was inspired by hawkmoth flight control, *M. sexta* with body proportions rounded to the nearest 0.1 cm. The simulated moth was made up of two ellipsoid body segments, the head-thorax mass (m_1) and the abdomen mass (m_2). The body segments are connected by a pin joint consisting of a torsional spring and a torsional damper as seen in [9]. The simulated moth model could translate in x-y plane, and both the head-thorax mass, and the abdominal mass could rotate with angles (θ,ϕ) in the x-y plane. See Fig 4.1 for more description of the simulated insect, A.6 for the global model parameters, and A.7 for the calculated model variables.

The computational model of the moth had three control variables and four state-space variables (as well as the respective state-space derivatives). This model is by definition underactuated because the number of control variables is less than the degrees of freedom. The controls are as follows: F, the average force applied by the wings during each downstroke and upstroke; α the direction of force applied (with respect to the midline of the head-thorax mass); and τ, the abdominal torque exerted about the pin joint connecting the two body segment masses (with its equal and opposite response torque). In addition to the downstroke or upstroke averaged forces, the model includes gravitational forces, abdominal

(A) Schematic of simulated moth



(B) Simulated moth data generation



(C) Neural network inverse model

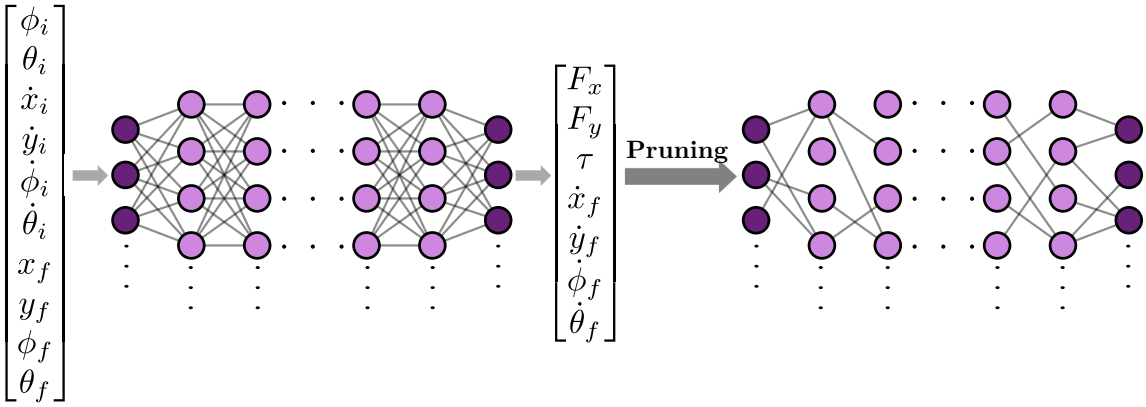


Figure 4.1: Inverse problem of flight control. (A) The moth body is made of two ellipses attached with a spring. There are three control variables (F, α, and τ) and four parameters to describe the state space (x, y, θ, and ϕ). See A.6 for the global parameters and A.7 for the calculated variables. (B) The differential equation solver solves the forward problem of insect flight control. (C) The neural network is an attempt to solve the inverse problem of flight control.

torques, and drag forces on the body. The controls are randomized every 20 ms, which is approximately the period of the wing downstroke or upstroke for *M. sexta* (25 Hz wing beat frequency) [82]. Thus our model is basically a simplified two-body dynamical system with thrust vectoring. Since our dominant focus is on hovering flight, this provides a reasonable basis for examining the control consequences of pruning a deep neural network. Fig 4.2 shows three example hovering trajectories of the simulated insect. All trajectories begin at the origin ($(x, y) = (0, 0)$). The grey dotted lines show the trajectory of the center of mass of each body segment and the red dotted line shows the trajectory of the thorax-abdomen joint.

The motion of the moth state-space is described by four parameters (x: horizontal position, y: vertical position, θ: head-thorax angle, and ϕ: abdomen angle), as well as the respective state-space derivatives ($\dot{x}$: horizontal velocity, $\dot{y}$: vertical velocity, $\dot{\theta}$: head-thorax angular velocity, and $\dot{\phi}$: abdomen angular velocity). The x and y position indicate the position of the pin joint where the head-thorax connects with the abdomen.

4.2.2 Generating training data

We used the ordinary differential equations from [14] (See Appendix, Equations 30-33) to generate a data set for training the deep neural network. All simulated trajectories were started from the origin (*i.e.*, (x_0,y_0) = (0,0)). We randomly sampled initial horizontal velocity ($\dot{x}_0$), vertical velocity ($\dot{y}_0$), head-thorax angle (θ_0), abdomen angle (ϕ_0), head-thorax angular velocity ($\dot{\theta}_0$), and abdomen angular velocity ($\dot{\phi}_0$) as shown in A.8. We also randomly sampled force (F), force angle (α), and torque (τ) as shown in A.8. The training data set is comprised of 10 million simulated trajectories and the test set contains an additional 5 million trajectories. The trajectories were simulated using the Python (Python Software Foundation, `https://www.python.org/`) function, `scipy.integrate.odeint` [96]. Fig 4.1 shows which variables were inputs and outputs from the differential equation solver.

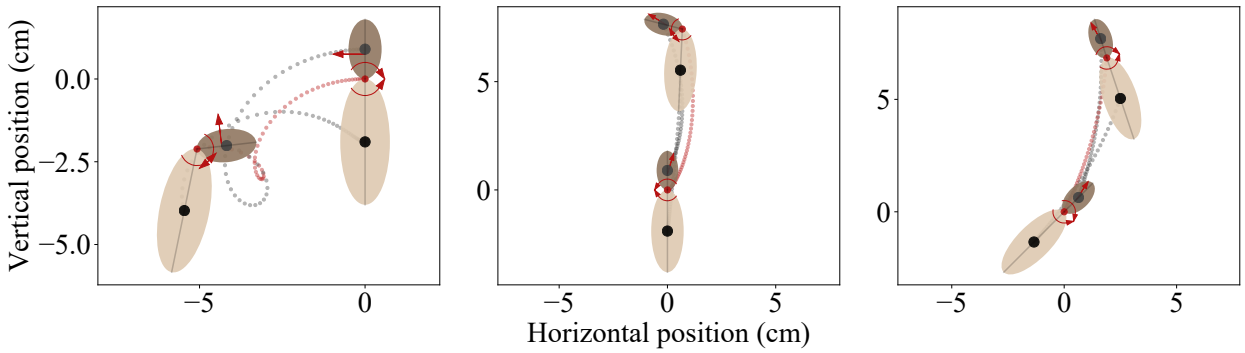


Figure 4.2: Example trajectories of the simulated insects. Each trajectory is 20 ms, and each starts at (x,y) = (0,0). Force (F) is indicated with the straight red arrow, and torque (τ) is shown with the curved arrows at the thorax-abdomen joint (red dot). The center of mass of each body segment is shown with black dots.

4.2.3 Data preparation for deep neural network training

The force (F) and force angle (α) were converted to horizontal and vertical components (F_x and F_y), using the following equations: $F_x = F \cdot cos(\alpha)$ and $F_y = F \cdot sin(\alpha)$. The data were split into training and validation sets for cross validation (80:20 split). The validation data is a sample used to provide an unbiased evaluation of a model fit while tuning the hyper parameters (such as number of hidden units, number of layers, optimizer, *etc.*). The data were scaled using a min-max scaler according to the training data set and transformed values to be between -0.5 and $+0.5$. The same scaler was then used to transform the validation and test data.

4.2.4 Training and pruning a deep neural network

The deep, fully-connected neural network was constructed with ten input variables and seven output variables (see Fig 4.1). The initial and final state space conditions are the inputs to the network: ($\dot{x}_i$, $\dot{y}_i$, ϕ_i, θ_i, $\dot{\phi}_i$, $\dot{\theta}_i$, x_f, y_f, ϕ_f, θ_f). The network predicts the control variables and the final derivatives of the state space in its output layer (F_x, F_y, τ, $\dot{x}_f$, $\dot{y}_f$, $\dot{\phi}_f$, $\dot{\theta}_f$).

The final derivatives of the state space were made outputs to be able to chain together 20 ms solutions to allow the moth to complete a complex trajectory for use in future work. The training and pruning protocols were developed using Keras [18] with the TensorFlow backend [2]. To scale up training for the statistical analysis of many networks, the training and pruning protocols were parallelized using the Jax framework [12].

To demonstrate the effects of pruning, the network was chosen to have a deep, feed-forward architecture with wide hidden layers (many more nodes than in the input and output layer). The network had four hidden layers with 400, 400, 400, and 16 nodes, respectively. Wide hidden layers were used rather than using a bottleneck structure (narrower hidden layer width) to allow the network to find the optimal mapping with little constraint, however, the specific choices of layer widths were arbitrary. The inverse tangent activation function was used for all hidden layers to introduce nonlinearity in the model. To account for the multiple outputs, the loss function was the uniformly-weighted average of the mean squared error for all the outputs combined.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{4.1}$$

For optimizing performance, there are several hyper-parameter differences in the TensorFlow model and the Jax model. In developing the training and pruning protocol in TensorFlow, the network was trained using the rmsprop optimizer with a batch size of 2^{12} samples. However, to scale up and speed up training we used the Jax framework, the Adam optimizer [55], and the batch size was reduced to 128 samples. Regularization techniques such as weight regularization, batch normalization, or dropout were not used. However, early stopping (with a minimum delta of 0.01 with a patience of 1000 batches) was used to reduce over-fitting by monitoring the mean squared error.

After the fully-connected network is trained to a minimum error, we used the method of neural network pruning to promote sparsity between the network layers. In this work, a target sparsity (percentage of pruned network weights) is specified and those weights are

forced to zero. The network is then retrained until a minimum error is reached. This process is repeated until most of the weights have been pruned from the network.

We developed two methods to prune the neural network: 1) a manual method that involves setting a number of weights to zero after each training epoch and 2) a method using TensorFlow's Model Optimization Toolkit [2] which involves creating a masking layer to control sparsity in the network. Both methods are described in detail in the following sections.

4.2.5 Manual Pruning

Algorithm 3 describes the a method of pruning in which the n weights whose magnitudes are closest to zero are manually set to zero. If N is the total number of weights in the network, the n weights are chosen such that n/N is equivalent to a specified pruning percentage (e.g. 15%, 25%, ..., 98%). After the n weights are set to zero, the network is retrained for one epoch. This process is repeated until the loss is minimized. After the network has been trained to a minimum loss, we select the next pruning percentage from the predetermined list and repeat the retraining process. The entire pruning process is repeated until the network has been pruned to the final pruning percentage in the list.

Upon retraining, the weights are able to regain a non-zero weight and the network is evaluated using these non-zero weights. Although this likely still captures the effects of pruning the network over the full training time, it is not true pruning in the sense that connections that have been pruned can regain weight.

4.2.6 Pruning using Model Optimization Toolkit

The manual pruning method described above has the downside of allowing weights to regain non-zero value after training. These weights are subsequently set back to zero on the next epoch, but the algorithm does not guarantee that the same weights will be pruned every time.

Algorithm 3: Sequential pruning and fine-tuning

```
Train fully-connected model until loss is minimized;
Define list of sparsity percentages;
for Each sparsity percentage do
    while Loss is not minimized do
        for Each epoch do
            Set n weights to zero s.t. n/N equals the sparsity percentage;
            Evaluate loss;
            Update weights;
        end
    end
end
```

To ensure weights remain pruned during retraining, we implemented the pruning functionality of a TensorFlow built toolkit called the Model Optimization Toolkit [2]. The toolkit contains functions for pruning deep neural networks. In the Model Optimization Toolkit, pruning is achieved through the use of binary masking layers that are multiplied element-wise to each weight matrix in the network. A four-layer neural network can be mathematically described the following way.

$$\hat{y} = \sigma_4(\mathbf{A_4}...(\sigma_1(\mathbf{A_1}x)) \tag{4.2}$$

In Eq 4.2, the inputs to the network are represented by x, the predictions by $\hat{y}$, the weight matrices by $\mathbf{A_i}$, and the activation function by σ_i, where $i = 1, 2, 3, 4$ for the four layers of the network. During pruning, the binary masking matrix, $\mathbf{M_i}$ is placed between each layer.

$$\hat{y} = \mathbf{M_4} \circ \sigma_4(\mathbf{A_4}...(\mathbf{M_1} \circ (\sigma_1(\mathbf{A_1}x))) \tag{4.3}$$

In Eq 4.3, the binary masking matrices, $\mathbf{M_i}$, are multiplied element-wise to the weight

Algorithm 4: Sequential pruning with masks and fine-tuning

```
Train fully-connected model until loss is minimized;
Define list of sparsity percentages;
for Each sparsity percentage do
    Define pruning schedule using ConstantSparsity;
    Create prunable model by calling prune_low_magnitude;
    Train pruned model until loss is minimized;
end
```

matrices ($\circ$ denotes the element-wise Hadamard product). The sparsity of each layer is controlled by a separate masking matrices to allow for different levels of sparsity in each layer. Before pruning, all elements of $\mathbf{M_i}$ are set to 1. At each pruning percentage (e.g. 15%, 25%, ..., 98%), the n weights whose magnitudes are nearest to zero are found and the corresponding elements of the the $\mathbf{M_i}$ are set to zero. The network is then retrained until a minimum error is achieved. The masking layers are non-trainable, meaning they will not be updated during backpropagation. Then, the next pruning percentage is selected and the process is repeated until the network has been pruned to the final pruning percentage.

In the TensorFlow Model Optimization Toolkit, the binary masking layer is added by wrapping each layer into a prunable layer. The binary masking layer controls the sparsity of the layer by setting terms in the matrix equal to either zero or one. The masking layer is bi-directional, meaning it masks the weights in both the forward pass and backpropagation step, ensuring no pruned weights are updated [109]. Algorithm 4 shows the pruning paradigm utilizing the Model Optimization Toolkit.

Rather than controlling for sparsity at each epoch of training, as was done in the manual pruning method described above, we control for sparsity each time we want to prune more weights from the network. Sparsity is kept constant throughout each pruning cycle and therefore we can use TensorFlow's built-in functions for training the network and regular-

ization.

4.2.7 Preparing for statistical analysis of pruned networks

To be able to train and analyze many neural networks, the training and pruning protocols were parallelized in the Jax framework [12]. Rather than requiring data to be in the form of tensors (such as in TensorFlow), Jax is capable of performing transformations on NumPy [39] structures. Jax however does not come with a toolkit for pruning, therefore pruning by way of the binary masking matrices was coded into the training loop.

The networks were trained and pruned using a NVIDIA Titan Xp GPU operating with CUDA [78]. At most, 400 networks were trained at the same time and the total number of networks used in the analysis was 1320. These networks were all trained with identical architectures, pruning percentages, and hyper-parameters. The only difference between the networks is the random initialization of the weights before training and pruning. The Adam optimizer [55] and a batch size of 128 were used to speed up training and cross-validation was omitted. However, early stopping was used on the training data to avoid training beyond when the loss was adequately minimized. Additionally, early stopping was used to evaluate the decrease in loss across batches, rather than epochs.

4.3 Results

4.3.1 Network pruning results

Fig 4.3 shows the learning curve for a network trained using the sequential pruning protocol with TensorFlow's Model Optimization Toolkit (see Methods section for details) [2]. The network is trained until a minimum error is reached, and then pruned to a specified sparsity percentage and then retrained until the loss is once again minimized. The sparsity (or pruning) percentages are shown in Fig 4.3 where they occur in the training process. An arbitrary threshold error of 10^{-3} (shown as a red, dashed line) was chosen to define the optimally sparse network (i.e. sparsest possible network that performs under the specified loss). This specific

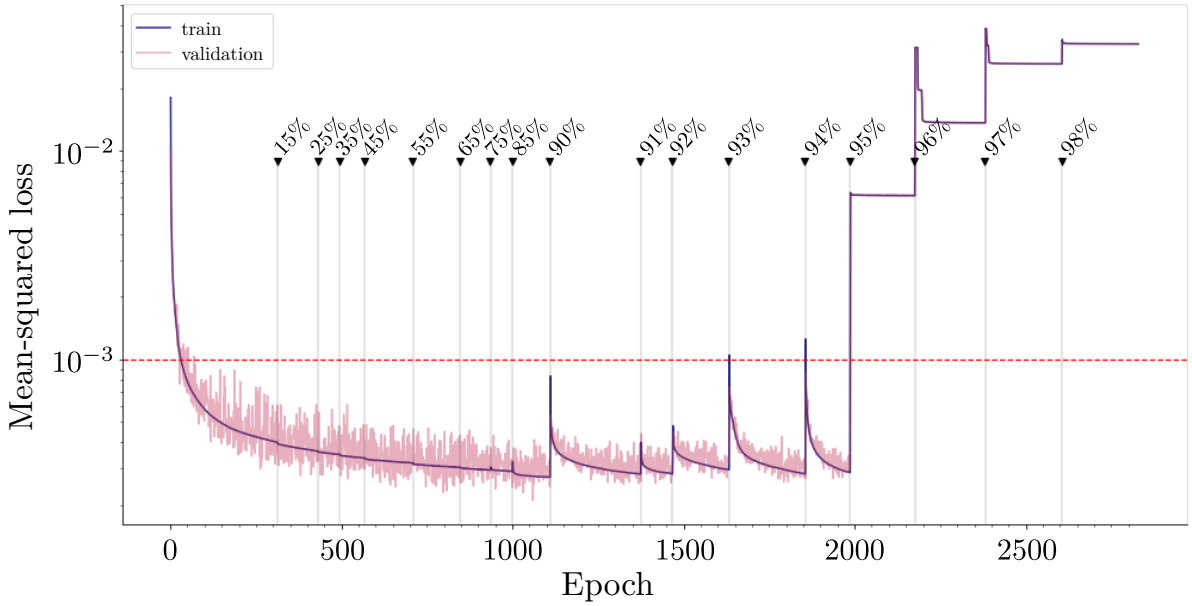


Figure 4.3: Learning curve for sequential pruning of network. Fully-connected neural network is trained until the mean-squared error is minimized. Then, the network is sequentially pruned by adding in masking layers and trained again. The performance of the network improves below the minimum error achieved by the fully-connected network for low levels of pruning, but performs comparably to the fully-connected network until 94% of the network is pruned.

threshold value was chosen because it is near the performance of the trained, fully-connected network. In practice, the red line represents the noise level encountered in the flight system. Specifically, given a prescribed signal-to-noise ratio, we wish to train a DNN to accomplish a task with a certain accuracy that is limited by noise. Thus high-fidelity models, which can only practically exist with perfect data, are traded for sparse models which are capable of performing at the same level for a given noise figure. In the example in Fig 4.3, the optimally sparse network occurs at 94% sparsity (or when only 6% of the connections remain). Beyond 94% sparsity, the performance of the network breaks down because too many critical weights

have been removed.

4.3.2 Monte Carlo results

To compare the effects of pruning across networks, we trained and pruned 1320 networks with different random initializations on the same data set. In this experiment, the hyper-parameters, pruning percentages, and architecture are held constant. Fig 4.4 shows the training curves of 9 sample networks. The red, dashed line in each of the panels represents the same threshold as in Fig 4.3 (10^{-3}). The black, solid lines in Fig 4.4 represent the optimally sparse networks. Although the majority of networks in this subset breakdown at 93% sparsity, a few breakdown at higher and lower levels of connectivity.

Fig 4.5 shows the loss after pruning the 1320 networks at varying pruning percentages (from 0% sparsity to 98% sparsity). The box plot in Fig 4.5 is directly comparable to Fig 4.3, but it is the compilation of the results for many different networks. The networks do not all converge to the same set of weights, which is evident by the numerous outliers, as well as the variance around the median loss.

The median minimum loss achieved by the networks before pruning is 7.9 x 10^{-4}. The first box in the box plot in Fig 4.5 corresponds to the losses of all the trained networks before any pruning occurs. The variance on the loss is relatively small, but there are several outliers. Once again, the red, dashed line in the box plot in Fig 4.5 represents the threshold below which a network is optimally sparse. Many networks follow a similar pattern and perform under the threshold until they exceed 93% sparsity. Also, many networks perform better than the median performance of the fully-connected networks when pruned up to 85% sparsity.

The number of optimally sparse networks in each sparsity category is shown in the bar plot at the top of Fig 4.5. Of the 1320 networks trained, 858 of the networks are optimally sparse at 93% sparsity. A small number of networks (5) remain under the threshold up to 95% pruned. Note that the total number of networks represented in the bar plot does not add up to 1320. This is because several networks never perform below the threshold throughout

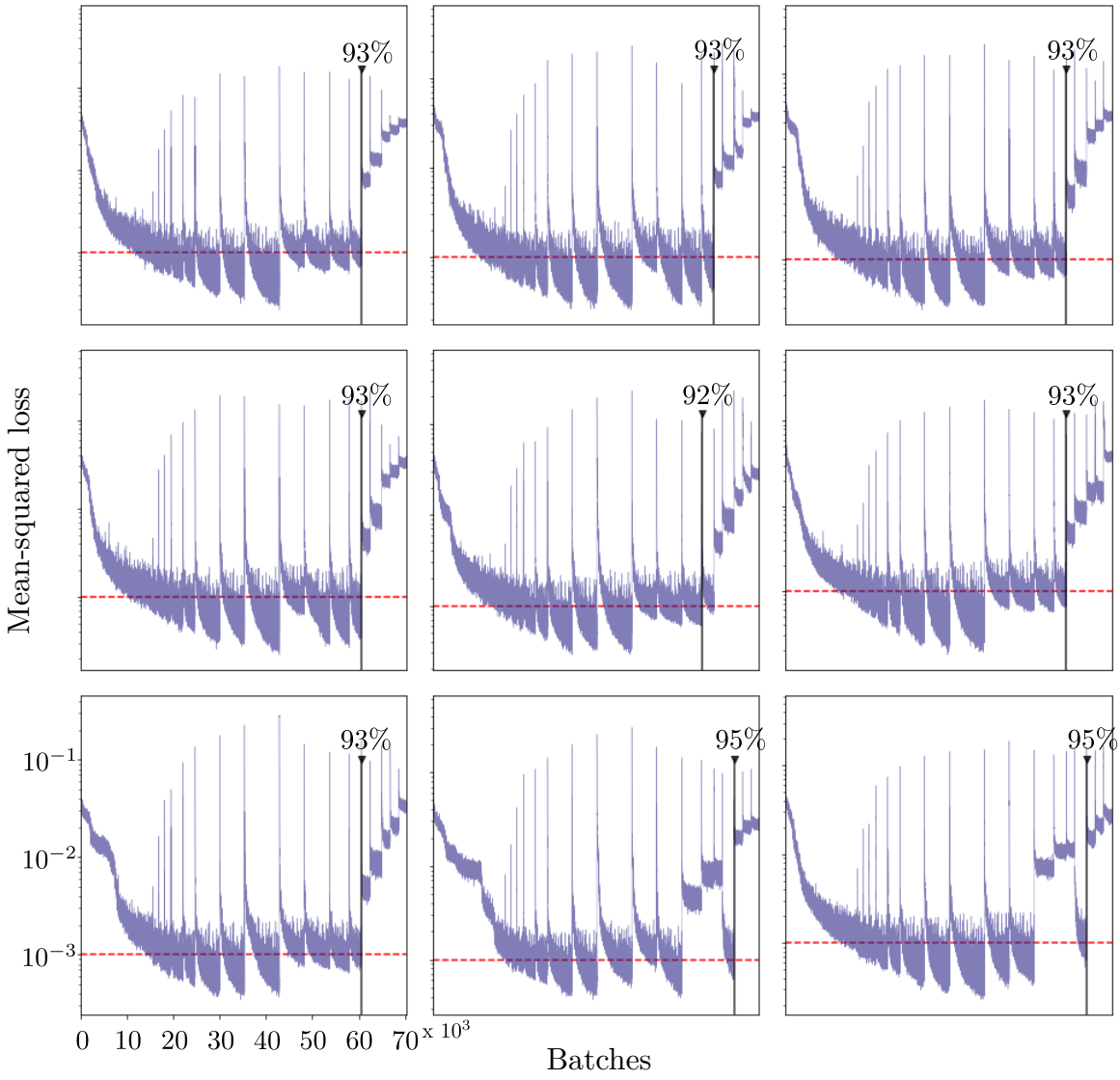


Figure 4.4: Performance breakdown of 9 sample pruned networks. The networks are sequentially pruned. Each network is evaluated to find the optimally sparse network. The red dashed line represents the performance threshold (10^{-3}). The sparsest network that performs below this threshold is shown by the solid, black vertical line.

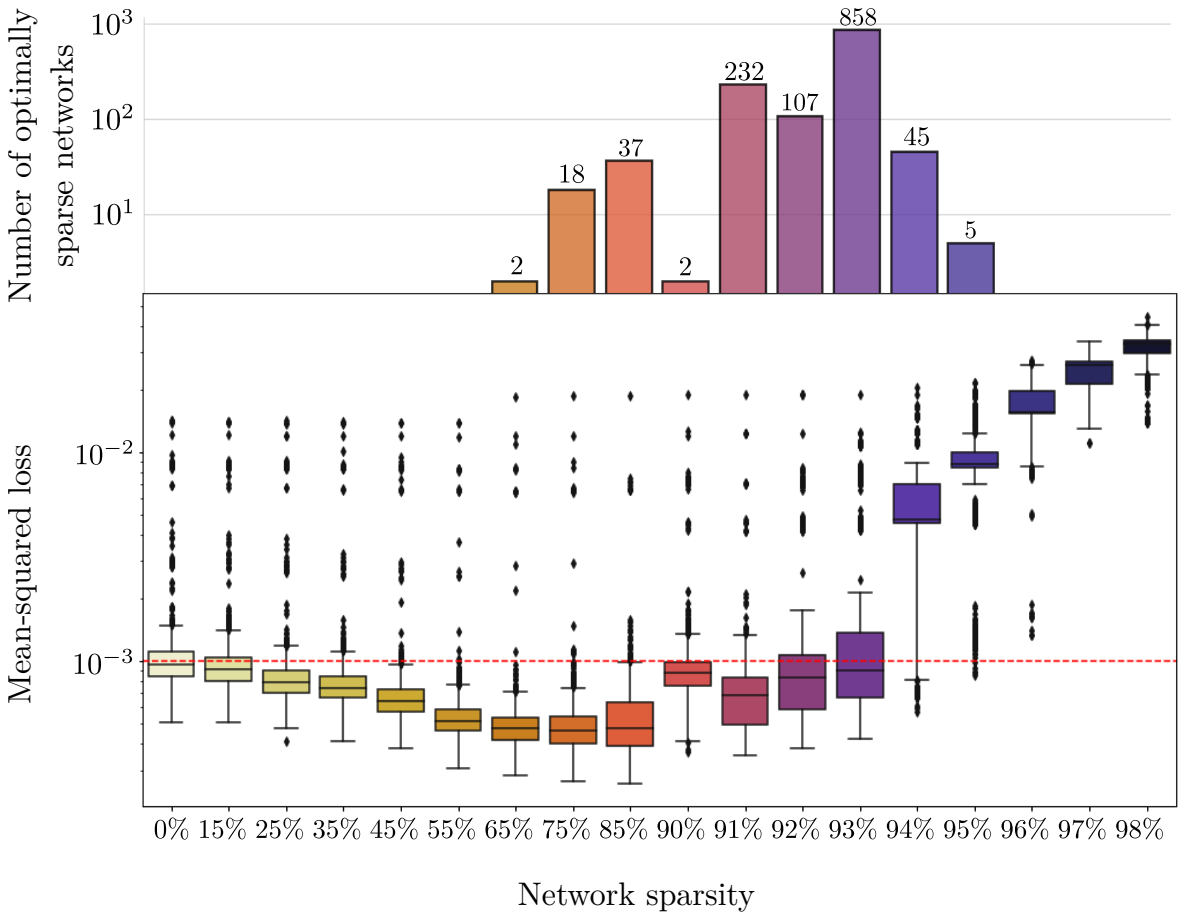


Figure 4.5: Monte Carlo analysis of pruned networks. 1320 networks are sequentially pruned and loss of the pruned networks at each sparsity percentage is recorded in the box plot. The bar plot records the number of networks that make it to the corresponding sparsity percentage before exceeding the hypothetical threshold (10^{-3}).

Table 4.1: **Number of remaining parameters in networks pruned to 93% sparsity** This table gives the average number of remaining weights in each layer of the networks pruned to 93% sparsity. The variance on the number of connections, as well as the fraction of remaining connections are also given.

Layer i	Average number remaining	Variance	Percentage remaining
1	280	23	0.07
2	11199	0	0.07
3	11199	0	0.07
4	447	0.04	0.07
5	8	6	0.07

the sequential pruning process (see outliers in Fig 4.5).

4.3.3 Analysis of layer sparsity

The subset of optimally sparse networks pruned to 93% is used in the following analysis of network structure (858 networks). The sparsity across all the layers was found to be uniform (7% of weights remain in each layer) despite not explicitly requiring uniform pruning in the protocol. Table 4.1 shows the average number of remaining connections across the 858 networks, as well as the variance and the fraction of remaining connections.

Fig 4.6 shows a box plot of the number of connections from the input layer to the first hidden layer for the subset of pruned networks. Interestingly, the initial head-thorax angular velocity was completely pruned out of all of the networks in the subset, meaning it has no impact on the output and predictive power of the network. Additionally, the initial abdomen angular velocity connects to either zero, one, or two nodes in the second hidden layer, while all the other inputs have a median connection to at least 5% of the weights in the first hidden layer.

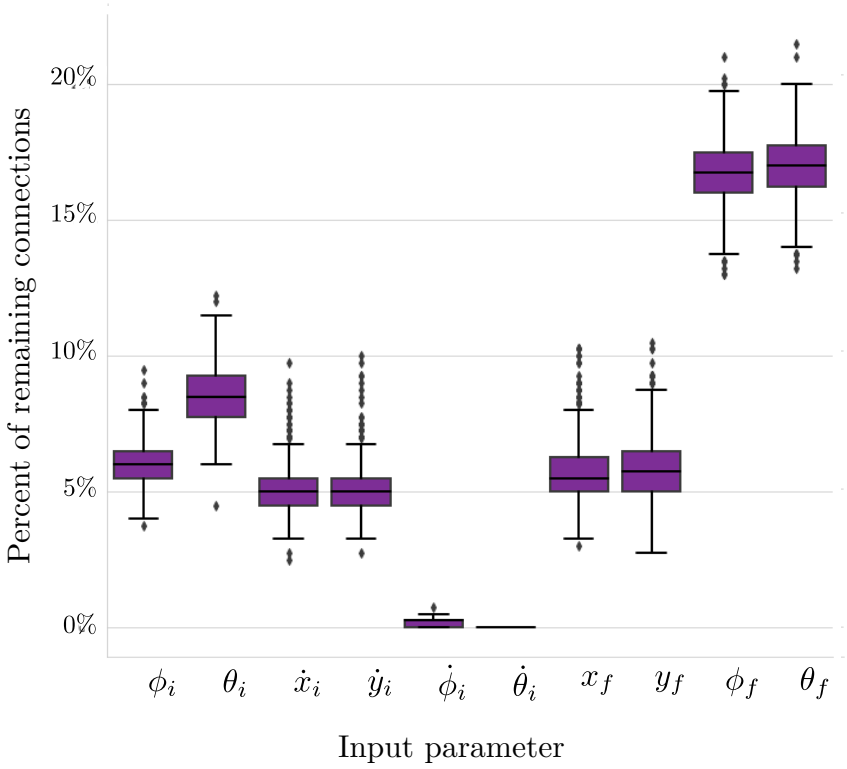


Figure 4.6: Sparsity of input layer of networks pruned to 93% sparsity. Each box represents the number of connections remaining between a parameter in the input layer and the first hidden layer. For all 858 networks in this group, $\dot{\theta}_i$ was pruned entirely from the network.

4.4 Discussion

In this study, we set out to investigate whether a sparse DNN can control a biologically relevant motor-task – in our case the dynamic control of a hovering insect. Taking inspiration from synaptic pruning found across wide ranging animal taxa, we pruned a DNN to different levels of sparsity in order to find the optimal sparse network capable of controlling moth hovering. The DNN uses data generated by the inertial dynamics model in [14] which models the forward problem of flight control. In this work, the DNN models the inverse problem of flight control by learning the controls given the initial and final state space variables.

Through this work, we found that sparse DNNs are capable of solving the inverse problem of flight control, i.e. predicting the controls that are required for a moth to hover to a specified state space. In addition, we demonstrate that across many networks, a network can be pruned by as much as 93% and perform comparably to the median performance of a fully-connected network. However, there are sharp performance limits and most networks pruned beyond 93% see a breakdown in performance. We found that although uniform pruning was not enforced, on average, each layer in the network pruned to match the overall sparsity (i.e. sparsity of each layer was 93% for networks pruned to overall sparsity of 93%). Finally, we looked at the sparsity of individual layers and found that the initial head-thorax angular velocity is consistently pruned from the input layer of networks pruned to 93% sparsity, indicating a redundancy in the forward original model.

Though we have shown that a DNN is capable of learning the controls for a flight task, there are several limitations to this work. Firstly, though the model in [14] used to generate the training data provided control predictions for accurate motion tracking in a two-dimensional task, biological reality is more rich and complex than can be captured by the forward model. Thus, since the DNN is trained with this data, it is only capable of learning the dynamics captured in the model in [14]. Furthermore, the size, shape, and body biomechanics of this systems all matter. This study uses the same global parameters across the data set (see A.6), but, in reality, these parameters vary significantly (across insect taxa and

within the life of an individual) and this likely affects the performance of the DNN.

We have shown here that DNNs are capable of learning the inverse problem of flight control. The fully-connected DNN used here learned a nonlinear mapping between input and output variables, where the inputs are the initial and final state space variables and the outputs are the controls and final velocities. A fully-connected network can learn this task with a median loss of 7.9 x 10^{-4}. However, due to the random initialization of weights preceding training, some networks perform as much as an order of magnitude worse (see Fig 4.5). This suggests that the performance of a trained DNN is heavily influenced by the random initialization of its weights.

We used magnitude-based pruning to sparsify the DNNs in order to find the optimal, sparse network capable of controlling moth hovering. For the task of moth hovering, a DNN can be pruned to approximately 7% of its original network weights and still perform comparably to the fully-connected network. The results of this analysis show that when trained to perform a biological task, fully-connected DNNs are indeed over-parameterized. Much like their biological counterparts, DNNs do not require fully-connected connectivity to accomplish this task. Additionally, flying insects maneuver through high noise environments and therefore perfect flight control is traded for robustness to noise. It is therefore assumed that the data has a given signal-to-noise ratio or performance threshold. The performance threshold represented by the red dashed line in Fig 4.3, Fig 4.4, and Fig 4.5 was arbitrarily chosen to represent a loss comparable to the loss of the fully-connected network (i.e. 0.001). In other words, this line represents a noise threshold, below which the network is considered well-performing and adapted to noise. It has been shown that biological motor control systems are adapted to handle noise [30]. Biological pruning may be a mechanism for identifying sparse connectivity patterns that allow for control within a noise threshold.

On average, when the networks are pruned beyond 7% connectivity, there is a dramatic performance breakdown. Beyond 93% sparsity, the performance of the networks break down because too many critical weights have been removed. A significant proportion of the 1320 networks breakdown before they reach 7% connectivity (approximately 30% of the networks).

This again supports the aforementioned claim that the random initialization of the weights before training affects the performance of a DNN and can be exacerbated by neural network pruning. Additionally, this shows that there exists a diversity of network structures that perform within the bounds of the noise threshold.

To investigate the substructure of the well-performing, sparse networks, we looked closer at the subset of networks that were optimally sparse at 93% pruned (858 networks). We have shown that the average sparsity of each layer in this subset is uniform, meaning each of the five layers have approximately 7% of their original connections remaining. However, the variance on the number of remaining connections between input layer and first hidden layer and between the final hidden layer and the output layer is markedly higher than the variance in the weight matrices between the hidden layers. This suggests that in networks pruned to 93% sparsity, the greatest amount of change in network connectivity occurs in input and output layers. However, there are notable features in the connectivity between the input and first hidden layer that are consistent across the 858 networks. Fig 4.6 shows that the input parameter, initial head-thorax angular velocity ($\dot{\theta}_i$), is completely pruned from all of the 858 networks. The initial abdomen angular velocity ($\dot{\phi}_i$) is also almost entirely pruned from all of the networks. All of the other input parameters maintain an median of at least 5% connectivity to the first hidden layer. The complete pruning of $\dot{\theta}_i$ suggests a redundancy in the original forward model. However, this redundancy makes physical sense because θ_i and ϕ_i are coupled in the original forward model.

The results of this study pose an interesting question about how the size of the initial network architecture affects the resultant pruning statistics. The networks pruned in this study are feed-forward, each with four hidden layers with 400, 400, 400, 16 nodes, respectively. This choice of architecture is somewhat arbitrary, however through the process of tuning and cross-validating the fully-connected network, we converged to a set of hyper-parameters (including the size of the hidden layers) which resulted in the most optimally performing network. To begin to explore the effect that initial network architecture size has on the pruning statistics, we repeated the experiment with increasingly smaller network

architectures. For example, in A.6 we trained 400 networks, each with four hidden layers with 200, 200, 200, 8 nodes. A.2 shows the same results for networks of sizes 100, 100, 100, 8 and A.3 shows the results for networks of size 50, 50, 50, 8. These decreases in hidden layer widths correspond to a decrease in the total number of weights across the networks from $330,512$ (for the original networks in Fig 4.5) to $83,656$ (A.1), $21,856$ (A.2), and $5,956$ (A.3). Across all networks, there is a slight improvement in performance for low levels of pruning. All networks show a performance breakdown, however the sparsity at which the breakdown occurs changes with the size of the network. For example, the networks in A.3 show a performance breakdown at 65% or when there are $2,084$ weights remaining. This is compared to the original 1320 networks which showed a performance breakdown at 93% or when there are $23,135$ weights remaining. The initial architecture of the network affects the achievable sparsity by the pruning protocol employed here. Additionally, smaller network architectures result in more volatility when higher levels of sparsity are reached. However, the results of these preliminary experiments only begin to explore the relationship between initial network architecture and resultant pruning statistics. We found that as the network architecture is made smaller, the raw number of parameters post-pruning is fewer. Whether these extra small networks are as robust to noise or better at generalizing to unseen data is yet to be seen. It is also unclear what the optimal starting architecture size should be because large, over-parameterized networks are thought to be more efficient to optimize via gradient descent [66]. As stated, these preliminary experiments open up many interesting questions to be explored in future work.

In this work, we have shown that a sparse neural network can learn the controls for a biological motor task and we have also shown, via Monte Carlo simulations, that there exists at least some aspects of network structure that are stereotypical. There are several computationally non-trivial extensions to the work presented here. Firstly, network analysis techniques (such as network motif theory) could be used to further compare the pruned networks and investigate the impacts of neural network structure on a control task. Network motifs are statistically significant substructures in a network and have been shown to

be indicative of network functionality in control systems [48]. Other areas of future work include investigating the sparse network's response to noise and changes in the biological parameters. Biological control systems are adapted to function adequately in the presence of noise. Pruning improves the performance of neural networks up to a certain level of sparsity, however the effects of noise on this bio-inspired control task are yet to be explored. Furthermore, the size and shape of a real moth can change rapidly (e.g. change of mass after feeding). The question of whether sparsity improves robustness in the face of such physical parameters could also be a future extension of this work.

4.5 Conclusion

Synaptic pruning has been shown to play a major role in the refinement of neural connections, leading to more effective motor task control. Taking inspiration from synaptic pruning in biological systems, we apply the equally thoroughly investigated method of DNN pruning to the inverse problem of insect flight control. We use the inertial dynamics model in [14] to simulate examples of *M. sexta* hovering flight. This data is used to train a DNN to learn the controls for moving the simulated insect between two points in state-space. We then prune the DNN weights to find the optimally sparse network for completing flight tasks. We developed two paradigms for pruning: via manual weight removal and via binary masking layers. Furthermore, we pruned the DNN sequentially with retraining occurring between prunes. Monte Carlo simulations were also used to quantify the statistical distribution of network weights during pruning to find similarities in the internal structure across pruned networks. In this work, we have shown that sparse DNNs are capable of predicting the controls required for a simulated hawkmoth to move from one state-space to another.

Chapter 5

MOTIF DISTRIBUTION AND FUNCTION IN SPARSIFIED DEEP NEURAL NETWORKS

In this work, we characterize the connectivity structure of feed-forward, deep neural networks (DNNs) using network motif theory. To address whether a particular motif makeup is characteristic of the training task, we compare the connectivity structure of 350 trained DNNs trained to simulate the same bio-mechanical flight control system with different randomly initialized parameters. We developed and implemented algorithms for counting 2^{nd}- and 3^{rd}-order motifs and calculate their significance using their z-score. The DNNs are trained to model the inverse of the flight dynamics model in [15] (i.e., predict the controls necessary for controlled flight from the initial and final state-space inputs) and are sparsified through an iterative pruning and retraining algorithm [104]. We show that, despite random initialization of network parameters, enforced sparsity causes DNNs to converge to similar connectivity patterns as characterized their network motif makeup. We propose several ideas for future experiments in the Discussion of this manuscript.

5.1 Introduction

Complex networks are prevalent in nature, technology, and in mathematical and computational modeling. Unlike random and lattice networks, complex networks are characterized by a non-trivial topology which enables complex collective dynamics. The study of complex networks spans disciplines from discrete mathematics to the social sciences, but it has historically focused on natural, physical, and real-world networks. For example, the spread of disease is affected by the connectivity and organizing principles of societal networks [80]. In animal neural systems, network structures and specialized synaptic pathways evolved for

specific behaviors [25] or encode critical function [57, 58]. While it is generally accepted that the behavior of a complex network is intrinsically tied to its structure and function, little is known about how (if at all) local connectivity patterns affect the overall behavior or function of a network. [92, 77, 3]. We show here that the connectivity structure of sparse deep neural networks can be characterized through its statistically significant sub-graphs, or network motifs, in order to encode function and dynamics.

Two well known topological properties of complex networks are the small-world property and the scale-free property. In small-world networks, a node is connected to most other nodes by way of neighboring connections [98]. Concretely, a small-world network is one in which the average distance between two nodes scales with the logarithm of the number of nodes in the network. This property allows almost all nodes in a sparsely connected network to communicate with all others in the network. The gene co-expression network in yeast, *Saccharomyces cerevisiae*, exhibits both the small-world and scale-free property of complex networks [95]. A scale-free network is one where the distribution of nodal degrees (i.e., the number of connections a given node has) follows a power law [6]. The scale-free property is exhibited in many different networks, from links between web pages to citations on publications. Both of these discoveries support the assumption that complex networks are not randomly connected, but are instead structured to enable specific dynamics, function and behavior.

The connectivity of complex networks can also be characterized by the makeup of the sub-graphs within the network. A network motif is a sub-graph within a larger network that occurs significantly more than it would occur in an equivalent randomly connected network [4]. Network motifs have been discovered in natural networks ranging from gene transcription to ecological networks [4, 72, 91]. In some contexts, individual network motifs have clear, interpretable functions. For example 2^{nd}-order chain sub-graphs (three nodes chained together in a feed-forward manner), occur with very high significance in food-chain networks, representing the hierarchical relationship between predators and prey. Relating the function of individual sub-graphs to the overall dynamics of a complex network is more

difficult. In [49], the authors relate the dynamics of a network to the statistics of its network motifs. The authors constrain their study to linear, time-invariant networks and derive the network transfer function (i.e., function that transforms the time-dependent input to the time-dependent output) in terms of motif cumulants (simple statistics of network motifs). They apply the method to several example real-world networks (power grids and *C. elegans* neuronal networks) and show that a few low-order motifs are needed to model the network transfer function.

Meanwhile, in the fields of computer science and engineering, computational models for complex tasks such as human language [79] and high-dimensional non-linear fluid flow [70, 10] have seen great success thanks primarily to advances in machine learning and specifically deep neural networks (DNNs). DNNs are complex networks that can be characterized by various statistical and/or topological approaches. In [76], the authors found that the topological complexity of a binary classification data set is reduced as the data passed through the network. In another study, researchers generate bio-instantiated recurrent neural networks, novel DNN architectures that are built from empirical data on animal neural networks [34]. The challenge of studying the connectivity structure of a DNN is made difficult by the sheer number of parameters in the network. Consequently, DNN sparsification is a helpful tool in the effort to uncover the relationship between DNN connectivity patterns and their function.

There are several different methods for reducing the number of parameters in a trained DNN, including by initially defining a sparse architecture or through regularization. One popular method is neural network pruning, which involves the systematic removal of parameters from a trained DNN. Pruning was first introduced in [62], in which the authors show that sparsifying a DNN via pruning improves generalization and efficiency. Pruning is inspired by a biological process called *synaptic pruning*; the elimination of synaptic connections during development. Synaptic pruning plays a role in the refinement of neural pathways and contributes, in part, to efficient cognitive function [19]. In machine learning, several works have shown that pruning can reduce the number of parameters in a trained DNN by as much as 93% [37, 104]. Pruning has also been used to discover sub-networks

that, when trained in isolation, will achieve comparable performance to the fully-connected network [29]. Sparsification via pruning is a simple method to reduce the number of network parameters and elucidate relevant and necessary connections for DNN performance.

In this work, we seek to answer the question: are trained DNNs composed of a set of statistically significant sub-graphs or are they more or less randomly connected? We use the distribution of network motifs to characterize the connectivity structure of sparse DNNs trained to simulate a bio-mechanical flight control system. To address whether a particular motif makeup is characteristic of the training task, we compare the results across 350 DNNs trained to model the same system, but with different randomly initialized parameters. We developed and implemented algorithms for counting 2^{nd}- and 3^{rd}-order motifs in feed-forward, sparse deep neural networks and calculate their significance using their z-score. The DNNs are trained to model the inverse of the flight dynamics model in [15] (i.e., predict the controls necessary for controlled flight from the initial and final state-space inputs). We sparsify the DNNs through an iterative pruning and retraining algorithm [104]. This work shows that, despite random initialization of network parameters, enforced sparsity will cause DNNs to converge to similar connectivity patterns as characterized their network motif landscape. Concretely relating the connectivity patterns to the function of a trained DNN is beyond the scope of this work. However, we proposed several ideas for future experiments in the Discussion of this manuscript.

5.2 Methods

The DNNs used in this study were trained to model the insect flight dynamics model described in [15, 104]. The methods for training and pruning closely follow the procedure described in [104]. The training and pruning procedure is summarized here and differences between the two studies will be highlighted. All code associated with the simulations, training and pruning DNNs, and network motif analysis is available on Github [**?**]. The network characterization is performed on a specific model given that the control objectives, or the ground truth objective, is known. Thus the results can be validated against an interpretable

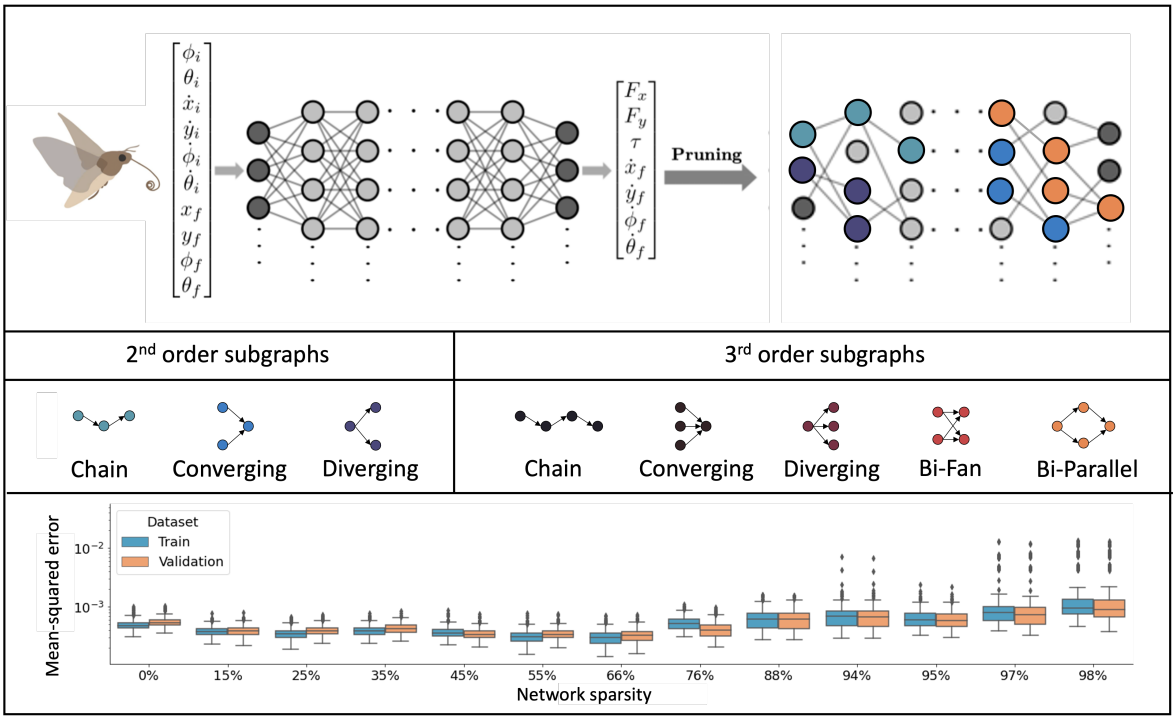


Figure 5.1: Top: A densely connected DNN is trained to predict the control variables for the task of insect hovering. Initial and final state-space variables are used as inputs to the network. The trained network is pruned to maximal sparsity with little decrease in performance. Middle: A subset of 2^{nd}- and 3^{rd}-order network sub-graphs that can exist in a feed-forward DNN. Bottom: Training and validation loss over 350 networks pruned to different sparsity levels.

and well-studied model. The algorithms developed, however, can be more broadly applied to complex networks in general with the goal of extracting insight into the underlying dynamics.

5.2.1 Network training data

The insect flight simulation uses an inertial dynamics model developed in Bustamante et al., 2022 [15] and is inspired by the flight mechanics of the hawkmoth, *Manduca sexta*. The insect is modeled by two conjoined ellipses representing the head-thorax and abdomen

body segments. The dynamics of the insect is constrained to two dimensions (x and y) and is controlled by three control variables, F, the average force applied by the wings, α the direction of force applied, and τ, the abdominal torque exerted about the pin joint connecting the two body segment masses. The system's state-space is described by four parameters (x: horizontal position, y: vertical position, θ: head-thorax angle, and ϕ: abdomen angle), as well as the respective state-space derivatives ($\dot{x}$: horizontal velocity, $\dot{y}$: vertical velocity, $\dot{\theta}$: head-thorax angular velocity, and $\dot{\phi}$: abdomen angular velocity). The model described in [15] is a forward model, where ordinary differential equations are used to predict the insect's final position and velocities from the initial state-space variables. More details about the moth model can be found in [15, 104].

5.2.2 Neural network training and pruning

The above described model is used to generate data to train the DNNs studied in this work. All simulated trajectories were started from the origin and initial state-space and control variables were randomly sampled. The training data set is comprised of 10 million simulated trajectories and the test set contains an additional 5 million trajectories.

The deep, fully-connected neural network has ten input variables ($\dot{x}_i$, $\dot{y}_i$, ϕ_i, θ_i, $\dot{\phi}_i$, $\dot{\theta}_i$, x_f, y_f, ϕ_f, θ_f) and seven output variables (F_x, F_y, τ, $\dot{x}_f$, $\dot{y}_f$, $\dot{\phi}_f$, $\dot{\theta}_f$). Prior to pruning, all of the networks are initialized with a deep, feed-forward structure, with four hidden layers with 400, 400, 400, and 16 nodes. The inverse tangent activation function is used in every layer to introduce non-linearity to the model.

This is a multi-output regression model, so during training, the uniformly-weighted average of the mean squared error across the outputs was used as a loss function. We used the Jax deep learning library to parallelize and consequently speed up training of the models. The Adam optimizer and a batch size of 128 samples were likewise chosen to decrease training time. No regularization techniques (such as weight regularization or dropout) were used, but early stopping (with a minimum validation loss delta of 0.01 and a patience of 1000 batches) was used to halt training at model convergence.

The goal of this study is to compare the connectivity structure across many DNNs. To speed up the training and pruning of the DNNs, we parallelized the training and pruning of 350 networks. This was achieved by optimizing over the performance of all networks collective error (i.e., loss was minimized over all 350 networks). After the networks are trained to convergence, a round of neural network pruning reduces the total number of weights by some fixed and predetermined amount (e.g. 15%, 25%, etc.). The result of the experiment are snapshots of 350 networks pruned at varying levels of sparsity.

5.2.3 Post-pruning

After the networks are trained and pruned as described above (and in detail in [104]), there still exist some weights that are obselete. This occurs when an upstream weight is removed such that all downstream nodes no longer have an input. Since the pruning algorithm utilized here does not prevent such prunes, these weights are retroactively removed. The precise removal of these connections is described in detail in the Supplementary Material.

5.2.4 Motif Significance

We used network motifs to access the similarity between trained DNNs connectivity structures. Finding a network motif of a larger network is a two step process. First, the total number of a desired sub-graph must be counted. That number is then used to calculate the significance of the the sub-graph. Here we use the z-score to determine the significance of a given sub-graph.

Sub-graph counting algorithms

There are many open source network motif counting software available [**?**]. However, these software are usually built for larger, more complex networks and can be slow to use. We decided to write our own counting algorithms specifically tailored to the task of exactly counting sub-graphs in feed-forward, sparse DNNs. These algorithms take advantage of the

connectivity matrices of the feed-forward DNNs and utilize simple matrix operations and combinatorics to quickly calculate the number of a given sub-graph.

The counting algorithms can all be found in the Supplementary Material of this manuscript. Every counting algorithm describes the counting process for the total number of occurrences of the specified sub-graph in a single network. We will walk through one example here, as all of the counting algorithms follow the same general pattern. In the following algorithm, every pruned network is considered a list of sparse, binary matrices or masks. The variable *mask list* refers to the list of sparse matrices associated with one pruned network. Likewise, *mask* refers to a single sparse, binary matrix. Algorithm 5 shows the steps for counting the 2^{nd}-order converging sub-graph in a sparse, feed-forward DNN.

Algorithm 5: 2^{nd}-order converging sub-graph counting

```
total = 0;
for mask in mask list do
    for column in mask do
        n ← count number of non-zero elements;
        if n ≥ 2 then
            total += (n 2);
    end
end
```

The first loop in Algorithm 5 loops over the layers of a network. The second loop loops over the input nodes of the layer. The number of output nodes, n is the number of non-zero elements connected to an input node. If an input node has two or more output nodes, the total number of 2^{nd}-order converging sub-graphs is $\binom{n}{2}$. The total number of sub-graphs is tallied until all layers of the DNN have been iterated over.

Z-score calculation

We use the z-score to determine sub-graph significance and find network motifs. The z-score for a given motif, Z_m, is defined as

$$Z_m = \frac{N_{real} - \langle N_{random} \rangle}{\sigma_{random}} \tag{5.1}$$

where N_{real} is the total number of times the given sub-graph occurs in a sparse, DNN. This is found by one of the sub-graph counting algorithms described above and in the Supplemental Material. N_{random} is the total number of times the given sub-graph occurs in an equivalently sized, but randomly connected, sparse DNN. This number is found by generating a randomly connected graph, with the same feed-forward structure as the trained DNNs and counting the number of sub-graphs according to the sub-graph counting algorithms. Z_m must be calculated over the average across many randomly connected networks. For each of the 350 networks analyzed here (and across all sparsity levels), we generate 1000 randomly connected networks for this calculation. The generation of equivalently sized, but randomly connected networks is described in detail in the Supplementary Material. The average number of sub-graphs and the variance of the number of sub-graphs is found and used to calculate the final z-score of the motif.

5.3 Results

The main finding of this work is that enforced sparsity during DNN training encourages network connectivity structure as characterized by network motifs. We find that there are distinct patterns in the network motif structure across networks as well as throughout the sparsification process (via pruning).

5.3.1 Patterns across networks

Figure 5.2 shows the summary statistics of the network motif distributions across the 350 networks pruned to 98%. Each distribution in the violin plot shows the distribution of z-scores for each motif over the 350 networks pruned to 98%. Across the networks, distinct

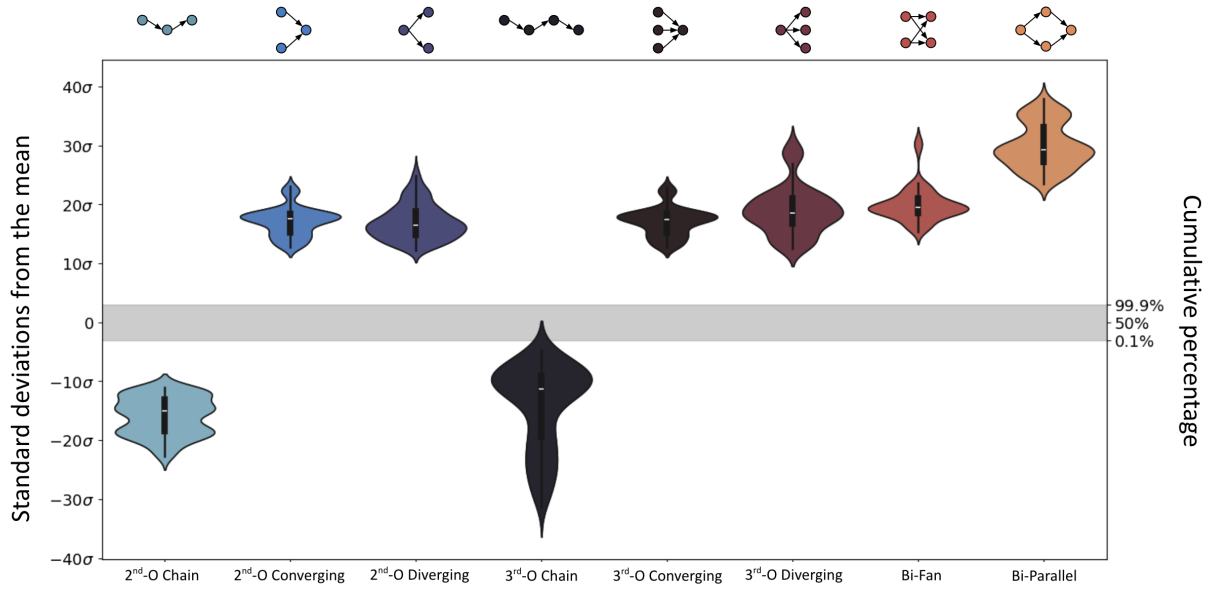


Figure 5.2: Distributions of z-scores across 350 DNNs pruned to 98% sparsity. Top axis shows the motif, left axis shows the standard deviations from the mean (or z-score), and the right axis shows the cumulative percentage.

patterns of over- and under-representation can be seen for different network motifs. Furthermore, at this level of sparsity, all motifs are either highly over- or under-represented, with approximate average motif representation well above (or below) $\pm 10\sigma$.

5.3.2 Patterns across sparsity levels

Figure 5.3 shows how the network motif distribution changes as the network is pruned (according to the pruning paradigm described in the Methods section). Each panel shows how the representation of the motif pictured changes with the sparsity levels labeled on the x-axis (i.e., 0 to 98%). From dense to sparse, network motifs are either over- or under-represented throughout the pruning process, except for a few outliers in the low sparsity networks (see

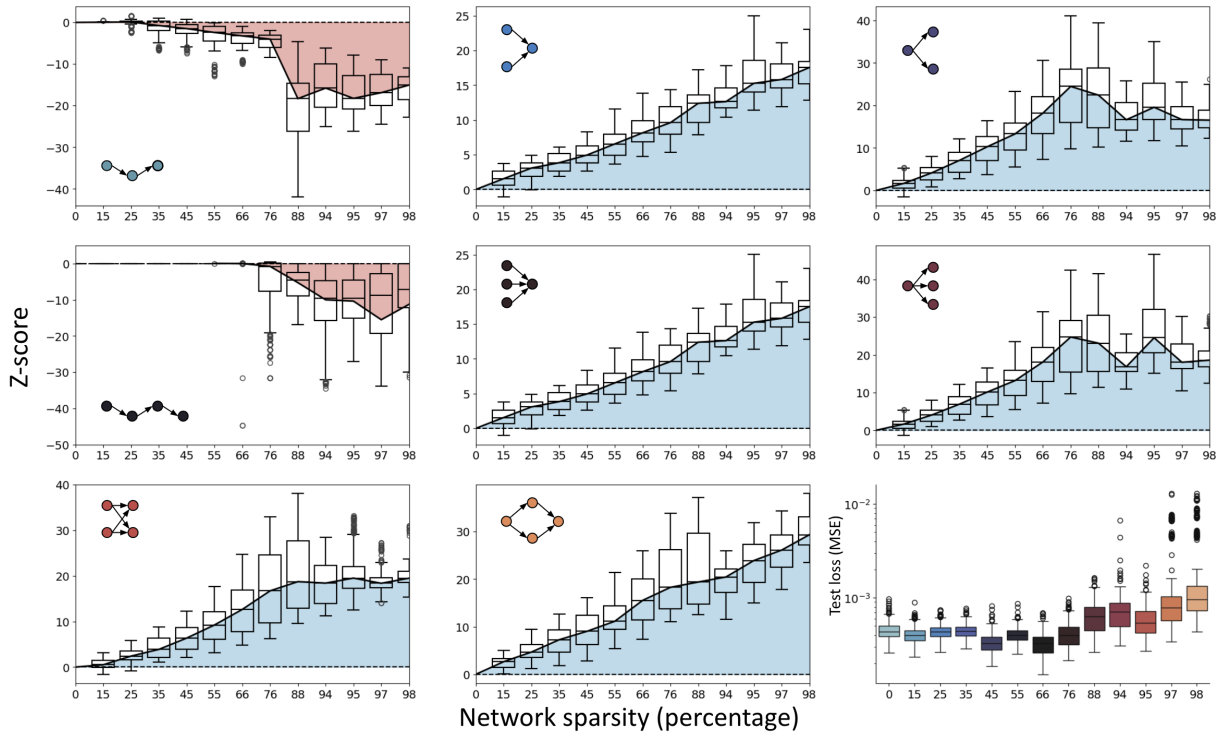


Figure 5.3: Z-score distributions across sparsity levels. Each panel shows how the z-score of the pictured motif changes throughout the pruning process. The bottom right panel shows the test MSE across all 350 networks at increasing levels of sparsity.

variance bars at 15% sparsity in the panels for 2^{nd}-order converging and diverging, 3^{rd}-order converging and diverging, and bi-fan motifs). Otherwise, network motifs are pushed to higher levels of over- or under-representation from very early in the pruning process.

The significance of some motifs (2^{nd}-order converging, 3^{rd}-order converging, and bi-parallel) appear to increase monotonically and show no signs of leveling-off even at very high levels of sparsity. The other motifs tend to reach convergence, leveling off at some maximum or minimum level of significance. Some of these motifs (see 2^{nd}-order diverging) show signs of decreasing after reaching maximum significance at 76% sparsity.

5.4 Discussion

In this study, we intended to characterize the network motif distribution of sparse, deep neural networks trained to simulate insect flight control. Using the flight dynamics model in [15] and the pruning paradigm developed in [104], we counted the 2^{nd}- and 3^{rd}-order network motifs of 350 networks pruned to increasing levels of sparsity. The DNNs were trained to predict the control variables for controlled hovering from the initial and final state-space variables as inputs. All 350 networks were trained and pruned in parallel where the pruning process followed the sequential pattern described in [104]. We developed motif counting algorithms that take advantage of the feed-forward structure of the DNNs studied in this work. The total number of motifs counted for each network was used to calculate the motif significance (z-score) for each network and results were compared across all 350 networks.

The significance of a motif is determined by comparing its number of occurrences in the network to its number of occurrences in a equivalently sized, but randomly connected network. We constrained the random networks to a feed-forward architecture with the same number of nodes and connections per layer as the network being evaluated. As a network is sparsified via pruning, motifs become more positively or negatively significant (depending on the motif). Figure 5.2 shows the distributions of network motif z-scores at 98% sparsity. At high levels of sparsity, network motifs are either highly over- or under-represented. Motifs below the midline of Figure 5.2 are highly under-represented. In other words, these motifs (2^{nd}-order chain and 3^{rd}-order chain, specifically) occur much less in the trained and pruned networks than in an equivalent random counterpart. The opposite is true for the the motifs above the midline. These motifs occur in greater numbers in the trained and pruned networks than in the random networks. The magnitude of the z-score for all motifs in the 98% pruned networks is very large. Magnitude-based weight pruning (the method employed here) results in networks with highly significant sub-graph structures.

Motif significance increases (or decreases) with network sparsity. Figure 5.3 shows how the motif significance relates to the network sparsity. At low levels of sparsity (in other

words, nearly fully-connected layers) motifs have very low z-scores. The low z-scores are due to the fact that there are few unique ways for the connections of a nearly fully-connected network to be organized. As networks are pruned and become more sparse, the over- or under-representation of a given motif becomes more pronounced. Some motifs become more under-represented as the network is made more sparse (i.e., 2^{nd}-order chain and 3^{rd}-order chain motifs). All of the other motifs become over-represented as more connections are pruned from the network. Furthermore, the significance levels for the 2^{nd}-order converging, 3^{rd}-order converging, and the bi-parallel motifs all monotonically increase. In contrast, at higher levels of sparsity, all other motifs display a leveling-off in their significance levels.

There are clear patterns when looking at different orders of the same motif type (e.g. the 2^{nd}-order and 3^{rd}-order chain motifs) Although the 2^{nd}-order and 3^{rd}-order chain motifs motifs becomes more significantly negative over the course of pruning, a leveling-off in the z-score occurs for high levels of sparsity. This is most clearly seen in the uppermost left panel of Figure 5.3 where at around 88% sparsity, the average significance of the 2^{nd}-order chain motif becomes constant despite increasing sparsity. From this result, it can be concluded that as a feed-forward network is pruned, the chain motifs become more rare, but a minimum number of chain sub-graphs are necessary for the network to perform its function.

A similar leveling-off pattern can be seen in the motif significance distributions of the 2^{nd}-order and 3^{rd}-order diverging motifs and the bi-fan motif (lower-most left panel of Figure 5.3). The bi-fan and bi-parallel motifs are made up of two connected 2^{nd}-order converging and diverging motifs. The significance of both the 2^{nd}-order diverging and bi-fan motifs levels-off at high levels of sparsity around 20σ. Interestingly, the 2^{nd}-order converging and bi-parallel motifs do not display the same leveling-off at the tested sparsity levels. It is possible that with further fine-grain pruning the motif z-scores for the 2^{nd}-order and 3^{rd}-order converging motifs and the bi-parallel motif would level-off.

All motif pairs tested in this study (i.e., converging, diverging, and chain) have similar motif significance distributions across varying levels of sparsity. The close similarity between the significance distributions of the 2^{nd}-order and 3^{rd}-order converging motifs, for example,

are natural due to the similarity between the sub-graphs themselves. While the distributions of the z-scores look very similar, differences in the distributions of the components of the z-scores are more pronounced. These distributions can be seen in the Appendix B at the end of this thesis.

In the motif significance distributions in Figure 5.3, the raw count of each sub-graph is very large, despite high-levels of global network sparsity (see Figure B.4 in Appendix B. This results in low resolution differences in the visualization of the motif significance distributions, specifically in the panels for the 2^{nd}-order and 3^{rd}-order converging motifs. For example, in a given network pruned to 98% sparsity, the raw count of 2^{nd}-order converging sub-graphs is $64,249$ and the raw count for 3^{rd}-order converging sub-graphs is $64,182$. Differences in the sub-graph count that are on the order of 10s or 100s are not visible in the visualizations provided here. However, further experimentation and a different pruning paradigm may help distinguish these motifs. This experiment was done with global neural network pruning. For a given network, pruning was halted when network performance dropped significantly which typically occurred when outputs were pruned from the network. More pronounced differences in the motif significance distributions may arise with further pruning of only the hidden layers (i.e., with the input and output layers frozen at 98% sparsity.

The DNNs analyzed here were randomly initialized prior to training, but were architecturally all the same. Additionally, every network underwent the same pruning paradigm. We followed the sequential, magnitude-based pruning schedule developed in [104]. While magnitude-based pruning is an effective DNN sparsification technique, it requires the assumption that the parameters that are critical to predictive performance are the ones that impart high activation to the nodes of the network. Therefore, high-activation nodes and connections are used to calculate motif significance. An interesting experiment would be to compare the motif makeup of a fully-connected network to its sparse counterpart, where the connections and nodes used in the sub-graph counting algorithms are ones with high activation. It is possible that a fully-connected network is equivalent to its sparse counterpart topologically, but with the advantage of parameter redundancy, which may increase

its robustness to noisy data. Another interesting followup experiment would be to test if different pruning techniques affect the performance and motif makeup of a sparse network. Random pruning and retraining can also result in a performant DNN, however the resultant network usually performs worse than underwent structured pruning [11]. Comparing the motif makeup of a randomly pruned DNN to a magnitude-based pruned DNN may provide insight to the importance of network sub-structures to network performance.

Finally, this work was done with feed-forward DNNs, which are the simplest example of DNNs after perceptrons. We limited the scope of this work to feed-forward DNNs because they are relevant, but tractable. Feed-forward DNNs are able to model many complex regression problems found in physics, biology, and engineering. Additionally, they are computationally easy to understand and their structure allowed us to quickly calculate a subset of the possible low-order network motifs. There are more motifs possible in feed-forward DNNs than the ones that are presented here. We constrained our study to the motifs that are relevant in the literature and natural networks [4, 105]. Furthermore, [49] showed that low-order motifs are the most important in relation to global network behavior. More complex motifs are possible in more complex network architectures. For example, networks with feed-back (such as recurrent neural networks) contain feed-back motifs. The sub-graphs we highlight connect subsequent layers, but in residual networks, information can travel further downstream without passing through every feed-forward layer. This work demonstrates our efforts to apply a method from complex network theory to the state-of-the-art technology.

5.5 Conclusion

The work done here is a first step in using the tools of network theory, namely network motif theory, to characterize the connectivity landscape of a DNN trained to model a biomechanical task. The work in [104] showed that DNNs trained to model the same system will converge to similar levels of sparsity when sparsified via pruning. This work is a direct successor to that study and further demonstrates the similarity between the networks studied. The gradient descent and backpropagation algorithms do not guarantee the randomly

initialized networks to converge to the same solution even when trained on the same data. However, we have shown here that training a DNN results in a network structure that has a characteristic network motif landscape.

Chapter 6

CONCLUSION AND OUTLOOK

Deep learning is and will continue to be a popular tool in wide ranging applications across many fields and sectors. Thanks to modern computing capabilities and novel deep learning architectures, DNNs are able to model a variety of complex systems. Due to their large number of learnable parameters, DNNs can model high-dimensional, highly non-linear relationships between inputs, parameters, and outputs. However, the number of parameters in a DNN makes it impossible to know how each parameter contributes mechanistically to its overall function.

Deep neural networks are not mechanistic models. However, the problem should drive the modeling approach. Specifically, DNNs are useful tools when it is unimportant to understand the underlying mechanisms of the system being studied. The size of DNNs makes understanding the relationships between parameters difficult. For example, there are two different approaches to modeling the bio-mechanical control system discussed in Chapter 4. The first is to construct a model that can predict many different trajectories of the insect given a large set of input state-space conditions and control variables. The second approach is to construct an accurate mechanistic model that describes the underlying mechanics of the control problem. Deep neural networks have great predictive power, but are often difficult to interpret. In contrast, mechanistic models are interpretable, but can often have less predictive power and are less amenable to changes in their inputs and parameters.

In this thesis, sparse deep neural networks were applied as models to a variety of systems including computer vision tasks, sensor placement, and bio-mechanical control. Chapters 1 and 2 provides motivation and background to the field of deep neural network sparsification. Chapter 3 describes the method of neural network pruning, which is used in the remainder of

the thesis to sparsify deep learning models. Additionally, Chapter 3 explores DNNs as sparse models in a sparse sensor placement problem. In Chapter 4, DNNs are trained to model the inverse problem of insect flight control. The mechanistic model used to generate the training data for the DNN provides a direct comparison to the deep learning model. An attempt to understand the modeling mechanisms of a DNN using tools from complex network theory is explored in Chapter 5.

Chapter 3 describes the method of neural network pruning utilized in this body of work and shows its application to problems in computer vision and sparse sensor placement [99]. Section 3.2.2 details the results of using a shallow decoder network to model optimal sensor location in a variety of systems. As an example, global sea-surface temperature is reconstructed from a limited number of sensors [1]. This high-dimensional, non-linear problem is an exemplar of a problem that does not necessitate understanding the underlying system dynamics. This problem also serves as example where the mechanism of neural network pruning is interpretable. In this experiment, neural network pruning is used on the input layer of the shallow decoder network (SDN) to identify salient input nodes and effectively discover the optimal sensor placement. Surprisingly, random sensor placement outperforms pruned sensor placement at all numbers of sensors considered. Nodal pruning was performed by removing entire groups of parameters associated with a specific input node. A group of parameters was removed based on the magnitude of its L^2 norm. In other words, nodes associated with collectively small weights were pruned. Magnitude-based pruning proved to be a poor choice for identifying important sensors, because SDNs require a compression of the input space. Input nodes with very low temperatures gained large connections to the latent space through training so that the network can reconstruct colder regions, as well as warmer ones. This is a case where the pruning methods based on parameter importance (described in detail in Section 2.1.2 may result in more ideal sensor locations.

In Chapter 4, a mechanistic model for bio-mechanical control is compared to a deep neural network trained to model the same system. We show that sparse deep neural networks are capable of learning the controls for a biological motor task. It is also shown, via Monte

Carlo simulations, that there exists at least some aspects of DNN connectivity structure (i.e., the number of remaining connections after pruning) that are stereotypical across networks. Synaptic pruning has been shown to play a major role in the refinement of neural connections, leading to more effective motor task control. In this work, we applied the method of DNN pruning to the inverse problem of insect flight control. We used the inertial dynamics model developed in [14] to simulate examples of *M. sexta* hovering flight. This data was then used to train a deep neural network to learn the controls for moving the simulated insect between two points in state-space. Neural network pruning was used to find the optimally sparse network for controlling the bio-mechanical system. Sparse neural networks, as well as fully-connected networks, can model a bio-mechanical system defined by a mechanistic model.

To push the comparison of mechanistic and deep learning models, areas of future work should include investigating the sparse DNNs response to noise and changes in the biological parameters. Natural bio-mechanical control systems are adapted to function adequately in the presence of noise. While pruning improves the performance of neural networks up to a certain level of sparsity, the effects of noise on the deep learning models are yet to be explored. Additionally, the physical parameters (such as size and shape) of a real insect can change rapidly (e.g. change of mass after feeding). Whether sparsity improves DNN robustness (over the mechanistic model) to changing physical parameters could also be a future extension of this work.

In Chapter 5, the connectivity structure of the sparse DNNs found in [104] was characterized using tools from complex network theory, namely network motif theory. The work in [104] showed that DNNs trained to model the same system will converge to similar levels of sparsity when sparsified via pruning. While the gradient descent and backpropagation algorithms do not guarantee that randomly initialized networks will converge to the same solution (even when trained on the same data), we find that deep learning training paradigms do lead to models with stereotypical connectivity patterns.

Sparse deep neural networks have been shown to be capable of modeling previously

intractable problems in many domains. While there may be structure in the connectivity of deep neural networks, they are far from mechanistic models and should not be treated as such. They are, however, extremely useful tools in problems where the underlying dynamics are unknown or are not necessarily important for the problem at hand. Deep learning models will continue to be popular tools for modeling across the science and engineering disciplines and further research into their connectivity and learning dynamics is necessary for trustworthy mechanistic modeling.

BIBLIOGRAPHY

[1] (2017). *NOAA Optimum Interpolation (OI) Sea Surface Temperature (SST) V2*. Available:`https://www.psl.noaa.gov/data/gridded/data.noaa.oisst.v2.html`.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[4] Uri Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8(6):450–461, 2007.

[5] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[6] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[7] Peter Bartlett. For valid generalization the size of the weights is more important than the size of the network. *Advances in neural information processing systems*, 9, 1996.

[8] Eric Baum and David Haussler. What size net gives valid generalization? *Advances in neural information processing systems*, 1, 1988.

[9] Tsevi Beatus and Itai Cohen. Wing-pitch modulation in maneuvering fruit flies is explained by an interplay between aerodynamics and a torsional spring. *Physical Review E*, 92(022712):1–13, 2015.

[10] Katharina Bieker, Sebastian Peitz, Steven L. Brunton, J. Nathan Kutz, and Michael Dellnitz. Deep model predictive flow control with limited sensor data and online learning. *Theoretical and Computational Fluid Dynamics*, 34(4):577–591, Mar 2020.

[11] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[14] Jr Bustamante, Jorge, Mahad Ahmed, Tanvi Deora, Brian Fabien, and Thomas L Daniel. Abdominal movements in insect flight reshape the role of non-aerodynamic structures for flight maneuverability I: Model predictive control for flower tracking. *Integrative Organismal Biology*, 09 2022. obac039.

[15] Jr Bustamante, Jorge, Mahad Ahmed, Tanvi Deora, Brian Fabien, and Thomas L Daniel. Abdominal movements in insect flight reshape the role of non-aerodynamic structures for flight maneuverability I: Model predictive control for flower tracking. *Integrative Organismal Biology*, 09 2022. obac039.

[16] Yves Chauvin. A back-propagation algorithm with optimal use of hidden units. *Advances in neural information processing systems*, 1, 1988.

[17] Gal Chechik, Isaac Meilijson, and Eytan Ruppin. Synaptic Pruning in Development: A Computational Account. *Neural Computation*, 10(7):1759–1777, 10 1998.

[18] François Chollet et al. Keras. `https://keras.io`, 2015.

[19] Fergus I.M. Craik and Ellen Bialystok. Cognition through the lifespan: mechanisms of change. *Trends in Cognitive Sciences*, 10(3):131–138, 2006.

[20] Charles B Delahunt, Jeffrey A Riffell, and J Nathan Kutz. Biological mechanisms for learning: a computational model of olfactory learning in the manduca sexta moth, with applications to neural nets. *Frontiers in computational neuroscience*, 12:102, 2018.

[21] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[22] John Denker, Daniel Schwartz, Ben Wittner, Sara Solla, Richard Howard, Lawrence Jackel, and John Hopfield. Large automatic learning, rule extraction, and generalization. *Complex systems*, 1(5):877–922, 1987.

[23] Michael H. Dickinson. Unsteady mechanisms of force generation in aquatic and aerial locomotion. *American Zoologist*, 36(6):537–554, 1996.

[24] Jonathan P. Dyhr, Kristi A. Morgansen, Thomas L. Daniel, and Noah J. Cowan. Flexible strategies for flight control: an active role for the abdomen. *The Journal of Experimental Biology*, 216:1523–1536, 2013.

[25] Sven OE Ebbesson. Evolution and ontogeny of neural circuits. *Behavioral and Brain Sciences*, 7(3):321–331, 1984.

[26] N. Benjamin Erichson, Lionel Mathelin, Zhewei Yao, Steven L. Brunton, Michael W. Mahoney, and J. Nathan Kutz. Shallow neural networks for fluid flow reconstruction with limited sensors. *Proc. R. Soc. A*, 476, May 2020.

[27] Travis E Faust, Georgia Gunner, and Dorothy P Schafer. Mechanisms governing activity-dependent synaptic pruning in the developing mammalian cns. *Nature Reviews Neuroscience*, 22(11):657–673, 2021.

[28] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. Perforatedcnns: Acceleration through elimination of redundant convolutions. *Advances in neural information processing systems*, 29, 2016.

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

[30] David W. Franklin and Daniel M. Wolpert. Computational mechanisms of sensorimotor control. *Neuron*, 72(3):425–442, 2011.

[31] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[32] Aidan N Gomez, Ivan Zhang, Siddhartha Rao Kamalakara, Divyam Madaan, Kevin Swersky, Yarin Gal, and Geoffrey E Hinton. Learning sparse networks using targeted dropout. *arXiv preprint arXiv:1905.13678*, 2019.

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[34] Alexandros Goulas, Fabrizio Damicelli, and Claus C Hilgetag. Bio-instantiated recurrent neural networks: Integrating neurobiology-based network topology in artificial networks. *Neural Networks*, 142:608–618, 2021.

[35] Claire Guerrier, John A. Hayes, Gilles Fortin, and David Holcman. Robust network oscillations during mammalian respiratory rhythm generation driven by synaptic dynamics. *Proceedings of the National Academy of Sciences*, 112(31):9728–9733, 2015.

[36] 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.

[37] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[38] Stephen Hanson and Lorien Pratt. Comparing biases for minimal network construction with back-propagation. *Advances in neural information processing systems*, 1, 1988.

[39] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[40] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pages 293–299. IEEE, 1993.

[41] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[42] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018.

[43] Tyson L. Hedrick and Thomas Lewis Daniel. Flight control in the hawkmoth *Manduca sexta*: the inverse problem of hovering. *The Journal of Experimental Biology*, 209:3114–3130, 2006.

[44] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent, 2012.

[45] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *The Journal of Machine Learning Research*, 22(1):10882–11005, 2021.

[46] Kyle S Honegger, Robert AA Campbell, and Glenn C Turner. Cellular-resolution population imaging reveals robust sparse coding in the drosophila mushroom body. *Journal of neuroscience*, 31(33):11772–11785, 2011.

[47] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[48] Yu Hu, Steven L. Brunton, Nicholas Cain, Stefan Mihalas, J. Nathan Kutz, and Eric Shea-Brown. Feedback through graph motifs relates structure and function in complex networks. *Physical Review E*, 98(6), Dec 2018.

[49] Yu Hu, Steven L Brunton, Nicholas Cain, Stefan Mihalas, J Nathan Kutz, and Eric Shea-Brown. Feedback through graph motifs relates structure and function in complex networks. *Physical Review E*, 98(6):062312, 2018.

[50] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[51] Steven A Janowsky. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600, 1989.

[52] Ehud D Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE transactions on neural networks*, 1(2):239–242, 1990.

[53] Paul S Katz. Evolution and development of neural circuits in invertebrates. *Current Opinion in Neurobiology*, 17(1):59–64, 2007. Development.

[54] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[55] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[56] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[57] James M Kunert, Joshua L Proctor, Steven L Brunton, and J Nathan Kutz. Spatiotemporal feedback and network structure drive and encode caenorhabditis elegans locomotion. *PLoS computational biology*, 13(1):e1005303, 2017.

[58] J Nathan Kutz. Neurosensory network functionality and data-driven control. *Current Opinion in Systems Biology*, 13:31–36, 2019.

[59] Andrey Kuzmin, Markus Nagel, Saurabh Pitre, Sandeep Pendyam, Tijmen Blankevoort, and Max Welling. Taxonomy and evaluation of structured compression of convolutional neural networks, 2019.

[60] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

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

[62] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[63] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[64] Esther Levin, Naftali Tishby, and Sara A Solla. A statistical approach to learning and generalization in layered neural networks. *Proceedings of the IEEE*, 78(10):1568–1574, 1990.

[65] Richard B Levine and James W Truman. Metamorphosis of the insect nervous system: changes in morphology and synaptic interactions of identified neurones. *Nature*, 299(5880):250–252, 1982.

[66] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E. Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers, 2020.

[67] 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*, pages 2736–2744, 2017.

[68] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning, 2017.

[69] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization, 2018.

[70] Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. Deep learning for universal linear embeddings of nonlinear dynamics. *Nature Communications*, 9(1), Nov 2018.

[71] Zelda Mariet and Suvrit Sra. Diversity networks: Neural network compression using determinantal point processes. *arXiv preprint arXiv:1511.05077*, 2015.

[72] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[73] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1), Jun 2018.

[74] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11264–11272, 2019.

[75] Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. *Advances in neural information processing systems*, 1, 1988.

[76] Gregory Naitzat, Andrey Zhitnikov, and Lek-Heng Lim. Topology of deep neural networks. *The Journal of Machine Learning Research*, 21(1):7503–7542, 2020.

[77] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[78] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[79] OpenAI. Gpt-4 technical report, 2023.

[80] Romualdo Pastor-Satorras, Claudio Castellano, Piet Van Mieghem, and Alessandro Vespignani. Epidemic processes in complex networks. *Reviews of modern physics*, 87(3):925, 2015.

[81] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[82] Brandon Pratt, Tanvi Deora, Thomas Mohren, and Thomas Daniel. Neural evidence supports a dual sensory-motor role for insect wings. *Proceedings of the Royal Society B: Biological Sciences*, 284, 2017.

[83] Siying Qian, Chenran Ning, and Yuepeng Hu. Mobilenetv3 for image classification. In *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 490–497, 2021.

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

[85] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[86] Sanjay P. Sane. The aerodynamics of insect flight. *Journal of Experimental Biology*, 206(23):4191–4208, 2003.

[87] Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, Jun 2017.

[88] Carla J Shatz and Michael P Stryker. Ocular dominance in layer iv of the cat's visual cortex and the effects of monocular deprivation. *The Journal of physiology*, 281(1):267–283, 1978.

[89] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

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

[91] Lewi Stone, Daniel Simberloff, and Yael Artzy-Randrup. Network motifs and their origins. *PLoS computational biology*, 15(4):e1006749, 2019.

[92] Steven H Strogatz. Exploring complex networks. *nature*, 410(6825):268–276, 2001.

[93] Taiji Suzuki, Hiroshi Abe, Tomoya Murata, Shingo Horiuchi, Kotaro Ito, Tokuma Wachi, So Hirai, Masatoshi Yukishima, and Tomoaki Nishimura. Spectral pruning: Compressing deep neural networks via spectral analysis and its generalization error. *arXiv preprint arXiv:1808.08558*, 2018.

[94] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[95] Vera Van Noort, Berend Snel, and Martijn A Huynen. The yeast coexpression network has a small-world, scale-free architecture and can be explained by a simple model. *EMBO reports*, 5(3):280–284, 2004.

[96] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.

[97] Fernando Vonhoff and Haig Keshishian. Activity-dependent synaptic refinement: New insights from drosophila. *Frontiers in Systems Neuroscience*, 11:23, 2017.

[98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[99] Jan Williams, Olivia Zahn, and J Nathan Kutz. Data-driven sensor placement with shallow decoder networks. *arXiv preprint arXiv:2202.05330*, 2022.

[100] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5687–5695, 2017.

[101] Seul-Ki Yeom, Philipp Seegerer, Sebastian Lapuschkin, Alexander Binder, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Pruning by explaining: A novel criterion for deep neural network pruning. *Pattern Recognition*, 115:107899, 2021.

[102] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9194–9203, 2018.

[103] O Zahn. Mothpruning. `https://github.com/oliviatessa/MothPruning`, 2021.

[104] Olivia Zahn, Jorge Bustamante Jr, Callin Switzer, Thomas L Daniel, and J Nathan Kutz. Pruning deep neural networks generates a sparse, bio-inspired nonlinear controller for insect flight. *PLoS Computational Biology*, 18(9):e1010512, 2022.

[105] Matteo Zambra, Amos Maritan, and Alberto Testolin. Emergence of network motifs in deep neural networks. *Entropy*, 22(2):204, 2020.

[106] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, pages 818–833. Springer, 2014.

[107] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

[108] H Zhu, H Liu, A Ataei, Y Munk, T Daniel, and IC Paschalidis. Learning from animals: How to navigate complex terrains. *PLoS computational biology*, 16(1):e1007452, 2020.

[109] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

Appendix A

SUPPLEMENTAL MATERIAL FOR CHAPTER 3

The following three figures are repeated experiments from Chapter 4 with increasingly smaller network architectures. For example, in A.6 we trained 400 networks, each with four hidden layers with 200, 200, 200, 8 nodes. A.2 shows the same results for networks of sizes 100, 100, 100, 8 and A.3 shows the results for networks of size 50, 50, 50, 8. These decreases in hidden layer widths correspond to a decrease in the total number of weights across the networks from $330,512$ (for the original networks in Fig 4.5) to $83,656$ (A.1), $21,856$ (A.2), and $5,956$ (A.3).

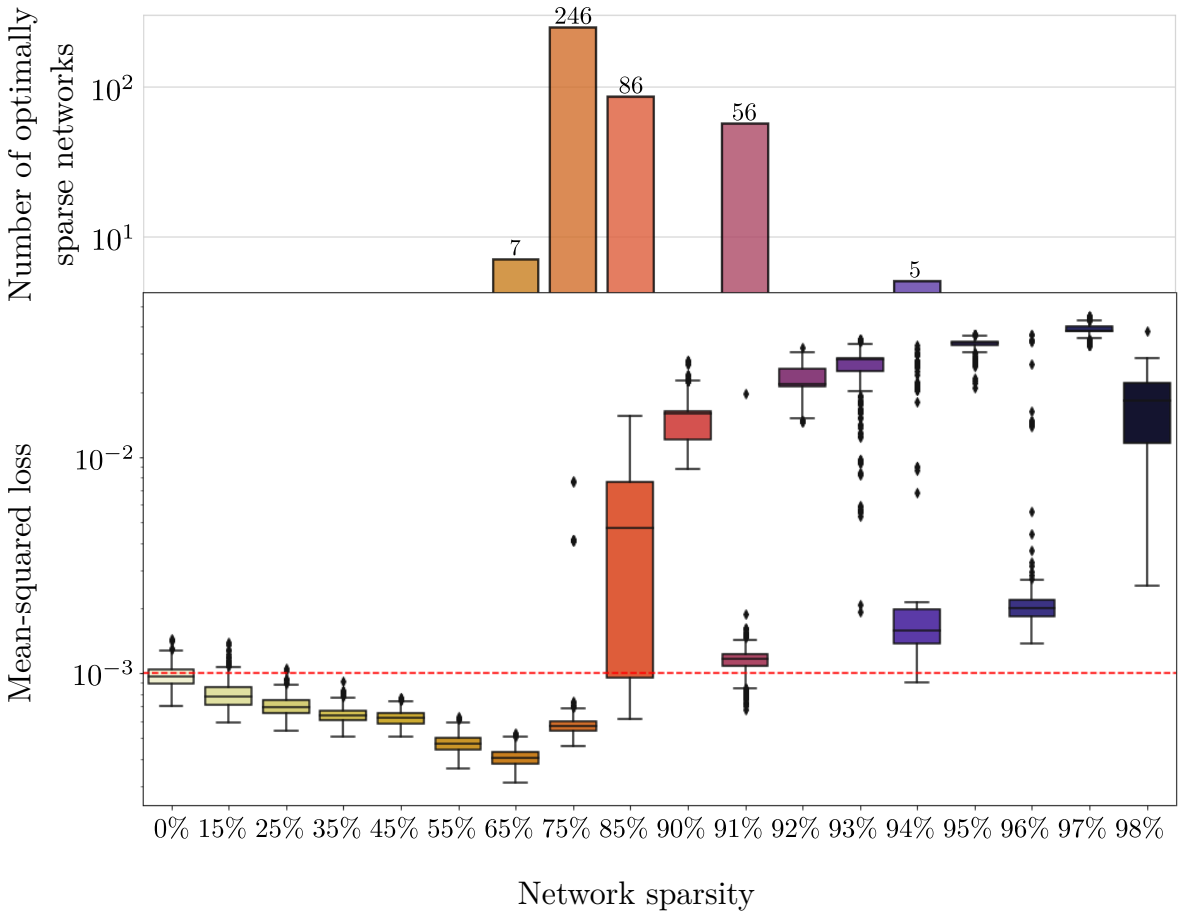


Figure A.1: Monte Carlo analysis of pruned networks. 400 networks, each with four hidden layers with 200, 200, 200, 8 nodes are sequentially pruned and loss of the pruned networks at each sparsity percentage is recorded in the box plot. The bar plot records the number of networks that make it to the corresponding sparsity percentage before exceeding the hypothetical threshold (10^{-3}).

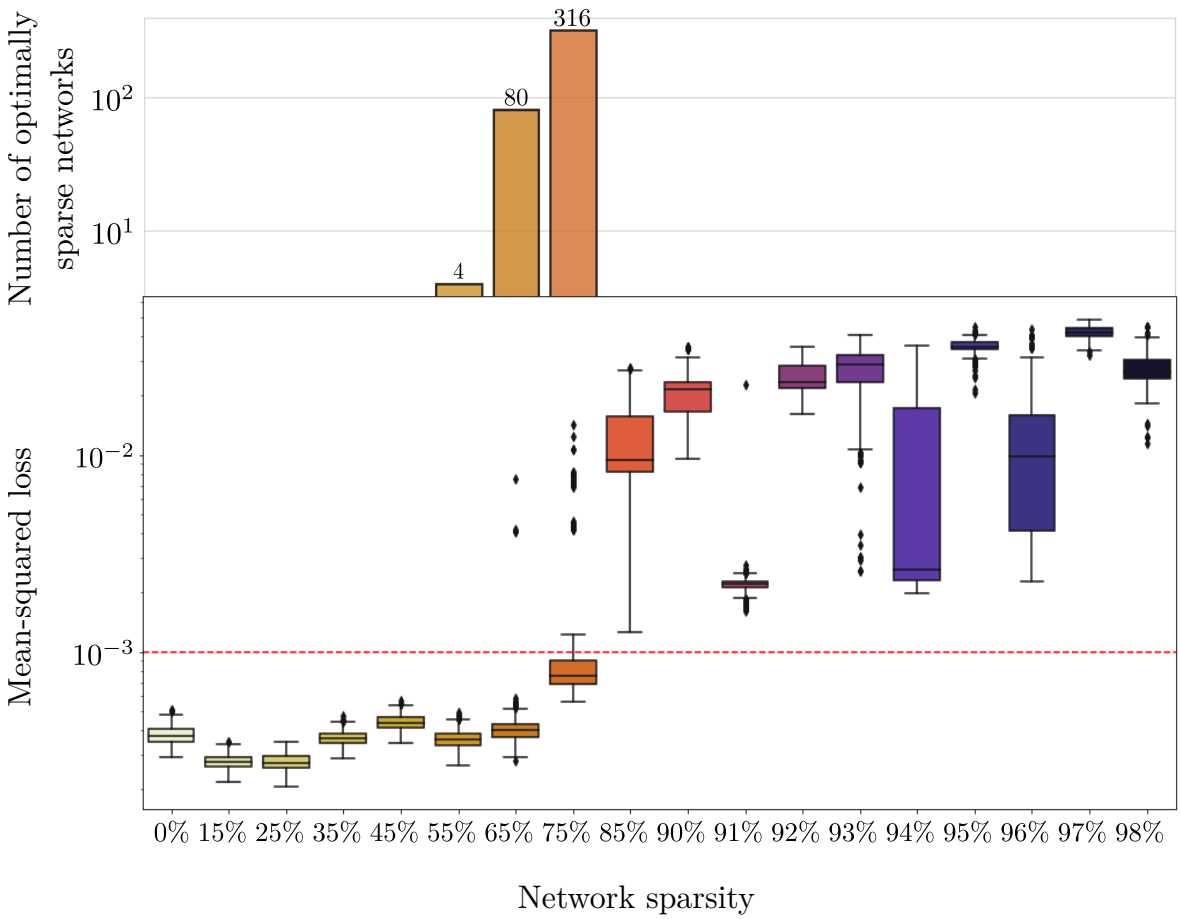


Figure A.2: Monte Carlo analysis of pruned networks. 400 networks, each with four hidden layers with 100, 100, 100, 8 nodes are sequentially pruned and loss of the pruned networks at each sparsity percentage is recorded in the box plot. The bar plot records the number of networks that make it to the corresponding sparsity percentage before exceeding the hypothetical threshold (10^{-3}).

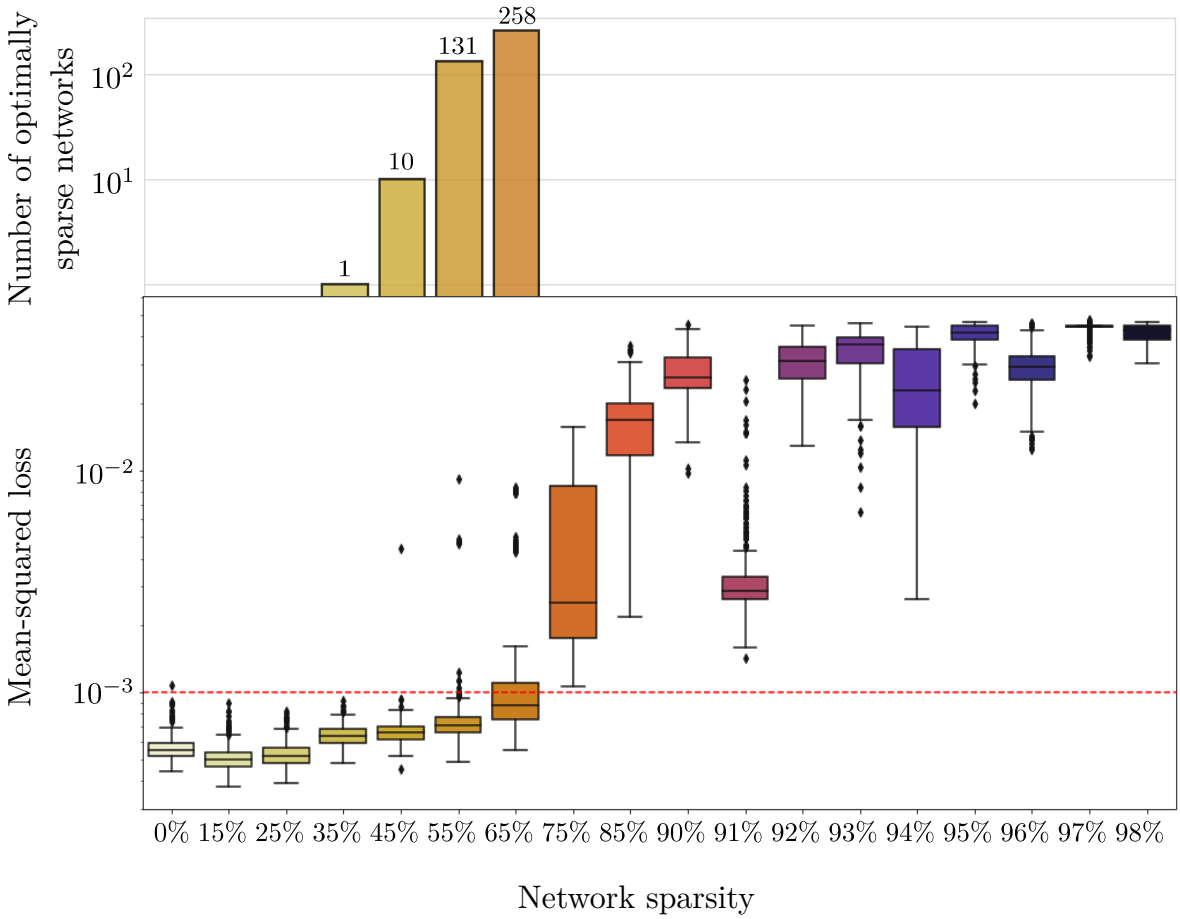


Figure A.3: Monte Carlo analysis of pruned networks. 400 networks, each with four hidden layers with 50, 50, 50, 8 nodes are sequentially pruned and loss of the pruned networks at each sparsity percentage is recorded in the box plot. The bar plot records the number of networks that make it to the corresponding sparsity percentage before exceeding the hypothetical threshold (10^{-3}).

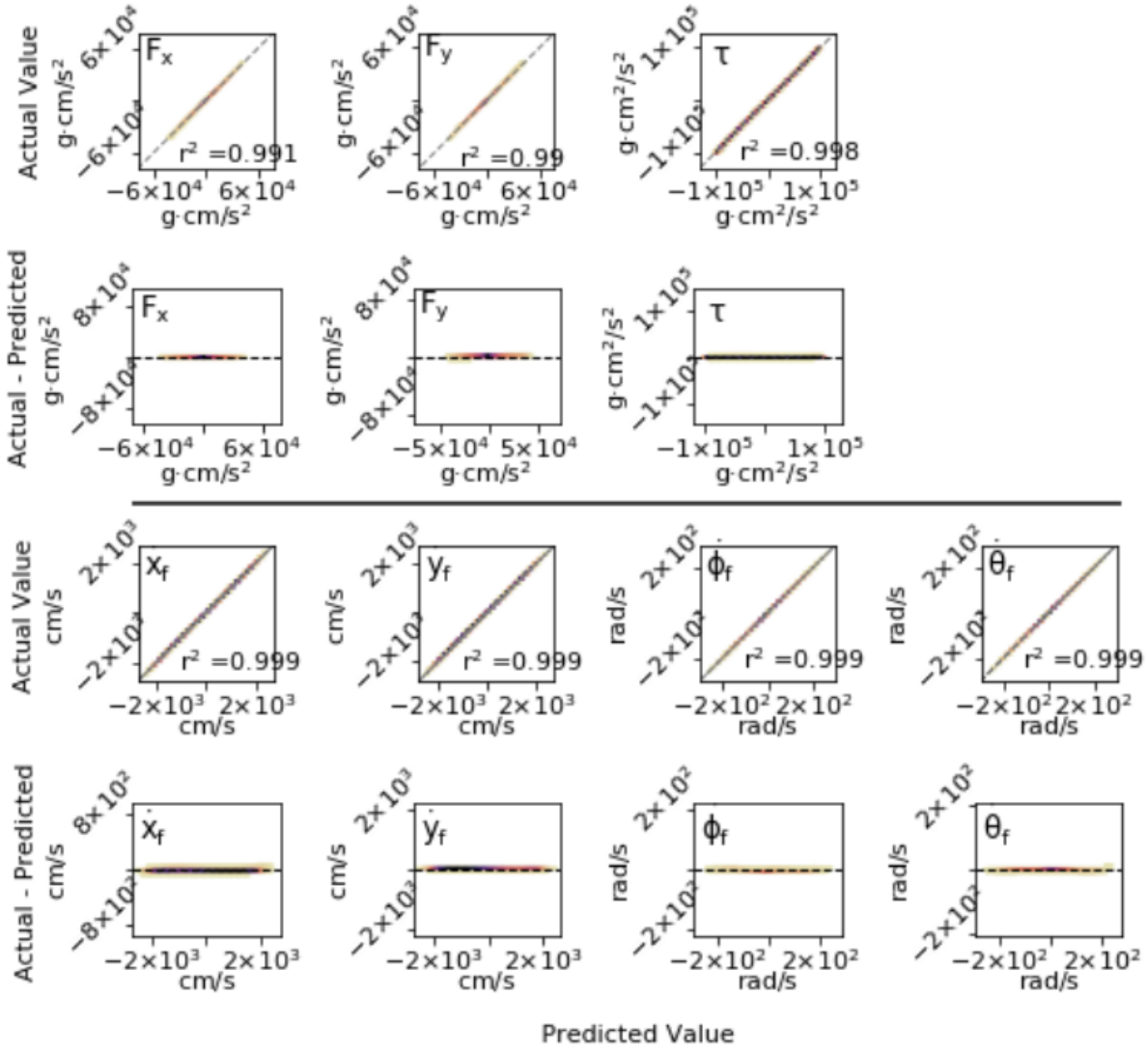


Figure A.4: Error evaluation (before pruning). Error evaluation of a fully-connected network before any pruning. The seven parameters shown are the outputs of the network, the three control variables and the final derivatives of the state space. The residual plots are also shown (denoted by Actual—Prediction).

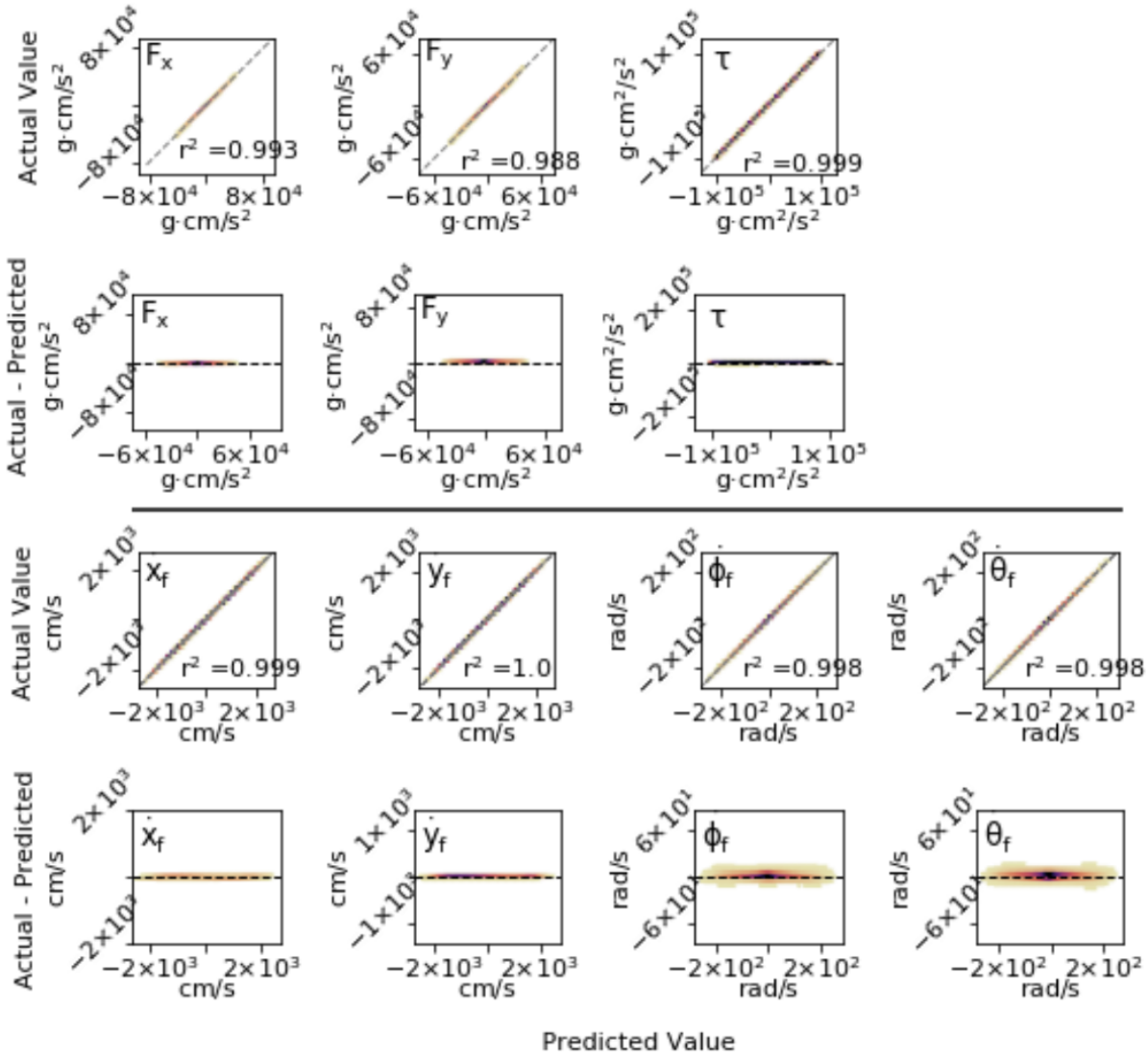


Figure A.5: Error evaluation (with pruning). See A.4 Note that axes for residual plots are scaled to include the max outliers.

Global parameters for the moth model. Below, we show a table of all the required parameters to recreate the simulated model of a moth.

Label	Value	Units	Description
$L1$	0.9	cm	Length from the thorax-abdomen joint to the center of mass of the head-thorax
$L2$	1.9	cm	Length from the thorax-abdomen joint to the center of mass of the abdomen
$L3$	0.75	cm	Length from the thorax-abdomen joint to the aerodynamic force vector
ρ_{head}	0.9	g/cm^2	The density of the insect head-thorax
ρ_{butt}	0.4	g/cm^2	The density of the insect abdomen
ρ_A	0.00118	g/cm^3	The density of air
μ_A	0.000186	g/cm·s	The dynamic viscosity of air at 27°C
a_{head}	0.9	cm	Length of 1/2 major axis of head-thorax ellipsoid
a_{butt}	1.9	cm	Length of 1/2 of major axis of abdomen ellipsoid
b_{head}	0.5	cm	Length of 1/2 of minor axis of head-thorax ellipsoid
b_{butt}	0.75	cm	Length of 1/2 of minor axis of abdomen ellipsoid
K	23000	$cm^2 \cdot g/(rad \cdot s^2)$	Torsional spring constant of the thorax-abdomen joint
c	14075.8	$cm^2 \cdot g/s$	Torsional damping constant of the thorax-abdomen joint
g	14075.8	cm/s^2	Acceleration due to gravity
β_R	0	rad	Resting configuration of the torsional spring = (initial abdomen angle) - (initial head-thorax angle) - π
t	0.02	s	Time step

Figure A.6: Global parameters for the moth model. A table of all the required parameters to recreate the simulated model of a moth.

Calculated variables for moth model. Below, we show a table of all the required calculated variables to recreate the simulated model of a moth.

Variable	Expression	Units	Description
m_1	$\rho_{head} \cdot \frac{4}{3}\pi \cdot (b_{head})^2 \cdot a_{head}$	g	Mass of head-thorax
m_2	$\rho_{butt} \cdot \frac{4}{3}\pi \cdot (b_{butt})^2 \cdot a_{butt}$	g	Mass of the abdomen
ec_{head}	a_{head}/b_{head}	N/A	Eccentricity of head-thorax
ec_{butt}	a_{butt}/b_{butt}	N/A	Eccentricity of abdomen
I_1	$\frac{1}{5}m_1 \cdot (b_{head})^2 \cdot (1 + (ec_{head})^2)$	g·cm^2	Moment of inertia of the head-thorax
I_2	$\frac{1}{5}m_2 \cdot (b_{butt})^2 \cdot (1 + (ec_{butt})^2)$	g·cm^2	Moment of inertia of the abdomen
S_{head}	$\pi \cdot (b_{head})^2$	cm^2	Surface area of the head-thorax. In this case, it is modeled as a sphere.
S_{butt}	$\pi \cdot (b_{butt})^2$	cm^2	**Surface area of the abdomen. In this case, it is modeled as a sphere.**
Re_{head}	$\rho_A \cdot \sqrt{\dot{x}^2 + \dot{y}^2} \cdot (2 \cdot b_{head})/\mu_A$	N/A	Reynolds number for the head-thorax
Re_{butt}	$\rho_A \cdot \sqrt{\dot{x}^2 + \dot{y}^2} \cdot (2 \cdot b_{butt})/\mu_A$	N/A	Reynolds number for the abdomen
Cd_{head}	$24/\lvert Re_{head}\rvert + 6/(1 + \sqrt{\lvert Re_{head}\rvert}) + 0.4$	N/A	Coefficient of drag for the head-thorax
Cd_{butt}	$24/\lvert Re_{butt}\rvert + 6/(1 + \sqrt{\lvert Re_{butt}\rvert}) + 0.4$	N/A	Coefficient of drag for the abdomen

Figure A.7: Calculated variables for moth model. A table of all the required calculated variables to recreate the simulated model of a moth.

Initial state space and controls for generating training data. The following variables are the required state space and control variables along with their ranges for random uniform sampling.

Var.	Distribution	Units	Variable type	Description
x_0	0	cm	State Space	Initial horizontal position
$\dot{x}_0$	$Unif(-1500, 1500)$	cm/s	State space	Initial horizontal velocity
y_0	0	cm	State space	Initial vertical position
$\dot{y}_0$	$Unif(-1500, 1500)$	cm/s	State space	Initial vertical velocity
θ_0	$Unif(0, 2\pi)$	rad	State space	Initial head-thorax angle
$\dot{\theta}_0$	$Unif(-25, 25)$	rad/s	State space	Initial head-thorax angular velocity
ϕ_0	$Unif(0, 2\pi)$	rad	State Space	Initial abdomen angle
$\dot{\phi}_0$	$Unif(-25, 25)$	rad/s	State space	Initial abdomen angular velocity
F	$Unif(0, 44300)$	g·cm/s^2	Control	Force magnitude
α	$Unif(0, 2\pi)$	rad	Control	Force angle
τ	$Unif(-100000, 100000)$	g·cm/s^2· cm	Control	Torque

Figure A.8: Initial state space and controls for generating training data. The following variables are the required state space and control variables along with their ranges for random uniform sampling.

Final state space variables The following variables are the output from the differential equation solver. These variables describe the final state space of the insect after 20ms.

Variable	Units	Description
x_f	cm	Initial horizontal position
$\dot{x}_f$	cm/s	Final horizontal velocity
y_f	cm	Initial vertical position
$\dot{y}_f$	cm/s	Initial vertical velocity
θ_f	*rad*	Final head-thorax angle
$\dot{\theta}_f$	*rad*/s	Final head-thorax angular velocity
ϕ_f	*rad*	Final abdomen angle
$\dot{\phi}_f$	*rad*/s	Final abdomen angular velocity

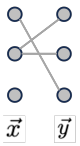
Figure A.9: Final state space variables. The following variables are the output from the differential equation solver. These variables describe the final state space of the insect after 20ms.

Appendix B

SUPPLEMENTAL MATERIAL FOR CHAPTER 4

B.0.1 Sub-graph counting algorithms

The following sub-graph counting algorithms take advantage of the network connectivity information contained in the sparse, binary masking layers. For example, take the sparse, two layer network depicted below. The number of connections that are still "live" in the pruned network can be ascertained entirely from the binary mask (matrix $\mathbf{M}$ in Figure B.1) by simply counting the number of ones in $\mathbf{M}$. Moreover, the number of more complex sub-graphs can be calculated using the binary masks of subsequent layers.



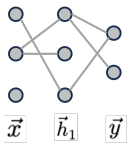
$$\vec{y} = \sigma((\mathbf{W} \odot \mathbf{M})\vec{x})$$

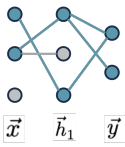
$$\underset{\mathbf{W}}{\begin{bmatrix} 0.03 & 0.02 & 0.50 \\ 0.55 & 0.60 & 0.01 \\ 0.01 & 0.02 & 0.01 \end{bmatrix}} \odot \underset{\mathbf{M}}{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}}$$

Figure B.1: Left: Example of two-layer sparse network with inputs, $\vec{x}$, and outputs, $\vec{y}$. Center: Forward pass computation where $\mathbf{W}$ represents the weight matrix $\mathbf{M}$ represents the mask matrix, and σ represents the nonlinear activation function (bias is excluded for simplicity). Right: Example of weight matrix $\mathbf{W}$ and mask matrix $\mathbf{M}$.

For example, to find the number of second-order chain sub-graphs in the example network depicted in Figure B.2, we need the masks between each of the layers. The highlighted version of the network (center panel in Figure B.2) shows that there are three second-order chain sub-graphs in the network. The network has one hidden layer, therefore, its connectivity is entirely described by two masks $\mathbf{M}_{\mathbf{x},\mathbf{h_1}}$ and $\mathbf{M}_{\mathbf{h_1},\mathbf{y}}$. Using these two matrices, the number of

second-order chain sub-graphs in the 3-layer network can be calculated by matrix multiplying $\mathbf{M_{x,h_1}}$ and $\mathbf{M_{h_1,y}}$ and taking the sum of the elements in the resultant product.





$$\mathbf{M_{x,h_1}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{M_{h_1,y}} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$$

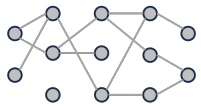
Figure B.2: Left: Example of sparse feed-forward network with inputs, $\vec{x}$, outputs, $\vec{y}$, and one hidden layer, $\vec{h}_1$. Middle: Same network with second-order chain sub-graphs highlighted. Right: Masks representing the connectivity of the network between the layers (e.g., $\mathbf{M_{x,h_1}}$ for the weights between layers $\vec{x}$ and $\vec{h}_1$).

$$\mathbf{M_{x,h_1}M_{h_1,y}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix} = P_{i,j}$$

$$\sum_{i=1}^{m}\sum_{j=1}^{n} P_{i,j} = 3$$

To count the second-order chain sub-graphs in a n-layer feed-forward network, one has to repeat the above calculation for each consecutive pair of layers in the network and sum all of the resultant products. Algorithm 6 describes the second-order chain sub-graph calculation in algorithmic form.

In the following algorithms, every pruned network is considered a list of sparse, binary matrices or masks. A simple example of this conversion is shown in Figure B.3. The variable *mask list* refers to the list of sparse matrices associated with one pruned network. Likewise, *mask* refers to a single sparse, binary matrix. Every algorithm describes the counting process for the total number of occurrences of the specified sub-graph in a single network.



$$\Longrightarrow \left(\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \right)$$

Figure B.3: Example of sparse, feed-forward network converted to list of masks, or *mask list* as referred to in algorithms.

Second-order sub-graph counting algorithms

Algorithm 6: 2nd-order chain sub-graph counting

```
total = 0;
for mask in mask list do
    product ← matrix multiply mask and subsequent mask;
    n ← sum elements of product;
    total += n
end
```

Algorithm 7: 2nd-order converging sub-graph counting

```
total = 0;
for mask in mask list do
    for column in mask do
        n ← count number of non-zero elements;
        if n ≥ 2 then
            total += (n choose 2);
    end
end
```

Algorithm 8: 2nd-order diverging sub-graph counting

```
total = 0;
for mask in mask list do
    for row in mask do
        n ← count number of non-zero elements;
        if n ≥ 2 then
            total += (n 2);
    end
end
```

Third-order sub-graph counting algorithms

The 3rd-order chain sub-graph counting algorithm requires the multiplication of three consecutive masks to obtain the number of 3rd-order chains, which involves 4 nodes. In the following algorithm m_i is used to denote mask i within the network.

Algorithm 9: 3rd-order chain sub-graph counting

```
total = 0;
for mask in mask list do
    product ← matrix multiply m_i, m_{i-1}, and m_{i-2};
    n ← sum elements of product;
    total += n
end
```

Algorithm 10: 3rd-order converging sub-graph counting

```
total = 0;
for mask in mask list do
    for column in mask do
        n ← count number of non-zero elements;
        if n ≥ 3 then
            total += (n 3);
    end
end
```

Algorithm 11: 3rd-order diverging sub-graph counting

```
total = 0;
for mask in mask list do
    for row in mask do
        n ← count number of non-zero elements;
        if n ≥ 3 then
            total += (n 3);
    end
end
```

Algorithm 12: Bi-fan sub-graph counting

```
total = 0;
for mask in mask list do
    for row in mask do
        list ← dot product of row with all other rows in mask;
        for element in list do
            if element ≥ 2 then
                total += (n 2);
        end
    end
end
```

Algorithm 13: Bi-parallel sub-graph counting

```
total = 0;
for mask in mask list do
    product ← matrix multiply m_i, m_{i+1};
    combination matrix ← (product 2);
    n ← sum elements of combination matrix;
    total += n
end
```

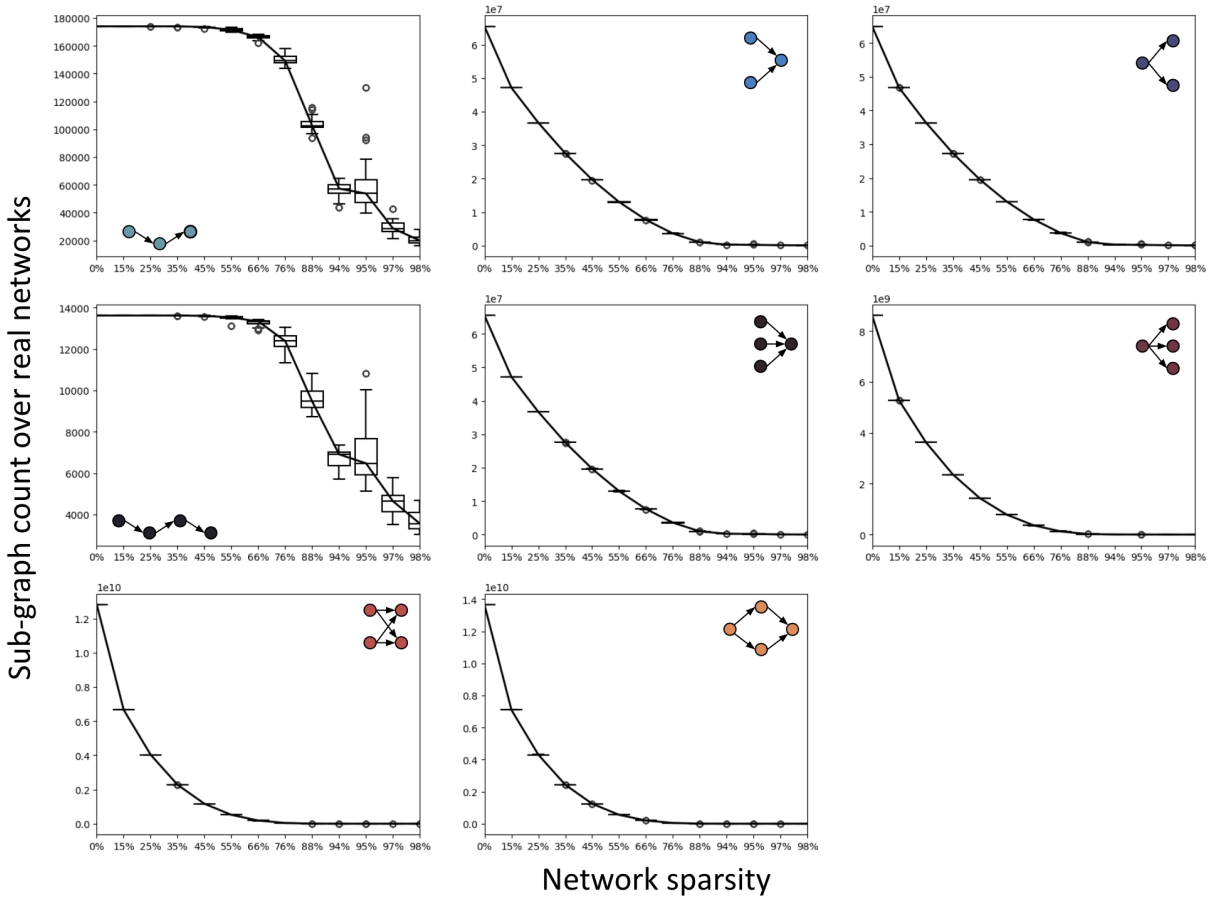


Figure B.4: Total sub-graph count in 350 sparse networks across sparsity levels. Each panel gives results for each sub-graph type. Used in the z-score calculation to produce results in Figure 5.3.

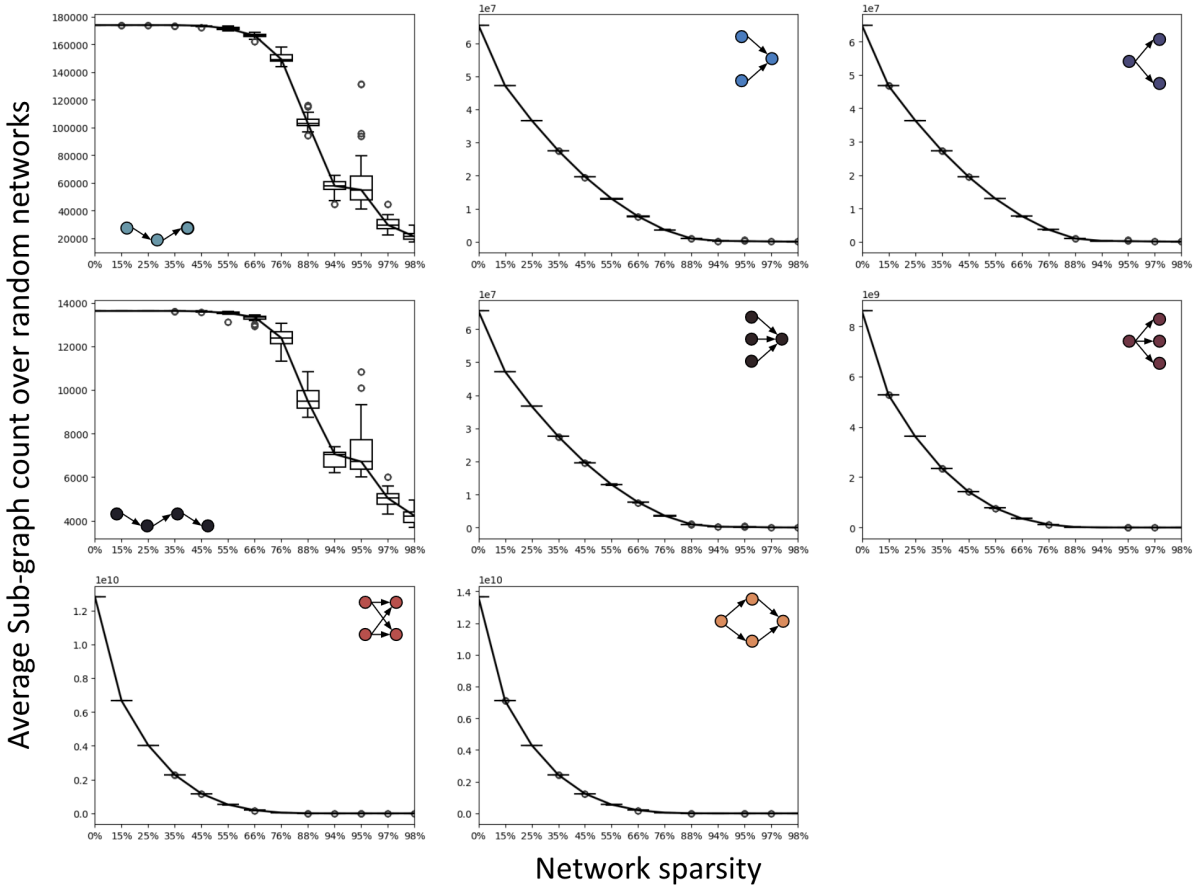


Figure B.5: Average sub-graph count across 1000 random networks generated for each of the 350 sparse networks across sparsity levels. Each panel gives results for each sub-graph type. Used in the z-score calculation to produce results in Figure 5.3.

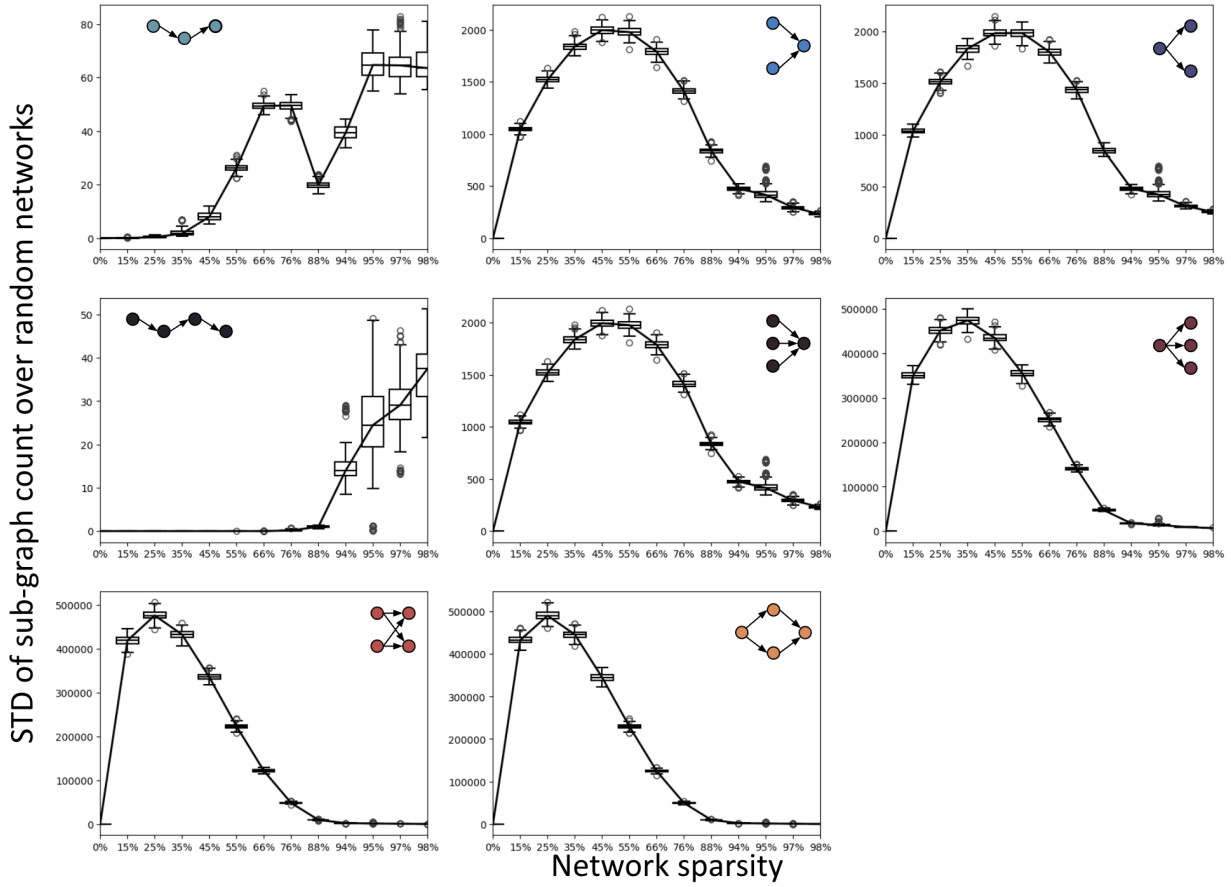


Figure B.6: Standard deviation of the sub-graph count across 1000 random networks generated for each of the 350 sparse networks across sparsity levels. Each panel gives results for each sub-graph type. Used in the z-score calculation to produce results in Figure 5.3.