AUTO-PARAMETRIZED KERNEL METHODS FOR BIOMOLECULAR MODELING

By

Timothy Andrew Szocinski

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Mathematics – Doctor of Philosophy

2021

ABSTRACT

AUTO-PARAMETRIZED KERNEL METHODS FOR BIOMOLECULAR MODELING

By

Timothy Andrew Szocinski

Being able to predict various physical quantities of biomolecules is of great importance to biologists, chemists, and pharmaceutical companies. By applying machine learning techniques to develop these predictive models, we find much success in our endeavors. Advanced mathematical techniques involving graph theory, algebraic topology, differential geometry, etc. have been very profitable in generating first-rate biomolecular representations that are used to train a variety of machine learning models. Some of these representations are dependent on a choice of kernel function along with parameters that determine its shape. These kernel-based methods of producing features require careful tuning of the kernel parameters, and the tuning cost increases exponentially as more kernels are involved. This limitation largely restricts us to the use of machine learning models with less hyper-parameters, such as random forest (RF) and gradient-boosting trees (GBT), thus precluding the use of neural networks for kernel-based representations.

To alleviate these concerns, we have developed the auto-parametrized weighted element-specific graph neural network (AweGNN), which uses kernel-based geometric graph features in which the kernel parameters are automatically updated throughout the training to reach an optimal combination of kernel parameters. The AweGNN models have shown to be particularly success in toxicity and solvation predictions, especially when a multi-task approach is taken. Although the AweGNN had introduced hundreds of parameters that were automatically tuned, the ability to include multiple kernel types simultaneously was hindered because of the computational expense. In response, the GPU-enhanced AweGNN was developed to tackle the issue.

Working with GPU architecture, the AweGNN's computation speed was greatly en-

hanced. To achieve a more comprehensive representation, we suggested a network consisting of fixed topological and spectral auxiliary features to bolster the original AweGNN success. The proposed network was tested on new hydration and solubility datasets, with excellent results. To extend the auto-parametrized kernel technique to include features of a different type, we introduced the theoretical foundation for building an auto-parametrized spectral layer, which uses kernel-based spectral features to represent biomolecular structures.

In this dissertation, we explore some underlying notions of mathematics useful in our models, review important topics in machine learning, discuss techniques and models used in molecular biology, detail the AweGNN architecture and results, and test and expand new concepts pertaining to these auto-parametrized kernel methods.

Copyright by
TIMOTHY ANDREW SZOCINSKI
2021

To my Lord Jesus Christ, for his infinite love and grace.

ACKNOWLEDGEMENTS

First of all, I would like to thank Dr. Guowei Wei for being my advisor. His guidance and patience throughout my doctoral journey is greatly appreciated. I would also like to thank Dr. Duc Nguyen for his help and advice in all of my projects. He has been an indispensable resource, frequently taking time out of his schedule for my benefit. I also thank my officemates, Menglun Wang and Rui Wang, for their willingness to help, their encouraging work ethic, and their openness to friendly conversation.

I would like to thank my parents, Hanna and Roman Szocinski, for their great love and support throughout my time here, along with the rest of my loving family. Their encouragement was unyielding and steadfast. May their love for me bear fruit in my future endeavors, and bring much joy in their lives as I strive for success in the time to come.

For those who have revealed to me the possibility of a better future, I thank Christen and Trevor Pollo. Where I thought much was lost, I could find hope in a true mission. Their leadership in the greatest cause of our time is truly inspiring and heartwarming. It will not go without notice.

To the friends of my life: Daniel Stewart, Matthew Bartels, Joseph Tuski, J.M. Hoyt, Joseph Matthias, and others; who have led me to a path of truth and honor, I thank you whole-heartedly. Though my eyes could not see, and my heart could not feel, these souls have brought me clarity of vision, peace of mind, and renewal of spirit.

To the men of the Order of the Cube, who have pledged themselves to a great enterprise, I thank you. Our destinies are now bound together as we journey on in life, and thus we must rejoice together in our accomplishments.

To Laura Durocher, the wonderful lady who has provided me with many joyful distractions and motivation for success, I give you my utmost love and affection.

Above all, I thank God, the Maker of all things. To Him I dedicate this dissertation,

for all my work should be for His glory.

TABLE OF CONTENTS

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ALGORITHMS xvii

CHAPTER 1 INTRODUCTION 1
- 1.1 Machine Learning and its Applications 1
- 1.2 Biomolecular Modeling 2
- 1.3 Abstract Mathematical Representations 3
- 1.4 Kernel-based Methods and Auto-parametrization 4
- 1.5 Motivation 5
- 1.6 Outline 6

CHAPTER 2 MATHEMATICAL TECHNIQUES 8
- 2.1 Introduction 8
- 2.2 Graph Theory 9
  - 2.2.1 Basic Definitions 10
  - 2.2.2 Adjacency and Laplacian Matrices 12
  - 2.2.3 Adjacency and Laplacian Matrices for Weighted Graphs 15
- 2.3 Algebraic Topology 16
  - 2.3.1 Simplicial Complexes 17
  - 2.3.2 Simplicial Homology 18
  - 2.3.3 Persistent Homology 21
  - 2.3.4 Čech and Vietoris-Rips Complexes 23

CHAPTER 3 MACHINE LEARNING 25
- 3.1 Introduction 25
- 3.2 Fundamental Concepts 26
  - 3.2.1 Data Representations 26
  - 3.2.2 Loss Functions 29
  - 3.2.3 Evaluation Metrics and Data 31
  - 3.2.4 Model Weights and Gradient Descent 35
  - 3.2.5 Regularization 37
- 3.3 Linear Regression 40
- 3.4 Decision Trees 42
- 3.5 Ensemble Methods 44
  - 3.5.1 Bagging 45
  - 3.5.2 Boosting 45
  - 3.5.3 Random Forest 46
  - 3.5.4 Gradient Boosting Trees 47
- 3.6 Artificial Neural Networks and Deep Learning 48

3.6.1 Activation Functions 50
3.6.2 Batch Normalization 52
3.6.3 Gradient Descent and Adaptive Learning Rates 54
3.6.4 Multi-task Networks and Transfer Learning 57

CHAPTER 4 BIOMOLECULAR MODELING AND KERNEL METHODS 60
4.1 Introduction 60
4.2 Physical Modeling 67
4.2.1 Electrostatic Interaction 67
4.2.2 Van der Waals Interaction 67
4.2.3 Molecular Surface Area 68
4.3 Biomolecular Datasets of Small Molecules 69
4.3.1 Toxicity 70
4.3.2 Solubility 71
4.3.3 Solvation Free Energy and Hydration 72
4.4 Basic Kernel Functions 72
4.5 Kernel Methods applied to biology 74
4.5.1 Element-specific Groups 74
4.5.2 Element-specific Graph Representations 75
4.5.3 Geometric Graphs 76
4.5.4 Algebraic Graphs 77
4.5.5 Algebraic Topology 78
4.6 Multi-scale Kernel Methods 80

CHAPTER 5 AWEGNN: AUTO-PARAMETRIZED WEIGHTED ELEMENT-SPECIFIC GRAPH NEURAL NETWORKS FOR SMALL MOLECULES 82
5.1 Introduction 82
5.2 Theory and Methods 85
5.2.1 Neural Networks 85
5.2.2 Overview of AweGNN 88
5.2.3 Normalization Function 92
5.2.4 Biomolecular Geometric Graph Representations 92
5.2.5 Parameter Adjustment and Initialization 96
5.2.6 Derivatives of the Representation Function 98
5.2.7 How to Update the Parameters 100
5.2.8 Multi-scale Models 101
5.2.9 Model Architectures and Hyper-parameters 101
5.3 Results 104
5.3.1 Evaluation Metrics 106
5.3.2 Toxicity Data Sets 107
5.3.2.1 LC_{50}-DM (Daphnia Magna) Set 108
5.3.2.2 Fathead Minnow LC_{50} Set 109
5.3.2.3 Tetraphymena Pyriformis IGC_{50} Set 111
5.3.2.4 Oral Rat LD_{50} Set 112
5.3.3 Solvation Data Set 114

5.4 Discussion 116
5.4.1 Impact of Automating Selection of Kernel Parameters 116
5.4.2 Feature Importance and Analyzing the Trajectories of the Kernel Parameters 117
5.4.3 Limitations and Advantages 119
5.5 Conclusion 122

CHAPTER 6 GENERAL AUTO-PARAMETRIZED KERNEL METHODS AND FUTURE WORK 124
6.1 Introduction 124
6.2 GPU-Enhanced AweGNN and Multi-scale Methods 124
6.2.1 GPU programming 124
6.2.2 Data Structure and Algorithm 125
6.2.3 New Capabilities 127
6.3 Comprehensive AweGNN with Fixed Topological and Spectral Descriptors 127
6.3.1 Model Features and Architecture 128
6.3.2 FreeSolv and ESOL Data Set 129
6.3.3 Results and Discussion 130
6.4 Auto-parametrized Spectral Layer 132
6.4.1 Differentiability of Eigenvalue and Eigenvector Functions 132
6.4.2 Derivatives of the Eigenvalue Functions and the Update Rule 135
6.4.3 Discussion 137
6.5 Auto-parametrized Kernel Models and Future Possibilities 138

CHAPTER 7 DISSERTATION CONTRIBUTION 141

BIBLIOGRAPHY 143

LIST OF TABLES

Table 5.1: Set statistics for quantitative toxicity data 108

Table 5.2: Comparison of prediction results for the LC50DM test set 110

Table 5.3: Comparison of prediction results for the LC50 test set 111

Table 5.4: Comparison of prediction results for the IGC50 test set 113

Table 5.5: Comparison of prediction results for the LD50 test set 114

Table 5.6: Comparison of prediction results for the solvation test set 115

Table 5.7: Supplementary results for solvation data set from Nguyen et al.[81] . . . 116

LIST OF FIGURES

Figure 2.1: This figure displays a vertex, edge, triangle, and tetrahedron from left to right, which are representations of the 0, 1, 2, and 3-simplices respectively. These are all realizations of the simplices by embedding them into 3 dimensional space [93]. 18

Figure 2.2: This figure displays the calculation of a homology group. The simplicial complex, K, is made up of one 2-simplex (triangle), abc, 3 1-simplices (lines), $ab, bc,$ and ac, and 3 0-simplices (points), $a, b,$ and c. If we focus on the 1-dimensional chain groups, we notice that there is one cycle represented geometrically by the loop going from a to b to c. This loop, however, is cancelled by the boundary of the simplex, abc, by "closing" the loop, thereby displaying the topological idea that the loop can be deformed continuously across the triangle surface. Therefore, there are no non-trivial loops in the complex, and so $\dim H_1(K) = 0$. If the simplicial complex did not contain the 2-simplex, then the complex, K, would have one non-trivial loop and we would have $\dim H_1(K) = 1$. 20

Figure 2.3: This figure displays the geometric construction of a Čech complex from a point cloud, $S \subset \mathbb{R}^2$, based on the radius parameter, r. The simplices are added based on the number of balls that have non-empty intersection. 23

Figure 3.1: This figure captures the challenge of finding just the right fit for a polynomial whose tunable parameters include the degree of the polynomial being fit. The true function that generated the distribution is shown in orange and the models that are fit to the data set are shown in blue. We can see that as we increase the degree of the polynomial, we more accurately fit the training data, but the model is becomes less able to generalize to new data points. We must balance the complexity of our model based on complexity of the distribution of our data [51]. 38

Figure 3.2: In this figure we show how varying the regularization constant when training models using ridge regularization affects how the model fits the data. On the left we see linear regression, and on the right we see polynomial regression. As the regularization constant increases, the model is "flattened". This "flattening" decreases the variance of the model, but increases bias [36]. 40

Figure 3.3: In this figure, we show a decision tree model making a prediction for a new data point, $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$. The threshold values for the nodes are denoted by T_1, T_2, and T_3. At each node, one of the features is compared to the threshold at that node, determining the direction that the data point will travel down the tree. Notice that only the features, x_1, x_2, and x_5, are used for any prediction. Decision tree models do not always consider all features of data point when making a prediction, especially when they are pruned or limited by a maximum depth parameter during construction. 43

Figure 3.4: This figure depicts the architecture of a simple feed-forward neural network with an input layer, hidden layers, and an output layer with one output. The neurons in the input layer, hold the values of the feature vectors of a sample from the data set, and the connections between the layers have weights associated with them, which determine how the values are transformed as they move to the neurons in the next layer of the network. The yellow lines above show the flow of a sample through the network that yields a prediction at the output layer, while the turquoise arrows pointing backwards show the flow of the calculations of the gradients at each layer before an update. . 49

Figure 3.5: An illustration of a general multi-task network with three tasks. The input data from datasets 1, 2, and 3 are fed forward simultaneously, but through the hidden layers corresponding to their task. Then the last layers of each individual task's hidden layers are pushed through the shared layers. Finally, the data points are separated again according to their task, and pushed through their corresponding second set of hidden layers and outputs to get the predictions. The average loss of the predictions is calculated, and the weights are then updated simultaneously. 59

Figure 4.1: This figure illustrates the solvent accessible surface compared to the Van der Waal surface. The solid red and gray colors represent the Van der Waals radii of the atoms in the molecule, and the striped yellow area represents the solvent accessible surface that is "rolled out" by the center of the translucent blue probe sphere (or solvent). 69

Figure 4.2: This figure shows a barcode representation being extracted from a biomolecular structure. On the left, we have the biomolecule in which the point cloud is taken by looking at the positions of all the atoms in the structure. On the right, we have the persistence information for the first 3 dimensions. The 0-dimensional persistence (points) for each class is on top, the 1-dimensional persistence (rings) is in the middle, and the 2-dimensional persistence (cavities) is on the bottom. The persistence is measure in Angtroms (Å) [22]. 79

Figure 4.3: This figure depicts the featurization of the barcode data from persistent homology calculations. On the top, the barcode representation is shown, with 4 bars representing the "lives" of 4 different homology classes as a simplicial complex "evolves" through a filtration. Below the barcode data, the birth, death, and persistence feature vectors are constructed using a max length of 4 and a resolution of 0.5. 80

Figure 5.1: Pictorial visualization of the AweGNN training process. The first stage is splitting up the molecules in the data set into the element-specific groups, with initialized η and κ kernel parameters for each group. Then, a molecular representation is generated based on the kernel parameters, given by **F** in the diagram. This representation is fed into the neural network, then the kernel parameters are updated by back propagation through the neural network and through the representation function, **F**. 90

Figure 5.2: Diagram depicting the structure and automation of the MT-AweGNN. The model is trained by collecting samples from each data set proportionally and then generating their representations according to shared parameters. The representations are normalized together, then pushed through the artificial neural network and then to their corresponding outputs. Back propagation goes through the feature matrix, normalization function, and finally the representation function to update the kernel parameters. 91

Figure 5.3: This shows an element specific subgraph corresponding to the element types C and O of aminopropanoic acid, $C_3H_6NO_2$ (with the hydrogens omitted). The dashed orange edges represent the edges of the subgraph that are weighted by the chosen kernel function, Φ, while the solid blue lines represent covalent bonds. Notice the 3rd carbon atom does not connect to the oxygen atoms since it is covalently bonded to both of them. 95

Figure 5.4: Evolution of the kernel function of the C-N group for a MT-AweGNN. The picture below shows a series of kernel functions that were used to evaluate the 4 features corresponding to the C-N element-specific group at different points in the training of an MT-AweGNN. We choose specific snapshots during training that show a smooth change in the kernel function as the η-κ pair is updated. 98

Figure 5.5: Internal architecture of our Auto-parametrized weight element-specific Graph Neural Network (AweGNN) model for the toxicity data sets. The input layer consists of our novel Geometric Graph Representation (GGR) layer and a batch norm with no affine transformation. Hidden layers include a linear transformation followed by regular batch normalization and ReLU activation, while the output layer has the standard linear activation. 103

Figure 5.6: Pictorial representation of the overall strategy. The arrows show the flow of the processes of training, extracting representations from trained AweGNNs, and making predictions. Blue corresponds to molecular data sets, light green corresponds to machine learning models, orange corresponds to molecular representations, and dark green corresponds to predictions. 105

Figure 5.7: Feature importance of MT-AweGNNs when applied to different data sets. We generate heat maps based on the average of feature importance across 42 RF models trained on the corresponding NEARs of the MT-AweGNNs. The average feature importance of the 4 features of each element-specific group are summed together to get the final scores shown above in the maps. Figure 5.7a shows the results for the LD_{50} MT-RF models and figure 5.7b shows the results for the IGC_{50} MT-RF models. Note that the maps are asymmetric due to the way that the electrostatics-based features of a group are generated. 119

Figure 5.8: Average trajectories of kernel parameters for 42 MT-AweGNN models. The graph on the left shows how the average of the η values of our 42 MT models for 8 different element-specific groups changes as the models are trained. On the right, we see the κ counterpart of this analysis. 120

Figure 6.1: This figure gives a graphical representation of the parallel addition algorithm for calculating the sum of an array. We start with an initial array, a, of length, n. Then we begin by doing pairwise additions, $a_i := a_i + a_{2i}$, for $i = 1, ..., n/2$. Then the next step includes pairwise additions, again $a_i := a_i + a_{2i}$, but now for $i = 1, ..., n/4$. The pairwise additions, $a_i := a_i + a_{2i}$, continue for $i = 1, ..., n/2^k$, at step k, until there is only one number remaining to be summed, and then the array sum is extract from the first element, a_0. The image uses green to show elements used in the sum at each step, grey to show inactive elements, and the red circle highlighting the final answer after all the steps are completed. 126

Figure 6.2: This figure shows the general structure of the comprehensive network described in this section. The geometric graph representation (GGR) layer from the AweGNN takes center stage with auxiliary topological and spectral features all merging together into one shared layer. Each set of features can have their own hidden layers to effectively process the features before merging. 128

Figure 6.3: This figure displays the results gathered from the MoleculeNet paper. 8 model types were tested on a validation set and a test set and the average performance using the RMSE metric were recorded. The error bars record the standard deviation of the model results [111]. 131

LIST OF ALGORITHMS

Algorithm 1: Gradient Descent 36
Algorithm 2: Batch Normalization 53
Algorithm 3: Descent with Momentum 55
Algorithm 4: AdaGrad 56
Algorithm 5: RMSProp 56
Algorithm 6: Adam 57

CHAPTER 1

INTRODUCTION

1.1 Machine Learning and its Applications

Machine learning can be defined as a field of computer science that uses algorithms that help a computer learn from data. More precisely, an algorithm is said to be a machine learning algorithm if for some task, T, that can be measured with a performance measure, P, the algorithm directs the computer in some way that will use experience, E, so that it will improve the performance on task T measured by P [36]. A simpler way to describe a machine learning algorithm is to say that the algorithm directs a computer to improve its performance on a task without being explicitly programmed to do so. This is in contrast to programming that would have to find an output response for each and every input response, which, in many cases, is totally infeasible. When data becomes too complex for the human mind to generate a simple set of rules to decide what to do in a given situation, then we might consider applying machine learning methods.

Machine learning algorithms have been applied in a variety of instances: computer vision, speech recognition, spam filters, medicine, self-driving vehicles, amongst others [84, 24, 29, 91, 35]. As the years pass by, many more more applications spring up every year as new ways of training machine learning models are discovered and computational capabilities naturally increase over time. Accessibility to massive amounts of data too, has added to the excitement of developing machine learning models, in which common people have access to millions and millions of pieces of data that are ready for use, or require little cleaning. In the field of molecular biology, interest has heightened as detailed molecular data has become available and demand for medical and bio-technologies has increased. Many successful machine learning models have been employed in the biomolecular field, and have caught the attention of many pharmaceutical companies,

which have been relying more and more on machine learning models to avoid costly experimentation to narrow down candidates for drugs.

1.2 Biomolecular Modeling

Molecular modeling is the science and art of studying molecular structure and function through model building and computation. The model building can be as simple as plastic templates or metal rods, or as sophisticated as interactive, animated color stereographics and laser-made wooden sculptures. The computations encompass ab initio and semi-empirical quantum mechanics, empirical (molecular) mechanics, molecular dynamics, Monte Carlo, free energy and solvation methods, quantitative structure/activity relationships (QSAR), chemical/biochemical information and databases, and many other established procedures. The refinement of experimental data, such as from nuclear magnetic resonance (NMR) or X-ray crystallography, is also a component of biomolecular modeling.

In particular, we want to create models that will help us predict various quantities or qualities of different biomolecules or biomolecular structures. For example, predictions of solvation free energy, toxicity, and solubility of molecular compounds are highly useful for biomedical purposes. The prediction of the binding free energy of protein-ligand complexes is also extremely important for drug development and design, saving precious time and money on costly experimentation. As there has recently been a shift from the qualitative and phenomenological analysis of biomolecular systems to more detailed and quantitative approaches, we would like to follow in that path.

We seek to apply machine learning algorithms to obtain models that can help us develop these predictive models, but this requires deep analysis and plenty of data. Fortunately, there has been an explosion in the availability of detailed biomolecular data, and methods of computer modeling have improved dramatically, making it possible to extract important information from the molecular structures to construct accurate pre-

dictive models. The computational capabilities of modern computers has spurred much advancement in the area of biomolecular modeling, and has shown no signs of stopping.

1.3 Abstract Mathematical Representations

Advanced mathematics can be used to understand many real-world problems through modeling and approximation techniques. In particular, we would like to apply these mathematical tools to the problem of biology. Algebraic topology, geometry, graph theory, and other techniques have been applied before to obtain successful predictive models that have state-of-the-art performance [78, 110, 22, 80, 81, 79, 108]. These mathematical tools are used to describe the geometry, structure, and underlying physics of the biomolecules and biomolecular constructs through calculation of abstract features. These features are then combined into representation vectors that describe each molecule in the data set, from which we may apply a multitude of machine learning algorithms to create predictive models.

Persistent homology, which measures persistence of the connected components, rings, and cavities of a cloud of points, can be used to extract features that provide a picture of the topological and geometric landscape of a biomolecular structure. Differential geometry can be used to measure the curvature of the interactions of the elements within a biomolecule or between the molecules in a biomolecular complex. Geometric graph theory can be used to describe the pairwise interactions between elements, and algebraic graphs can be used to provide spectral information, which may reveal information about the connectivity and other abstract properties about the arrangement of atoms. One can imagine that there are a plethora of possibilities when attempting to build representations for the various qualities of biomolecules that would be useful for applying machine learning algorithms, so we explore them ardently.

1.4 Kernel-based Methods and Auto-parametrization

Much of these mathematical representations of molecules and biomolecular systems rely on a chosen kernel function, and a careful tuning of parameters that determine the shape of the kernel function by setting the center of the cutoff and its sharpness. For instance, geometric graph features, as in the flexibility-rigidity index used in a previous paper, rely on the choice of a kernel function such as the Lorentz or exponential kernel function [78]. These are tuned by adjusting the τ and κ values that determine the distance of the cutoff and the sharpness of the cutoff. Representations based on curvature are also based on kernel choice and kernel parameter tuning, as well as the representations extracted from spectral data of adjacency and Laplacian matrices. Topological representations in previous works also did introduce kernel-based features, providing an opportunity to tune the models as well.

When using kernel-based features, we may develop multi-scale representations, where we combine multiple choices of kernels and kernel parameters that increase the number of features that we get for our representation. Multi-scale methods reveal extra detail that cannot be captured by the limited scope of just one kernel function and parameter choice, and have shown to be very successful in previous attempts [78, 80, 81], while single-scale models are very limited in comparison. The downfall of the multi-scale models is in the increase of kernel parameters that must be tuned.

Because of this increase in the amount of parameter choices when optimizing, it becomes increasingly time-consuming to tune them. It is computationally infeasible to go beyond a certain point without using techniques that miss key parameter combinations. In the case of element-specific features, the kernel and parameter choices per scale too must be fixed across all pairs, missing out on many possibilities for capturing various biological effects based on the interactions between different elements. This limits us significantly in how we can develop our models in that we are virtually constrained to non-deep learning models, such as gradient boosting trees and random forest algorithms,

since they are more easily trained with fewer hyper-parameters to adjust.

What we may do to bypass the problems above is to automate the process of parameter selection. By automating this process, we may tune the parameters during the training of our models. We can then take care of the parameter tuning with just one model instead of hundreds that must be compared to obtain the best result. Also, the parameters may be tuned specifically for each individual feature or small group of features. This means that hundreds of parameters may be tuned with one model instead of tuning 2 parameters after training hundreds of models. To automate the parameter selection, we utilized the back propagation of a neural network to tune these parameters, thereby introducing the possibility of using a neural network with kernel-based representations and solving the problem of the high cost for kernel parameter tuning. This yielded the Auto-parametrized weighted element-specific Graph Neural Network (AweGNN) [100], which used an element-specific kernel-based geometric graph representation for the molecules. This auto-parametrized kernel model was found to be very successful, and was later improved by increasing computational efficiency and adding additional features. Also, the idea of the AweGNN can be extended to other molecular representations using various abstract mathematical topics described previously.

1.5 Motivation

There has been exponential growth in the availability of data, even to the general population, and the rapid advances of machine learning and data science have provided us with a plethora of tools and techniques for modeling a variety of data sets and constructing very successful predictive models that can aid us in almost any situation. This has been increasingly applied to tackle the big problems in the biological sciences in recent years. The increase in detailed biomolecular datasets give us a large pool of data from which to develop these machine learning models. Due to this fountain of high-quality biomolecular data, machine learning has become the most powerful tool in com-

putational biology. Solvation free energy and hydration free energy, toxicity, solubility, protein-ligand binding free energy, etc. are all important physical quantities associated with biomolecules or biomolecular complexes that are useful in industry, medicine, and science. Machine learning models are commonly utilized to predict these values, in which advanced mathematical techniques can aid increasing performance greatly.

Kernel-based mathematical representations have been used to develop state-of-the-art machine learning models for a variety of molecular datasets. Despite the success of these models, the kernel-based models have a drawback. The kernels are dependent on parameters that have to be hand-tuned. This manual tuning can be somewhat time consuming, especially when multi-scale models are used. For multi-scale models, a parameter grid search is required, and when the number of kernels becomes too large, the grid search may be infeasible. Because of the computational cost of the grid search, neural networks are generally omitted from experimentation, and random forest and/or gradient boosting trees are applied in place of them. To remedy this issue, we sought to automate the kernel parameter-tuning by using the backpropagation of a neural network to update the parameters during training. This auto-parametrization technique, however, requires careful construction and computational efficiency to be applied successfully to these molecular datasets.

1.6 Outline

In chapter 2, we provide an some mathematical background in graph theory and persistent homology, providing vital theoretical detail to understand how it may be applied to extract molecular descriptors for molecular datasets. In chapter 3, we cover fundamental concepts in machine learning, including some details on specific machine learning algorithms, which we apply in later chapters. Chapter 4 introduces the concept of biomolecular modeling and describes the changes in technology and data availability that are spurring on the field. Various techniques in biomolecular modeling are covered,

including physical modeling, and kernel methods applied to machine learning. Chapter 5 details the entire AweGNN paper, which introduces the first auto-parametrized kernel model. Chapter 6 details the general idea of the auto-parametrized kernel methods and introduces improvements and theories that extend the original AweGNN concept. Some brief results for a solubility and hydration dataset are recorded and compared to other models on a benchmark dataset.

CHAPTER 2

MATHEMATICAL TECHNIQUES

2.1 Introduction

Mathematics has been applied to many different kinds of problems. Some of the first applications were found in commerce, land measurement, architecture, and astronomy. In our current times, we see that all sciences use a variety of mathematical tools to advance their subject in terms of modeling, understanding, and applying techniques to solve various problems, and many of these tools are also used in industrial applications. In many cases, there are teams that will enlist the help of a mathematician for the purposes of understanding various elements of problems arising within the many scientific disciplines due to the specialized nature of the matter. For an example of applying advance mathematical topics to scientific modeling, we may look to the physicist Richard Feynman, who invented the path integral formulation of quantum mechanics by combining his understanding of theoretical mathematical concepts and his insight in the field of physics [66]. Today's research on string theory, which is a scientific theory that attempts to unify the four fundamental forces of nature, continues to inspire mathematics and spur on many creative mathematicians to participate. There are many examples apart from this, such as the application of representation theory in chemistry, but the field of physics draws the most attention for its focus on mathematical modeling.

Some topics in mathematics are relevant only in the area that inspired it, and are applied to solve further problems in that particular area, while many times, the problems being solved even create new areas of mathematics that can be studied independently. Often times, mathematics inspired by one field proves useful in many different fields, and so merits the generalized study of a particular area of mathematics. There is often made a distinction between the study of pure mathematics, in which there is an emphasis

on strict proof-writing with a disregard for real-world application, and applied mathematics, where application of techniques are the focus and how to apply them to a variety of situations in the real world. There are many instances, however, of topics which were considered to be in the sphere of pure mathematics, but were applied to great effect in numerous problems, for example, the application of number theory in cryptography. Due to the explosion of knowledge in the modern scientific era, there have been scores of specializations within the field of mathematics, and there are certainly more to come. While the prospect of having well developed fields of mathematics and an understanding of the abstract mathematical objects therein is appealing for a number of reasons, we cannot lose sight of the understanding that there are important problems to tackle in this world, and abstractions do little on their own to pave the way to prevail against them.

Although there has been much effort to apply mathematics and computational techniques to the understanding of biological systems, we have not seen so many advances in the use of more abstract techniques as we have seen in physics and chemistry. Thus, we attempt to present some useful abstract concepts that may be useful in the field. In this chapter, we discuss special topics in two fields of abstract mathematics, graph theory and algebraic topology, and focus mainly on the applied elements of the discussed concepts. A base knowledge of linear algebra, abstract algebra, and topology are required to understand the details in this chapter. The theory described in this chapter will later be applied to topics in computational biology, as we will see in the later chapters.

2.2 Graph Theory

A simple graph is composed of a set of vertices and a set of edges in which the vertices represent objects of interest or points in space, and the edges represent some sort of relationship between the vertices, or simply a line between the points. This very basic definition can be extended to include additional information that describes the system in more detail. These additions can include a direction of the edges indicating that there is a

relationship from one vertex to another or that there is a mutual relationship. The edges might also take on values to represent some quantitative relationship between the two vertices, or the vertices themselves may have values of their own. The edges or vertices may be colored to categorize components of the graph that may represent a type of relationship or a type of vertex. Graphs can take on as deep of a complexity as they need, but in any case they are very useful and natural constructs for analyzing many different problems in our world.

Graph theory has many ties to different mathematical fields and sciences due to its simplicity and natural definitions. Set theory, number theory, algebra, topology, computational biology, computer security, geography, and many more [32, 40]. A deep tie between graph theory and linear algebra is present by virtue of the notion of the adjacency and Laplacian graphs that can be obtained from a graph. In our work, we use graphs to represent molecular structures, and relationships between the atoms that make up the molecular structures, where we can obtain numerical information from the matrices associated with the graph structure. This natural representation of biomolecular structures has been particularly fruitful [80, 78, 81, 79, 90], and so we will revisit the concepts learned in this section and apply them to biomolecular models. The theorems and definitions are mainly retrieved from the book, "Modern Graph Theory", by Bollobás [18].

2.2.1 Basic Definitions

An *undirected graph*, G, is a pair of disjoint sets, (V, E), such that $E \subset V^{(2)}$, where $V^{(2)}$ is the set of unordered pairs of elements of V. Since an element of the edge set, E, can be described by an unordered pair of vertices, $x, y \in V$, we denote an edge connecting those two vertices by the two-element set $\{x, y\} \in E$. We say that $G' = (V', E')$ is a *subgraph* of $G = (V, E)$, denoted $G' \subset G$, if we have $V' \subset V$ and $E' \subset E$. The *neighborhood* of a vertex, $x \in V$, denoted by $\Gamma(x)$, is defined as $\Gamma(x) = \{y \in V : \{x, y\} \in E\}$, the set of all vertices, y, that that are connected to x by an edge. Then the *degree* of the vertex, $d(x)$, is

naturally defined by $d(x) = |\Gamma(x)|$, the number of edges that are connected to the vertex, x. A graph, $G = (V, E)$, is called a *bipartite graph* if there are two vertex sets, $V_1, V_2 \subset V$, with $V = V_1 \bigcup V_2,\ V_1 \bigcap V_2 = \emptyset$, and the condition that for every $\{x, y\} \in E$, either $x \in V_1$ and $y \in V_2$ or $y \in V_1$ and $x \in V_2$. One important, yet trivial, type of graph is the *complete graph*, which is a graph that has every vertex connected to every other vertex by an edge. More specifically, a graph G of n vertices is called a *complete graph on n vertices* if the edge set of G, namely E, is exactly $E = \{\{x, y\} \in V^{(2)} \ : \ x, y \in V,\ x \neq y\}$. In this case, we denote G by $G = K_n$.

There are many properties that are important to graphs. One of the most basic properties of a graph is the number of connected components. For a graph, $G = (V, E)$, two vertices, $x, y \in V$, are *connected vertices* if the is a path from x to y. A *path from x to y* is defined to be a series of edges, $(\{x_1, y_1\}, \{x_2, y_2\}, ..., \{x_m, y_m\})$, such that $y_{i-1} = x_i$ for all $i = 2, ..., m$, with $x = x_1$ and $y = y_m$. The graph, G, is a *connected graph* if for every vertex pair of vertices, $x, y \in V$, such that $x \neq y$, x is connected to y. A maximal connected subgraph of the graph, G, is called a *connected component* of G, or simply a *component* of G. One useful property of a graph is its connectivity. To define this, we need some additional language.

Let $W \subset V$ be a subset of vertices of the graph, G. Then we define the *subtraction of G by W* as $G - W = (V - W, \{\{x, y\} \in E \ : \ x, y \in V - W\})$. If $G - W$ is disconnected, then we say that W *separates* G. If we let $k \geq 2$, then the graph, G, is *k-connected* if either $G = K_{k+1}$ or if G has at least $k + 2$ vertices and there is no set of $k - 1$ vertices that separates G. A graph is called *1-connected* if it is a connected graph. The maximal k value such that a graph, G, is k-connected is defined to be the *connectivity* of G, and is denoted by $\kappa(G)$. This value gives us an idea of the strength of a graph's overall connection.

In many cases, we may wish to extend the basic notion of a graph to give us extra detail. If a graph has an edge set of order pairs instead of unordered pairs, i.e. elements of the edge set, E, have a direction from one vertex to another that is represented by the

ordered tuple $(x, y) \in E$ going from the vertex x to the vertex y, then the graph is called a *directed graph*, or *digraph*. Direction in a graph may be very important, for example, if you are working with fluid flow or modeling the popularity of internet stars (you may follow an individual online, but they may not follow you back). A *weighted graph* is a graph in which each edge in the edge set has a corresponding value associated with it, or in other words, the graph G comes with a weight function, $w : E \to X$, from the edge set to a set of your choosing, where common choices are $X = \mathbb{R},\ \mathbb{Z},$ or $\mathbb{N}$. Weights are essential for encoding pertinent information that represents the strength of a connection between two vertices. This can be the strength of the flow within pipes at different junctions, the strength of an interaction between two atoms, etc.

2.2.2 Adjacency and Laplacian Matrices

In graph theory, an adjacency matrix is a square matrix that describes the important connective information of a graph. The entries of the matrix express whether the pairs of a set of vertices are adjacent in the graph or not. Many key features about the graph can be extracted such as information about the connected components of the graph, calculations of path and cycles, finding shortest paths, and other important information. Another useful matrix associated with a graph is the Laplacian matrix. This matrix contains very similar details, but it has other properties, especially related to the spectrum of the matrix, that can provide additional information that the adjacency matrix cannot.

The *adjacency matrix*, A, of an undirected graph, G, is defined by:

$$A_{ij} = \begin{cases} 1 \ : \ \{i, j\} \in E \\ 0 \ : \ \{i, j\} \notin E \end{cases}$$

It is easy to see that the adjacency matrix is a real symmetric matrix, i.e. A has real entries and $A = A^T$. This gives us useful properties that we will use later that involve the

spectrum of the matrix, which is the set of eigenvalues for the matrix. Namely, we have the important spectral theorem associated with real symmetric matrices:

Spectral Theorem. *Let A be a symmetric matrix. There is an orthogonal matrix P, i.e. $P^TP = I = PP^T$, such that $D = P^TAP$ is a diagonal matrix.*

The corollary of this theorem most importantly shows us that the eigenvalues and eigenvectors of an orthogonal matrix are real-valued. More precisely, D is a diagonal matrix of real entries, so the eigenvalues of D are real eigenvalues, say $\lambda_1, \lambda_2, ..., \lambda_n$ with corresponding eigenvectors being the standard basis, $\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n$. Now, let us define $v_i = P\mathbf{e}_i$ for every i. The theorem gives us that $D = P^TAP$ with $PP^T = I$, so $PD = AP$. Thus, we have that for every i, $\lambda_i v_i = \lambda Pe_i = P\lambda e_i = PDe_i = APe_i = Av_i$. This means that the eigenvalues of A are the real eigenvalues of D, and the eigenvectors of A are the columns of the orthogonal matrix of P (which has real entries), so they form an orthogonal basis of $\mathbb{R}^n$, with n being the size of the square matirx, A. Therefore, any real symmetric matrix has real eigenvalues and eigenvectors,

The other important matrix that we can associate with a graph is the Laplacian matrix, as was mention earlier. The Laplacian matrix is very similar to the adjacency matrix, but the difference is that the Laplacian matrix has the degree of each vertex encoded on the diagonal. More precisely, we define the *Laplacian*, L, as a matrix with the following entries:

$$L_{ij} = \begin{cases} -1 \ : \ \{i,j\} \in E \\ 0 \ : \ \{i,j\} \notin E, \, i \neq j \\ d(i) \ : \ i = j \end{cases}$$

The Laplacian can also be defined in a more concise way. Let us define Δ to be a diagonal matrix whose diagonal (i,i) elements are equal to the degree, $d(i)$, of vertex i of the graph G. Then, if A is the adjacency matrix associated with the graph, G, we define the Laplacian, L, as the difference of matrices, $L = \Delta - A$. We note that the Laplacian is

also a real symmetric matrix, so the same theorem above applies as well, namely that the eigenvalues are real numbers. The benefit of the Laplacian matrix comes from the results of the theorems associated with it, that reveal more about the spectrum of the Laplacian, and the relation of the spectrum to the graph. This theorem gives us some additional information about the spectrum of the Laplacian [18]:

Theorem. *The Laplacian matrix L of a graph, $G = (V, E)$, is a positive semi-definite matrix, i.e.* $\langle Lx, x\rangle \geq 0$ *for all* $x \in \mathbb{C}^n$*, where* $n = |V|$ *and*

$$\langle x, y\rangle = \sum_{i=1}^{n} x_i \cdot \overline{y}_i$$

for $x, y \in \mathbb{C}^n$*.*

With this theorem, we can easily see that all the eigenvalues of the Laplacian are all non-negative, since for any eigenvalue λ of L, and eigenvector $v \in \mathbb{C}^n$ normalized to unit length, i.e. $||v|| = 1$, we have $0 \leq \langle Lv, v\rangle = \langle \lambda v, v\rangle = \lambda\langle v, v\rangle = \lambda||v|| = \lambda$. Thus all eigenvalues of the Laplacian matrix of a graph are non-negative, and we also note that the zero eigenvalue occurs with multiplicy at least 1, since the vector $v = \langle 1, 1, ..., 1\rangle$ is always an eigenvector with eigenvalue $\lambda = 0$ because the sum of the elements in each row is zero. A second theorem that may be useful and reveals information about the graph is given below [18]:

Theorem. *Let G be a graph with Laplacian L. Then the dimension of the nullspace of L is the number of connected components of G.*

So, by applying computational techniques in linear algebra to find the dimension of the nullspace of a matrix, we may find the basic property of the number of connected components in a graph. We can also find more complex information from a deeper theorem dealing with the connectivity of a graph [18]:

Theorem. *The vertex connectivity of an incomplete graph G, i.e.* $G \neq K_n$*, is at least as large as the second smallest eigenvalue of the Laplacian of G.*

Now, we have a tool that can give us a lower bound for the connectivity of any graph just by using a standard concept from linear algebra, namely the eigenvalues of a matrix. Having calculated this value, we can understand how strongly connected a network or structure that can be put into the framework of a mathematical graph is. This second smallest eigenvalue is commonly referred to as the Fiedler value, and has many applications, such as the stability analysis of dynamical systems [5]. More generally, the theorem reveals that there may be some important information locked within the spectrum of a Laplacian, and so it might behoove us to analyze this.

2.2.3 Adjacency and Laplacian Matrices for Weighted Graphs

We consider now the notion of the adjacency matrix and the Laplacian for a weighted graph, $G = (V, E, \phi)$, where $\phi : E \to \mathbb{R}$ is the weight function that assigns the weights to the edges of the graph. Now, instead of constructing matrices with discrete values representing the presence or absence of edges, we construct similar matrices that account for the weights of the existing edges. More precisely, we define the *weighted adjacency matrix of G*, denoted by A_ϕ^G to be defined with entries:

$$(A_\phi^G)_{ij} = \begin{cases} \phi(\{i,j\}) \ : \ \{i,j\} \in E \\ 0 \ : \ \{i,j\} \notin E \end{cases}$$

We then define in a similar way the *weighted Laplacian of G*, denoted by L_ϕ^G to be defined with entries:

$$(L_\phi^G)_{ij} = \begin{cases} -\phi(\{i,j\}) \ : \ \{i,j\} \in E \\ 0 \ : \ \{i,j\} \notin E, \, i \neq j \\ \sum_{j=1}^{|V|} \phi(\{i,j\}) \ : \ i = j \end{cases}$$

Now that we have encoded the weights of the graph into the linear algebraic structures that are derived from them, we may turn our attention once again to the spectrums of

these matrices. In the case of the weighted adjacency and Laplacian graphs, we do not have the benefits of the aforementioned theorems, but we do have various instances of scholars revealing some connections between the spectrums of weighted Laplacians to geometry and graph theory [80, 85, 98, 20, 86].

2.3 Algebraic Topology

Although topology can be quite an abstract discipline, topological spaces can represent physical objects and notions so that they may be applied to problems in the real world [4, 1]. The biggest hurdle in applying topology is the effort required to take the smooth and stretchable nature of topological objects and transform them into something discrete or tangible that can be used for computation. One way to do this is to decompose the topological space into simpler topological pieces that ultimately can be encoded into a machine to make concrete computations. The spaces are sometimes approximated in a sense with these topological pieces, but they are able to give us accurate topological information because the structure produced is topologically the same. In fact, the field of algebraic topology gives us the means by which to extract discrete algebraic information from continuous topological data. A whole array of computational techniques in topology can be found in literature, including homology, cohomology, morse theory, and more [34].

We want to ultimately be able to apply computational topology to our problem of biomolecular modeling. Persistent homology is a geometry-sensitive extension of the idea of the homology of a topological space. The homology of a space captures the presence of n-dimensional holes, and persistent homology looks at the evolution of the homology of the space as it is being constructed over time. Persistent homology has seen widespread use in our group, providing a useful tool for building successful machine learning models [22, 90, 110, 70]. We seek to lay out the theoretical foundation of these ideas in this section, starting with the notion of the simplex, which are our building blocks

that we use to construct simplicial complexes. These complexes allow us to apply the rich theory of homology in a manner that is suitable for computation.

2.3.1 Simplicial Complexes

We wish to now formally present the idea of a simplicial complex, which can be used to approximate topological spaces so that we can obtain relevant topological information from computations. We will start with the geometric notion of a simplex, as outlined by Edelsbrunner [34]. If we let $x_0, x_1, ..., x_k$ be points in $\mathbb{R}^d$. Then we consider linear combinations, $x = \sum_{i=0}^{d} \lambda_i x_i$, in which each $\lambda_i \in \mathbb{R}$ and $\sum_{i=0}^{d} \lambda_i = 1$. These linear combinations are called *affine combinations*. The *affine hull* of these points is then defined to be the set of all affine combinations, where we notice that this makes a k-plane if the $k+1$ points are *affinely independent*, meaning that for any affine combination of points, $a = \sum_{i=0}^{d} \lambda_i x_i$ and $b = \sum_{i=0}^{d} \mu_i x_i$, then we have that $\lambda_i = \mu_i$ for all i. It can be seen that the $k+1$ points are affinely independent iff the k vectors $x_i - x_0$ are linearly independent for $i = 1, 2, ..., k$. Since $\mathbb{R}^d$ has at most d linearly independent vectors, we have that there are at most $d+1$ affinely independent points in $\mathbb{R}$.

An affine combination, $x = \sum_{i=0}^{k} \lambda_i x_i$, is a *convex combination* if $\lambda_i \geq 0$ for all i and the *convex hull* of the points is the set of all convex combinations of those points. A *k-simplex*, σ, is then defined to be the convex hull of $k+1$ affinely independent points, denoted by $\sigma = \text{conv}\{x_0, x_1, ..., x_k\}$, with its *dimension* defined to be $\dim \sigma = k$. On this note, we observe the first four dimensions of simplices can be represented by a point, line, triangle, and tetrahedron respectively. This fact is depicted in figure 2.1 below. Consider a non-empty subset of the points whose convex hull describes a simplex, $\sigma = \text{conv}\{x_1, x_2, ..., x_k\}$. We notice that the convex hull of any subset, $T \subset \{x_0, x_1, ..., x_k\}$, then defines another simplex, and we call this simplex a *face* of the simplex σ. A face of a simplex is called *proper* if the face is not the entire simplex, and we write $\tau \leq \sigma$ for any

face, τ, of the simplex, σ, and $\tau < \sigma$ if it is a proper face.

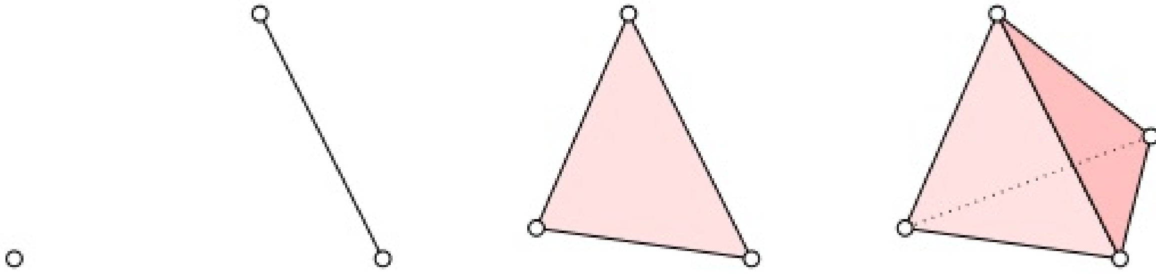


Figure 2.1: This figure displays a vertex, edge, triangle, and tetrahedron from left to right, which are representations of the 0, 1, 2, and 3-simplices respectively. These are all realizations of the simplices by embedding them into 3 dimensional space [93].

A *simplicial complex*, K, is a finite collection of simplices such that $\sigma \in K$ and $\tau \leq \sigma$ implies that $\tau \in K$, while $\sigma, \sigma' \in K$ implies $\sigma \cap \sigma'$ is empty or a face of both σ and σ'. In many cases, we want to represent these simplicial complexes in a more abstract way. We define an *abstract simplicial complex*, A, as a finite collection of sets such that $X \in A$ with $Y \subset X$ implies that $Y \in A$. The *dimension* of an abstract simplex is denoted $\dim X = |X|$. A *face* of the simplex, X, is a non-empty subset $Y \subset X$, and is *proper* if $Y \neq X$. We find that for any geometric simplicial complex, we can produce an abstract simplicial complex, and vice-versa [34]. Thus, we have an intimate connection between the geometric and abstract constructs, allowing us to make discrete computations with the abstract complexes which represent geometric realities. The main computations that we can make are through an algebraic object associated with a topological space called the homology of a space.

2.3.2 Simplicial Homology

The homology groups of a topological space provide an understanding of the k-dimensional holes that are present within the space. By looking at how k-simplices are continuously mapped into the space, we can extract a group structure that yields numerical information. To be able to make these calculations using our computational capabilities, we must use an equivalent version of the homology by using the simplicial structure of the space.

We begin by defining the objects of interest. For a simplicial complex, K, we define a *p-chain* to be a formal sum of p-simplices in K, $c = \sum_i a_i \sigma_i$, where the σ_i are the p-simplices in K and the a_i are the coefficients from a particular group. So as to allow for simple computations and inituitive understanding, we use the group $\mathbb{Z}/2\mathbb{Z}$, but other finite groups can also be used. In a theoretical setting, infinite groups such as $\mathbb{Z}$ are used, but groups like these may not allow us to obtain information from a computational source. The operation of addition can be defined for two p-chains by component-wise addition, i.e. for p-chains $c_1 = \sum a_i \sigma_i$ and $c_2 = \sum b_i \sigma_i$, we have $c_1 + c_2 = \sum (a_i + b_i)\sigma_i$. The set of p-chains of K under addition then define a group, $C_p(K)$, called the *chain group of K of dimension p*.

Now we wish to define a map between these chain groups, so that we can get a relation between simplices of different dimensions. Let us define the *boundary* of a p-simplex, $\sigma = [x_0, x_1, ..., x_p]$, by the sum of its faces, namely

$$\partial_p \sigma = \sum_{j=0}^{p} [x_0, ..., \hat{x}_j, ..., x_p],$$

where the hat over the x_j term means that it is omitted. Then, we can extend the idea of the boundary of one simplex to the whole chain group, $C_p(K)$, by defining $\partial_p : C_p(K) \to C_{p-1}(K)$ by making it linear, i.e. for a chain, $c = \sum_i a_i \sigma_i$, we have $\partial_p c = \sum_i a_i \partial_p(\sigma_i)$. In fact, this defines a homomorphism from the chain group of p-simplices to the chain group of $(p-1)$-simplices, and we call this the *boundary map* of chains. Now, an important algebraic structure that comes from these boundary maps is the *chain complex*, which is the sequence of chain maps and chain groups of all dimensions, namely

$$\cdots \xrightarrow{\partial_{p+2}} C_{p+1}(K) \xrightarrow{\partial_{p+1}} C_p(K) \xrightarrow{\partial_p} C_{p-1}(K) \xrightarrow{\partial_{p-1}} \cdots$$

To obtain the homology groups that we seek from this chain complex, we must define two special types of chains, namely p-cycles and p-boundaries. A *p-cycle* is a p-chain whose boundary is the 0-chain, i.e. $\partial_p c = 0$. Since the kernel of the homomorphism, ∂_p, is a subgroup of $C_p(K)$, then the set of p-cycles under addition, which we denote by

$Z_p(K) = \ker \partial_p$, is indeed a group. Next, we define a *p-boundary* as a p-chain that is the boundary of a $(p+1)$-chain, i.e. c is a p-boundary if there is a $d \in C_{p+1}(K)$ such that $\partial_{p+1} d = c$. Then, $B_p(K) = \operatorname{im} \partial_{p+1}$ is a group of the p-boundaries of $C_p(K)$. An important result that allows us to calculate the homology groups of a chain complex is the fact that $\partial_p \partial_{p+1} = 0$. This means that $B_p(K) = \operatorname{im} \partial_{p+1} \subset \ker \partial_p = Z_p(K)$, and so we may define the pth *homology group of the simplex K* as $H_p(K) = Z_p(K)/B_p(K)$. We then define the *dimension* of the homology group, denoted by $\dim H_p(K)$, to be the rank of $H_p(K)$ (or the dimension of the group when viewed as a vector space). The dimension of $H_p(K)$ is also known as the pth *Betti number*. This can be denoted by $\beta_p = \dim H_p(K)$.

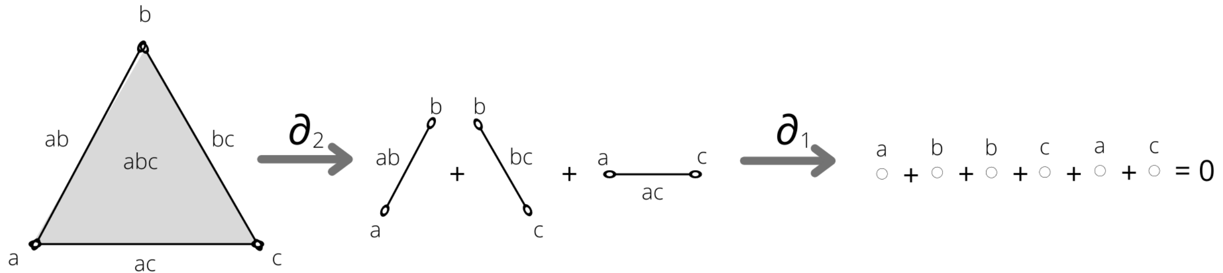


Figure 2.2: This figure displays the calculation of a homology group. The simplicial complex, K, is made up of one 2-simplex (triangle), abc, 3 1-simplices (lines), $ab, bc,$ and ac, and 3 0-simplices (points), $a, b,$ and c. If we focus on the 1-dimensional chain groups, we notice that there is one cycle represented geometrically by the loop going from a to b to c. This loop, however, is cancelled by the boundary of the simplex, abc, by "closing" the loop, thereby displaying the topological idea that the loop can be deformed continuously across the triangle surface. Therefore, there are no non-trivial loops in the complex, and so $\dim H_1(K) = 0$. If the simplicial complex did not contain the 2-simplex, then the complex, K, would have one non-trivial loop and we would have $\dim H_1(K) = 1$.

We notice that the cycles represent loops, cavities, and higher-dimensional "holes" that can be found in a complex, while boundaries represent the higher dimensional simplicial structures that "fill in" the space inside the holes. This idea is illustrated in figure 2.2, showing the calculation of the 1-dimensional homology group. The Betti numbers that we retrieve from the calculation of these homology groups give us numerical information about the topology of the complex, which may be useful in some applications. We will extend the ideas in this section to reveal deeper information about the structure of topological spaces, which are infinitely more useful than the betti numbers of a fixed simplicial

complex.

2.3.3 Persistent Homology

We wish to now obtain information from an evolving complex that slowly increases the number of simplices that form it. As new pieces enter the complex, the homology of the complex will change. Holes will come into and out of existence through merging classes of cycles, and we would like to capture the persistence of each of these topological features. We obtain this persistence by first explaining what is known to be the elder rule. This rule dictates how persistence is understood, by giving precedence to the older cycles when two cycles merge as a boundary is introduced to disrupt them. The older cycle persists as the younger cycle dies.

We consider a simplicial complex, K, with a monotonic function, $f : K \to \mathbb{R}$, in which $f(\tau) \leq f(\sigma)$ whenever $\tau \leq \sigma$. With this definition of monotonic, we notice that the sublevel sets, $K(a) = f^{-1}(-\infty, a] \subset K$ are a subcomplex of K. Since K is discrete, we have a finite amount of values, $a_0, a_1, ..., a_n \in K$, where we can define $K_i = f^{-1}(-\infty, a_i]$ for each i so that we get an increasing sequence of subcomplexes,

$$\emptyset = K_0 \subset K_1 \subset ... \subset K_n = K$$

.

We call this sequence of subcomplexes a *filtration of f*. Now, for every, $i \leq j$, we have an inclusion map from the ambient space containing K_i and K_j, namely $\iota^{i,j} : K_i \hookrightarrow K_j$, which induces a homomorphism, $\iota_p^{i,j} : H_p(K_i) \to H_p(K_j)$, for every dimension. If we consider the consecutive induced inclusion maps, we obtain a sequence of homology groups for every dimension, p,

$$0 = H_p(K_0) \longrightarrow H_p(H_1) \longrightarrow ... \longrightarrow H_p(K_n) = H_p(K).$$

As we go through the homology groups of the filtration along the induced inclusion maps, we find that non-trivial cycles in one homology group get merged with others as new boundaries are introduced. The length at which these non-trivial cycles persist will then be captured by doing some calculations of Betti numbers. We define the pth *persistent homology groups* to be the images of the induced maps, $H^{i,j}(K) = \operatorname{im} \iota_p^{i,j}$, and the pth *persistent Betti numbers* to be the dimensions of the corresponding groups, $\beta_p^{i,j} = \dim H^{i,j}$. First, notice that $H_p^{i,i} = H_p(K_i)$. Then notice that the persistent homology group, $H_p^{i,j}$, consists of the homology classes of K_i that have survived to the complex, K_j. More formally, we have that $H_p^{i,j} = Z_p(K_i)/(B_p(K_j) \cap Z_p(K_i))$. For $\gamma \in H_p(K_i)$, we say that γ is *born at* K_i if $\gamma \notin H_p^{i-1,i}$. Second, if γ is born at K_i and then merges with another class at K_j, i.e. $\iota_p^{i,j-1}(\gamma) \notin H_p^{i-1,j-1}$, but $\iota_p^{i,j}(\gamma) \in H_p^{i-1,j}$, then we say that γ *dies entering* K_j. This then is considered to be the elder rule mentioned earlier.

We now have a way to obtain information about the persistence of homology classes, or cycles, with respect to our filtration function, f. If γ is a class born at K_i and died at K_j, then we define the *persistence* of the class to be pers$(\gamma) = a_j - a_i$, where the a_i and a_j are the values that determined the sublevel sets, $K_i = f^{-1}(-\infty, a_i]$ and $K_j = f^{-1}(-\infty, a_j]$, of the filtration. We can also similarly define the birth and death by their filtration, i.e. birth$(\gamma) = a_i$ and death$(\gamma) = a_j$. These values are calculated with sophisticated packages such as Ripser, Gudhi, and Dionysus [16, 101, 77]. The filtration functions and the companion complexes are usually obtained from the evolution of various complexes springing from a point cloud. In the case computational biology, these point clouds are derived from the positions of atoms in a molecule. Next, explore the Čech Complex, which gives us a physically relevant filtration in which we can apply the idea of persistent homology to a given point cloud.

2.3.4 Čech and Vietoris-Rips Complexes

Suppose we have a finite set of point, $S \subset \mathbb{R}^n$. Let us define the closed ball around a point $\mathbf{x} \in S$ with radius, r, by $B_{\mathbf{x}}(r) = \{\mathbf{y} \in \mathbb{R}^n \ : \ ||\mathbf{x} - \mathbf{y}|| < r\}$. Then we define the *Čech complex* as:

$$\check{\text{C}}\text{ech}(r) = \{\sigma \subset S : \bigcap_{\mathbf{x} \in \sigma} B_{\mathbf{x}}(r) \neq \emptyset\}.$$

An abstract p-simplex is included in the Čech complex if there are $p+1$ points whose closed balls of radius, r, have anon-trivial intersection. Although the Čech complex is defined using abstract simplices, we can imagine the process using the notion of geometric simplices. This idea is illustrated in figure 2.3.

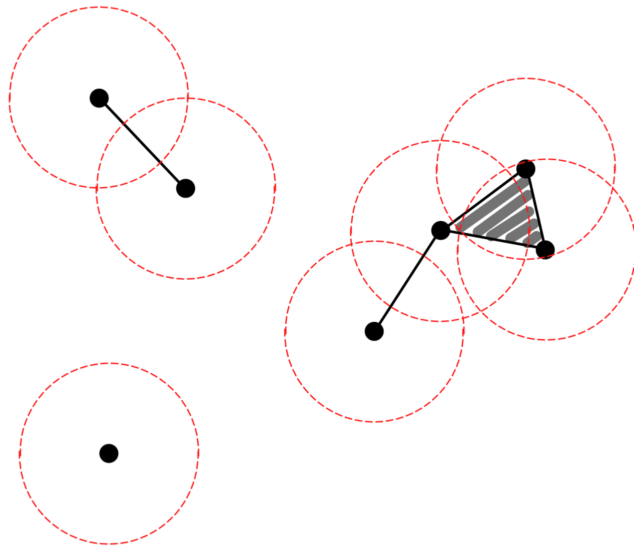


Figure 2.3: This figure displays the geometric construction of a Čech complex from a point cloud, $S \subset \mathbb{R}^2$, based on the radius parameter, r. The simplices are added based on the number of balls that have non-empty intersection.

For computational and geometric reasons, another complex was introduced, namely the Vietoris-Rips complex. The *Vietoris-Rips complex* of S and r is defined as:

$$\text{Vietoris-Rips}(r) = \{\sigma \subset S \,:\, \text{diam}(\sigma) \leq 2r\}$$

Instead of checking all subcollections of points to find out which simplices are in our complex as we do in the Čech complex, we may simply just check pairs and add 2-dimensional and greater simplices whenever we can. It can be shown also that we have the inequality, Vietoris-Rips$(r) \subset$ Čech$(r) \subset$ Vietoris-Rips$(\sqrt{2}r)$ [34], so we have a close relationship between the two complexes. In reality, the Vietoris-Rips complex is used extensively in application because of the ease of computation in relation to persistent homology.

We can see that the Čech and Vietoris-Rips complex is dependent on the choice of a point cloud, S, and a choice of parameter, r, determining the radius of the balls around those points. If we define $\mathcal{P}(S)$ to be the set of all subsets of S, then $\mathcal{P}(S)$ is a simplicial complex, and we can define a filtration function, $f : \mathcal{P}(S) \to \mathbb{R}$, where for a simplex σ, we have $f(\sigma) = r_\sigma$, where $r_\sigma \geq 0$ is the value in which $\sigma \in$ Čech(r) for $r \geq r_\sigma$, but $\sigma \notin$ Čech(r) for $r < r_{sigma}$. This can be equivalently defined for the Vietoris-Rips complex. In both cases, these filtration functions give us a way to obtain persistent homological data from a point cloud of atoms in a molecule, which gives us important physical features for their analysis. In addition to this, there is a very popular complex called the alpha complex, which is more complicated to describe, but is more physically representative of a molecular structure than the Čech and Vietoris-Rips complexes. This construction, however, will not be discussed here.

CHAPTER 3

MACHINE LEARNING

3.1 Introduction

Machine learning (ML) is a field of computer science that deals with algorithms that create models that improve automatically through experience and/or by the application of fixed data. ML is generally seen as a subfield of artificial intelligence (AI) because on the surface it seems as though the computer being programmed is "learning" from the data that is fed into it. In fact, the computer is programmed to do a particular task without being explicitly programmed to do so, giving it the appearance that it is "learning" from real-world data. In many cases, this might be the only feasible way to create a reasonable program since the direct programming of every single action and response will be far too much to handle for one or even many human beings. For example, one might consider the difficulty of producing an explicit algorithm that can identify and differentiate between various objects contained in an image. ML algorithms have a wide variety of applications: computer vision, speech recognition, spam filters, medicine, self-driving vehicles, amongst others [84, 24, 29, 91, 35]. More and more applications are discovered every year as new ways of training these ML models are invented and computational efficiency increases over time.

There are three main categories in machine learning that explore different ways of learning from the training data. These categories are supervised learning, unsupervised learning, and reinforcement learning. Some might include semi-supervised learning as a category of its own, but semi-supervised learning is really just supervised learning with some extra techniques that can handle unlabeled data. The most common type of learning is supervised learning, which learns from data that is labeled in the sense that inputs are paired with desired outputs usually hand-chosen by a human being through either

manual affirmation or scientific experimentation. Within supervised learning, there are two types of tasks that the models are trained for, namely regression and classification. Regression looks for predictions that may take a continuum of values, i.e. the prediction of the price of a home, whereby classification looks for outputs that place a piece of data into a particular category, i.e. the categorization of objects in an image. Unsupervised learning learns something about a dataset without any labels as described above. In this type of learning, the goal is to understand the structure of the data and find hidden patterns or insights of the input that can be used to understand how data should be labeled. Clustering algorithms are commonly used to categorize a given set of data, such as in the case of search results provided by search engines [2]. In reinforcement learning, the computer interacts with a dynamic environment through actions and inputs, and learns through trial and error how to to maximize its reward. An example of this would be the training of DeepMind's AlphaZero [102], which is a chess-playing algorithm that learned how to play by analyzing an inordinate amount of chess games against itself.

In this section we will be describing various concepts in machine learning, focusing solely on supervised learning algorithms, and going over basic and more advanced algorithms along with their advantages and disadvantages.

3.2 Fundamental Concepts

3.2.1 Data Representations

When implementing most machine learning algorithms, we represent each individual item in the data set as a vector whose components are numerical values that describe the object in some way. These vectors are commonly referred to as *feature vectors*, and the entries of the vectors are called features. In some cases, we use the term descriptors instead of features or refer to a feature vector as a representation of a data point, but they all refer to the same concepts. Although all entries of a feature vector must ultimately be some real number, we can have different categories of feature types. For instance,

common categories include numerical, binary, and categorical feature types.

The numerical feature type is the simplest to explain in that the entry in the feature vector is the exact number that describes that feature, i.e. the entry for height of a particular person in a data set would be a numerical feature of that person. Binary type features are entered as 1 or 0 based on whether they are in or out of a category, i.e. a 1 may represent that someone has a bachelors degree or a higher degree while 0 may represent that the person has not completed a bachelors degree or higher. Categorical feature types are originally represented by an integer taken from the range 0 to $k-1$, where k is the number of categories for that particular feature. This choice of category is then usually converted into a vector of length k by a one-hot encoder which outputs a vector with 0s everywhere except for a 1 in the ith place, in which i is the chosen category. All of the information from each feature type is concatenated into one long feature vector and the collection of these feature vectors from each data point is sometimes called a *design matrix*, where the rows of the matrix are the feature vectors of each individual data point.

There are some shortcomings to representing data points as the concatenation of a collection of numerical, binary, and categorical features as described above. The first thing to note is that categorical features that are one-hot encoded into a vector of 1s and 0s are not able to pick up on the correlations between two different categories. The representation then is conveying that all categories are equally dissimilar, which is a big assumption that is in most cases unrealistic. Another problem is that the weight of each feature is not appropriately captured. Some features will be incredibly important and contribute much to the construction of the model while others will do very little and perhaps even disrupt the process and ultimately result in an less accurate model. If some numerical values are very large in comparison to other features, then there may be an issue in which the large values outweigh other features in terms of the influence in computation, especially categorical or binary features, which do not exceed a value of 1. A problem that can occur with specifically categorical features and the prevalence of them in a representation is

that when concatenating a one-hot encoded choice of category to the feature vector, the feature vector can become sparse and waste a lot of memory and increase computing time significantly. Finally, we are not always able to easily figure out which features are going to be irrelevant to our analysis, and so we must do our best to eliminate redundant or uncorrelated features to save time, memory, and create more accurate models.

To deal with these shortcomings, there are several techniques that are applied. The main technique that is used is rescaling, but sometimes techniques like embedding are applied as well to deal with some of the remaining issues. Rescaling transforms each individual feature based on the values of that particular feature with respect to all the other data points. One way to do this is to take the maximum and minimum of the values of each individual features and scale them so that every value lies on or between 0 and 1, more specifically, if k_{min} and k_{max} are the minimum and maximum values respectively of the kth feature, then a feature value is transformed by the function $N(x) = \dfrac{x - k_{min}}{k_{max} - k_{min}}$. The most popular method of rescaling employed is the method that sets all features to have a value from a distribution of mean 0 and standard deviation 1. In this case, for the kth feature we take the mean, μ_k, and standard deviation, σ_k, with respects to all the values in the data set of the kth feature, then transform the values with $N(x) = \dfrac{x - \mu_k}{\sigma_k}$. There are also other popular rescaling methods, such as term frequency-inverse document frequency (tf-idf) [62], that are applied in natural language processing, but the above techniques are usually sufficient. This rescaling of all the features, which is also referred to as normalizing the features is useful for removing any differences in scale between the different features, and makes sure that the computations do not become too large during training before they have a chance to converge.

Aside from rescaling features, feature embedding is a technique that reduces the size of feature vectors that remove sparse elements of a feature vector and include mainly the important features. There are many ways of doing feature embedding that have been studied. Popular global dimensionality reduction algorithms include principle com-

ponent analysis (PCA), independent component analysis (ICA), and multi-dimensional scaling (MDS) [47, 42, 54]. Local dimensionality reduction algorithms, which are good for finding local patterns, include t-distributed neighbor embedding (tSNE) and uniform manifold approximation (UMAP) [58, 68]. Other types of feature embedding are representations learned from neural networks, which are encoded using a neural network, such as is done with auto-encoders, Word2Vec, FastText, BERT, etc [60, 39, 61, 31]. Finally mixture models have also been used, such as the guassian mixture models (GMM), Dirichlet multinomial mixture (DMM), etc [88, 41]. These dimensionality reduction algorithms can be incredibly useful for cleaning up your data, reducing computational weight, and for ultimately creating effective predictive models.

Data extraction, preparation, and cleaning can be just as important, if not more than choosing the right machine learning algorithm. There is a common saying, which is "garbage in, garbage out", and is commonly stated in the machine learning sphere. Your model can only be as good as the data that you train it on, no matter what machine learning algorithms you use. The most important thing you can do is gather a lot of data and extract relevant and important features to represent your data. The rest of the process is in finding a sufficient model to understand your data.

3.2.2 Loss Functions

Loss functions are usually used for a model during training to assess the progress that is being made in the process of fitting the machine learning model to the data. Different loss functions will have different advantages and disadvantages, and so should be chosen based on the goals that you have in mind regarding the accuracy of particular data points in your data set. Different distributions of the data at hand and the type of machine learning task, either regression or classification, may also guide your decision. The two most common loss functions for regression tasks are the mean squared error (MSE) and the mean absolute error (MAE). These are given below:

$$L_{MSE}(\widehat{y},\ y) = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2 \tag{3.1}$$

and

$$L_{MAE}(\widehat{y},\ y) = \frac{1}{n}\sum_{i=1}^{n}|\widehat{y}_i - y_i| \tag{3.2}$$

where the $\widehat{y}$ vector represents the predictions from the machine learning model, and the y vector represents the true labels. These two loss functions have their own advantages and disadvantages. The MSE loss is commonly used because it is a convex function, and so in many cases, this may allow us to find a global minimum. It is also sometimes possible to find a closed form solution to the minimization of the loss when considering the entire model, like in the linear regression case. The MSE is not however good at dealing with data sets that contain many outliers because the squaring of the error affects the loss more significantly in those case, so MSE is generally used as long as the data has a relatively normal distribution. MAE is not a convex function, and so is usually less desirable, however it does have the advantage of ignoring the outlier data points better than the MSE loss.

In terms of classification problems the most common loss function is the cross entropy loss (CEL). This is given below:

$$L_{CEL}(y,\ \widehat{y}) = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\ln\widehat{y}_{ij} \tag{3.3}$$

where in this case, y and $\widehat{y}$ are $n \times m$ matrices representing the probability that the ith data point is in the jth category, and so the rows all add up to one. $\widehat{y}$ represents the predicted probabilities, while y represents the true values, meaning that each row has a single 1 with the remaining entries being 0 since each data point is certainly in one of the categories and absolutely not in any of the other ones. The entropy of a random variable is the uncertainty inherent in the possible outcomes of that random variable. So a

random variable that is evenly distributed has a high entropy because you can never tell what value it might take, however, when the random variable is concentrated at one or a handful of points, then the entropy is significantly reduced. With the cross entropy loss, we are simultaneously measuring the entropy *and* the accuracy, where we are penalized for both giving high probability to incorrect labels and for being uncertain in our predictions. This allows the models that we are training to be optimized in a way that provides accurate and concrete predictions.

There are other loss functions that are used in classification problems, but we will not cover those explicitly. When dealing with machine learning algorithms with loss functions, we must choose these loss functions according to our problems at hand. The distribution of the data, the algorithm being used, etc. can all affect our choice of the loss function, but ultimately there are no one-size-fits-all solutions to this dilemma. We must use our experience and intuition to decide which loss to use, then apply it to the situation and see if it is an effective measure according to our task.

3.2.3 Evaluation Metrics and Data

While loss functions are used to train models, evaluation metrics are used to evaluate a model after it has been trained. Although in many cases it is appropriate to use the loss function to make this evaluation, we may find that another metric may be more desirable or insightful in certain circumstances. For instance, when building binary classification models, we are interested in analyzing the precision and recall of our classifiers. Each prediction that we have is a true positive (TP), true negative (TN), false positive (FP), or a false negative (FN). The precision and recall scores are given by:

$$\text{precision} = \frac{TP}{TP + FP} \tag{3.4}$$

and

$$\text{recall} = \frac{TP}{TP + FN} \tag{3.5}$$

The precision measures the accuracy of the classifier on the set of all positively identified data points, and the recall measures the ability to detect the positively labeled data points. Depending on our situation, we may value precision over recall or vice-versa. For example, suppose we have we are creating an app that tells you whether a plant you found in the wilderness is safe to eat (positive) or not (negative) by looking at an image of it. In this case, you would like to make sure that there is an extremely low chance that the app could be wrong if it labels it safe to eat. This means that we would like a very high precision for the model. On the other hand, a lower recall would be an acceptable trade-off since we would not be too upset if the app labeled many things that were safe to eat as potentially dangerous because the alternative would be an app that is very good at finding exotic plants that are safe to eat, but once in awhile gives the okay for someone to ingest a poisonous substance. If we consider the case of a model that detects that a particular location is possibly being infiltrated by intruders to a group of security guards, it is easy to see that we would prefer the reverse situation, in which the recall is much more important than the precision. In both of these illustrations, we note that although we most likely trained our model with the cross-entropy loss function, we did not use this metric to evaluate our model.

In this work, we are mainly interested in regression tasks, so we want to focus on evaluation metrics involving regression predictions. The most common metric used for regression is the root mean squared error (RMSE). This is just the square root of the MSE loss, i.e. for true labels, y, and predicted labels, $\widehat{y}$, we have:

$$\text{RMSE}(\widehat{y}_i,\ y_i) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2} \tag{3.6}$$

The main difference between MSE and RMSE is that the units for the RMSE are the same as that of the labels, so will provide some readability to the results. In some cases,

we may also use the MAE if we want to understand the tendency towards outliers. These can often be compared side-by-side to get a full picture of a models performance. A metric that is commonly used, but is not as intuitive of a measure of success as the RMSE, is the Pearson correlation coefficient. This metric is a measure of the linear correlation between the predicted labels and the true labels, and so gives us some information about how well the model fit the data set. We let $\text{cov}(x,\ y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})$ and $\sigma_z = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(z - \overline{z})^2}$ be the covariance and standard deviation of the distributions, x, y, and z, where $\overline{x}$, $\overline{y}$, and $\overline{z}$ are the respective means. Then we may define the Pearson correlation coefficient for predictions, $\widehat{y}$, and true labels, y, by:

$$r_{\widehat{y},y} = \frac{\text{cov}(\widehat{y}, y)}{\sigma_{\widehat{y}}\sigma_y} \tag{3.7}$$

We note that the value of $r_{\widehat{y},y}$ always satisfies $-1 \leq r_{\widehat{y},y} \leq 1$, where $|r_{\widehat{y},y}| \to 1$ means that the correlation between the predicted and true values is getting closer to being linear. A perfect prediction, i.e. $\widehat{y} = y$, gives a Pearson correlation of $r_{\widehat{y},y} = 1$, while predictions that are entirely independent of the labels, i.e. $\text{cov}(\widehat{y}, y) = 0$, gives a Pearson correlation of $r_{\widehat{y},y} = 0$. The advantage of using this metric is that it has no units associated with it, so we can get a rough measure of how well our model is performing based on how close the correlation coefficient is to 1. The problem with using this metric alone, however, is that there are instances in which the Pearson correlation is equal to 1, but the predictions are very inaccurate. For this reason, we still combine this metric with the RMSE and/or MAE. Other correlation coefficients also exist, such as Spearman's rank correlation coefficient and Kendall's tau rank correlation coefficient. The Spearman coefficient measures how well the relationship between two distributions can be described by a monotonic function and Kendall's tau is a measure of the portion of ranks that match between two data sets. Although these two metrics are somewhat different from the Pearson correlation, they can also provide us with the benefit of giving an understanding of the model performance without analyzing the error in units.

When we use these metrics to evaluate our model, we are ultimately trying to measure the generalization error, or the error that we would have with our chosen metric if tested on new data. For this problem, we generally split our data set into a train and test dataset in which the test set is made up of around 10-20% random data points of the whole dataset and the train set is made up of the remaining data points. The model would then be trained with specific model parameters on the train set to then get an estimate of this generalization error. Now if you have many parameters that you can tune for your model and are able to run many tests, you can use this test set to determine the optimal choice of model parameters. A problem that you may run into when doing this is that you may be chosing the best model parameters for that specific test set, and so the parameters chosen cannot generalize as well as you may want. The remedy to this is to introduce a validation set, which is usually an additional 10-20% of the whole dataset that is taken specifically from the train set to be able to tune the model parameters. In this case, we would have 60-80% of the dataset as the train set, 10-20% for the validation set, and 10-20% for a test set, in which the models are trained on the train set and validated on the validation set to decide the optimal parameters based on the scores of your performance metric chosen, then finally tested on the test set to determine generalization error. An alternative way to tune model parameters would be to do a k-fold cross-validation in which the dataset is broken into k pieces, where a model can be tested on each piece, being trained in each instance on the remaining $k-1$ pieces, where the average of the errors of the k instances of trained models of a particular set of model parameters gives an estimate of the generalization error. The more folds, the more accurate our estimate of the generalization error will be, but also the more computationally heavy the process will be. The number of folds used when applying cross-validation is about 5 to 10,

Different metrics can give us different sorts of information and we must use our intuition ultimately to figure out what evaluation metric is the best to use to assess the success of our models. Each metric has its advantages and disadvantages, and so in many cases,

we may consider using more than one to better understand their performance. Our main goal is to improve our generalization error so that we have a model that can understand how to deal with new instances of data, and so we must use statistical techniques to split the data up in a way that allows us to measure this error.

3.2.4 Model Weights and Gradient Descent

Many machine learning models make predictions based on intermediate or direct calculations involving scalar values that are adjusted throughout the training as the model "fits" the data that it is being trained on. We refer to these adjustable scalar values as weights, and we differentiate them from the model parameters (or hyperparameters) mainly by the two properties mentioned; that is, the direct use of them in calculations used to make a prediction, and the automatics adjustments that are made to them to fit the data at hand. Model parameters stay fixed throughout the training of a model, and so have to be hand-tuned by comparing their performance through running many models at once. Many machine learning models do not have weights that are adjusted throughout training, but build up a model through various criteria, and have many parameters that can be changed manually, that determine how these models are built. Machine learning models that rely on decision trees do not generally have any weights that are associated with their models, while simple algorithms like linear regression do have model weights that can be adjusted throughout training, although it turns out that linear regression models can be trained in one step in special circumstances (see below section). Neural networks are perhaps the most complicated models involving weights, and they involve a lot of them in fact. Neural network models tend to have millions of weights involved or more!

The most common way to adjust these weights is through a process called gradient descent. This process applies the idea of the gradient of a scalar function of multiple variables. The gradient of a function, f, is defined to be the vector: $\nabla f(x_1, ..., x_l) = \left(\dfrac{\partial f}{\partial x_1}, ..., \dfrac{\partial f}{\partial x_l}\right)$. The gradient of a function at a given point gives us a vector in the di-

rection of the greatest ascent, meaning that the directional derivative is optimal in the direction of the gradient. Since the gradient gives us the direction of greatest ascent, then the opposite of the direction of the gradient gives us the direction of the greatest descent. Thus, if we follow the opposite of the gradient at small enough intervals, then we will eventually reach a local minimum of the function. Suppose we choose a scalar value, $\alpha > 0$, in which the smaller α is, the shorter step that we take in the direction of the rate of steepest descent. If we do n iterations of the gradient descent after choosing an initial point, $(z_1, ..., z_l)$, then the following algorithm describes the gradient descent procedure quite concisely:

Algorithm 1 Gradient Descent

1: $(x_1, ..., x_l) \leftarrow (z_1, ..., z_l)$
2: **for** $i = 1, ..., n$ **do**
3: $\quad (x_1, ..., x_l) \leftarrow (x_1, ..., x_l) - \alpha \cdot \nabla f(x_1, ..., x_l)$

If we have chosen enough iterations, i.e large enough n, then we should have reached a local minimum. Now, if we think of the function, f as the loss function with respect to the weights of our model, i.e. $L(w_1, ..., w_l)$, then this gradient descent procedure will give us weights that will minimize our loss locally, if not globally. After we reach a local minimum, we would consider our model fully trained.

If there is a model with weights, then as long as the loss function is differentiable (or almost everywhere differentiable) with respect to its weights, then we are able to use gradient descent. In most cases, this is a feasible procedure, but sometimes the loss function is such that the gradient descent procedure would take too many iterations to effectively settle on a local minimum. Although a large learning rate may speed up the training process, if the learning rate is too large, we will not be able to settle at a local minimum. The loss function might have valleys that are too deep and the landscape of the function may be too erratic so that a very small learning rate may be required to traverse the parameter space to properly settle at a point. There are ways to find local minimums more

efficiently by using momentum and other considerations, so we can usually avoid the pitfalls of inefficient training times or inability to settle in a local minimums. The idea of a local minimum can be troubling in general when using a model based on weights. In general, we want to be able to find global minimums, and so convex functions are sought after for they can guarantee a global minimum. This is unrealistic, however, especially if you wish to consider relatively complex models, such as neural networks. Despite any disadvantages that models with weights may have, they are the easiest to implement, and have seen widespread use.

3.2.5 Regularization

There are two important problems that arise when applying machine learning algorithms to a data set to train a predictive model: underfitting and overfitting. *Underfitting* occurs when a model is fit to the data set, but does not perform well even on the data set that it was trained on. Overfitting is characterized by a high performance on the training data and a low performance on the test data. *Overfitting* occurs when a model fits the data that it is trained on so well that it cannot perform well on any new data. The model is so tailored to the particular data set by finding patterns that are very specific and not generalizable to new data so that it cannot effectively predict anything that might even be similar to the training data. Overfitting is characterized by a very high performance on the training data with low performance on the test data. Although underfitting is an important thing to avoid, the main focus for machine learning experts is on overfitting since most machine learning algorithms can create models that almost perfectly fit the data that they are trained on, but it is hard to find an algorithm that is very generalizable. In figure 3.1 below, we see a classic example of choosing an appropriate model to avoid both overfitting and underfitting.

A theoretical result that has important application in machine learning is that a model's generalization error (A measure of how accurately the model can predict previously un-

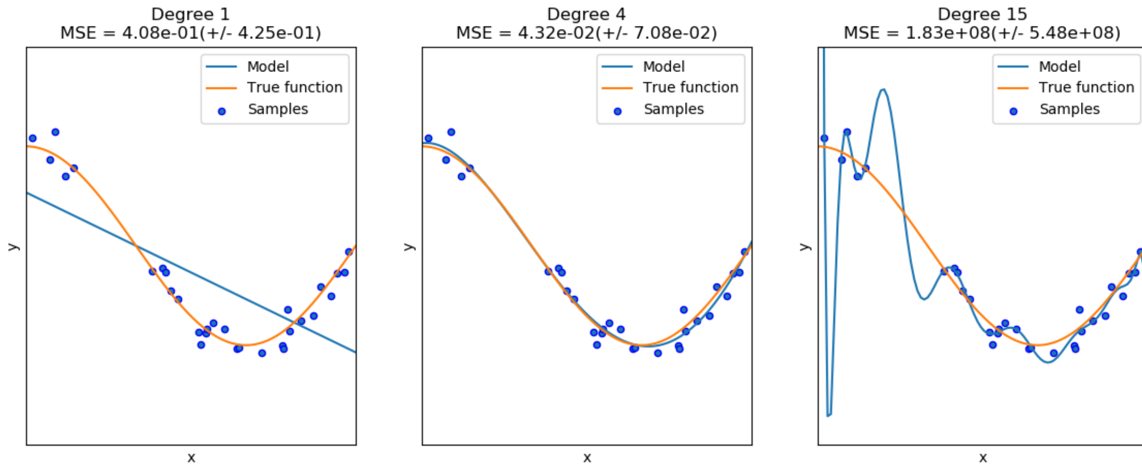


Figure 3.1: This figure captures the challenge of finding just the right fit for a polynomial whose tunable parameters include the degree of the polynomial being fit. The true function that generated the distribution is shown in orange and the models that are fit to the data set are shown in blue. We can see that as we increase the degree of the polynomial, we more accurately fit the training data, but the model is becomes less able to generalize to new data points. We must balance the complexity of our model based on complexity of the distribution of our data [51].

seen data) can be broken up into the sum of three very different components: bias, variance, and irreducible error. *Bias* is the error that comes from incorrect assumption about the shape of the data, which usually implies underfitting. *Variance* is the error that comes from sensitivity to variations in the training data. High variance generally comes from models with too many parameters, and usually results in overfitting. *Irreducible error* is the error that comes from the noisiness of the data itself and cannot be reduced by tuning the model. Irreducible error must be dealt with by retrieving better data, or by cleaning the data. As the complexity of the model increases, the variance typically increases will the bias decreases. This is customarily referred to as the bias/variance trade-off.

Regularization in machine learning refers to any attempts to avoid overfitting while training a machine learning model. In many cases, we try to use the simplest models that we can to improve our generalizability while still being able to fit both the train and test data, but we also can use a combination of techniques which are generally classified as regularization techniques. One common regularization technique is to modify the loss function that is being used to train the model so that the loss takes into account the com-

plexity of the model in some way thereby penalizing a model based on how complex it is with it's current parameters. The easiest way to do this in most models is to make sure that the magnitude of the parameters are kept small by incorporating the weights into the loss function in a way that penalizes the model if the weights are too large. The main assumption with penalizing large weights in a model is the belief that smaller weights create a less complicated model.

The two most popular ways to put the above idea into practice are through ridge regularization and the least absolute shrinkage and selection operator (LASSO). Ridge regularization, also known as l_2 regularization, focuses on penalizing excessively large weights and usually has a nice form for differentiation purposes. LASSO regularization, or l_1 regularization, has the ability to help with feature selection, but does not have some of the desirable features of ridge regularization. Suppose that we have weights associated with a particular machine learning model, say $\{\theta_l\}_{l=1}^{d}$, then we have the following terms for the ridge and LASSO techniques respectively:

$$L_{ridge} = \sum_{l=1}^{d} \theta_l^2 \tag{3.8}$$

and

$$L_{lasso} = \sum_{l=1}^{d} |\theta_l| \tag{3.9}$$

Then the loss, L_{error}, is modified by the ridge or LASSO loss to get a new loss: $L = L_{error} + \alpha L_{ridge}$ or $L = L_{error} + \alpha L_{lasso}$, where $\alpha \geq 0$ is a parameter called the regularization constant that can be tuned which determines the amount that the magnitude of the weights will contribute to the total loss. If α is very small, then the training will be effectively the same as with no regularization. If α is very large, then our model will be close to a constant function. Ultimately there is no good value that will give us the best performance, but we must tune this parameter when training our models to see which

one allows our model to generalize the most effectively. In the figure 3.2, we observe the effects of applying ridge regularization to linear and polynomial regression techniques.

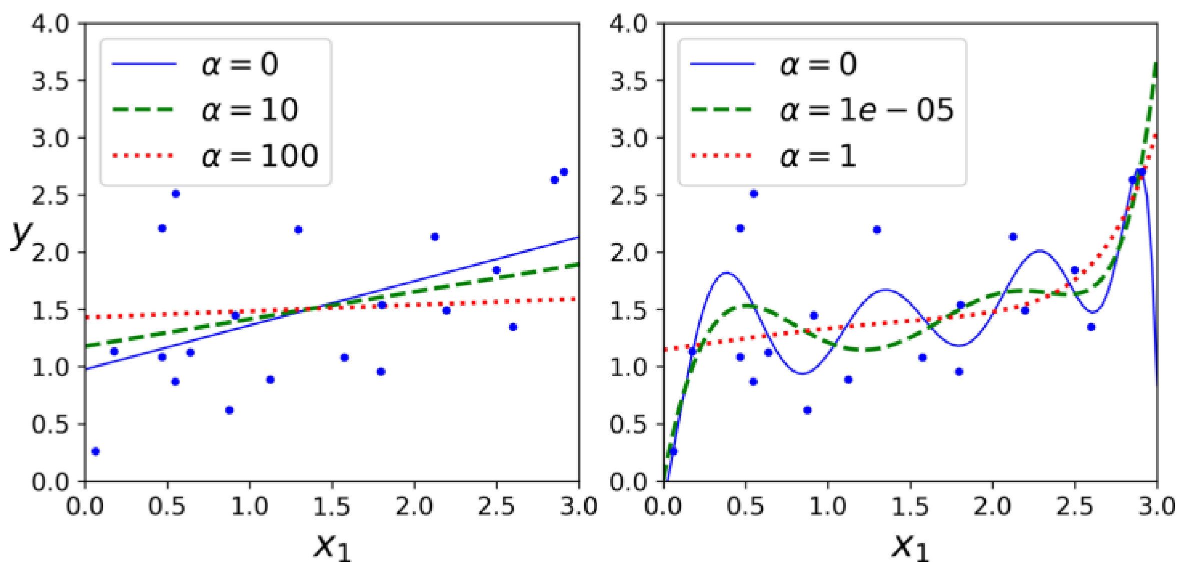


Figure 3.2: In this figure we show how varying the regularization constant when training models using ridge regularization affects how the model fits the data. On the left we see linear regression, and on the right we see polynomial regression. As the regularization constant increases, the model is "flattened". This "flattening" decreases the variance of the model, but increases bias [36].

There are also other forms of regularization that are used, particularly in neural networks, such as weight decay, dropout, and other indirect forms of regularization. Regularization is perhaps the most important addition to any machine learning model since it allows us to increase our model's generalizability if we are lacking a large sum of data.

3.3 Linear Regression

The linear regression algorithm is perhaps the simplest algorithm in the machine learning toolbox, but it is the most reliable, quickest to train, and the easiest model to tune. The task of linear regression is to create a linear model that fits the dataset most closely. The formulation is as follows: A loss function, L, is chosen; usually the mean squared error (MSE), which is again defined by

$$L_{MSE}(\widehat{y},\ y) = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2 \tag{3.10}$$

The predictions, $\widehat{\mathbf{y}}$, are given by the linear predictor:

$$p(\mathbf{x};\mathbf{a}) = \sum_{j=0}^{m} a_j x_j = \mathbf{a}\cdot\mathbf{x} \tag{3.11}$$

The vector, $\mathbf{a}$, is the set of coefficients for the linear function, and $\mathbf{x}$ is the vector of variables in which the component, x_0, is held to be a constant 1. In the case of the loss $L = L_{MSE}$, the optimization problem becomes:

$$\min_{\mathbf{a}} \sum_{i=1}^{n}(p(\mathbf{x}^{(i)};\mathbf{a}) - y_i)^2 = \min_{\mathbf{a}} \sum_{i=1}^{n}(\mathbf{a}\cdot\mathbf{x}^{(i)} - y_i)^2 \tag{3.12}$$

Where the $\mathbf{x}^{(i)}$ is the feature vector of the ith data point in the data set, which is described by the design matrix, X. Now, by applying basic optimization techniques using derivatives, we can get a solution by the multiplication of matrices:

$$\mathbf{a} = (X^T X)^{-1} X^T \mathbf{y} \tag{3.13}$$

This gives us a solution that requires the inversion of a matrix. In most practical situations, we need not worry about the existence of an inverse, and in the cases that we may have to worry about this, we have tools like the pseudo-inverse [14] that can take care of that particular issue. For matrices of a size below a certain threshold, this all may be computationally feasible, but for a larger matrices, there are other techniques, such as gradient descent, that can give us an extremely accurate solution. In the case that we use a different loss function, such as the mean absolute error (MAE), we do not have a closed equation and so must in fact use gradient descent. The same problem may occur if you wish to apply some form of regularization to the model, in which Ridge and LASSO regression are common choices.

ScikitLearn has resources which include as many options as you would need for any linear regression task, and is a useful tool for a myriad of other machine learning algorithms [83].

3.4 Decision Trees

Decision trees in machine learning are tree structures that can be constructed by fitting the data to provide a regression or classification-based predictive model that arrives at its prediction through a series of comparisons performed at each node until it reaches a leaf element of the tree. These are constructed greedily by using an appropriate cost function to determine the best split. Decision trees can be constructed by choosing a few options, such as the cost function, the number of splits per node, the minimum number of data points needed to make a split, the maximum depth of the tree, and others. There are not too many parameters that must be considered with decision trees, so they are relatively easy to tune.

In a regression task, a prediction is made by navigating a new data point through the nodes of the tree according to the split criteria until a leaf node is hit, then averaging the labels of all the data points in that leaf node to get the prediction. In the case of a classification task, the most common label in the batch at the leaf determines the final prediction. Regression tasks most commonly use the MSE cost function to determine the best split, while the GINI index measure (a measure that determines the purity of a set of data) is applied during classification tasks to reduce the overall entropy at each split in the tree. The other parameters can provide necessary rules for creating robust trees that can prevent overfitting, such as pruning, which removes branches that make use of features with low importance and don't improve the accuracy of the model. An example of a simple decision tree model is given below in figure 3.3.

The greatest advantage of decision trees is the ease at which we can determine the feature importance and the relations between the data points at each split. The model is

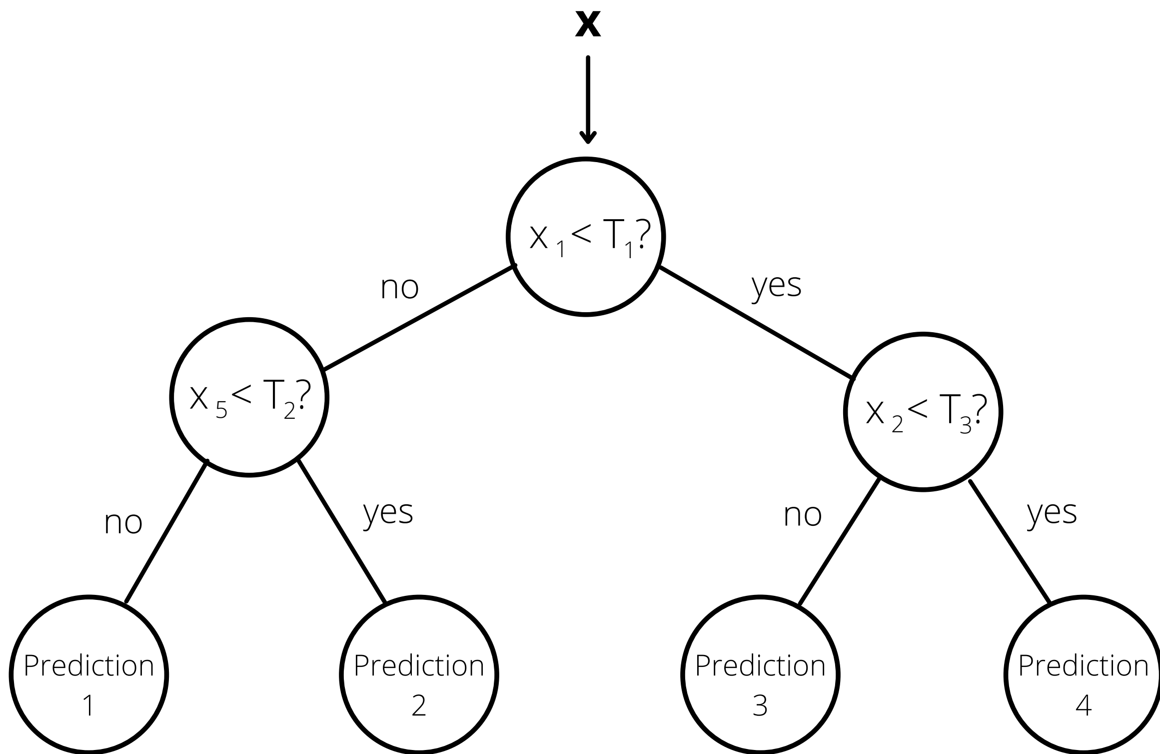


Figure 3.3: In this figure, we show a decision tree model making a prediction for a new data point, $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$. The threshold values for the nodes are denoted by T_1, T_2, and T_3. At each node, one of the features is compared to the threshold at that node, determining the direction that the data point will travel down the tree. Notice that only the features, x_1, x_2, and x_5, are used for any prediction. Decision tree models do not always consider all features of data point when making a prediction, especially when they are pruned or limited by a maximum depth parameter during construction.

natural and easy to understand as a prediction for a new data point is made since it has its features analyzed at each node of the path down to a leaf node. Decision trees also require little data preparation before being trained, i.e. normalization of the features, and any nonlinear relationships between the parameters do not effect the performance of the tree since the splits are always done comparatively instead of being based on some form of abstract distance. There are, however, many disadvantages to the decision tree models. For one, there is a large tendency to overfit if the tree is not properly pruned and/or the maximum depth, minimum samples to split, or other options are not wisely chosen. Secondly, there is a lot of variance involved in creating these trees since dropping or adding a feature may create a very different tree. Thirdly, greedy algorithms do not always give

an optimal model, and so we prefer to use other techniques if possible. Finally, the trees may be unbalanced, in which we mean that they are dominated by one particular class, thus creating a model with high bias.

There are many ways to combat the disadvantages of the decision trees. First, for practical use, we almost always choose to use a binary split for each node since it is faster and just as effective, but also reduces variance and overfitting. Also, balancing the data before training can reduce the effect of bias, which should be practiced for any model in fact. Next, methods such as bagging may be used, which are statistical techniques that use randomness and a consensus of multiple models to combat overfitting and variance, and help us get closer to a global minimum that the greedy nature of the algorithm cannot find for us in most cases. Finally, boosting is another technique that uses multiple trees to make a prediction, where the trees are built sequentially based on the performance of earlier trees, but the concepts of bagging and boosting will be covered in detail below.

3.5 Ensemble Methods

Ensemble methods are techniques that combine different models together into one to improve the accuracy of the results produced by the consensus of all the models as a whole. In many cases, ensemble methods improve the accuracy of models by a significant amount, and so have contributed to their popularity. The most popular ensemble methods are boosting, bagging, and stacking. There are also two main categories of ensembles, namely sequential and parallel, in which sequential ensemble methods combine models that depend on the previous models built and parallel models combine models that are built independently and rely solely on the random seed to establish a variety of models. Ensemble methods have been shown to reduce the variance and bias that exist within the single models.

Below we detail two main forms of ensemble learning, bagging and boosting, and then go over concrete examples of both bagging and boosting that use decision trees as

weak learners for their ensembles. Namely, these two ensemble models are random forest (RF) and gradient boosting trees (GBT), which have been used in many of our previous works. Random forest models are classified as parallel ensemble models and gradient boosting trees are classified as sequential ensemble models.

3.5.1 Bagging

Bagging stands for bootstrap aggregating. Bootstrapping is a sampling technique in which the samples are chosen from the data set using a randomized sampling with replacement. A model is trained on the chosen samples in which we have allowed repetition. In some cases, we may even chose a randomized subset of the set of features to increase the variety of models put forth. This procedure is repeated a number of specified times thereby giving us a number of weaker models that can be combined, or aggregated, to produce better results than if we simply had one model trained on all the data at once, namely with a far lower variance. The bagged model is made up of these m weaker models:

$$\widehat{f}_{bag}(X) = \frac{1}{m}\left(\widehat{f}_1(X) + \widehat{f}_2(X) + \cdots + \widehat{f}_m(X)\right)$$

Because of the aggregating process, the variance is lowered naturally by the averaging of these models. Bagging, however, does not do as well when the models that are aggregated are of low variance already. Decision trees and kNN models with low k value are good candidates for applying the bagging technique because of the ease in which they can be modified to create weak learners with high variance.

3.5.2 Boosting

Boosting is a sequential ensemble technique that learns from the mistakes of the previous models in the sequence to create a new model that will be combined with the previous ones to create a very successful final model. The main boosting techniques are AdaBoost,

Gradient Boosting, and XGBoost. AdaBoost uses models with stump splits that partition the data set based on a threshold of their feature values, and create models in a sequential manner that change the weight of each split after each instance of training. Gradient boosting uses the notion of a gradient to reduce errors made by previous models according to a chosen loss function and adds them to the ensemble. XGBoost uses gradient boosted trees in a computationally efficient way to make it possible to feasibly train on larger data sets.

3.5.3 Random Forest

There are many problems that require representations consisting of thousands of features and tens of thousands or even hundreds of thousands of data points. This can easily become too great for a single decision tree to handle. Random Forest (RF) is a parallel ensemble method that combines a group of weak learners, specifically decision trees, that would normally perform poorly as individual models into one single model which increases performance dramatically. Random Forest uses the bagging ensemble technique to deliver stellar performance and is known to be quite robust to overfitting, while also having few hyperparameters to tune, thereby minimizing the time it takes to optimize the models.

More specifically, we start by choosing an amount of trees in our forest to use in our final predictions. Then, for a fixed size (in which the size is chosen to be the whole dataset in practice, i.e. in the ScikitLearn implementation [83]), we randomly choose a number of data points equal to that fixed size with replacement for each individual tree. Next, for each of these trees, we also choose a subset of the features (this time without replacement) that will be used for the construction of each tree. The size of the subset of the features that will be use for each tree will usually be set to be the floor of the square root of the number of features, which is the default of the ScikitLearn random forest model. Finally, with each of these randomly determined data sets, we construct a decision tree. Now, our

model operates by making predictions from either a consensus of the predictions of the models in the forest (in the case of classification), or by the average of all the predictions of the models of the forest (in the case of regression).

3.5.4 Gradient Boosting Trees

Gradient Boosting Trees (GBT) are sequential ensemble models that uses decision trees as weak learners. The procedure involves adding one tree at a time to minimize a chosen loss function by following a gradient based on the performance of the previous trees. The gradient boosting trees have similar advantages to the RF algorithm in that they are also quite robust to overfitting and have relatively few hyperparameters to tune. The gradient boosting trees, however, do generally perform far better than the RF models after optimizing the hyperparameters. We want to detail how the gradient boosting algorithm works.

We choose a loss function, L, which is usually the standard MSE loss. Suppose now that F is our combined model which gives us predictions based on input representations, X is the matrix representing the data set, and y is the set of ground truth labels for each data point. Then $L(F(X), y)$ is the loss quantity measuring the error of our predictions, $F(X)$, when compared to the true labels, y. Let F also be considered as a sequence of functions, F_l, in which we begin with an initial weak learner, F_0, which is usually chosen to be a constant predictor which predicts the mean value, $\frac{1}{n}\sum_{i=1}^{n} F(X_i)$. Then we follow in sequence with a series of base learners, h_l, that are combined with the previous base learners to create an improved model, leaving us with a final model, $F = F_m$, after m iterations of improvement.

We wish to ultimately minimize our loss $L(F(X), y)$ by reducing the loss at each iteration. To reduce our loss, we train a new base learner, h_l, which will be a simple decision tree that is trained on the negative gradient, $-\frac{\partial L(F_l(X), y)}{\partial F_l(X)}$, then we follow the gradient by adding a scalar multiple of this h_l to the previous function, F_l. Thus, we update the

model, F, by a simple rule of the form: $F_{l+1} = F_l + \gamma_l \cdot h_l$, where the value γ_l is chosen by solving the below optimization problem at each step:

$$\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_l(X_i) + \gamma \cdot h_l(X_i)) \tag{3.14}$$

A learning rate, α, which takes a value from 0 to 1 is generally applied so that the actual update rule above looks like $F_{l+1} = F_l + \alpha \cdot \gamma_l \cdot h_l$.

3.6 Artificial Neural Networks and Deep Learning

Neural networks are predictive models that consist of layers which are made up of neurons in which each neuron is connected to every other neuron in the next layer of the layer sequence. A weight is associated with each connection that determines how much a neuron contributes to the input of the neuron in the next layer, and a bias term is introduced to shift the output to that neuron. Activation functions, such as the logistic sigmoid, grant the network a level of non-linearity that gives it sufficient complexity that sets it apart from a linear predictor. A feature vector, in which each feature corresponds to a neuron in the input layer, can be pushed through the layers of the network in sequence all the way to the output layer in a process called forward propagation. Neural networks can be trained as classification models that categorize data, or regression models that predict continuous quantities. Neural networks are trained by updating the weights and biases of the network in a series of steps through gradient descent, in which an appropriately chosen loss function is minimized. The process of calculating the errors of a neural network is called back propagation, in which derivatives are calculated in the backwards direction starting from the output layer, where the loss (or error) is calculated, and propagating backward through each layer in the opposite order all the way back to the input layer. The architecture and training process is depicted in figure 3.4.

Artificial Neural Networks (ANN) were originally inspired by vision systems and brain structure. The artificial neurons in a the ANN model are loosely analogous to the

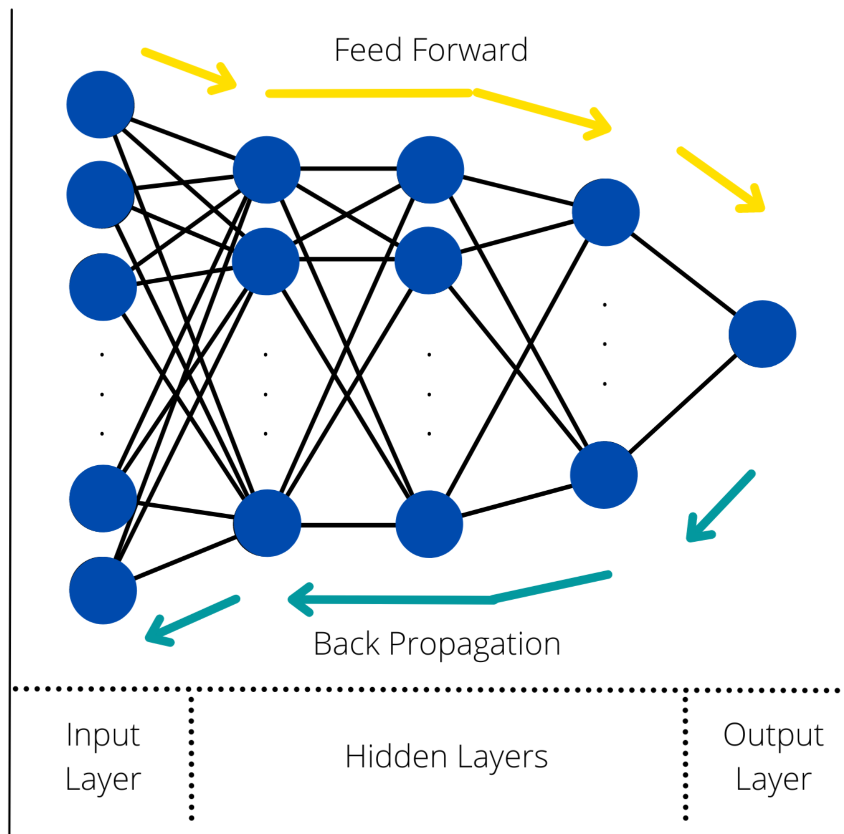


Figure 3.4: This figure depicts the architecture of a simple feed-forward neural network with an input layer, hidden layers, and an output layer with one output. The neurons in the input layer, hold the values of the feature vectors of a sample from the data set, and the connections between the layers have weights associated with them, which determine how the values are transformed as they move to the neurons in the next layer of the network. The yellow lines above show the flow of a sample through the network that yields a prediction at the output layer, while the turquoise arrows pointing backwards show the flow of the calculations of the gradients at each layer before an update.

neurons of a brain, and the connections between them are analogous to the action of synapses that transmit information throughout the brain. The concept of the ANN was first introduced in 1943 by McCulloch and Pitts [67], then the idea was expanded by researchers like Rosenblatt [89] until a roadblock was hit in the late 1960s in which a paper written by Minsky and Papert [71]. Interest in neural networks steadily increased as many factors changed over the decades. Progress in the field picked up when computational capabilities increased and computer technology became more advanced. Vast quantities of data became available to make training neural networks more effectively. Algorithms have improved, and theoretical hurdles were found to be harmless in practice. Graphical

processing units (GPUs), which were originally produced for graphical enhancement in the gaming industry, were applied to neural networks for an incredible computational speed boost, allowing for very large networks to be trained in a reasonable amount of time. Finally, a generous cycle of funding for neural network-related research stimulated advancement in neural network technology, and attracted many talented individuals.

Deep neural networks (DNNs) are defined to be networks with two or more hidden layers, but many contemporary networks contain more than a dozen layers, and sometimes even a hundred layers or more. Virtually all neural network research is referred to as deep learning research, and generally involves more complicated architectures, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), multi-task (MT) networks, and many more. Below, we review many important concepts involved in deep learning, especially those that were used for the networks described in this work.

3.6.1 Activation Functions

Activation functions in neural networks provide the necessary non-linearity that makes them far more successful than a linear model. They are generally applied at each layer of the network to create multiple levels of non-linearity that enhance the ability of the network to fit detailed data sets. There are various activation functions that are commonly used in deep learning today. They include the logistic sigmoid function, hyperbolic tangent function, Rectified Linear Unit (ReLU), etc. They are described below:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.15}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.16}$$

$$\text{ReLU}(x) = \begin{cases} x \ : \ x \geq 0 \\ 0 \ : \ x < 0 \end{cases} \tag{3.17}$$

As you can see, the sigmoid function keeps it's output in between 0 and 1 while the hyperbolic tangent function keeps it's output values between -1 and 1, so both functions have outputs within a relatively small band. This is to keep the values small at each layer so that the gradients do not get out of proportion, which is especially bound to happen in very deep networks. This normalization from the activation functions, however, comes with the problem of vanishing gradients when the activations are applied multiple times throughout a network. The problem of vanishing gradients then can be fixed by using the ReLU function, which does not suffer from vanishing gradients to the same extent as the sigmoid and tanh function, and has many other benefits.

The ReLU function has the derivative of 1 when non-zero, and so it is a constant 1 at each application, and does not shrink in magnitude as it is consecutively applied, thereby assuaging the deleterious effects of the vanishing gradient. The ReLU function also provides a sparsity to the neurons that are fired in the network, in which the neurons that have no use in the network are effectively turned off if they are not contributing to the improvement of the model. Finally, the computational requirements of the ReLU are far less than the sigmoid or tanh functions because ReLU does not have any exponential calculations included, thus reducing the computational cost of activation.

There are also drawbacks to the ReLU activation function. The first of which is the non-bounded nature of the output in the positive direction, which can create an unstable network if the initialization is not properly done. Another possible issue with the ReLU activation function is the dying ReLU problem, which occurs when too many of the weights corresponding to a neuron get close to zero and the ReLU activation cannot recover from repeatedly outputting zero, and this happens to too many neurons overall thus leading to a dead network. Thirdly, small outputs can benefit a neural network

model, but ReLU outputs only zero for all non-positive values, so doesn't allow that extra nuance within the model. Lastly, the ReLU function is not differentiable at all points and so creates a rocky optimization process, even though the gradient descent will still be able to find desirable local minimums.

To combat some of these issues, various activation functions were developed that mimic the shape of the ReLU, but tackle the problems with ReLU itself, and lead ultimately to better performance of the network. Leaky ReLU, Swish, and SeLU are common replacements for ReLU and have their own strengths and weaknesses while being able to perform better than the ReLU itself in many cases. We mainly stick to ReLU in many of our experiments mainly for ease of use and known effectiveness, but other activation functions have also been experimented with and have been very effective.

Activation functions can make quite a difference in your network models and a careful choice of activation function at each layer may improve performance noticeably. The non-linearity is essential for networks to be able to fit complex data sets and give accurate predictions.

3.6.2 Batch Normalization

Batch normalization is a technique that can be applied to the layers of a deep neural network that prevents the instability that comes along with deep networks, and allows a much faster convergence to a local minimum. Batch normalization layers normalizes the outputs of each layer to speed up the training time, improve performance, and allow deep networks and reduce internal covariant shift; which is the main source of instability in deep networks characterized by the vast shift in the distribution of the network weights that is exacerbated by the depth of the network due to the number of updates involved without account of the important sequence in mind.

The batch normalization technique is applied in similar fashion as the normalization of features, except the normalization occurs at each layer, and the normalization itself is

updated throughout the training. Suppose we have a given batch of the dataset, B, of size m. Suppose that the batch is described by the matrix $\mathbf{x}$ with rows $\{\mathbf{x}^{(i)}\}_{i=1}^{m}$, in which there are a total of l features, we may describe the batch normalization procedure with Algorithm 2 displayed below.

Algorithm 2 Batch Normalization

1: **for** $k = 1, ..., l$ **do**
2: $\quad \mu_k \leftarrow \dfrac{1}{m}\sum_{i=1}^{m} x_k^{(i)}$
3: $\quad \sigma_k^2 \leftarrow \dfrac{1}{m}\sum_{i=1}^{m} \left(x_k^{(i)} - \mu_k\right)^2$
4: $\quad$ **for** $i = 1, ..., n$ **do**
5: $\quad\quad \widehat{x}_k^{(i)} \leftarrow \dfrac{x_k^{(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$
6: $\quad\quad z_k^{(i)} \leftarrow \gamma_k\, \widehat{x}_k^{(i)} + \beta_k$
7: **return z**

The parameters, μ_k and σ_k, are the mean and standard deviation respectively of the kth feature, ϵ is a small constant used for numerical stability, while γ_k and β_k are scaling values for the kth feature that can be updated throughout training if we choose to use this (the PyTorch default is to use this scaling). The output is a $m \times l$ matrix, $\mathbf{z}$, in which $\mathbf{z}^{(i)}$ is the ith row of the matrix representing the ith data point in the batch with its kth component denoted by $z_k^{(i)}$. There are some different ways to implement this transformation over the various batches. The simplest way is to calculate the batch statistics after each individual forward pass and use those, as is show in Algorithm 2. Another way is to average the batch statistics over each batch, i.e. at each step, t, we update the mean and standard deviation vectors, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ respectively by the update rules: $\boldsymbol{\mu}^{t+1} = \dfrac{t\,\boldsymbol{\mu}^t + \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)}}{t+1}$ and $\boldsymbol{\sigma}^{t+1} = \dfrac{t\,\boldsymbol{\sigma}^t + \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}^{(i)} - \boldsymbol{\mu}^t)^2}{t+1}$. The most popular way, and the default for PyTorch, is to use a momentum parameter to keep a running average over the batches as the training progresses, i.e. we choose a momentum parameter, α, such that at each step, t,

we update the mean and standard deviation vectors, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ respectively by the update rules: $\boldsymbol{\mu}^{t+1} = \alpha\,\boldsymbol{\mu}^{t} + (1-\alpha)\left[\frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)}\right]$ and $\boldsymbol{\sigma}^{t+1} = \alpha\,\boldsymbol{\sigma}^{t} + (1-\alpha)\left[\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}^{t}\right)^{2}\right]$.

Batch normalization provides many benefits to a neural network. Aside from improving the stability of training by reducing covariant shift, it has also been shown to reduce training time dramatically, and improve the performance in general due to regularization inherent in the batch normalization process, thereby reducing the need for other types of regularization. Batch normalization has created many more possibilities for the uses of deep networks due to these positive qualities.

3.6.3 Gradient Descent and Adaptive Learning Rates

There are three main types of gradient descent: batch, stochastic, and mini-batch gradient descent. Batch gradient descent uses all training samples at each epoch to calculate the gradient, which gives you the correct gradient, but usually is slower to converge, does not do well in exploring new local minimums, and sometimes is not feasible to compute if your data set is extremely large. Stochastic gradient descent takes one random sample at a time and calculates the gradient before moving to the next one until it reaches all samples, in which an epoch is completed. In this case, the training time is usually decreased, many local minimums are explored, and the gradient can be computed without any hardware considerations since there is only one sample at a time. The downfall of this technique is that the gradient descent will then be extremely unstable because the true gradient is not calculated, but only a small portion of it, though it will be statistically similar over many iterations. Mini-batch gradient descent is a wonderful middle ground between the two techniques, batch and stochastic descent, that addresses the hardware concerns, training time considerations, local minimum exploration, and gradient accuracy and stability. Mini-batch gradient descent takes batches of random samples from the whole data set of sizes greater than 1, commonly in multiples of 2 such as 16, 32, 64, 128, or 256; and descends along the gradient calculated for each batch. This gives us the

erratic movements that allow us to explore the loss surfaces' landscape, gives us more updates per epoch to decrease training time, avoids large batches that would might put a strain on hardware, and is close enough to the actual gradient so as to not introduce an uncomfortable level of instability.

To help with speeding up the process of gradient descent, we use adaptive learning rates in addition to the various batch techniques. Adaptive learning rates are learning rates that change throughout the training process in a way that helps us find local minimums quicker than if we would use a constant learning rate. Common techniques use notions of momentum, acceleration, and knowledge of previous gradients. Popular adaptive learning rate techniques include simple momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSProp, Adam, Nadam, and others. For an example, for N iterations, chosen learning rate, η, and loss function, J, a simple momentum adaptive learning rate algorithm saves the velocity, $\mathbf{v}$, at each step t, and updates the given model parameters, $\boldsymbol{\theta}$, as follows in Algorithm 3 below.

Algorithm 3 Descent with Momentum

1: $\mathbf{v} \leftarrow \mathbf{0}$
2: **for** $t = 1, ..., N$ **do**
3: $\quad \mathbf{v} \leftarrow \beta\,\mathbf{v} - \eta\,\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
4: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$
5: **return** $\boldsymbol{\theta}$

This provides an effective algorithm for stochastic gradient descent to drastically speed up training time. Another type of adaptive learning rate is AdaGrad. This uses a decaying learning rate based on the accumulation of squared gradients that leads the deterioration. The update rule is detailed in Algorithm 4.

Where the ϵ term is a smoothing term that avoids division by zero (usually set to around $\epsilon = 10^{-8}$) the $\oslash$ operator simply means component-wise division, i.e. the ith component is defined as $\left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{G}_i + \epsilon}\right)_i = \dfrac{\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_i}{\sqrt{\mathbf{G}_i + \epsilon}}$. This algorithm decays the

Algorithm 4 AdaGrad

1: $\mathbf{G} \leftarrow \mathbf{0}$
2: **for** $t = 1, ..., N$ **do**
3: $\quad \mathbf{G} \leftarrow \mathbf{G} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
4: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{G} + \epsilon}$
5: **return** $\boldsymbol{\theta}$

learning rate and helps move the updates more towards the global minimum by scaling down the gradient in the steepest dimension. The scaling helps us move faster toward our global minimum as we encounter less steep slopes, and also puts less emphasis on tuning the learning rate. This is why we would call this algorithm an adaptive learning rate algorithm. This algorithm was one of the first successful adaptive learning rate algorithms, but has one major flaw of a disappearing gradient, since the accumulation of the squares of the gradients in the denominator eventually stops the gradient descent entirely if it does not make it to the bottom in time.

RMSProp was designed to take care of the problem of the vanishing learning rate by choosing to use a moving average of the squares of the gradients. This then gives us the update rule, detailed in Algorithm 5.

Algorithm 5 RMSProp

1: $\mathbf{v} \leftarrow \mathbf{0}$
2: **for** $t = 1, ..., N$ **do**
3: $\quad \mathbf{v} \leftarrow \beta\, \mathbf{v} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
4: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{v} + \epsilon}$
5: **return** $\boldsymbol{\theta}$

This process avoids the accumulation of gradients because a moving average will have its old terms exponentially decay, so the benefits of the AdaGrad algorithm are present in the RMSProp algorithm without the penalty of the vanishing gradient. Next, we move on to the most popular adaptive learning rate algorithm, which is Adam. This combines the success of RMSProp with the idea of momentum. In this case, we have two values to tune, β_1 and β_2, that control the momentum and the moving average of the squared

gradients respectively. The Adam procedure is detailed below in Algorithm 6.

Algorithm 6 Adam

1: $\mathbf{v}_1 \leftarrow \mathbf{0}$
2: $\mathbf{v}_2 \leftarrow \mathbf{0}$
3: **for** $t = 1, ..., N$ **do**
4: $\quad \mathbf{v}_1 \leftarrow \beta_1 \, \mathbf{v}_1 - (1 - \beta_1) \, \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
5: $\quad \mathbf{v}_2 \leftarrow \beta_2 \, \mathbf{v} + (1 - \beta_2) \, \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
6: $\quad \widehat{\mathbf{v}}_1 \leftarrow \dfrac{\mathbf{v}_1}{1 - \beta_1^t}$
7: $\quad \widehat{\mathbf{v}}_2 \leftarrow \dfrac{\mathbf{v}_2}{1 - \beta_2^t}$
8: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \, \widehat{\mathbf{v}}_1 \oslash \sqrt{\widehat{\mathbf{v}}_2 + \epsilon}$
9: **return** $\boldsymbol{\theta}$

Now, we note that step 6 and 7 in the algorithm are unique and are used to get a boost in the beginning stages of the gradient descent. Since Adam combines the best qualities of each of these algorithms and has had much success in training neural networks, it is the most popular algorithm out there along with its variants. It is also used in most of our works involving neural networks.

3.6.4 Multi-task Networks and Transfer Learning

Most neural networks that are used are single task models where they are being trained to accomplish one particular task at a time. Knowledge in the human mind is acquired over time, and new tasks are initially approached with an intuition resulting from performing similar tasks. Some data sets that we analyze are very similar in nature, and it would seem that we could take advantage of this fact in some way using a machine learning technique. Two ways that we can apply this concept is to use a model that is trained to perform well on one task, then retrain it to perform well on a very similar task, or alternatively train it to perform well on multiple tasks simultaneously. These two techniques are transfer learning and multi-task learning respectively, which allow us to benefit from learning on multiple data sets. This is especially helpful if we have sparse access to data, or if we have

a large data set that we can use to bolster our performance on a task in which we have sparse data.

Transfer learning is a technique that uses a model that was already trained on one set of data, to then train the model on a second data set that is similar to the first one. The final weights of the first model are used to initialize the second bout of training, putting us in a position in the landscape of the cost function surface that is more beneficial than the random weight initialization that we would have without it. Transfer learning works best if there is a large data set to initially pre-train and the labels are similar enough so that the initial training provides a suitable set of initial weights.

The multi-task procedure is more involved than transfer learning and is done with a neural network architecture in the following way: If we let T be the number of tasks and $\{(\mathbf{x}_i^t, y_i^t)\}_{i=1}^{N_t}$ is the training data for the tth task where N_t is the number of samples of the tth task, $\mathbf{x}_i^t$ is the feature vector of the ith sample of the tth task, and y_i^t is the label for the ith sample of the tth task. The training strategy is to minimize the chosen loss function, L, for all tasks simultaneously. We define the loss for task t, L_t, below.

$$L_t = \sum_{i=1}^{N_t} L(y_i^t, f_2^t(f^s(f_1^t(\mathbf{x}_i^t; \{\mathbf{W}_1^t, \mathbf{b}_1^t\}); \{\mathbf{W}^s, \mathbf{b}^s\}); \{\mathbf{W}_2^t, \mathbf{b}_2^t\})), \tag{3.18}$$

In this equation, f_1^t and f_2^t the parts of the network that are the layers before and after the shared layers respectively which are parametrized by respective weights, $\mathbf{W}_1^t$, $\mathbf{W}_2^t$, and biases, $\mathbf{b}_1^t$, $\mathbf{b}_2^t$. f^s is the part of the network that has shared weights, $\mathbf{W}^s$, and biases, $\mathbf{b}^s$. To train all models simultaneously, we provide a single loss function to minimize so our problem is then:

$$\text{argmin} \left[\alpha \cdot ||\mathbf{W}^s||_2^2 + \frac{1}{T}\sum_{t=1}^{T}\left(L_t + \beta_t \cdot ||\mathbf{W}_1^t||_2^2 + \gamma_t \cdot ||\mathbf{W}_2^t||_2^2\right)\right], \tag{3.19}$$

$||\cdot||_2$ denotes the L_2 norm and α, the β_ts, and the γ_ts represent the regularization constants that set the magnitudes of the weight decay during training. In practice, if

weight decay is used, we generally set $\alpha = \beta_t = \gamma_t$ for all t. The structure of a general multi-task layer is illustrated in figure 3.5 below:

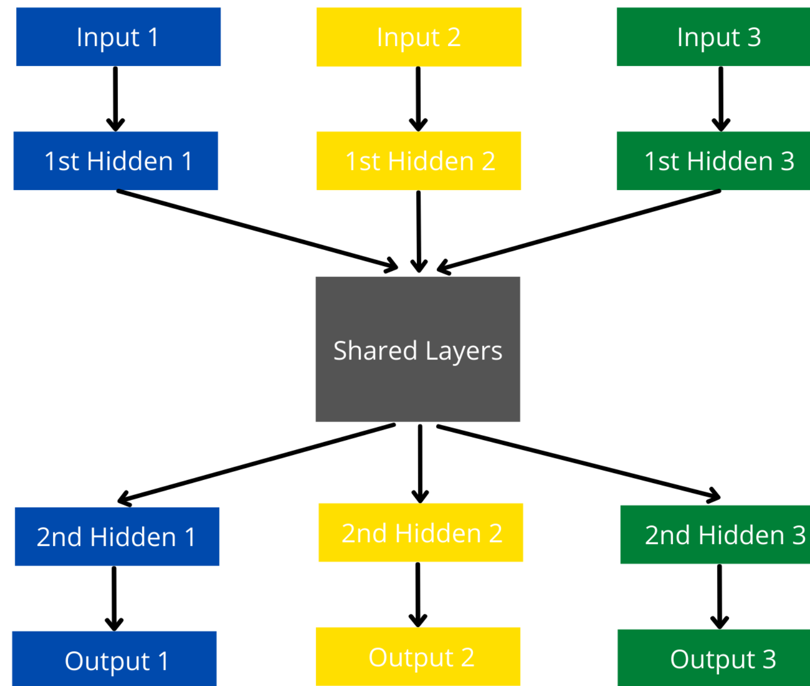


Figure 3.5: An illustration of a general multi-task network with three tasks. The input data from datasets 1, 2, and 3 are fed forward simultaneously, but through the hidden layers corresponding to their task. Then the last layers of each individual task's hidden layers are pushed through the shared layers. Finally, the data points are separated again according to their task, and pushed through their corresponding second set of hidden layers and outputs to get the predictions. The average loss of the predictions is calculated, and the weights are then updated simultaneously.

These two techniques are indispensable and in some cases necessary for us to be able to train effective models in many cases. It is not always easy to gather data, and so we must always look for ways to use the data that we already have, to then build better models from ones already constructed.

CHAPTER 4

BIOMOLECULAR MODELING AND KERNEL METHODS

4.1 Introduction

Biomolecules, particularly proteins and nucleic acids (DNA and RNA), are essential for all forms of life, so naturally this carries importance for us as we ourselves are living beings. For instance, proteins have a wide range of functions within living organisms: membrane channel transport, signal transduction, structural support for organisms, enzymatic catalysis for transcription and the cell cycle, etc. Nucleic acids function in association with proteins and are essential for encoding, transmitting, and expressing genetic information, which is stored in nucleic acid sequences and transmitted by transcription and translation processes. Understanding biomolecular structure, function, dynamics, and transportation mechanisms is paramount to understanding any biological system or organism. Understanding the protein-function relationship is so crucial to understanding protein specification, protein-protein interactions, and protein-drug binding, that are the key to solving problems in drug design and discovery.

Life science is one of the most prominent fields of natural science in the 21st century, as opportunities abound for depth of understanding and application to human health and technology. Molecular biology is the foundation of the biological sciences and biomolecular mechanisms, and as technology advances, the capabilities for analyzing biomolecular structures improves dramatically. Many traditional disciplines, such as epidemiology, neuroscience, zoology, physiology, and population biology are shifting from a macroscopic and phenomenological framework for analyzing to a molecular-based strategy. More generally, the field of biology is moving from a focus on qualitative and descriptive models to quantitative and predictive modeling. These changes over the decades have provided ample opportunity for the application of mathematics and machine-learning to

the biological sciences [108].

Biomolecular modeling is the science of studying the structure and function of molecules through modeling and computation. These computations include semi-empirical quantum mechanics, empirical molecular mechanics, Monte Carlo methods, free energy and solvation methods, quantitative structure/activity relationships (QSAR), biochemical databases and information, and various other procedures. Things like nuclear magnetic resonance (NMR) and X-ray crystallography are also categorized as a component of biomolecular modeling. Many times, our models may represent a very simplified form of the complex molecular structures that we study, but we may use these simplified models to glean important patterns and add insights that would otherwise not be seen in the full intricacy of a more complete model of the biological structure. The key is to develop models that are as simple as possible for us to understand, yet incorporate all of the appropriate elements that we require to understand all the components necessary to solve the problem at hand.

Most of the data concerning the structure and function of biomolecules and biomolecular functions has more recently come from experimental data collected through use of techniques such as X-ray crystallography, NMR, cryo-electron microscopy (cryo-EM) [103], electron paramagnetic resonance (EPR), multi-angle light scattering, and other techniques. The two techniques that have seen the most use are NMR and X-ray crystallography, and have been developed extensively over the past few decades. X-ray crystallography offers structural information at angstrom and even sub-angstrom precision, while the NMR technology provides structural information in a physiological context. X-ray crystallography and NMR are limited, however, in their application to proteasomes, subcellular structures, organelles, cells, tissues, and others. Cryo-EM has been able to do imaging in many of the areas mentioned that were not suitable for either NMR or X-ray crystallography, and many other techniques are also applied to deal with more specialized situations when they arise, allowing us to gather large amounts of biological data.

The rapid advances in experimental technology in recent history has led to massive stores of 3 dimensional biomolecular structural data and spurred large-scale development of bio-technology and pharmaceutical products. This 3-D structural information is applied in many tasks such as virtual screening of drugs, computer-aided drug design, binding pocket descriptors, quantitative structure-activity relationships (QSAR), protein design, RNA design, and molecular machine design, among other tasks. This substantial increase in molecular data has catalyzed the transition from the traditional qualitative approach to the quantitative and predictive approach that now pervades the biological sciences. The incredible growth of computer technology, which has now found application in every field of science, perhaps deserves the lions share of credit. Most quantitative endeavors would have little weight without the computational capacity of our modern computers, which made possible the advancement of biomolecular modeling and was able to bridge the gap between biological data and mathematical models [72, 27, 10, 11, 44, 56, 59, 9, 94, 6].

Studying structure-activity function in a biomolecular system is a major challenge in the field of computational biology. There are many quantities relating to bio-chemicals and biomolecular complexes that are of great interest to biologists, chemists, professionals in the pharmaceutical field, and other scientists. Quantitative and qualitative toxicity, solvation energy, solubility, protein-ligand binding affinity, and mutation effects of proteins are all various quantities associated with biomolecular systems that are important to know for a variety of purposes. FDA and EPA officials may wish to know whether a particular chemical is toxic to humans or to a surrounding environment, chemists or chemical engineers may be interested in the solubility of a range of biomolecules, pharmaceutical companies may be interested in the binding energy of a series of drugs in development to see if they should undergo further experimentation, and medical experts want to understand how the effects of a mutation in a protein affects its function. These are all common examples of the applications of biomolecular modeling that showcase the

importance of the field.

In many cases, biomolecular modeling involves creating models that can be used to predict the target quantities in question. Measuring these quantities may require tedious lab work which might be very time-consuming and expensive when considering a large set of chemical candidates. This may be prohibitive to small-scale companies and universities with a smaller budget. In the cases of larger organizations that may be able to afford the incredible costs and hire the specialized personnel that are required, there is always an incentive to reduce costs, bypass stringent laboratory regulations, and avoid any ethical concerns through use of in-silico methods. This can decrease waste significantly, provide cheaper options for consumers, and spur more innovation in the pharmaceutical and medical fields. One can think of a multitude of opportunities that effective predictive models can bring to provide serious benefits to our society.

The simplest geometric model that has been proposed is the space-filling Corey Pauling-Koltun (CPK) theory, which represents each atom as a solid sphere with Van der Waals(VDW) radius, in which the surface is referred to as the Van der Waals surface. Solvent accessible surfaces (SAS) and solvent-excluded surfaces (SES) were then introduced to create smooth modeling surfaces, where SES has been applied in the cases of protein folding, protein surface topography, protein-protein interactions, DNA binding, DNA bending, docking, enzyme catalysis, drug classification, and solvation energy prediction. The SES model is also used in dynamic simulations and understanding ion channel transports. SAS and SES, however, are not differentiable due to the cusps and tips that are prevalent on their surfaces. Gaussian surfaces were proposed to deal with this problem, which introduces a smooth Gaussian function to remove any singularities from the surface. With this fix, many analytical techniques were applied, such as differential geometry, that had found much success. Other basic physical properties of biomolecules are also applied to bolster the geometric information, including electrostatics, lipophilicity, elasticity, and some properties derived from molecular dynamics. These geometric models

have provided excellent predictions of the solvation free energy, protein-ligand binding affinities, protein mutation energy changes, and protein-protein interaction hotspots of various biomolecules and biomolecular complexes.

More advanced models have been constructed using advanced theoretical concepts in physics. Theoretical modeling of structure-function relationships generally based on fundamental laws of physics, such as quantum mechanics (QM), molecular mechanics (MM), continuum mechanics, statistical mechanics, thermodynamics, among other subfields of physics. QM methods are great for understanding chemical reactions and protein degradation, while MD is useful for understanding molecular conformational landscapes and collective motion/fluctuations. MD is at the forefront of theoretical biomolecular modeling and simulations, however, the modeling of large-scale biomolecules or biomolecular complexes can introduce an excessive amount of degrees of freedom involved, and can easily become prohibitively expensive in it's computational cost. For example, in some instances it might even take months to develop a computer simulation of protein folding, and usually is a poor comparison to the base truth.

There are many ways to create predictive models involving high-level physical and mathematical methods that flesh out the details of the biological system, but also there are more crude methods may be used if they can provide sufficient predictions for the problem at hand. More recently, in-silico methods are used that involve the application of machine learning algorithms to create accurate models that can predict all sorts of important physical quantities without any experimentation. Many advanced mathematical techniques have been used to generate molecular representations that have been very useful in training a number of machine learning models, from simple regression and other basic models with few hyper-parameters, to the more complicated deep-learning models with a wide range of hyper-parameters that require extensive tuning. These representations, which involve the use of many different fields in mathematics, have created extremely successful models which have provided us with state-of-the-art predictive ca-

pabilities. Algebraic topology, differential geometry, graph theory, and other fields have all been applied to model the complex interactions involved in these biomolecular systems.

These three fields; geometry, topology, and graph theory have been essential for the understanding of biomolecular function, dynamics, and transport. Geometry provides structural and surface representation, which helps to understand molecular structure and internal and external interactions between or within molecules. The application of geometry-based mathematics connects the physical molecular structure to theoretical mathematical models. Topology helps us understand atomic critical points, connectivity, and information about important topological invariants that represent physical details, such as independent components, rings (or pockets), and cavities, where the notion of the "size" of these components can be captured. Graph theory analyzes biomolecular interactions and reveals structure function relationship. Aside from those three mathematical fields that we discussed, there are many other tools that can be used. There are two main categories that we can place them in, discrete and continuous. Discrete approaches include graph theory, Gaussian network models, anisotropic network models, normal mode analysis, quasi-harmonic analysis, flexibility-rigidity index (FRI), molecular nonlinear dynamics, spectral graph theory, and persistent homology. The continuous approaches include discrete to continuum mapping, high dimensional persistent homology, biomolecular geometric modeling, differential geometric theory of surfaces, curvature evaluation, variational derivation of minimal molecular surfaces, atoms in molecular theory and quantum chemical topology.

In many of our works, we use kernel functions that measure similarity between two points on a continuous scale from 0 to 1 to obtain features for our molecular representations. These kernel functions are applied in feature extraction because they provide a normalization of the features in our representation, they provide an important aspect of non-linearity to the feature generation which adds complexity to the representation,

and they allow for further parameter tuning, which gives us the ability to refine our features in a simple way. The shapes of these kernel functions are based on parameters that determine the inclusiveness of the similarity and also the sharpness of the decay of the similarity measure at a point of cutoff. As stated earlier, the parameters that determine the shape of the function can be naturally tuned, allowing a greater chance for more successful models to be found. The above concepts of flexibility-rigidity index (FRI), homology, spectral graph theory, and differential geometry all have versions in which we have applied kernels to abstract the features and allow additional tuning.

Why should one care for the application of abstract topics mathematical techniques to model these biological situations? Given the abstract nature of the models, we will usually not be able to determine a natural explanation about each piece of the model, but we may speculate about the proximate reasons for their success. For example, features generated by the spectrum of Laplacian matrices may not give us an immediate understanding of what they may mean to a particular molecular system, but there may be hidden attributes, such as information about the connectivity of the molecule when viewed as a graph, that might prove useful in the context of the solubility of a molecule. One important observation is that in larger and more complex biomolecular systems, we find that the more precise physical models that give us predictive models do not scale well. As the system complexity increases, the physical calculations necessary to get appropriate accuracy are computationally infeasible. When abstracting the properties of the molecular systems, the fine details are not extracted, but a more general qualitative description of the system and its workings are presented. Important information is gathered, while omitting unnecessary detail. For example, we may obtain topological, geometric, and spectral data from a molecule, representing the system as a whole with the intrinsic physics and geometry necessary through abstraction.

In this chapter, we will be exploring simple concepts in biomolecular modeling, and then dive deeper into more advanced topics. The application of kernel functions, graph

representations of molecules, and specific mathematical modeling methods will be covered.

4.2 Physical Modeling

Physics provides us with many tools to analyze physical systems in a concrete mathematical way. The mathematical models that have been experimentally proven to be true are indispensable to us in a variety of contexts. Our focus is specifically on the use of physical modeling in biology and bio-chemistry. In this section, we will cover some basic physical notions that are pertinent for understanding biomolecular systems.

4.2.1 Electrostatic Interaction

The electrostatics of pairwise interactions between molecules in a homogeneous medium is frequently modeled by Coulomb's law, which measures the electrostatic force between two models by taking into account the distance and partial charges of the atoms. If we let the distance between two atoms, i and j, be denoted by d_{ij} and the pair, q_i and q_j, be the charges of atom i and atom j respectively, then the electrostatic force between the two atoms can be described by the equation:

$$F_{clb} = k_e \frac{q_i q_j}{d_{ij}^2}, \tag{4.1}$$

where k_e is coulomb's constant. This is important because it can tell us where there might be attraction or repulsion between atoms or molecules, which is incredibly useful if we are trying to model how a system of molecules will react with one another.

4.2.2 Van der Waals Interaction

Van der Waals forces are weak forces between two atoms or molecules induced by electrical interactions. They are the weakest of the main intermolecular forces, such as hydrogen bonds, hydrophobic interactions, and ionic interactions, but they can be very strong if the

forces accumulate from multiple sources. There are two main types: the weak London Dispersion forces and the stronger dipole-dipole forces. The chance that an electron is in a certain area is determined by a probability cloud, and so the density of electrons within a molecule or atom will be changing at all times. The tendency for electrons to concentrate themselves to one side of the molecule or atom and momentarily change the status of its polarity to subsequently exert a force on other molecules or atoms are generally referred to as the London dispersion forces. Dipole-dipole interactions are interactions that result simply from the natural polarity of a molecule.

The Lennard-Jones potential models the Van der Waal interaction energy, E_{ij} between the ith and j^{th} atoms in a system. This combines two forces: one attractive force that comes from the London dispersion forces, and, one shorter range repulsive force that comes from the Pauli exclusion principle. This mathematical model is given by the equation:

$$E_{ij} = \epsilon_{ij}\left(\left(\frac{r_i + r_j}{d_{ij}}\right)^6 - 2\left(\frac{r_i + r_j}{d_{ij}}\right)^{12}\right), \tag{4.2}$$

where r_i and r_j are the Van der Waal radii of the atoms i and j respectively, d_{ij} is the distance between them, and ϵ is the depth of the potential well, which is given based on the pair of atoms.

4.2.3 Molecular Surface Area

The solvent-accessible surface area is the area of a molecule that is accessible by a solvent or other type of molecule, usually measured in Angstroms (Å). This surface are is usually calculated using the rolling ball method, which uses a sphere of a particular radius to probe the surface of the molecule [95]. This process is illustrated in figure 4.1 below.

There are other algorithms that can calculate this surface, such as the LCPO method and the power diagram method, which are much faster than the rolling ball algorithm [109, 7]. The solvent accessible surface area is used in calculating the transfer free energy,

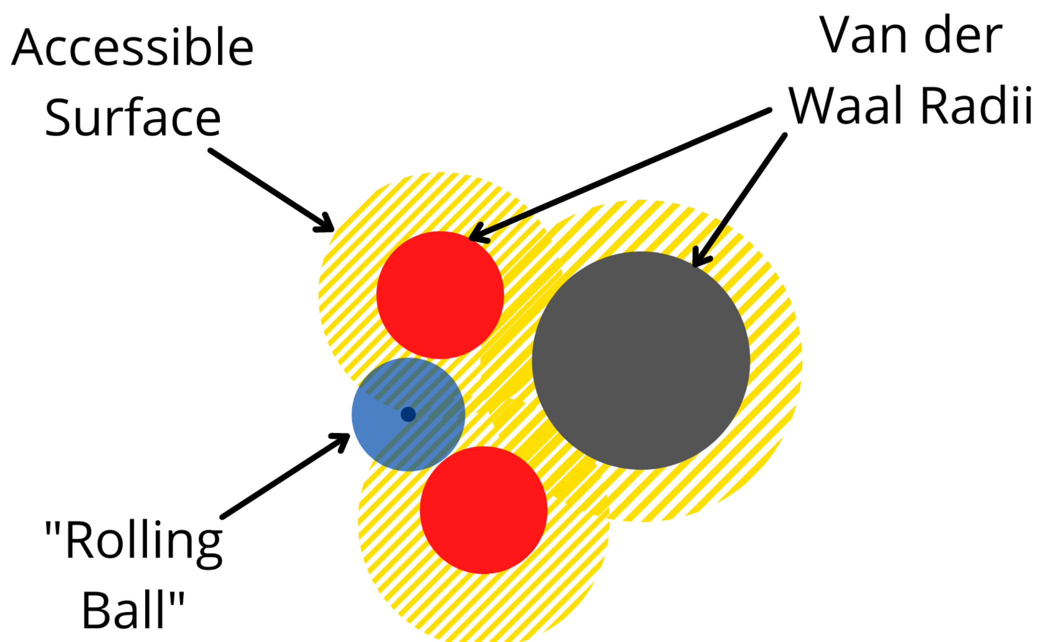


Figure 4.1: This figure illustrates the solvent accessible surface compared to the Van der Waal surface. The solid red and gray colors represent the Van der Waals radii of the atoms in the molecule, and the striped yellow area represents the solvent accessible surface that is "rolled out" by the center of the translucent blue probe sphere (or solvent).

molecular dynamics simulations, and it has been recently suggested the surface area can even be used to improve the predictions of protein secondary structures [76, 8].

4.3 Biomolecular Datasets of Small Molecules

Small molecules play an important role in all biological processes, mainly through interactions with larger biomolecules such as proteins and nucleic acids. They bind with proteins as parts of complex chemical reactions and processes that are part of larger processes important for the preservation of biological lifeforms. Small molecules often act as triggering signals or inhibitors for these processes, and in fact, many of these small molecules are prepared as drugs, alleviating problems in the human body and for animals as well. Being able to accurately describe the effects of these small molecules on

larger biological complexes is of utmost importance to drug design and the understanding of this is attractive to pharmaceutical companies and medical professionals.

In this section, we want to explore some problems involving small molecules and the important physical quantities that are associated with them. We cover toxicity, solubility, and the prediction of solvation and hydration free energy. Data sets of this type will be explored in later chapters.

4.3.1 Toxicity

Toxicity is a measure of the degree to which a chemical can harm an organism. The harmful effects can be measured qualitatively or quantitatively. A qualitative approach will categorizes chemicals as toxic or nontoxic, while quantitative toxicity data records the minimal amount of a substance that would produce lethal effects. Experimental measurement of toxicity can be expensive, time-consuming, and subject to ethical constraints. Machine learning models can alleviate all of these concerns, and so are extremely useful in toxicity prediction. The working principal of the machine learning approach for toxicity analysis is that molecules with similar structures have similar activities, which is called the quantitative structure-activity relationship (QSAR) approach. By analyzing the relationship between molecular structures, one can predict their biological functions.

The toxicity of a small molecule is a fundamental concern for drug design, simply for the fact that a drug is generally designed to alleviate or heal problems arising in the human body, and so toxic drugs are naturally avoided. Toxicity analysis is also useful for us for environmental concerns. If applying a chemical to our environment for purposes of protection, we may wish to know how this will affect the area in which it is applied. Environmental damage may be cause concern for number of reasons, and the government agencies like the EPA might have a great interest in understanding the level of toxicity for various chemicals. There are so many different organisms that may be affected by different chemicals, and the complexity of many organisms can present many challenges to

creating predictive models, however, we have seen much success in our group nonetheless [110, 45, 81].

4.3.2 Solubility

Solubility is the ability of a chemical to form a solution with another chemical, in which the chemicals are referred to the solute and the solvent respectively. If a solute cannot form a solution with a solvent, the the solute is said to be insoluble in the solvent. The solubility of a molecule is generally measured in terms of the concentration of the solute in a saturated solution, which is a solution in which no more solute can be dissolved, where the solute and solvent are said to be in equilibrium. Quantifying and predicting the solubility of a solute has many important applications in industry and sciences besides chemistry, such as geology, biology, engineering, medicine, agriculture, paint production, brewing, and many other fields. The most common solvent used in industrial processes and scientific experimentation is water, and so understanding how a solute behaves in water is of utmost importance.

In drug design, pharmaceutical professionals are interested in understanding the solubility of a drug candidate that could occur during the delivery of a drug to the body. A drug must have some level of aqueous solubility (solubility in water) for it to properly be absorbed into systemic circulation. There are many factors that affect the solubility of a drug. Some of these are the shape, temperature, pH, and crystal characteristics. As far as shape is concerned, we have tools to create representations revealing the geometric properties of solutes and methods for implementing them in machine learning models to great effect. The QSAR approach that we take in toxicity may also be fruitful in determining solubility.

4.3.3 Solvation Free Energy and Hydration

Solvation describes the interaction of a solvent with dissolved molecules, in which the strength of the interaction from both ionized and uncharged molecules are measured. The strength of these interactions influence many properties of the solute, which involve solubility and reactivity, while also influencing properties of the solvent, such as viscosity and density [12]. Solvation is the process of reorganizing the molecules of the solvent and solute into solvation complexes, which are formed by having molecules of the solvent form shells around ions in the solution. Solvation involves physical interactions and processes such as bond formation, hydrogen bonding, and Van der Waals forces. A special case, which is common enough to give it a name, is the solvation of a solute in water. This solvation process is called hydration.

Given the physical processes involved, we would like to look for representations that can encode the intrinsic physics of the system. Solvent polarity is perhaps the most important factor in solvation analysis. Polar solvents have molecular dipoles, and are more susceptible to forming the solvent shells. When modeling, we must include partial charges and polar information. The quantity that we want to measure in solvation analysis is the solvation free energy. This is the change in the Gibbs free energy of the system, which is a thermodynamic quantity, measured in joules, that is given by the change in enthalpy minus the temperature times the change in entropy, i.e. $\Delta G = \Delta H - T\Delta S$. Solvation free energy is a very important quantity in solvation analysis, which can be very beneficial in studying other complex or biological processes [63, 74, 105, 55].

4.4 Basic Kernel Functions

A kernel function in machine learning usually refers to a function that is used to map data points in the feature space into another feature space so that the data becomes linearly separable, or at least more separable than before. These functions are based on a measure of similarity and are usually limited between 0 and 1, where the closer the dis-

tance, the more similar the two data points are. More precisely, for a kernel function, $\Phi : \mathbb{R}_{\geq 0} \rightarrow [0, 1]$, we have the following conditions:

$$\Phi(x) \rightarrow 0 \text{ as } x \rightarrow \infty$$

$$\Phi(x) \rightarrow 1 \text{ as } x \rightarrow 0$$

$$\Phi(x) \leq \Phi(y) \text{ for } x \leq y$$

These conditions help us choose functions that can allow us to more deeply consider the similarity of data points that are much closer together in the feature space and eliminate the influence of data points that are much further away. This gives us a more accurate categorization or grouping of the data points that would otherwise frustrate our models' attempts to fit the data nicely. In essence, the kernel function many times acts as a low-pass filter, which has a soft cutoff centered at a parameter, η, and quality control parameter, κ, that determines the sharpness of the cutoff. There are two kernel functions that are very popular, and that have been used extensively in research coming from our group. These kernels are the Lorentz and Gaussian kernels:

$$\Phi_L(x; \eta, \kappa) = \frac{1}{1 + (x/\eta)^\kappa}$$
$$\Phi_E(x; \eta, \kappa) = e^{-\left(\frac{x}{\eta}\right)^\kappa}$$

They have both shown to be great kernel functions in practice and have aided in the creation of many successful models, with multi-kernel methods having the greatest success. Another special function that has been tested briefly and performed well is the piece-wise linear function that is defined as:

$$\Phi_{plin}(x; \eta, s) = \begin{cases} 1 - s \cdot \frac{x}{\eta} \ : \ x \leq \eta \\ s \cdot \left(2 - \frac{x}{\eta}\right) \ : \ x > \eta \end{cases}$$

Where we usually set $s = 0.5 \cdot e^{-\kappa^2}$ so that there are no values that cut the graph off from what it should look like, since the s value should be in between 0 and 0.5. Then κ can be adjusted without worry. This kernel gives us a very sharp cutoff and allows us to adjust the contribution near the cutoff point while totally eliminating the contribution of very far off points.

In our experiments we use the kernel functions for different applications. We us the kernel functions to obtain features. These functions are usually applied to atomic distances to get interaction strength and electrostatics information. They work as low-pass filters that put more importance on atoms that are closer and drowns out the interactions of further off atoms that should not provide much to the overall function of the system.

4.5 Kernel Methods applied to biology

4.5.1 Element-specific Groups

The concept of an element-specific group is a key concept in many papers published by our research group. They are a way to group the elements of an atom to provide multiple avenues for feature extraction by providing additional relevant information about the internal or external interactions of the biomolecules of a system. Different elements interact with each other in different ways and so analyzing the restricted sets of atoms defined by element pairs can give important insights for your machine learning models. This has been a powerful tool for feature extraction and has been responsible for many successful models.

Being more specific, if we wish to choose element-specific groups to analyze, we must choose a set of elements that will determine our groups. If we are looking at single molecules, we will generally be analyzing the pairs within the molecules themselves. If we are dealing with a biomolecular complex, usually of a small molecule and a large protein, the element pairs will refer to the atoms of one type in the first biomolecule and the atoms of the element type in the second biomolecule. If there are more than two

biomolecules, then we may begin to look at triples, quadruples, etc. In this work, we will focus on the case of pairs of elements, so we choose two sets of elements, $\mathcal{T}_1$ and $\mathcal{T}_2$, that contain element types; i.e. $\mathcal{T}_1, \mathcal{T}_2 = \{\text{H, C, N, O, S, P, F, Cl, Br, I}\}$ in the case of the internal element pair interactions of small molecules, or $\mathcal{T}_1 = \{\text{C, N, O, S}\}$ and $\mathcal{T}_2 = \{\text{H, C, N, O, S, P, F, Cl, Br, I}\}$ for the external interactions between a protein and a ligand. The choices of the element type sets are based on many factors, but commonly hydrogen atoms are withheld from large molecules because of the unreliable nature of the 3-dimensional positions that are obtained through biomolecular software when applied to things like proteins, and also other elements are withheld because they do not appear regularly in molecules and so do not add any relevant information to the mix.

4.5.2 Element-specific Graph Representations

From the previous section, we discussed the concept of element specific groups consisting of atoms with particular element types. One useful way of organizing the structure of a molecule of a biomolecular system is to place this information into a nice graph structure. We can naturally obtain a set of vertices by allowing the atoms in the molecule or biomolecular complex to be a set of vertices. To get enough detail, we wish to consider a vertex-labeled and weighted graph whose vertices are labeled by element type of the atoms and possibly other information, and the edges are weighted by the distance between the atoms, or by a measure of their closeness as judged by a chosen kernel function. This gives us a complete graph which might be quite large in most applications. By restricting ourselves to the subgraphs that consist of the element pairs of our choosing, we may then extract the useful features that we desire.

To make pairs for element specific groups, we combine the two sets of element types, $\mathcal{T}_1$ and $\mathcal{T}_2$, into a set of pairs: $\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2$. We then describe each atom in the biomolecular complex as a pair itself of a position and an element type, $(\mathbf{r}, t) \in \mathbb{R}^3 \times \mathcal{T}_1 \cup \mathcal{T}_2$, and let $\mathcal{V}$ be the vertex set which contains all the atoms in the system. It may also be the case that

we wish to include another descriptor along with each atom, such as the partial charge, so in some cases we may look at atoms as triples, i.e. $(\mathbf{r}, t, c) \in \mathbb{R}^3 \times \mathcal{T}_1 \cup \mathcal{T}_2 \times \mathbb{R}$. This provides us with additional features to be extracted per element-specific group. The edge set, $\mathcal{E}$, is the complete graph that connects all the vertices by an edge. In fact, we define $\mathcal{E}$ to be the weighted graph, $\mathcal{E} = \left\{ \left(\big((\mathbf{r}_1, t_1), (\mathbf{r}_2, t_2)\big),\ d \right) \in \mathcal{V} \times \mathbb{R}_{\geq 0}\ :\ d = \Phi(||\mathbf{r}_1 - \mathbf{r}_2||) \right\}$, for some function Φ, which can either be the euclidean distance, or can be a chosen kernel function.

Now for each element pair, $\mathcal{P} = (T_1, T_2) \in \mathcal{T}$, we choose all the vertices that have the element types of either pair, so our vertex set for the element-specific group is $\mathcal{V}_{\mathcal{P}} = \{(\mathbf{r}, t) \in \mathcal{V} : t = t_1 \text{ or } t = t_2\}$. Then we define the element specific edge set as follows: $\mathcal{E}_{\mathcal{P}} = \{\big((\mathbf{r}_1, t_1), (\mathbf{r}_2, t_2)\big) \in \mathcal{E}\ :\ (t_1, t_2) = \mathcal{P}\}$. This then gives us a complete description of the element-specific subgraph, $\mathcal{G}_{\mathcal{P}} = \big(\mathcal{V}_{\mathcal{P}}, \mathcal{E}_{\mathcal{P}}\big)$.

4.5.3 Geometric Graphs

Our first example of kernel methods includes the use of the element-specific subgraphs as described above, in which we extract features for each subgraph by summing the weights of the edges with other possible statistics. An example of this process was described in a paper by Nguyen et al. [78]. To be more precise, we start with an element specific subgraph, G, within our molecule or biomolecular complex. Then we have a choice of kernel function, Φ, which is usually the Lorentz or the exponential kernel, and has chosen parameters, η and κ. Then, we obtain a value, μ_i at each atom i that represents the strength of the interactions between all the other, in which covalent bonds are usually excluded:

$$\mu_i = \sum_{j=1}^{d} w_j \phi(||\mathbf{x}_i - \mathbf{x}_j||; \eta, \kappa), \tag{4.3}$$

where d is the number of edges in the element specific graph that is connected to i, $\mathbf{x_i}$ and $\mathbf{x}_j$ are the coordinates of the atoms i and j respectively, and the w_js are either all set to the value 1, or set to be a physical quantity related to the atom, j. The value μ_i can

be referred to as the rigidity index at the atom i. To get a feature from the group, we can simply take the sum of all the rigidity indices at each atom in the subgraph, i.e.

$$RI_G = \sum_{i=1}^{N} \mu_i, \tag{4.4}$$

where N is the number of atoms in the subgraph, G. Now, if we wish to get additional features from the group, we may consider taking various statistics of the μ_is such as the maximum, minimum, standard deviation, mean, etc. For every element-specific group, we can obtain one or more features and combine them together to describe each data point in our data set.

4.5.4 Algebraic Graphs

Another way we can generate molecular descriptors is through analyzing the graph structure by understanding the spectral decomposition of their adjacency and Laplacian matrices, as was done in Nguyen et al. [80]. Concerning the case of an element specific subgraph, G, as described before with a chosen kernel function, Φ, we may obtain the weighted Laplacian matrix, L_Φ. We obtain the spectrum for the Laplacian, $\lambda_1, \lambda_2, ..., \lambda_n$, with eigenvectors, $\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n$. Then we can form atom-specific features for each atom, i:

$$\mu_i^L = \sum_{l=1}^{n} \frac{1}{\lambda_l} [\mathbf{u}_l \mathbf{u}_l^T]_{ii} \tag{4.5}$$

As before, we get the element specific descriptor by summing the μ_i values:

$$RI_G^L = \sum_{i=1}^{N} \mu_i^L \tag{4.6}$$

Additional statistics can also be gathered as before. For additional molecular descriptors, we may wish to draw features form another graph, one that takes edges based on the

covalent bonds between the atoms. This may reveal more about the structural properties of the molecule since the physical structure is encoded in the graph.

4.5.5 Algebraic Topology

Persistent homology can be applied to molecular structures to gain an understanding of the topological structure of a molecule or a biomolecular complex. We have seen much success in its application and in some cases, a kernel was applied to provide opportunity for parametrization and optimization of those parameters to construct more refined models [22, 110]. We want to consider the case of small molecules in which for each molecule, we may obtain a point cloud from the positions of the atoms in the molecule. The point cloud from a molecule then can yield persistence values based on the the distance matrix. The Vietoris-Rips, Čech complex, or the alpha complex may then be used to produce a filtration.

The births and deaths of the homology classes define a sort of barcode representation of the molecules. This representation is illustrated in figure 4.2 below from a paper by Zixuan et al [22].

Since we want to apply kernels to our features, we want to define the correlation between two atoms, i and j, with positions x_i and x_j respectively, by the formula $1 - \Phi(||x_i - x_j||)$. This then gives us an alternative to the distance between two atoms by looking at the correlation value defined instead, giving us a modified distance matrix. We then apply the same method as described above to get the barcode representation.

From these barcode representations, i.e. persistence information, we can extract topological features to use in machine learning models. We start by choosing a maximum length, L, and a resolution value, m. Then, for a given dimension, p, we may define 3 feature vectors which describe the barcode data, that is the births, deaths, and persistence vectors. The main technique is to split the maximum length into a number of bins, $N = L/m$, in which bin i is the bin that corresponds to the interval, $\frac{(i-1)m}{L}$ to $\frac{im}{L}$. We

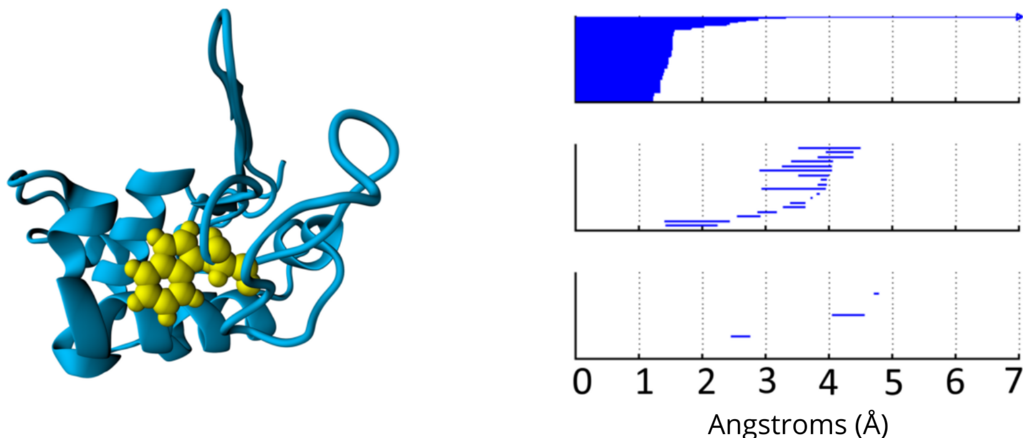


Figure 4.2: This figure shows a barcode representation being extracted from a biomolecular structure. On the left, we have the biomolecule in which the point cloud is taken by looking at the positions of all the atoms in the structure. On the right, we have the persistence information for the first 3 dimensions. The 0-dimensional persistence (points) for each class is on top, the 1-dimensional persistence (rings) is in the middle, and the 2-dimensional persistence (cavities) is on the bottom. The persistence is measure in Angtroms (Å) [22].

then place homology classes (i.e. barcodes) into the bins and record the number of items placed in a bin to get a feature vector. For births and deaths, we place a homology class in bin i if the homology classes is born or died respectively in the length corresponding to its bin. We can alternatively state this as placing a barcode in a bin if its tail is contained in the interval corresponding to the bin (birth), or if its head is contained in the interval corresponding to the bin (death). The persistence feature vector is calculated in a slightly different way in that a homology class (or barcode) is placed into every bin whose length is entirely included within the whole stretch from birth to death of that class (or alternatively, the bins in which the barcode covers their interval entirely). This process is illustrated in figure 4.3.

For each dimension then, we obtain 3 sets of $N = L/m$ (number of bins) features. If we use 3 dimensions, then we get 3 sets of 3 feature vectors, for a total of $9L/m$ features. This can be a lot, since we would usually want to set the length to be enough to capture all of the atoms, and the resolution to be set to something that is certainly smaller than 1, but not so small that the feature vectors are too sparse. Generally, the length will be set

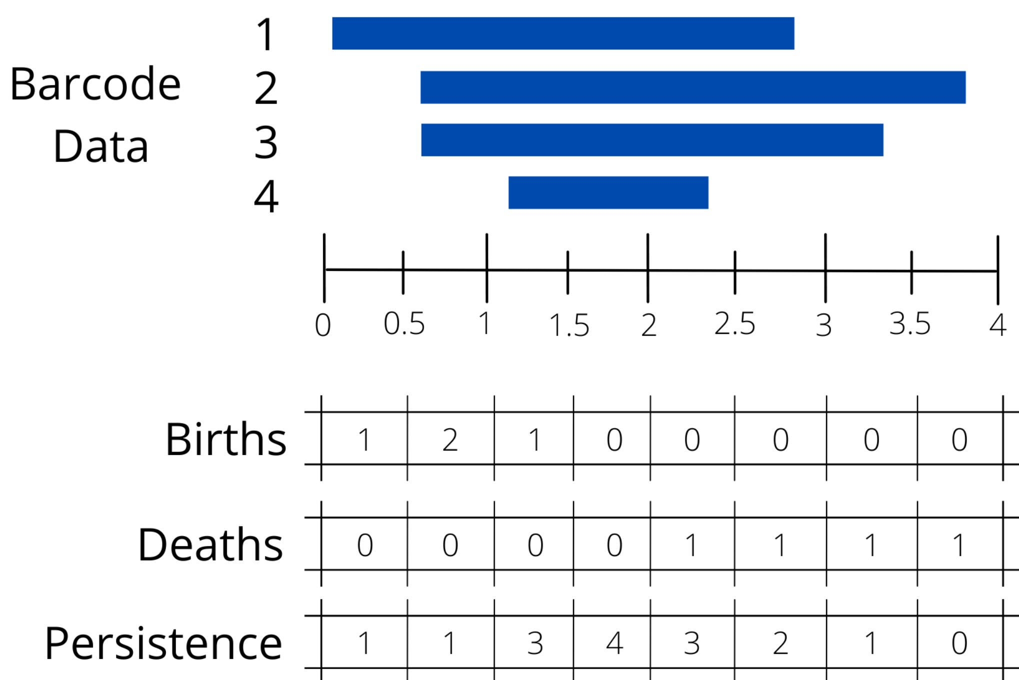


Figure 4.3: This figure depicts the featurization of the barcode data from persistent homology calculations. On the top, the barcode representation is shown, with 4 bars representing the "lives" of 4 different homology classes as a simplicial complex "evolves" through a filtration. Below the barcode data, the birth, death, and persistence feature vectors are constructed using a max length of 4 and a resolution of 0.5.

to 10 or lower for small molecules, and up to 30 for large bio-molcules. The resolution is usually set to 0.2 or 0.25, but should probably not be set to anything below 0.1. Now, we might be willing to look at element-specific topological features and obtain many more features, given the amount of element specific groups that we can usually analyze, but we may lose some physical meaning if we do this since the intuition of these homological features is to measure components, rings, and cavities, etc. It has been, however, shown to be fruitful in either case [22].

4.6 Multi-scale Kernel Methods

Multi-scale kernel methods refer to methods involving the use of multiple kernels functions or parameters to generate additional features. For instance, we can generate representations based on a choice of kernel, such as Lorentz and exponential kernel, each

with a choice of η and κ parameters. Let us call these representations, $R(L, \eta_1, \kappa_1)$ and $R(E, \eta_2, \kappa_2)$. Then, we can simply combine the two representations by concatenating them to get a new representation, $R(L, \eta_1, \kappa_1) : R(E, \eta_2, \kappa_2)$. This can be extended for an arbitrary number of kernel choices and parameters, i.e. for a sequence of kernel types, $(K_1, ..., K_n)$, and parameter choices, $(\eta_1, \eta_2, ..., \eta_n)$ and $(\kappa_1, \kappa_2, ..., \kappa_n)$, we can get a multi-scale representation of their concatenated representations, $R(K_1, \eta_1, \kappa_1) : R(K_2, \eta_2, \kappa_2) : \cdots : R(K_n, \eta_n, \kappa_n)$. Each kernel type has its own benefits and drawbacks, and the different parameters give the kernels various shapes which might pick up unique features, so the combination of these different kernel types and parameters gives us an edge when building models. We have seen multi-scale models showing great success when applied [78, 80, 81].

CHAPTER 5

AWEGNN: AUTO-PARAMETRIZED WEIGHTED ELEMENT-SPECIFIC GRAPH NEURAL NETWORKS FOR SMALL MOLECULES

This chapter includes much of the AweGNN paper by Szocinski et al. [100], and should be read as a self-contained work.

5.1 Introduction

Automated feature extraction techniques, like those used in convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have been very successful in deep learning applications [53, 92]. They have made inroads in a variety of fields now, producing state-of-the-art results in signal and information processing fields [30], such as speech recognition, image recognition [25], and natural language processing [99]. These kinds of automated feature extraction techniques, however, are only highly successful on data that is relatively simple in structure, such as with images, text, etc. For data sets with complex internal structures, such as molecules or macro-molecules, hand-crafted descriptors, or representations, are indispensable for developing top-quality predictive models [79].

There are many physical, biological and man-made objects that have intricate internal structures. For example, proteins, chromosomes, human bodies, and cities have very complex structures. Tools from abstract mathematics such as algebraic topology, differential geometry, and combinatorics can be utilized to simplify the structural complexities of data. [79, 69, 107]. For molecules and macro-molecules, molecular fingerprint representations obtained from persistent homology, curvature analysis of surface electrostatic potentials, and eigenvalues of weighted adjacency matrices derived from atomic distances have all be used for this endeavor [79]. From the point view of machine learning, molecular representations which include detailed chemical and physical information can be extremely high-dimensional, especially when the biomolecules in question are made of

thousands of atoms, such as in the case of protein-ligand complexes [19, 21, 26, 13]. By using lower dimensional mathematical representations to train simple machine learning models, such as gradient-boosting trees (GBT) and random forest (RF) models, one can achieve stellar performance.

Some of our hand-crafted mathematical features are somewhat limited in their scope [110], in which we mean they are generated by a fixed procedure, with little possibility for making fine adjustments to produce more favorable representations. Other mathematical features that we have used are based on a choice of kernel function or functions along with adjustable parameters, in which a careful tuning of these parameters can deliver optimal performance [78, 80]. This process of tuning, however, can be very time-consuming and require experience. Although the kernel parameters introduce new dimensions to enhance our models, the cost of tweaking becomes tedious when more than a small handful are introduced.

In this work, motivated by the automation of simple feature-extraction techniques as in CNNs, we seek to automate the parameter optimization for our mathematical representations. Specifically, we automate the selection process of kernel parameters either partially or fully, meaning that we would like to have some or all of the parameters automatically chosen for us during training. Neural networks are trained in a series of epochs where the weights of the network are updated at each step through gradient descent, and so they are a perfect starting point for exploring this idea of automated parameter selection. In our approach, back propagation is extended further to include the kernel parameters of our molecular representations, leading to the simultaneous update of our kernel parameters and neural weights at each batch of training. To this end, we introduce auto-parametrized weighted element-specific graph neural networks (AweGNNs). The AweGNN implements the above scheme of parameter updates by using a representation based on kernel-weighted geometric subgraphs, with features similar to the correlation functions of flexibility-rigidity index (FRI) used in a previous work[112, 78]. The repre-

sentations resulting from updating the kernel parameters through training the network are then referred to as network-enabled automatic representations (NEARs). To validate these NEARs, we implement them on RF and GBT models, leading to accurate predictions. We further show that ensemble learning can in some cases bolster our network predictions through consensus.

To test our AweGNN models, we employ four toxicity data sets of various sizes. Toxicity is a measure of the degree to which a chemical can harm an organism [110]. The harmful effects can be measured qualitatively or quantitatively. A qualitative approach only categorizes chemicals as toxic or nontoxic, while quantitative toxicity data records the minimal amount of a substance that would produce lethal effects. Experimental measurement of toxicity is expensive, time-consuming, and subject to ethical constraints. In this light, machine learning models are extremely useful in that they do not have these same challenges. The working principal of the machine learning approach for toxicity analysis is that molecules with similar structures have similar activities, which is called the quantitative structure-activity relationship (QSAR) approach. By analyzing the relationship between molecular structures, one can predict their biological functions. We can create AweGNN models that predict these toxicity endpoints without having to conduct any lab experiments [15, 96, 57, 48]. To further improve our AweGNN predictions on the sparsity of data [46], we construct multi-task (MT) AweGNNs. MT learning or transfer learning [23] learns from related tasks to improve the performance on the smaller data sets in particular. It is frequently used to compensate if there are similar data sets at hand. We find that AweGNN models perform well on these data sets when we compare our results to state-of-the-art QSAR techniques, such as the ones that were pursued by our group [110, 81] previously and by others[64, 65], with particularly excellent performance on the larger data sets.

To further showcase our AweGNN, we apply our models on a different data set; a small data set concerning the solvation of molecules. Solvation free energy is a very im-

portant quantity in solvation analysis, which can be very beneficial in studying other complex or biological processes [63, 74, 105, 55]. This data set is relatively small compared to the toxicity data sets, but we found that the AweGNN was able to perform extremely well, beating all other methods in the literature [81]. This proves that the AweGNN is useful in application to molecular problems that are not just restricted to toxicity analysis. In addition to this, we can find that the concept of the AweGNN can be extended further to other kernel-based methods, in which we can automate the kernel parameters as we train a network, such as with the differential geometry work done by Nguyen et al [81]. Finally, we note that although we have not used multi-scale approaches in this work, we have seen them perform well in previous work [78, 81], so it is likely to be worthy of exploration in further works.

5.2 Theory and Methods

In this section, we will briefly review single and multi-task neural networks, then describe our AweGNN models. We discuss AweGNN through the frameworks of element-specific geometric graph representations, dynamic normalization functions, explicit derivative calculations, and instructions on how to update the parameters. Some variations on the generation of features are included even though these variations might have not been used to train the final models.

5.2.1 Neural Networks

Neural networks are predictive models that consist of layers which are made up of neurons in which each neuron is connected to every other neuron in the next layer of the layer sequence. A weight is associated with each connection that determines how much a neuron contributes to the input of the neuron in the next layer, and a bias term is introduced to shift the activations. Activation functions, such as the logistic sigmoid, grant the network a level of non-linearity for additional complexity. A feature vector, in which

each feature corresponds to a neuron in the input layer, can be pushed through the layers of the network all the way to the output layer in a process called forward propagation. Neural networks can be trained as classification models that categorize data, or regression models that predict continuous quantities. In this work, we only develop regression models.

Neural networks are trained by updating the weights and biases of the network in a series of steps through gradient descent, in which an appropriate loss function is minimized. The process of calculating the errors of a neural network is called back propagation, in which derivatives are calculated in the backwards direction starting from the output layer, where the loss (or error) is calculated, and propagating backward through each layer all the way back to the input layer, after which the weights are then updated by following the negative direction of the gradient. The gradient descent process finds a local minimum of the loss function when viewed with respect to the weights. This process fits the network to the data set in question.

A general neural network consists of an input layer, an output layer, and any number of hidden layers in between. Deep neural networks are characterized by having many more layers and neurons, allowing the network to be more complex. There are many additions to the neural network architecture that can be included for training a model. Activation functions bring non-linearity to a network, and a proper choice of activation function can have an effect on training and performance of a network. Dropout [97] layers prevent over-fitting by dropping out random neurons during each step of training. Weight decay is another technique that regularizes the network to prevent over-fitting by decreasing the magnitude of each weight throughout the training. Batch normalization [43] layers normalizes the outputs of each layer to speed up the training time, improve performance, and allow deep networks to be trained with stability by reducing internal covariant shift. Adaptive learning rates are learning rates that change during the training of the network, such as in a momentum approach to gradient descent that "gains speed

as it descends down a valley", that can find local minimums faster and sometimes have the ability to jump out of “undesirable" local minimums. We use batch normalization, ReLU activation, and the AMSGrad [87] variant of the popular Adam [50] optimization adaptive learning rates in our work. Dropout and weight decay increased the training time dramatically and so were not used in our final models, but may be explored further in the future.

Single-task (ST) neural networks are networks trained on a single data set and have a single output (at least in the regression case). An ST network is trained by the minimization problem:

$$\min_{\mathbf{W},\mathbf{b}} \left[\alpha \cdot ||\mathbf{W}||_2^2 + \sum_{i}^{N} L(y_i, f(\mathbf{x}_i; \{\mathbf{W}, \mathbf{b}\}))\right], \tag{5.1}$$

where $||\cdot||_2$ denotes the L_2 norm, α represents the regularization constant, f is a function parametrized by the weights, $\mathbf{W}$, and biases, $\mathbf{b}$, that represents the output of the network, and $\mathbf{x}_i$ and y_i are the feature vector and the label of the ith data point respectively of a data set with N samples.

Single-task networks are, however, limited in their usefulness when being trained on small data sets. Multi-task (MT) networks have been developed to take advantage of large data sets to bolster the training of a network on a smaller data set. The idea is to train the models on multiple tasks simultaneously by sharing weights, thus highlighting relevant features that are important across all the related tasks. The details of their training are as follows:

If we let T be the number of tasks and $\{(\mathbf{x}_i^t, y_i^t)\}_{i=1}^{N_t}$ is the training data for the tth task where N_t is the number of samples of the tth task, $\mathbf{x}_i^t$ is the feature vector of the ith sample of the tth task, and y_i^t is the label for the ith sample of the tth task. The training strategy is to minimize the chosen loss function, L, for all tasks simultaneously. We define the loss for task t, L_t, below.

$$L_t = \sum_{i=1}^{N_t} L(y_i^t, f^t(f^s(\mathbf{x}_i^t; \{\mathbf{W}^s, \mathbf{b}^s\}); \{\mathbf{W}^t, \mathbf{b}^t\})), \tag{5.2}$$

where f^t is the part of the network that predicts the labels for the tth task parametrized by weights, $\mathbf{W}^t$, and bias, $\mathbf{b}^t$; f^s is the part of the network that has shared weights, $\mathbf{W}^s$, and bias, $\mathbf{b}^s$. To train all models simultaneously, we provide a single loss function to minimize so our problem is then:

$$\text{argmin} \left[\beta \cdot ||\mathbf{W}^s||_2^2 + \frac{1}{T} \sum_{t=1}^{T} \left(L_t + \alpha \cdot ||\mathbf{W}^t||_2^2 \right) \right], \tag{5.3}$$

where $|| \cdot ||_2$ denotes the L_2 norm and α and β represent the regularization constants that set the magnitudes of the weight decay during training. In practice, if weight decay is used, we generally set $\alpha = \beta$.

5.2.2 Overview of AweGNN

We wish to describe the concept of AweGNNs applied to molecular data sets. As in most machine learning endeavors, we seek to frame the task at hand as a minimization problem. To detail our supervised learning algorithm, we first start with a biomolecular data set, χ, where χ_i will represent the ith element of the training data set. We want a function $\mathbf{F}(\chi_i; \{\boldsymbol{\eta}, \boldsymbol{\kappa}\})$ that encodes the geometric and chemical information into an abstract representation parametrized by a set of parameters, $\{\boldsymbol{\eta}, \boldsymbol{\kappa}\}$. Then, the minimization problem becomes:

$$\min_{\boldsymbol{\eta}, \boldsymbol{\kappa}, \mathbf{W}, \mathbf{b}} \alpha \cdot ||\mathbf{W}||_2^2 + \sum_{i \in I} L(\mathbf{y}_i, f(\mathbf{F}(\chi_i; \{\boldsymbol{\eta}, \boldsymbol{\kappa}\}); \{\mathbf{W}, \mathbf{b}\})), \tag{5.4}$$

where L is the chosen scalar loss function to be minimized, α is the regularization constant that determines the magnitude of the weight decay, and $\mathbf{y}_i$ is the label of the ith data point in our biomolecular data set. The function, f, is the function parametrized by the weights, $\mathbf{W}$, and biases, $\mathbf{b}$, that represent the output of the neural network. Notice the difference

with the formulations in the previous section. The output of the function, $\mathbf{F}$, on the data χ_i acts as the feature vector, $\mathbf{x}_i$, given previously. The parameters, $\{\boldsymbol{\eta}, \boldsymbol{\kappa}\}$, that parametrize $\mathbf{F}$ are included in the minimization process. This simultaneous update of the parameters of the representation function is the novel idea driving this work.

The AweGNNs are trained for the regression task of predicting the toxicity endpoints and the solvation free energy of various small molecules, with the standard choice of mean squared error (MSE) as our loss function, L. The regularization constant, α, as noted before, is omitted in our work for both the single and multi-task models due to a lack of noticeable increase in performance, but with a moderate increase in training time. $\mathbf{F}$ will be based on element-specific calculations that are controlled by pairs of tunable kernel parameters that capture different geometric features of the molecules based on the values of those kernels chosen. Each element-specific group will be assigned a pair of these parameters and will be updated throughout the training by back propagation. The representations generated by $\mathbf{F}$ after the training of the AweGNN are called network-enabled automatic representations (NEARs), and will be used later for training other models to demonstrate their performance. The procedure is summarized in Figure 5.1 below.

More specifically, we wish to understand how to back propagate through $\mathbf{F}$. This will require us to, at each batch or epoch, back propagate throughout the entire network, then use that information to back propagate through $\mathbf{F}$ all the way to the kernel parameters that $\mathbf{F}$ is dependent on. In reality, $\mathbf{F}$, should be a composition of a function that outputs vector representations of biomolecules along with the normalization of the output of that function. Thus, we can represent $\mathbf{F}$ as $\mathbf{F} = N \circ R$, where N is the normalization function, and R is the raw representation function. The normalization function is important because it avoids instability in the network. Now when considering back propagation through the function, $\mathbf{F}$, we have to keep in mind the chain rule, $\frac{\partial \mathbf{F}}{\partial x} = \frac{\partial N}{\partial R}\frac{\partial R}{\partial x}$, as we calculate our partial derivatives.

If we want to be able to calculate these derivatives, then we must make sure that

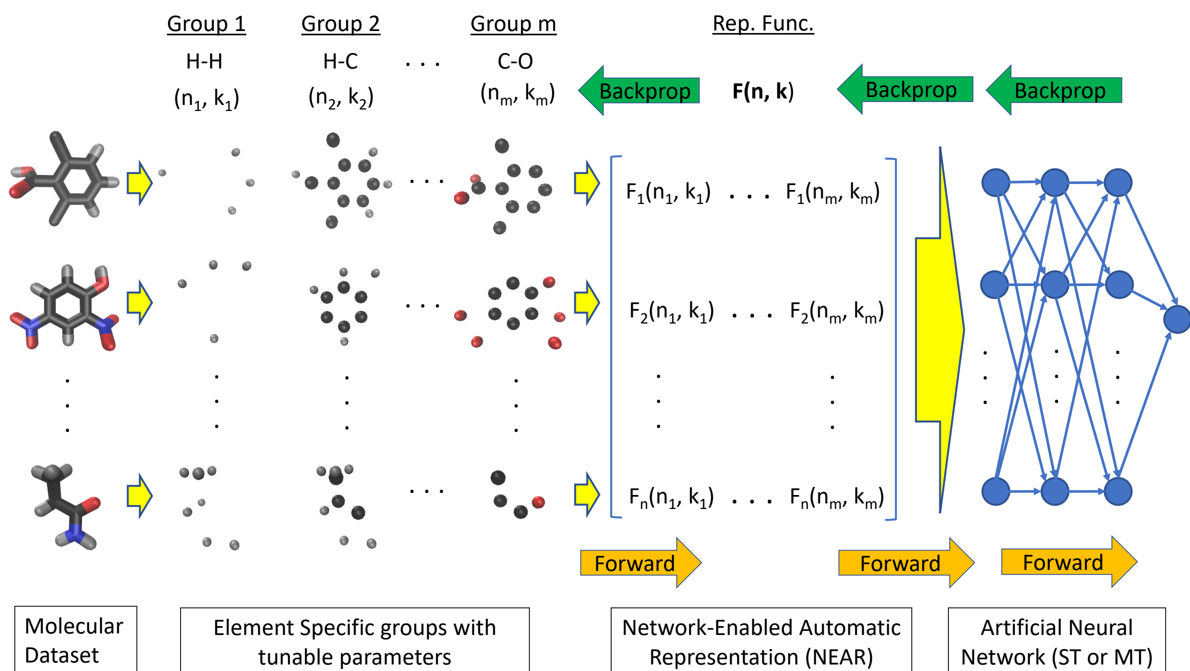


Figure 5.1: Pictorial visualization of the AweGNN training process. The first stage is splitting up the molecules in the data set into the element-specific groups, with initialized η and κ kernel parameters for each group. Then, a molecular representation is generated based on the kernel parameters, given by $\mathbf{F}$ in the diagram. This representation is fed into the neural network, then the kernel parameters are updated by back propagation through the neural network and through the representation function, $\mathbf{F}$.

they are differentiable functions. Some normalization functions, however, will not be differentiable everywhere, but will usually still be differentiable *almost* everywhere and continuous *almost* everywhere, so they will be practical functions to use. Representation functions themselves can be very discontinuous and non-differentiable depending on the method of calculating features, so we must be especially careful in selecting the function, R. We would also like both the representation function and normalization function to be easy to calculate since we must calculate the normalized features at each epoch of training. We will later describe in detail the geometric graph representation function, R, that we used for our AweGNN that is continuous and differentiable everywhere, and easy to calculate.

We will be testing both single-task (ST) and multi-task (MT) AweGNN models that follow much the same structure, again as outlined in figure 5.1. The ST model takes

batches and from a data set, generates a representation, then normalizes it with batch normalization before feeding it into the artificial neural network portion of the AweGNN and then updating the parameters as above. The MT-AweGNN consists of a common set of parameters that are used for all 4 toxicity data sets, shared weights for the hidden layers of the artificial neural network portion, but separate weights for the 4 output layers. The normalization is done also in batches, by placing a number of data points into the batch from each data set proportional to the size of that data set with respect to all the data sets combined. For example, if the batch size is 100, and there are data sets with size 200, 400, 600, and 800, then each batch would have 10 samples from the first data set, 20 from the second, 30 from the third, and 40 from the last data set. This MT structure is illustrated below in Figure 5.2.

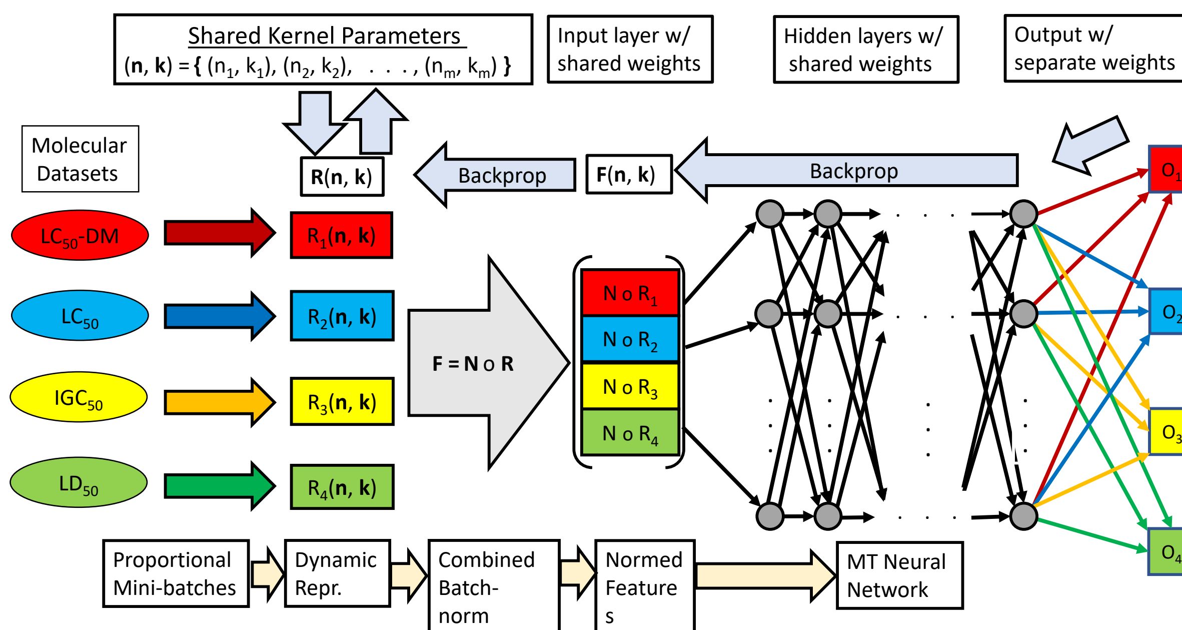


Figure 5.2: Diagram depicting the structure and automation of the MT-AweGNN. The model is trained by collecting samples from each data set proportionally and then generating their representations according to shared parameters. The representations are normalized together, then pushed through the artificial neural network and then to their corresponding outputs. Back propagation goes through the feature matrix, normalization function, and finally the representation function to update the kernel parameters.

5.2.3 Normalization Function

For the normalization function, we choose the standard normalization function, N_σ, which centers each feature about the mean and scales by the standard deviation of the training samples. More precisely, if x is the feature, μ the mean, and σ the standard deviation, then $N_\sigma(x) = \frac{x-\mu}{\sigma}$. This function is continuous and differentiable everywhere except when the standard deviation is zero. By a slight modification, we can add an error constant for numerical stability and to make N_σ continuous and differentiable everywhere, where $N_\sigma(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}$ with ϵ being a very small number such as 10^{-5}. When training in batches, we normalize each batch individually with separate batch mean, $\mu_\mathcal{B}$, and batch standard deviation, $\sigma_\mathcal{B}$. These statistics are tracked and a batch momentum value of 0.1 is used to record accumulated batch statistics to obtain a more stable reading of the distribution of the features in the whole data set, as detailed in Ioffe and Szegedy[43]. During evaluation time, the recorded statistics are used to transform the output of the representation function, R.

The normalization procedure is similar to batch normalization in neural networks, except that the affine transformation is omitted, and the expression for $\frac{\partial N}{\partial R}$ can be found in the paper written by Ioffe and Szegedy [43], along with the rest of the details for batch normalization. Although the above procedure omits the affine transformations, we chose to apply the affine transformations in our work becuase the models performed well with them. In practical terms, the PyTorch[82] library is utilized for applying this batch normalization. A BatchNorm1d layer is placed in front of the molecular representation layer with parameters: eps=1e-05, momentum=0.1, affine=True, and track_running_stats=True, which are the default settings in PyTorch[82].

5.2.4 Biomolecular Geometric Graph Representations

We seek a general formulation of a geometric graph representation of biomolecular structures from which we can extract features to create our sought after representation func-

tion, R. We use this method to generate descriptors for the toxicity analysis of small molecules, but the idea can extend beyond this concept quite naturally to provide useful representations. In our work, we want to use a weighted and vertex-labeled graph in which we can generate descriptors from special subgraphs of the graph representation of each molecular structure. We begin by looking for a set, $\mathcal{T}$, of element types that are suitable for our analysis. Generally the choice of element types will be chosen by looking at the commonly occurring element types found in a given data set, or we might choose to omit certain element types to avoid elements that have uncertain positioning or negligible influence in molecular interactions. Usually, we will have at least C, N, O, S $\in \mathcal{T}$, but in most cases we will also include H, P, Cl, and Br. We define:

$$\mathcal{V} = \{(\mathbf{r}_j, \alpha_j) \in \mathbb{R}^3 \times \mathcal{T} \mid \mathbf{r}_j;\ \alpha_j \in \mathcal{T};\ j = 1, 2, ..., N\}$$

to be the vertex set, in which N denotes the number of atoms in the molecule that are of an element type that is a member of $\mathcal{T}$. This vertex set has a labeling that describes the atom coordinates and the element type of the atom at each vertex. We will use these labels to determine element specific subgraphs from which to extract features.

Our edge set, $\mathcal{E}$, is described by a choice of parameters, $\{\eta_{kk'}\}$ and $\{\kappa_{kk'}\}$, and a choice of a type of kernel function, $\Phi : \mathbb{R} \to \mathbb{R}$, which is parametrized by them

$$\mathcal{E} = \{\Phi(||\mathbf{r}_i - \mathbf{r}_j||; \eta_{kk'}, \kappa_{kk'}) \mid \alpha_i = k \in \mathcal{T}, \alpha_j = k' \in \mathcal{T}\ ;\ i, j = 1, 2, ..., N\ ;\ ||\mathbf{r}_i - \mathbf{r}_j|| > v_i + v_j + \sigma\}$$

,

where $||\cdot||$ denotes the Euclidean distance, v_i and v_j denote the Van der Waals radius of the ith and j^{th} atoms respectively, $\mathbf{r}_i$ and $\mathbf{r}_j$ represent the ith and j^{th} atom coordinates, and σ is the average of the standard deviations of v_i and v_j in the data set. Notice, the distance constraint eliminates covalent interactions from being used in our calculations. We remove the covalent interactions because many biomolecular properties are known to be

determined by non-covalent interactions, As noted in Nguyen et al., many biomolecular properties are known to be determined by non-covalent interactions, so we remove the covalent interactions because we do not want their contribution to overwhelm or otherwise interfere with the contributions of the non-covalent interactions. The parameters, $\eta_{kk'}$ and $\kappa_{kk'}$, are values tied to the element specific pair, kk', which can be tied to the properties of the element types in question. The types of kernels that determine the weights of our edges are always chosen to be decreasing functions that have the properties:

$$\Phi(x) \to 1 \text{ as } x \to 0 \tag{5.5}$$

and

$$\Phi(x) \to 0 \text{ as } x \to \infty. \tag{5.6}$$

This characterization gives the property that atoms that are closest will contribute the most, and atoms that are too far away, will contribute almost nothing. Most radial basis functions can be used, but the most popular choice of kernel types are the generalized exponential and the generalized Lorentz functions, given by

$$\Phi^E(x;\ \eta, \kappa) = e^{-(\frac{x}{\eta})^\kappa} \tag{5.7}$$

and

$$\Phi^L(x;\ \eta, \kappa) = \frac{1}{1 + (\frac{x}{\eta})^\kappa}, \tag{5.8}$$

respectively. These kernel types have been used to create successful models for protein-ligand binding prediction, predicting toxicity endpoints, and many other applications. They act as low-pass filters that capture the most important element interactions, where the η value determines the cutoff point, and the κ value determines the sharpness of the cutoff. In summary, we have procured a vetex-labeled, weighted graph, $G(\mathcal{V}, \mathcal{E})$, in which we can extrapolate features from special element specific subgraphs.

Let $G_{kk'}$ be the subgraph of G whose vertex set, $\mathcal{V}_{kk'}$, contains the vertices that are labeled element type k or k'; and the edge set, $\mathcal{E}_{kk'}$, contain only the edges connecting a vertex labeled with element k to a vertex labeled with element type k'. This is called the element specific subgraph generated by the element pair kk'. For a given element specific subgraph, $G_{kk'}$, we define the following descriptor associated with that subgraph:

$$\mu_{G_{kk'},\Phi} = \sum_{(i,j)\in\mathcal{E}_{kk'}} w_j \cdot \Phi(||\mathbf{r}_i - \mathbf{r}_j||;\ \eta_{kk'}, \kappa_{kk'}) \tag{5.9}$$

where the w_j term is usually a constant 1 or a value associated with the jth atom, such as a partial charge. The $\eta_{kk'}$ and $\kappa_{kk'}$ values represent the respective η and κ values for the kk' group, while $\mathbf{r}_i$ and $\mathbf{r}_j$ represent the atom coordinates as before. We get a sum over the weights of the subgraph in which additional atom specific weights can be applied to enhance the meaning of the feature. This abstractly defines the total strength of the non-covalent interactions between the atoms of element type k and the atoms of element type k'. This is illustrated in Figure 5.3 below.

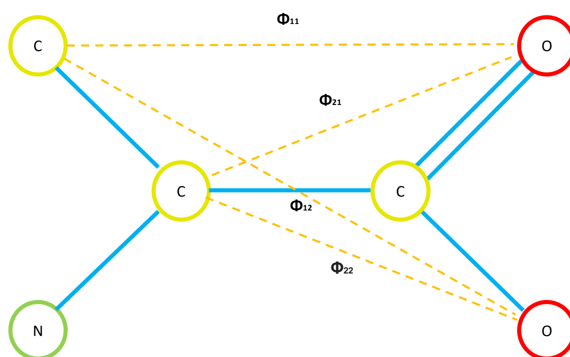


Figure 5.3: This shows an element specific subgraph corresponding to the element types C and O of amino-propanoic acid, $C_3H_6NO_2$ (with the hydrogens omitted). The dashed orange edges represent the edges of the subgraph that are weighted by the chosen kernel function, Φ, while the solid blue lines represent covalent bonds. Notice the 3rd carbon atom does not connect to the oxygen atoms since it is covalently bonded to both of them.

With these descriptors detailed above, we can describe a representation function by

obtaining one or more features from each element specific subgraph. We can extract multiple features per subgraph by considering different statistics of our features such as the mean of the sum above, the standard deviation of the terms in the sum, choosing various values of w_j above according to chemical or physical properties of specific atoms, etc. In summary, the features that we generate using this approach measures the aggregate interaction strength between groups of atoms of specific element types, which we call element-specific groups. This gives us a representation of the internal structure of a molecule that fits into the QSAR paradigm, which promotes the idea that similar molecular structures have similar physical properties.

The representation function used in our work for the toxicity data sets consisting of small molecules considers the element type set $\mathcal{T} = \{H, C, N, O, S, P, F, Cl, Br, I\}$. For each element specific pair, we calculate 4 descriptors: The first is calculated with each $w_j = 1$, the next with the w_j's set to be equal to the partial charge of the jth atom, and the remaining 2 descriptors are generated by dividing the first 2 descriptors by the number of edges in the element specific group, giving us a measure of the mean interaction of the elements. Since we have 10 different element types, we can form 100 element specific groups; and since we have 4 descriptors per element specific group, we have 400 descriptors for every set of parameters and choice of kernel. In our work, we only use the Lorentz functions, $\Phi^L_{\eta,\kappa}$, for our feature generation and results.

5.2.5 Parameter Adjustment and Initialization

When considering our molecular representation function in the context of the parameter automation task, we can make a few choices in terms of the parameters that we will be adjusting at each epoch or batch. Below are 3 possible avenues to consider:

η-adjustable Here, we set the η values for each element specific pair randomly within some range. We note that these assignments that are made for each element specific pair are common for all of the biomolecular structures in you data set. We can combine several

sets of features to make multi-scale models that would have multiple initializations of the η values, but we will be assuming a 1-scale model at present. If we set the η values this way, these will be the parameters that we will be updating in our gradient descent.

τ**-adjustable** For an element specific pair, kk', we set the characteristic value to be $\eta_{kk'} = \tau(v_k + v_{k'})$, where v_k is the Van der Waals radius of element type, k, and $v_{k'}$ is the Van der Waals radius of element type, k'. Similar to the previous situation, the τ value assigned determines the element specific η values for the entire data set. Again, if we consider multi-scale models, then we may choose different values of τ for each set. In the τ-adjustable case, we will be seeking to update the τ value at each step of the training.

κ**-adjustable** In the previous two situations, we were assuming a fixed κ value, but we can also update this parameter in conjunction with the others. There are two ways to introduce this technique. The first way is to introduce one κ and update that with respects to all of the element specific groups. The second way is to introduce a κ value for all the element specific groups and update those separately. Multi-scale models are handled in a similar way as above.

Aside from choosing which parameters to update throughout the training of the AweG-NNs, we also must consider how we want to initialize each parameter before training. We notice that τ, κ, and η all have to take values greater than zero to maintain continuity and/or satisfy the conditions of a radial basis function. Also, we want to make certain that the values are not too large. A large η value will capture all of the atoms in a molecule and the kernel value will be effectively a constant 1 at all measured distances. A large κ value will make the kernel function approach the form of an ideal low-pass filter, where it is either a constant 0 or constant 1. In both cases, the magnitude of the derivatives will be extremely small and the model will not be able to update effectively.

In our experimentation, we use 100 η-adjustable and 100 κ-adjustable parameters per scale (1 η and 1 κ value per element specific group) giving us 200 total adjustable variables per scale for our molecular representation function. The η-initialization for each scale is

chosen by a single random tau value from a uniform distribution with a range of 0.5 to 1.5, in which all the η values are set according to the Van der Waals radii as described above. The κ-initialization for each scale is done by chosing a value of κ for each element-specific group from a uniform distribution with a range of 5 to 8.

The adjustment of parameters throughout the training of our AweGNN will be like an evolving kernel function at each element-specific group that attempts to create an optimal filter that provides the best representation for the network. The η values determine the center of the cutoff region and the κ values control the sharpness of the cutoff. In Figure 5.4, we see the evolution of the kernel function of a MT-AweGNN model, where the kernel functions are graphed at various choices of the number of epochs of training, so that the transformation of the kernel function can be clearly seen.

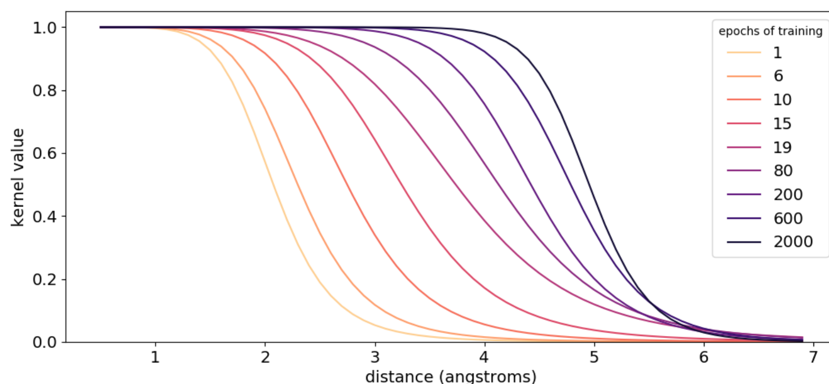


Figure 5.4: Evolution of the kernel function of the C-N group for a MT-AweGNN. The picture below shows a series of kernel functions that were used to evaluate the 4 features corresponding to the C-N element-specific group at different points in the training of an MT-AweGNN. We choose specific snapshots during training that show a smooth change in the kernel function as the η-κ pair is updated.

5.2.6 Derivatives of the Representation Function

Our main goal is to update the parameters of our feature representation during the training of our neural network. We must first show how to calculate the derivatives of the

representation functions with respect to parameters κ, η, and τ values. We will then use these derivatives to show how to complete the back propagation through all the way to the parameters. We begin with a representation function, $R : \mathbb{R}^n \to \mathbb{R}^{d\times m}$, and focus on a given element-specific subgraph; where this could be a single or multi-scale representation function.

Suppose that $\boldsymbol{\nu}$ is the set of update-able parameters for R. Then we can represent R by:

$$R(\boldsymbol{\nu})_{i,j} = \sum_{(a,b)\in G} q_b \cdot \Phi(||\mathbf{r}_a - \mathbf{r}_b||; \boldsymbol{\nu}_G), \tag{5.10}$$

Where G is the element specific subgraph which is used in the calculation of the jth descriptor of the ith sample, q_b represents a value that may be associated with the atom labeled b, $\mathbf{r}_a$ and $\mathbf{r}_b$ represent the coordinates of the atoms labeled a and b respectively, and $\boldsymbol{\nu}_G$ is the subset of $\boldsymbol{\nu}$ that corresponds to the calculations that are associated with G.

Now, since differential operators are linear and our features are calculated by a summation of differentiable functions, we need only know how to calculate the derivative of each term, i.e., for any variable, x, we have:

$$\left(\frac{\partial R(\boldsymbol{\nu})}{\partial x}\right)_{i,j} = \sum_{(a,b)\in G} q_b \cdot \frac{\partial}{\partial x}\Phi(||\mathbf{r}_a - \mathbf{r}_b||; \boldsymbol{\nu}_G). \tag{5.11}$$

Thus, we want to find out the derivatives of the functions, $\Phi^E_{\eta,\kappa}$ and $\Phi^L_{\eta,\kappa}$, with respect to parameters η, τ, and κ. We start by finding the derivative with respect to η.

The derivative of the generalized exponential function is:

$$\frac{\partial \Phi^E_{\eta,\kappa}(r)}{\partial \eta} = \left(\frac{\kappa}{\eta}\right)\left(\frac{r}{\eta}\right)^{\kappa} e^{-(r/\eta)^{\kappa}}, \tag{5.12}$$

and for the Lorentz generalized function, we have the derivative:

$$\frac{\partial \Phi^L_{\eta,\kappa}(r)}{\partial \eta} = \frac{\left(\frac{\kappa}{\eta}\right)\left(\frac{r}{\eta}\right)^{\kappa}}{\left(1 + (\frac{r}{\eta})^{\kappa}\right)^2} \tag{5.13}$$

Also, we calculate the derivatives with respect to κ parameters starting with the generalized exponential function:

$$\frac{\partial \Phi^E_{\eta,\kappa}(r)}{\partial \kappa} = \left(\frac{r}{\eta}\right)^{\kappa} \ln\left(\frac{\eta}{r}\right) e^{-(r/\eta)^{\kappa}}, \tag{5.14}$$

and for the Lorentz generalized function, we have the derivative:

$$\frac{\partial \Phi^L_{\eta,\kappa}(r)}{\partial \kappa} = \frac{\left(\frac{r}{\eta}\right)^{\kappa} \ln\left(\frac{\eta}{r}\right)}{\left(1+\left(\frac{r}{\eta}\right)^{\kappa}\right)^2}. \tag{5.15}$$

Now, for the derivative with respect to the τ parameter, we can just apply the chain rule with $\eta = \tau(v_1 + v_2)$ being a function of τ. Note that $\dfrac{\partial \eta}{\partial \tau} = (v_1 + v_2) = \dfrac{\tau(v_1 + v_2)}{\tau} = \dfrac{\eta}{\tau}$. Then, for the exponential derivative, we have:

$$\frac{\partial \Phi^E_{\eta,\kappa}}{\partial \tau} = \frac{\partial \eta}{\partial \tau} \cdot \frac{\partial \Phi^E_{\eta,\kappa}}{\partial \eta} = \left(\frac{\eta}{\tau}\right)\left(\frac{\kappa}{\eta}\right)\left(\frac{r}{\eta}\right)^{\kappa} e^{-(r/\eta)^{\kappa}} = \left(\frac{\kappa}{\tau}\right)\left(\frac{r}{\eta}\right)^{\kappa} e^{-(r/\eta)^{\kappa}}, \tag{5.16}$$

and for the lorentz function, we have:

$$\frac{\partial \Phi^L_{\eta,\kappa}}{\partial \tau} = \frac{\partial \eta}{\partial \tau} \cdot \frac{\partial \Phi^L_{\eta,\kappa}}{\partial \eta} = \left(\frac{\eta}{\tau}\right) \frac{\left(\frac{\kappa}{\eta}\right)\left(\frac{r}{\eta}\right)^{\kappa}}{\left(1+\left(\frac{r}{\eta}\right)^{\kappa}\right)^2} = \frac{\left(\frac{\kappa}{\tau}\right)\left(\frac{r}{\eta}\right)^{\kappa}}{\left(1+\left(\frac{r}{\eta}\right)^{\kappa}\right)^2} \tag{5.17}$$

Thus, we now know the derivatives $\frac{\partial R}{\partial \eta}$, $\frac{\partial R}{\partial \tau}$, and $\frac{\partial R}{\partial \kappa}$; and we can use them to calculate the gradient for our gradient descent when training our neural network.

5.2.7 How to Update the Parameters

Now that we have the derivatives of the representation function, we will be able to calculate the derivatives of the loss function that we have chosen, L, with respect to our parameters, η, τ, and κ. We must remember that in our process of feature calculation, we have decided to normalize the features. We chose to use the standard normalization, N_σ, and we denoted the normalized representation function to be $F = N \circ R$, with R being the un-normalized representation function. By the chain rule, we obtain $\frac{\partial F}{\partial x} = \frac{\partial N}{\partial R}\frac{\partial R}{\partial x}$, where $x = \eta$, κ, or τ. We have also discussed what $\frac{\partial N}{\partial R}$ looks like, so in fact we should now have a

full description of the derivatives of the normalized representation function with respect to our parameters.

Now, to get our final update rule, we have to extend the back propagation of our neural network to the features, obtaining the derivatives, $\frac{\partial L}{\partial F}$. Then our derivatives that we calculated earlier, $\frac{\partial F}{\partial x}$, will be used in the chain rule again to get: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial F}\frac{\partial F}{\partial x}$, where $x = \eta$, κ, or τ. This gives us the update rule for our parameters. In more detail, the update rule actually looks like $\frac{\partial L}{\partial x} = \sum_{i,j} \frac{\partial L}{\partial F_{ij}}\frac{\partial F_{ij}}{\partial x}$. Remember that the τ, κ, and η parameters are all constrained to be greater than zero. As a precaution we propose that after every epoch, the parameters be clipped so that they are constrained to values that are 0.01 or higher to avoid forbidden values.

5.2.8 Multi-scale Models

Multi-scale models are models that are trained on features generated by multiple sets of parameters. More specifically, for 2 representation functions, $R_1 : \mathbb{R}^n \to \mathbb{R}^{d\times m}$ and $R_2 : \mathbb{R}^n \to \mathbb{R}^{d\times m}$, we can get the 2-scale representation function, $[R_1, R_2] : \mathbb{R}^n \to \mathbb{R}^{d\times 2m}$, by concatenating the outputs of R_1 and R_2 so that the features of each data point match up.

We can extend this idea of a 2-scale model to any scale. Let $R_1 : \mathbb{R}^n \to \mathbb{R}^{d\times m}$, ... , $R_k : \mathbb{R}^n \to \mathbb{R}^{d\times m}$ be representation functions with the same parameter types. Then, combining the functions together one at a time as above, then we get a new multi-scale representation function, $[R_1, R_2, ..., R_k] : \mathbb{R}^n \to \mathbb{R}^{d\times (k\cdot m)}$. These multi-scale methods have been shown to be very effective in improving the performance of single-scale models[81, 78].

5.2.9 Model Architectures and Hyper-parameters

We wish to now describe the specific network architecture of the AweGNN model. For the artificial neural network (ANN) portion of the AweGNN, we choose a very simple

4-layer network with 400 neurons in the first two hidden layers and 20 neurons in the last two hidden layers. The ANN architecture and parameters were chosen through a quick parameter search of the multi-task network where the models were tested on a randomized validation set of 10% of the training data of each toxicity data set to obtain good average predictions across all 4 data sets and was modified to ensure relative convergence for the kernel parameters. The convergence assures that we have stable choices for the parameters of the kernel function, which we will use for later analysis. Dropout and weight decay are omitted since they did not notably increase performance, but increased the training time significantly.

The AMSGrad [87] variant of the Adam optimizer is used because it has been shown to improve the convergence of the kernel parameters while still achieving large gradients that allow the model to explore a larger parameter space, and allows a more interesting analysis of the parameter trajectories. A large learning rate of 0.1 is also applied with a learning rate decay of 0.999 per epoch of training, for the total of 2000 epochs, leaving the final learning rate at a value of approximately 0.01352 at the end of the training. The decay gives the network and kernel parameters an opportunity to settle down in a local minimum, while the large learning rate gives the models a chance to explore more parameter choices while also regularizing the network. The network was further improved by applying batch normalization, which is known to speed up the training of a network and provides an additional source of regularization that reduces the need for dropout[43]. Finally, the batch size used was 100 for the ST models, while the MT models used a total of 40 batches per epoch, with sizes as even as possible and number of samples from each data set proportional to their respective size.

For all of the network models, we use the PyTorch[82] API in conjunction with cython[17] and Numba[52] to speed up the calculation of the geometric graph descriptors. All hidden layers are actually composed of 3 different layers: a linear unit, followed by a batch normalization layer with affine transformations active and momentum set to 0.1 (default

PyTorch setting), then a ReLU activation. The input layer is constructed with 2 layers: a Geometric Graph Representation (GGR) layer followed by a batch normalization layer again with affine transformations and the same momentum setting. The GGR layer is the representation function, R, and the batch normalization layer is the N_σ normalization function, as described in detailed previously to generate our normalized representation function, $F = N_\sigma \circ R$. Finally, the output layer is simply a linear activation into either 1 or 4 outputs, depending on if we are using a multi-task or single-task model. The details of the full AweGNN architecture are illustrated in Figure 5.5 below.

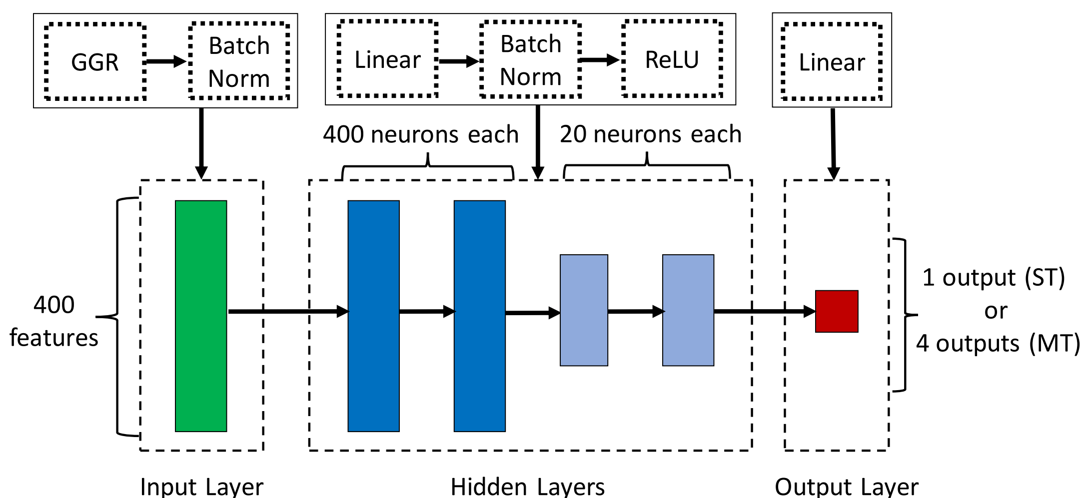


Figure 5.5: Internal architecture of our Auto-parametrized weight element-specific Graph Neural Network (AweGNN) model for the toxicity data sets. The input layer consists of our novel Geometric Graph Representation (GGR) layer and a batch norm with no affine transformation. Hidden layers include a linear transformation followed by regular batch normalization and ReLU activation, while the output layer has the standard linear activation.

As discussed earlier, the network-enabled automatic representation (NEAR) generated from training an AweGNN can be used as inputs for ensemble models. By examining the performance of these models, we can see if the representations generated by AweGNNs can produce meaningful features for a broad range of machine learning models. We use two ensemble models, random forest regressor (RF) and gradient boosting regressor (GBT), from the scikit-learn v0.23.2 package [83]. A search for reasonable parameter choices was made on the same validation sets above. For the RF models, we

use the parameters: n_estimators = 8 000, max_depth = 27, min_samples_split = 3, and max_features=sqrt, with any remaining parameters considered to be set at the default setting. For the GBT models, we use the same parameters as the RF models above along with the additional parameters: loss = 'ls', learning_rate = 0.1, and subsample = 0.2, with any remaining parameters set to the default choice.

5.3 Results

In this section, we give a description of the different quantitative toxicity data sets, and then we compare the results of our best models to the models generated by other published methods [64, 65, 110, 81]. We note that in the work by Nguyen et al.[81], we only report the results from one consensus model trained and tested on the Tetrahymena pyriformis IGC_{50} data set that performed the best, which is labeled as *Nguyen-best*. For the work done by Wu et al.[110], we only report the results for the models that utilized all the descriptors detailed in that paper. The results of the random forest models are referred to as Wu-RF, the gradient boosting models are referred to as Wu-GBT, the ST models are Wu-ST, the MT models are Wu-MT, and the consensus of Wu-GBT and Wu-MT is Wu-consensus. Except in the case of the *Daphnia Magna* LC_{50}, the best scoring Wu models were those that relied on all descriptors, but any such important detail will be mentioned when relevant.

We also introduce a solvation data set, Model III, as in the paper by Nguyen et al [81]. The performance of our AweGNN models, when trained on this data set, is compared to those of differential geometry based models developed earlier [81], and other state-of-the-art methods [106, 104]. The models in this particular section are named as they are in the previously referenced paper [81], where the names refer to various choices of kernel options/kernel parameters used in the calculation of differential geometric descriptors.

As for our own models, we report them in many ways and in many combinations. We train multiple models for each data set and each model type, then we combine those in

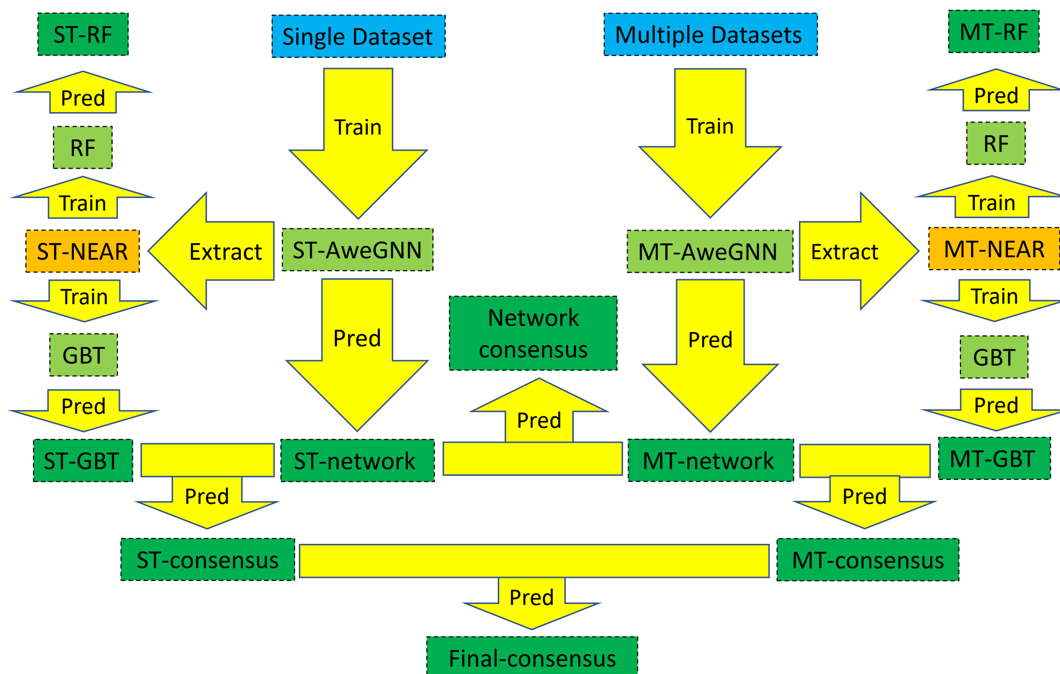


Figure 5.6: Pictorial representation of the overall strategy. The arrows show the flow of the processes of training, extracting representations from trained AweGNNs, and making predictions. Blue corresponds to molecular data sets, light green corresponds to machine learning models, orange corresponds to molecular representations, and dark green corresponds to predictions.

that instance by averaging their predictions to get one consensus prediction. This is done to reduce variance from random weight initializations and to improve our final results. We also combine the predictions of different model types in consensus to see if those will possibly yield better results, as we see in Wu[110]. We note that models were trained in sets of 42 instead of sets of 40 or 50 models due to the constraints of our computing architecture. If there are computational concerns, we could also significantly reduce the number of models trained (perhaps to 5 or 10 models), and still obtain similar results. For each data set, we train 42 ST and MT AweGNNs with randomized initial weights and obtain a consensus prediction for both types, labeled ST-network and MT-network respectively. Then, we train one ensemble model for each of the NEARs generated by training the networks. Thus, for random forest, we train 42 models based on the NEARs resulting from the ST models and 42 more on the NEARs resulting from the MT models, in which we get the corresponding consensus predictions, ST-RF and MT-RF. Similarly, we

get predictions from the GBT models, ST-GBT and MT-GBT. Finally, we obtain various consensus predictions by combining the previous predictions mentioned above. First, we average the prediction ST-network and MT-network to get the consensus, network-consensus. Next, we combine ST-network and ST-GBT to get ST-consensus. Similarly, we obtain MT-consensus by combining MT-network and MT-GBT. Finally, we get the prediction, final-consensus by taking the consensus between ST-consensus and MT-consensus. These predictions and their performances on the test sets are recorded in the tables in section 5.3.2, and a summary of this entire process is depicted in Figure 5.6.

5.3.1 Evaluation Metrics

A protocol was proposed by Golbraikh et al.[38] to determine if a QSAR model has true predictive power, as summarized in the following four points:

1. $q^2 > 0.5$
2. $R^2 > 0.6$
3. $\frac{R^2 - R_0^2}{R^2} < 0.1$
4. $0.85 \leq k \leq 1.15$

Where q^2 is the square of the leave one out correlation coefficient for the training set, R^2 is the square of the Pearson correlation coefficient between the experimental and predicted toxicity endpoints of the test set, R_0^2 is the square of the correlation coefficient between the experimental and predicted toxicity for the test set in which the intercept is set to zero for a linear regression performed on the predicted and experimental labels given by X and Y respectively so that the regression is given by $Y = kX$ (from which the k value is extrapolated).

For each numerical experiment involving toxicity, we record the metrics above, although we omit the q^2 coefficient due to the computational cost. We also record the root-

mean-squared error (RMSE) and the mean-absolute error (MAE) since these are standard metrics for evaluating the performance of a model. The final metric to report is coverage, or the fraction of chemicals predicted, which is important because a lower coverage can unfairly improve prediction accuracy. For the solvation data sets, we only report the MAE, RMSE, and the R^2 score as in Nguyen et al.[81]

5.3.2 Toxicity Data Sets

This paper analyzes 4 different quantitative toxicity data sets[64]. These are the 96h fathead minnow LC_{50} data set, the 48h Daphnia magna LC_{50}-DM data set, the 40h Tetraphymena pyriformis IGC_{50} data set, and the oral rat LD_{50} data set. The LC_{50} (LC_{50}-DM) set report the concentration in milligrams per liter of test chemicals placed in water that it takes to cause 50% of fathead minnows (Daphnia magna) to die after 96(48) hours. These were downloaded from the ECOTOX aquatic toxicity database via the web site e http://cfpub.epa.gov/ecotox/ and were preprocessed using filter criterion including media type, test location, etc. The IGC_{50} data set measures the 50% growth inhibitory concentration of the Tetrahymena pyriformis organism after 40 hours and was obtained by Schultz and co-workers[3, 113]. The last data set, LD_{50}, represents the concentration of chemicals that kill half of rats when orally ingested. This data set was constructed from the ChemIDplus database (http://chem.sis.nlm.nih. gov/chemidplus/chemidheavy.jsp) and then filtered according to several criteria[64].

The final sets in this work are identical to the sets preprocessed to develop the Toxicity Estimation Software Tool (TEST)[64]. The 2D sdf format molecular structures and toxicity endpoints are available on the TEST website. 3D mol2 format molecular structures were created with the Schrödinger software in an earlier work[110]. We should note that the units of the toxicity endpoints are not uniform between the data sets. The LD_{50} set endpoints are in -$\log_{10}(T \text{ mol/kg})$ while the endpoints of the remaining sets are in units of -$\log_{10}(T \text{ mol/L})$. No attempt has been made to rescale the values. Finally, the data sets

all have differing sizes and compositions, and so the effectiveness of our method varies greatly between them.

Statistics of each data set are detailed below in Table 5.1. Numbers inside parentheses indicate the actual number of molecules that were used for training and evaluating models. The first 3 data sets include all available molecules, but for the last data set (LD_{50}), some molecules were dropped out due to force field failures when applying the Schrödinger software. Despite this, our coverage is greater than any of the TEST models and so is more widely applicable in use.

Table 5.1: Set statistics for quantitative toxicity data

data set	# of molecules	train set size	test set size	max value	min value
LC_{50}-DM	353	283	70	10.064	0.117
LC_{50}	823	659	164	9.261	0.037
IGC_{50}	1792	1434	358	6.36	0.334
LD_{50}	7413 (7398)	5931 (5919)	1482 (1479)	7.201	0.291

5.3.2.1 LC_{50}-DM (Daphnia Magna) Set

The *Daphnia magna* LC_{50} set is the smallest data set with 283 molecules in the training set and 70 molecules in the test set. Given the small size of the data set, it can be difficult to train robust QSAR models, thus multi-scale models are extremely important for obtaining reasonable results. Table 5.2 shows the results of various QSAR models on the LC50DM data set. The TEST consensus had the highest R^2 score ($R^2 = 0.739$) out of all the models shown, although Wu et al.[110] reported a much higher R^2 score of 0.788, which does not appear in the table, when only using topological descriptors for a multi-task model. This high result with fewer descriptors may be due to the nature of neural networks to overfit when trained on small data sets and many descriptors. When comparing the result of Wu-MT as reported in the table (using all descriptors) to the TEST consensus, we notice that although the R^2 score was lower, the RMSE and MAE are better. The group contribution

scored exceptionally well in RMSE and MAE, but doubt was cast on the accuracy of those results[110].

The performance of our ensemble models was fairly poor in comparison to the results from Wu. Our RF-ST and RF-MT models scored 0.439 and 0.443 respectively vs. the 0.460 R^2 score achieved by Wu-RF. Our MT-GBT model was a bit better in performance than the Wu-RF model, but it could not beat the Wu-GBT model. We do see however that our MT-GBT model had outperformed the ST-GBT model noticeably, going from an R^2 score of 0.457 to 0.471, showing a slight benefit from using the MT-NEAR for GBT models. As for the network models, we see a descent show for our ST-network model vs. the Wu-ST model. Although ST-network has an R^2 score of 0.448 vs. the 0.459 for Wu-ST, we see that the ST-network RMSE of 1.315 beats quite decisively the RMSE of 1.407 for Wu-ST. Our MT-network model had unfortunately performed much worse than the Wu-MT model, even though its results are comparable with many of the TEST models reported. The Wu-MT model R^2 score of 0.726 is significantly higher than the score of 0.664 from MT-network. All of our other models do not do well enough to the comment on.

5.3.2.2 Fathead Minnow LC$_{50}$ Set

The fathead minnow LC$_{50}$ set was randomly divided into a training (80% of the entire set) and a test set (20% of the entire set)[64]. Table 5.3 shows the performance of all of the various models trained and tested on the LC50 data. This is the second smallest data set that we are analyzing in this work. The best TEST model is again the TEST consensus, at an R^2 score of 0.728. Notice that this is lower than the previous TEST consensus on the LC$_{50}$ at an R^2 of 0.739, although the coverage increase from 0.900 to 0.951. For the Wu models, the best result is a whopping 0.789 R^2 score for Wu-consensus. Wu-consensus also boasts the best overall RMSE and MAE. The other 2 high scoring models from that category are Wu-MT and Wu-GBT, which come at R^2 scores of 0.769 and 0.761, respectively.

Our ensemble models do comparably well when compared to the TEST models, except

Table 5.2: Comparison of prediction results for the LC50DM test set

model	R^2	$\frac{R^2-R_0^2}{R^2}$	k	RMSE	MAE	coverage
Results with TEST models						
hierarchical[64]	0.695	0.151	0.981	0.979	0.757	0.886
single model[64]	0.697	0.152	1.002	0.993	0.772	0.871
FDA[64]	0.565	0.257	0.987	1.190	0.909	0.900
group contribution[64]	0.671	0.049	0.999	**0.803**	0.620	0.657
nearest neighbor[64]	0.733	0.014	1.015	0.975	0.745	0.871
TEST consensus[64]	**0.739**	0.118	1.001	0.911	0.727	0.900
Results with previous methods from our group						
Wu-RF[110]	0.460	1.244	0.955	1.274	0.958	1.000
Wu-GBT[110]	0.505	0.448	0.961	1.235	0.905	1.000
Wu-ST[110]	0.459	0.278	0.933	1.407	1.004	1.000
Wu-MT[110]	0.726	0.003	1.017	0.905	**0.590**	1.000
Wu-consensus[110]	0.678	0.282	0.953	0.978	0.714	1.000
Ensemble models						
ST-RF	0.439	0.014	0.956	1.312	0.983	1.000
ST-GBT	0.457	0.004	0.966	1.280	0.954	1.000
MT-RF	0.443	0.012	0.960	1.304	0.985	1.000
MT-GBT	0.471	0.003	0.970	1.261	0.953	1.000
AweGNN models						
ST-network	0.448	0.060	0.959	1.315	0.959	1.000
MT-network	0.664	0.000	0.983	1.002	0.741	1.000
Consensus models						
network-consensus	0.583	0.005	0.984	1.112	0.826	1.000
ST-consensus	0.465	0.015	0.933	1.272	0.933	1.000
MT-consensus	0.602	0.000	0.981	1.090	0.820	1.000
final-consensus	0.541	0.001	0.979	1.171	0.872	1.000

for the TEST consensus. Our ST-network model did similarly in performance to many of the TEST models, notably FDA, group contribution, and nearest neighbor. Our best model, MT-network, with an R^2 score of 0.749 and RMSE of 0.735 beats all TEST models, and all of our consensus models do equal to or better than the TEST consensus model, except for the ST-consensus, which was weighed down by the oddly poor performance of the ST-GBT model (ST-GBT performed more poorly than did the ST-RF model, with an R^2 score of 0.687 vs. 0.697).

In comparison to the Wu models, we notice that all of our ensemble models are far behind in performance to either Wu-RF or Wu-GBT, with our best R^2 score of 0.706 from our MT-GBT model being overtaken by the lowest ensemble score of 0.727 from the Wu-RF model. The performance of our ST-network model is comparable to Wu-ST with an R^2 of 0.682 to 0.692, with even closer RMSE and MAE scores. As far as the MT models

go, we see a much greater difference between MT-network and Wu-MT. We cannot come close to the R^2 score of 0.789 from the Wu-MT model, which is the best of the models from Wu for this data set[110].

Table 5.3: Comparison of prediction results for the LC50 test set

model	R^2	$\frac{R^2-R_0^2}{R^2}$	k	RMSE	MAE	coverage
Results with TEST models						
hierarchical[64]	0.710	0.075	0.966	0.810	0.574	0.951
single model[64]	0.704	0.134	0.960	0.803	0.605	0.945
FDA[64]	0.626	0.113	0.985	0.915	0.656	0.945
group contribution[64]	0.686	0.123	0.949	0.810	0.578	0.872
nearest neighbor[64]	0.667	0.080	1.001	0.876	0.649	0.939
TEST consensus[64]	0.728	0.121	0.969	0.768	0.545	0.951
Results with previous methods from our group						
Wu-RF[110]	0.727	0.322	0.948	0.782	0.564	1.000
Wu-GBT[110]	0.761	0.102	0.959	0.719	0.496	1.000
Wu-ST[110]	0.692	0.010	0.997	0.822	0.568	1.000
Wu-MT[110]	0.769	0.009	1.014	0.716	0.466	1.000
Wu-consensus[110]	**0.789**	0.076	0.959	**0.677**	**0.446**	1.000
Ensemble models						
ST-RF	0.697	0.028	1.018	0.836	0.589	1.000
ST-GBT	0.687	0.000	0.995	0.820	0.562	1.000
MT-RF	0.703	0.029	1.019	0.829	0.582	1.000
MT-GBT	0.706	0.001	0.998	0.795	0.543	1.000
AweGNN models						
ST-network	0.682	0.003	0.987	0.830	0.566	1.000
MT-network	**0.749**	0.000	0.999	**0.735**	**0.481**	1.000
Consensus models						
network-consensus	0.739	0.000	0.996	0.748	0.505	1.000
ST-consensus	0.711	0.000	0.994	0.788	0.540	1.000
MT-consensus	0.747	0.001	1.001	0.739	0.494	1.000
final-consensus	0.737	0.001	0.998	0.753	0.507	1.000

5.3.2.3 Tetraphymena Pyriformis IGC$_{50}$ Set

The IGC$_{50}$ data set is the second largest of the data sets we are analyzing. The diversity of the molecules is relatively low compared to the other data sets, which allows for more coverage in the TEST models. The amount of data points in the set is large enough to train robust models, which translates into the high R^2 scores for the models that are compared in this section. We notice that the TEST models are far more variant in their results than in the previous 2 data sets, with R^2 scores ranging from 0.600 to 0.764. Again, the TEST consensus gets the highest R^2 score (0.764). Among the Wu models, the Wu-consensus

model is again the supreme champion with an R^2 score of 0.802. As usual, the Wu-MT model also did very well, trumping all TEST models with respect to every metric. We also introduce the best IGC_{50} model from the work done by Nguyen et al.[81] in which GBT models were trained on representations derived from differential geometry based descriptors. Though the model was trained without help from other data sets, it was able to defeat the Wu-MT model in all metrics. The GBT Nguyen-best model is however narrowly defeated by the Wu-GBT model with an R^2 score of 0.781 compared to 0.787.

Our ensemble models are relatively low in comparison to the performance of the rest of the models, although our MT-GBT model does perform better than all TEST models except for the TEST consensus model. Our ST-network model outperforms all TEST models single-handedly with an R^2 score of 0.778. All of our remaining models outperform our own ST-network model, and thus also every TEST model as well.

Our ensemble models under-perform yet again in comparison to the RF and GBT models set forth by Wu. When comparing our network models, we see that our ST-network model strongly defeats the Wu-ST model with an R^2 score of 0.749 and even outperforms the Wu-MT model with an R^2 of 0.770. ST-network is, however, beaten by Wu-consensus, and even by the model put forth by Nguyen, that is, Nguyen-best. Our MT-network model is the best scoring model in the whole table in all metrics except for MAE, with an R^2 score of 0.803, RMSE of 0.436, and MAE of 0.310. Only Wu-consensus has a lower MAE of 0.305, but that MAE is beaten by our model, network-consensus, with an MAE of 0.304. Although MT-network technically performs slightly better than Wu-consensus, they are effectively identical in performance.

5.3.2.4 Oral Rat LD_{50} Set

The oral rat LD_{50} set contains the most molecules. The data set is quite large (7413 molecular compounds), so naturally full coverage is not available for any of the methods proposed, although most models still have very high coverage. The labels of this data set

Table 5.4: Comparison of prediction results for the IGC50 test set

model	R^2	$\frac{R^2-R_0^2}{R^2}$	k	RMSE	MAE	coverage
Results with TEST models						
hierarchical[110]	0.719	0.023	0.978	0.539	0.358	0.933
FDA[110]	0.747	0.056	0.988	0.489	0.337	0.978
group contribution[110]	0.682	0.065	0.994	0.575	0.411	0.955
nearest neighbor[110]	0.600	0.170	0.976	0.638	0.451	0.986
TEST consensus[110]	0.764	0.065	0.983	0.475	0.332	0.983
Results with previous methods from our group						
Wu-RF[110]	0.736	0.235	0.981	0.510	0.368	1.000
Wu-GBT[110]	0.787	0.054	0.993	0.455	0.316	1.000
Wu-ST[110]	0.749	0.019	0.982	0.506	0.339	1.000
Wu-MT[110]	0.770	0.000	1.001	0.472	0.331	1.000
Wu-consensus[110]	**0.802**	0.066	0.987	**0.438**	**0.305**	1.000
Nguyen-best[81]	0.781	0.004	1.003	0.463	0.324	1.000
Ensemble models						
ST-RF	0.713	0.013	1.006	0.535	0.377	1.000
ST-GBT	0.745	0.000	0.994	0.496	0.329	1.000
MT-RF	0.716	0.013	1.006	0.532	0.377	1.000
MT-GBT	0.753	0.000	0.995	0.489	0.327	1.000
AweGNN models						
ST-network	0.778	0.000	1.003	0.463	0.309	1.000
MT-network	**0.803**	0.000	0.999	**0.436**	0.310	1.000
Consensus models						
network-consensus	0.799	0.000	1.001	0.440	**0.304**	1.000
ST-consensus	0.777	0.000	1.000	0.464	0.309	1.000
MT-consensus	0.795	0.000	0.998	0.445	0.305	1.000
final-consensus	0.789	0.000	0.999	0.451	0.306	1.000

are quite difficult to predict because of the high experimental uncertainty in obtaining the toxicity endpoints, as noted in Zhu et al.[114]. Table 5.5 shows the results. As in Wu et al.[110], we omit the single model and group contribution TEST methods from the table in our analysis. As always, the TEST consensus provides the greatest results amongst the TEST models. For this data set, the effectiveness of the Wu models wanes, with the Wu-MT model failing to perform decisively better than the TEST consensus (although, Wu-MT does indeed do slightly better in RMSE and MAE in comparison to TEST consensus).

For this data set, we see the best performance from our models. Even our ensemble models do quite well. In fact, our ST-GBT and MT-GBT models, with respective R^2 scores of 0.643 and 0.641, outperform any TEST model and any Wu models aside from the Wu-consensus model. All of our non-ensemble models out-perform every single other model, including Wu-consensus, with our best performing model, final-consensus, having an R^2

of 0.667, RMSE of 0.557, and MAE of 0.405. This decisively beats the Wu-consensus model having an R^2 score of 0.653, RMSE of 0.568, and MAE of 0.421.

We notice that for this data set, we finally required the consensus of the GBT models to get our best prediction, as opposed to the other data sets in which we obtained the best result from our MT-network models. Our AweGNN models seem to shine relative to other groups models when the data sets are the largest, in which the networks are able to generate their most generalizable representations, and the predictions are the most accurate.

Table 5.5: Comparison of prediction results for the LD50 test set

model	R^2	$\frac{R^2-R_0^2}{R^2}$	k	RMSE	MAE	coverage
Results with TEST models						
hierarchical[64]	0.578	0.184	0.969	0.650	0.460	0.876
FDA[64]	0.557	0.238	0.953	0.657	0.474	0.984
nearest neighbor[64]	0.557	0.243	0.961	0.656	0.477	0.993
TEST consensus[64]	0.626	0.235	0.959	0.594	0.431	0.984
Results with previous methods from our group						
Wu-RF[110]	0.619	0.728	0.949	0.603	0.452	0.997
Wu-GBT[110]	0.630	0.328	0.960	0.586	0.441	0.997
Wu-ST[110]	0.614	0.006	0.991	0.601	0.436	0.997
Wu-MT[110]	0.626	0.002	0.995	0.590	0.430	0.997
Wu-consensus[110]	**0.653**	0.306	0.959	**0.568**	**0.421**	0.997
Ensemble models						
ST-RF	0.606	0.013	1.003	0.612	0.452	0.998
ST-GBT	0.643	0.002	0.995	0.578	0.424	0.998
MT-RF	0.596	0.012	1.003	0.619	0.456	0.998
MT-GBT	0.641	0.002	0.995	0.580	0.426	0.998
AweGNN models						
ST-network	0.660	0.001	0.995	0.563	0.406	0.998
MT-network	0.658	0.001	0.993	0.565	0.411	0.998
Consensus models						
network-consensus	0.665	0.000	0.995	0.558	**0.404**	0.998
ST-consensus	0.665	0.001	0.997	0.559	0.406	0.998
MT-consensus	0.664	0.001	0.996	0.560	0.409	0.998
final-consensus	**0.667**	0.001	0.997	**0.557**	0.405	0.998

5.3.3 Solvation Data Set

In this section, we explore the results of the solvation data set, model III. Model III has a total of 387 molecules (excluding ions), and is split into a training set of 293 molecules and a test set of 94 molecules. We use the same training set, but we omit molecules based

on obscure chemical names in the PubChem database and due to some difficulties with the Schrödinger software in generating mol2 files. The test set is unaltered, so these difficulties leading to our smaller training set of 280 molecules should disfavor our method. In addition to this, there is only one data set for solvation that we analyze, so we cannot apply our MT method. We simply train ST models and report one consensus with the GBT model and the network model, following the same procedure as before. The results are shown below in Table 5.6, where they are compared to other models mentioned previously[81, 106, 104].

Table 5.6: Comparison of prediction results for the solvation test set

model	MAE (kcal/mol)	RMSE (kcal/mol)	R^2
Results from outside sources			
WSAS[106]	0.66	-	-
FFT[104]	0.57	-	-
Results from Nguyen et al.[81]			
$\text{EIC}^H_{E,3.5,0.3}$	0.575	0.921	0.904
$\text{EIC}^H_{E,3.5,0.3;E,2.5,1.3}$	$\mathbf{0.558}$	$\mathbf{0.857}$	$\mathbf{0.920}$
$\text{EIC}^H_{L,3,1.3}$	0.592	0.931	0.906
$\text{EIC}^H_{L,3,1.3;L,6.5,0.3}$	0.608	0.919	0.907
Consensus^H	0.567	0.862	$\mathbf{0.920}$
Ensemble models			
ST-RF	0.698	1.001	0.893
ST-GBT	0.496	0.666	0.951
AweGNN model			
ST-network	$\mathbf{0.373}$	$\mathbf{0.569}$	$\mathbf{0.963}$
Consensus model			
ST-consensus	0.401	0.583	0.962

We see that in this instance, we get our best performance relative to other group's models. This is quite surprising, as we found that with the toxicity data, the AweGNN seemed to perform very well only when the data sets were very large. Our greatest model, the ST-network model, significantly outperforms every other model in every metric. Even our GBT model is able to greatly outperform all other models outside this group, though our RF model performs only satisfactory. In fact, the supplementary material from Nguyen et al. [81] contains models whose kernel parameters were optimized specifically on the test data, giving a further advantage. These can be seen in Table 5.7. Again, we see that even out GBT model is able to win in every metric against every model in this table, let alone

our ST-network with an MAE gap of 0.145, an RMSE gap of 0.229, and an R^2 score gap of 0.032 when compare to the best metrics of any chosen opposing model. This analysis shows that there is great promise for applying the AweGNN method for solvation free energy prediction.

Table 5.7: Supplementary results for solvation data set from Nguyen et al.[81]

Method	MAE (kcal/mol)	RMSE (kcal/mol)	R^2
*$\text{EIC}^{H}_{E,3.5,0.3}$	0.575	0.921	0.904
*$\text{EIC}^{HH}_{E,3.5,0.3;E,4.0,1.3}$	**0.518**	0.812	0.929
*$\text{EIC}^{H}_{L,5,0.3}$	0.579	0.862	0.917
*$\text{EIC}^{HH}_{L,5,0.3;L,0.5,0.9}$	0.559	0.842	0.922
*Consensus^{H}	0.524	**0.798**	**0.931**

5.4 Discussion

In this section, will we discuss the impact of automated parameter selection, analyze some elements of feature importance, and discuss a new paradigm of auto-parametrized kernel-based networks that could lead to many more possibilities.

5.4.1 Impact of Automating Selection of Kernel Parameters

When selecting kernel parameters to optimize the choice of representation, our group was previously using the grid search method to determine which parameters were optimal[79]. This is not so much of a hindrance if there are only a handful of parameters to tune, but as the number of parameters to tune increases, the number of models that need to be trained increases exponentially. In addition, grid searches require a discrete set of parameters to experiment with, which is a coarse way to tune parameters and leads to some inefficiency. These problems can severely restrict the potential of these kernel-based representations, and largely restrict us to the use of machine learning algorithms that do not have many tunable parameters, such as RF or GBT.

One of our goals in this work was to alleviate these issues. Automatically updating kernel parameters is a great solution in theory and in practice, as we have seen with the

success of our AweGNN in this work. The AweGNN seeks the optimal choice of parameters within a continuous space, so the parameter selection is done in a smooth way and can select the values that are not present in a grid search. In our case, we incorporate the parameter selection into the gradient descent process of training a neural network. This association with the gradient descent allows us to update very many kernel parameters at the same time as the weights of the neural network, but has a moderate computational cost associated with it that is commensurate with the computational cost of a moderately sized network. More specifically, a quick analysis on the IGC_{50} data shows that a network consisting of the GGR layer (producing 400 features) with normalization takes the same amount of time on average to cycle though an epoch as does the network with the GGR+normalization layer along with a 4-layer network of 1600 neurons each, thus the GGR+normalization layer is equivalent to a 4-layer 1600 neuron network in terms of computational cost. This cost is certainly not prohibitive, and ultimately saves us an inordinate amount of time, especially since we are able to tune 200 kernel parameters simultaneously (which would be impossible with a grid search).

As kernel methods require careful tuning to reach their full potential, neural networks were effectively off limits for training effective models with kernel-based representations. The AweGNN proves that kernel-based representations can be used to train high performing models and brings many new possibilities for developing excellent predictive models for biomolecular data.

5.4.2 Feature Importance and Analyzing the Trajectories of the Kernel Parameters

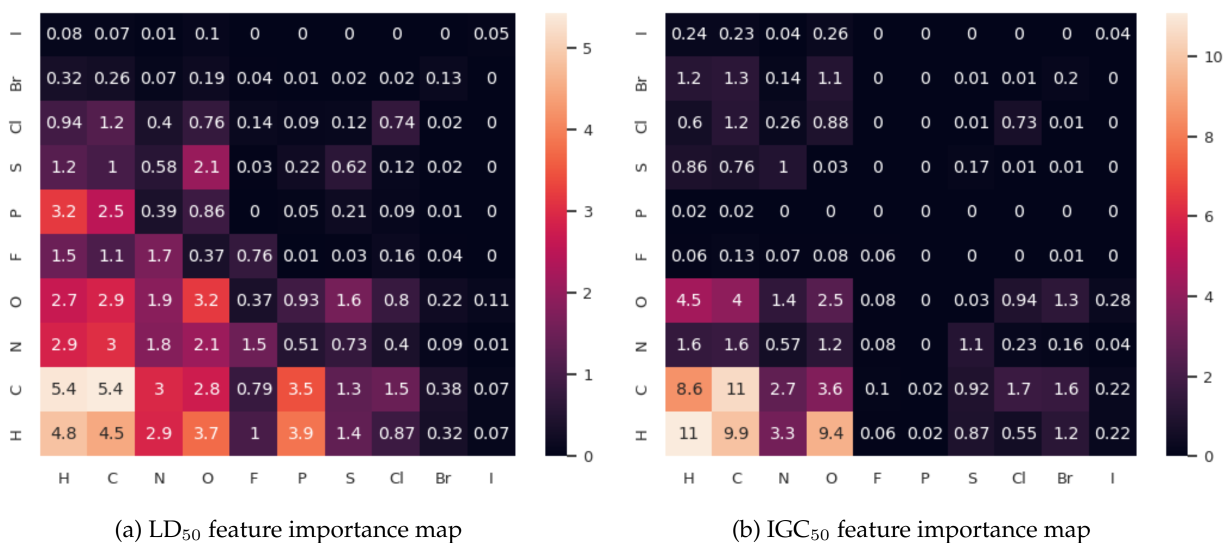
To find the most important features that influenced our predictions, we look at the feature importance ranking of the random forest (RF) models that were trained on the network-enabled automatic representations (NEARs) generated by the AweGNNs. This analysis will pinpoint the specific element-pair interactions that contributed the most to the creation of our predictive models. Then, we can analyze the average trajectories and final

states of the kernel functions from the element-specific groups that generate the most important features. The average η values can tell us about the most beneficial interaction distances between those element pairs, and the average κ values can tell us about the most favorable degree of sharpness of the cutoff for including the edge cases centered around those η values.

By closely examining the feature importance over all data sets and all models, we notice that, in general, the most important interactions tend to be the ones between hydrogen-hydrogen, carbon-carbon, and the interaction between those two elements. This would make sense, since the carbons and hydrogens are the most numerous elements in these structures and will be interacting most with the environment. The second most important features are generated from the interactions between the hydrogen-oxygen and carbon-oxygen pairs. This again seems to be a natural result, for this can be indicating the importance of the polar bonds. Aside from these groups, however, there seems to be no broader pattern. Different data sets seem to prioritize different element specific groups, especially the largest data set, LD_{50}, in which there are a great diversity of element pair interactions that are important, as can be seen in Figure 5.7a, which illustrates the feature importance associated with each pair through a heat map.

A more contained display of the feature importance is portrayed in figure 5.7b, based on the RF models trained to predict the toxicity for the IGC_{50} data set with the NEARs generated from the multi-task (MT) AweGNN models. The feature importance values are concentrated within the pairs consisting of the hydrogen, carbon, nitrogen, and oxygen elements. For the complete list of feature importance heat maps of all the data sets, including feature importance measured by the gradient boosting models, you may reference the Supporting information, Figures S1-S16.

To go into more detail of what is happening in each element-specific group, we analyze how the parameters of the kernel functions of the top 10 groups evolve on average. This evolution can be seen in Figure 5.8.



(a) LD_{50} feature importance map

(b) IGC_{50} feature importance map

Figure 5.7: Feature importance of MT-AweGNNs when applied to different data sets. We generate heat maps based on the average of feature importance across 42 RF models trained on the corresponding NEARs of the MT-AweGNNs. The average feature importance of the 4 features of each element-specific group are summed together to get the final scores shown above in the maps. Figure 5.7a shows the results for the LD_{50} MT-RF models and figure 5.7b shows the results for the IGC_{50} MT-RF models. Note that the maps are asymmetric due to the way that the electrostatics-based features of a group are generated.

As the training progresses, element pairs have their kernel parameters tend towards various η-κ combinations. Some element pairs have high η values and moderate κ values, such as the H-H and C-N groups, while others tend towards low η and low κ values (i.e, H-O), or even low η and high κ (O-C). There are many physical interpretations that could be valid, for example, the low η and κ value for the H-O group could be a measure of the potential for hydrogen bond interactions with the environment, or even internal short-range interactions that assess the stability of the molecule. In any case, we may note the convergence of the η values especially for they might reveal important cutoff distances that contribute to the measurement of toxicity.

5.4.3 Limitations and Advantages

One of the limitations present in the study is with regards to applying the AweGNN to very large molecules or systems of molecules. If too many atoms are involved in the

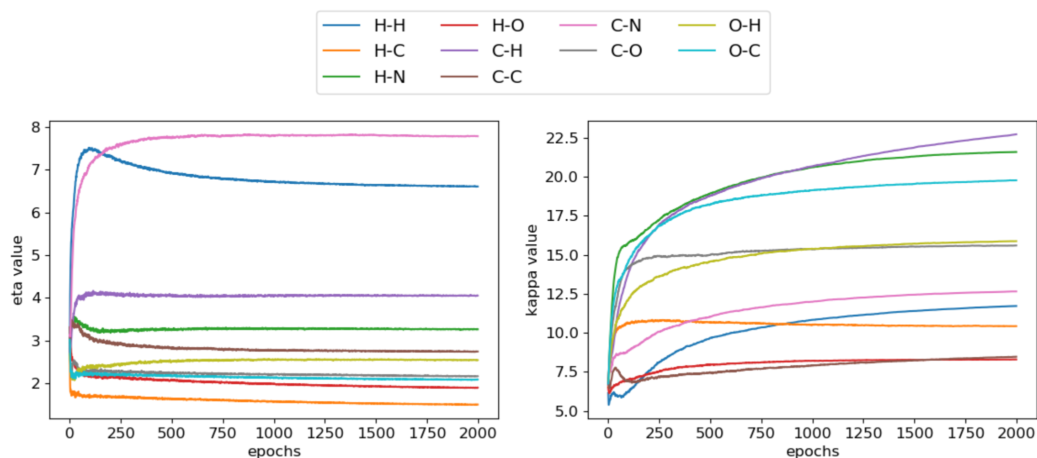


Figure 5.8: Average trajectories of kernel parameters for 42 MT-AweGNN models. The graph on the left shows how the average of the η values of our 42 MT models for 8 different element-specific groups changes as the models are trained. On the right, we see the κ counterpart of this analysis.

calculation of these features, then the computational cost may become unbearable. Some provisions or rules can be made to avoid heavy computational costs while dealing with large scale biomolecular structures or systems of molecules. For example, our method may be naturally extended to the problem of protein-ligand binding. We can measure the interaction strength between atoms of specific element type in the protein and atoms of another element type in the ligand, instead of the interactions between the atoms of two element types contained within a single molecule. A standard cutoff distance from the ligand is applied to focus on the strongest interactions, thereby leaving a reduced number of atoms for the analysis. A cutoff that is very large might include too many atoms from the protein and thereby incur a heavy computational cost that is far beyond that of the small molecular data sets that we have used in this study. Small cutoffs can be feasible and perhaps still quite effective since long-range interactions may not carry as much weight in terms of performance.

The advantage of the AweGNN is its ability to shape the kernel function for each

element-specific group so as to provide the optimal filter for capturing the most important interactions between elements within a molecule. Though there is much potential in this approach, there is a danger of over-fitting with smaller data sets. This is perhaps why we see the greatest performance with the largest LD_{50} data set, and so practically we may want to train on the largest data sets that we can. However, we notice also that the AweGNN was incredibly successful with the small solvation energy data set; and for the two smallest data sets, LC_{50}-DM and LC_{50}, the results for the single task network are similar to the RF models, which are known to be quite robust against over-fitting. Thus, the danger of over-fitting may not be as present as originally thought. Also, it may be the case that the multi-task method is less effective when applying the AweGNN, for it may fit its features too specifically to a particular task, and therefore be influenced by the larger data sets more so than other methods, like Wu's[110].

The computational cost associated with training our multi-task (MT) AweGNN model, which is our largest computational endeavor, is under 135 minutes. The largest single task model (LD_{50} model), takes under 120 minutes, while the second largest IGC_{50} data set is trained in approximately 20 minutes, with the training time for the smaller data sets being significantly lower (a handful of minutes). While the training time of our MT-AweGNN takes almost 70% longer than, for instance, the MT-Wu model (80 minute training time)[110], we can see that the MT AweGNN can still perform better on the larger data sets, so the 70% increase in training time may be worth the performance boost when provided with more samples. In addition to this, we note that our features are simple kernel-based features, and do not require auxiliary physical descriptors to achieve stellar performance. The AweGNN can in fact use those same auxiliary features to bolster its own predictions, and is more flexible in that it can also combine multiple kernel types to generate multi-scale representations that been highly successful in previous works[78, 81]. Finally, as our features are kernel-based, we noted in section 5.4.1 that these features would usually require a grid-search to optimize the kernel parameters, but in our case,

we are able to automate this process while simultaneously training our network, thereby saving time in that respect.

5.5 Conclusion

Recent years have witnessed much effort in developing mathematical representations for the machine learning predictions of chemical and biological properties that are crucial to drug discovery. Advanced mathematical tools from fields such as graph theory [78][80], differential geometry[81], algebraic topology[110], and other mathematical area, have been developed and demonstrated their superb performance. Many of these representations were generated with a choice of kernel functions that depend on a choice of parameters. These kernel-based molecular representations must be fine tuned to optimize their effectiveness for training machine learning models. Increasing parameter choices lead to a fast increasing in the computational cost of optimization. This problem deteriorates in the case of deep neural networks because of the already cumbersome task of choosing network hyperparameters. Motivated by the automated feature extraction in convolutional neural network (CNN), we propose an auto-parametrized element-specific graph neural network (AweGNN) to automate and optimize the parameter selection in our geometric graph approach. The resulting representation from training the AweGNN is called a network-enabled automatic representation (NEAR). NEARs can be used as input features for other machine learning models, such as random forest (RF) and gradient-boosting tree (GBT) models.

AweGNN and NEAR-based ensemble methods are validated five data sets from quantitative toxicity and solvation predictions. Both toxicity and solvation are important for lead hit, and structure optimization in drug discovery. Four quantitative toxicity data sets: 96 h fathead minnow LC_{50}, 48 h *Daphnia Magna* LC_{50} data set (or LC_{50}-DM), 40 h Tetrahymena pyriformis IGC_{50} data set, and the oral rat LD_{50} data set were used in our work. Our models were compared to state-of-the-art models in various literature, i.e.,

the Toxicity Estimation Software Tool (TEST)[64] listed by the United States Environmental Protection Agency (EPA), a previous work by Wu et al.[110], and another work by Nguyen et al.[81]. Given that we had 4 data sets of various sizes with similar prediction task, we were able to employ the multi-task (MT) learning method to greatly improve our performance. Our top models were able to out-compete all TEST models in all but the smallest data sets. When comparing with the top Wu models, our top models were able to perform just as well or better when restricting to the largest 2 data sets, although our general models for the smaller data sets were still able to perform relatively well when compared to many of the earlier models.

To broaden our application, we also tested the AweGNN on the solubility data set, Model III, used in the work by Nguyen al. [81]. For this instance, we were not able to apply MT learning because we only had one data set to work with. Despite this, we were able to greatly outperform any other models that we compared [81, 106, 104]. Although there is a positive correlation between the relative effectiveness of the AweGNN and the data set size with respect to the 4 toxicity data sets, we see that in the case of solubility, the AweGNN can perform exceptionally well even with a very small data set. Our work showcases the impressive predictive capabilities of the AweGNN and ultimately introduces new potential to improve the effectiveness of previous mathematical methods.

CHAPTER 6

GENERAL AUTO-PARAMETRIZED KERNEL METHODS AND FUTURE WORK

6.1 Introduction

In this section we will discuss various extensions of the original AweGNN work and present future possibilities that can advance the auto-parametrized kernel method paradigm.

6.2 GPU-Enhanced AweGNN and Multi-scale Methods

The original AweGNN did not have a geometric graph representation (GGR) layer that was compatible with the graphical processing unit (GPU) technology. This hampered the feasibility of applying the multi-scale kernel methods that have been shown to be wildly successful in the past. Although computational speed was improved though the use of the Cython and Numba packages, the GGR layer did not have the necessary speed to place the computation time within the proper scope. This led to the implementation of the GPU-Enchanced AweGNN which uses a different data structure that allows the GPU to make quick calculations of the element-specific geometric graph features. We will discuss the new structure, algorithm, and performance.

6.2.1 GPU programming

Graphical processing units, or GPUs, have immense computing power due to the thousands of tiny processing cores contained within their architectures. These cores can run thousands of threads in parallel, and so benefits greatly in terms of low computational time when applied to algorithms that rely heavily on computations that can be done simultaneously. GPUs were originally used to display graphics in video games, and later found use in machine learning, especially in application to deep neural networks. Since deep neural networks involve the multiplication of many large matrices and matrix mul-

tiplication involves many operations that can be performed in parallel, GPUs have been extremely beneficial for the advancement of these networks. NVIDIA CUDA programming is the most popular programming model at the moment, although there are many packages that allow various programming languages to interact with the GPU programming interface and improve the ease of use. OpenMP is a popular software that help facilitate GPU programming with the C++ language while Numba and PyCuda are used for GPU programming with the Python programming language.

GPUs launch kernels one at a time, which are GPU tasks that consist of many blocks of computing threads. Threads are individual computations that are completed independent of one another and can be synchronized with other computations occurring within the same block. Threads are scheduled by CUDA in warps of 32 threads which are executed SIMD (single instruction, multiple data) style across all threads in a warp. Blocks usually contain around 1024 threads grouped in 32 warps per block. When programming a GPU, you must be mindful of the way in which you schedule the execution of parallel instructions to achieve optimal speed. For additional efficiency from parallelism, you may want to consider running multiple streams of kernels, which allow multiple whole tasks to be executed in parallel. GPUs are quite flexible despite the restrictions that one may have to work with.

6.2.2 Data Structure and Algorithm

For the original AweGNN, the element-specific biomolecular data, i.e. the distances between the atoms and the electrostatics information was stored in python lists and Numpy matrices which could then be accessed easily for Numba operations. To make use of GPU programming with Numba, the data had to be stored without the use of lists, and so the data was stored only in arrays, with indexing arrays that described the location of distances corresponding to specific element-specific groups of the molecules in the data set.

More precisely, we have a 1-D array detailing the pairwise distances of all the atoms in each element specific group in each sample all in this one array. An electrostatics information array of the same size is also present detailing the electrostatic information relevant to the corresponding position in the array. Two index arrays, again of the same length, indexing the element-specific group in the sequence of the dataset in one and indexing the distances within that element-specific group in the other. With this data structure, we can apply a common parallel computation algorithm that sums up the elements of an array with some slight modifications to then get a sum for all of our element-specific groups at once. This algorithm is illustrated in figure 6.1.

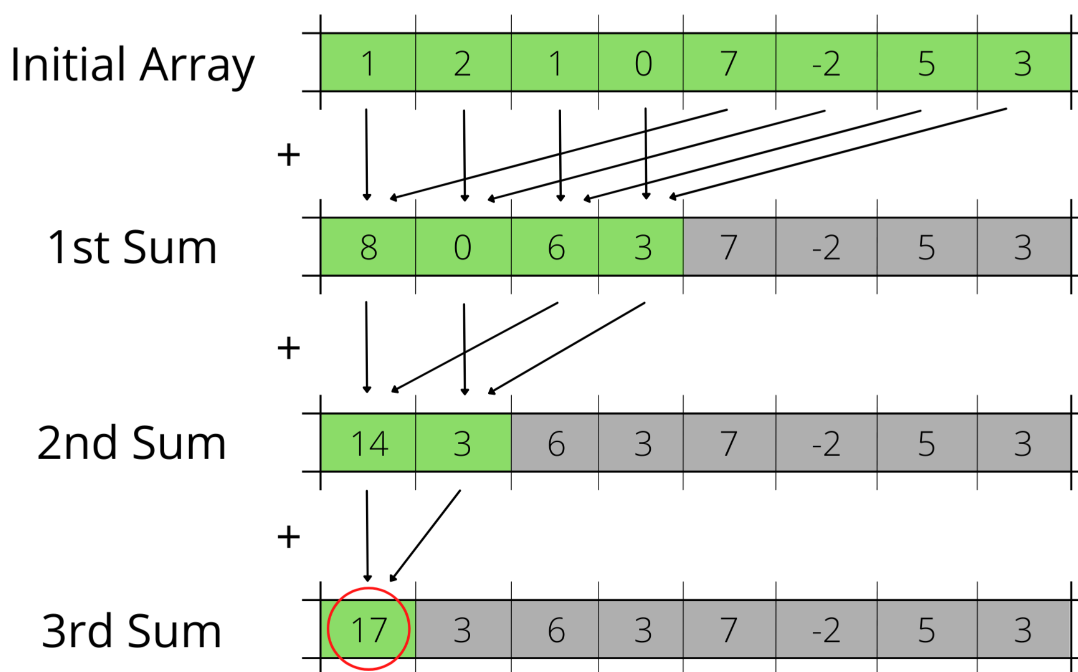


Figure 6.1: This figure gives a graphical representation of the parallel addition algorithm for calculating the sum of an array. We start with an initial array, a, of length, n. Then we begin by doing pairwise additions, $a_i := a_i + a_{2i}$, for $i = 1, ..., n/2$. Then the next step includes pairwise additions, again $a_i := a_i + a_{2i}$, but now for $i = 1, ..., n/4$. The pairwise additions, $a_i := a_i + a_{2i}$, continue for $i = 1, ..., n/2^k$, at step k, until there is only one number remaining to be summed, and then the array sum is extract from the first element, a_0. The image uses green to show elements used in the sum at each step, grey to show inactive elements, and the red circle highlighting the final answer after all the steps are completed.

6.2.3 New Capabilities

In addition to the new algorithm adapted for the GPU, it has been decided that the GGR layer from the AweGNN should output features based on a choice of either distance, electrostatics, average distance, or average electrostatics information for each element-specific group. This is in contrast to the original GGR layer, which would output all 4 categories per element-specific group. This was done to be able to allow the kernels to fit the data based on the different types of features, i.e. a different kernel function may be appropriate for capturing the electrostatic interactions of an element specific group as opposed to capturing the distance data within an element-specific group. We also we have a new kernel that is now operational, namely the piece-wise linear kernel detailed in section 4.4, which has performed well in brief experimentation, as can be seen in the section below. Finally, the multi-scale capabilities are operational due to the increase in speed, and have been briefly tested, again pointing to the section below.

6.3 Comprehensive AweGNN with Fixed Topological and Spectral Descriptors

The GGR layer can perform very well on its own, as it has been shown by the performance of the AweGNN. There is, however, more improvement to be made. The GGR layer lacks descriptors that detail the topological and geometric information of the biomolecules. In addition to that, spectral information can be gleaned from the graph structure inherent in the molecular structure. It would seem reasonable that the combination of the molecular descriptors from all of these categories can produce an effective cumulative model. The descriptors generated by the AweGNN provide element-specific features that measure relative distance and electrostatic interaction strength. Topological descriptors measuring the persistence of rings and cavities present within the molecules gives us a representation of the geometric properties. Finally, the spectral descriptors of the molecular graph can give us information about the connectivity of the molecule among other

geometric information. We seek to create and test a cumulative model that incorporates all these descriptors in a neural network structure.

6.3.1 Model Features and Architecture

We follow a simple architecture that involves the geometric graph representation (GGR) layer from the AweGNN, and one or two auxiliary layers which facilitate the topological and/or spectral descriptors which are then merged into the main part of the network. The idea for this architecture is illustrated in figure 6.2.

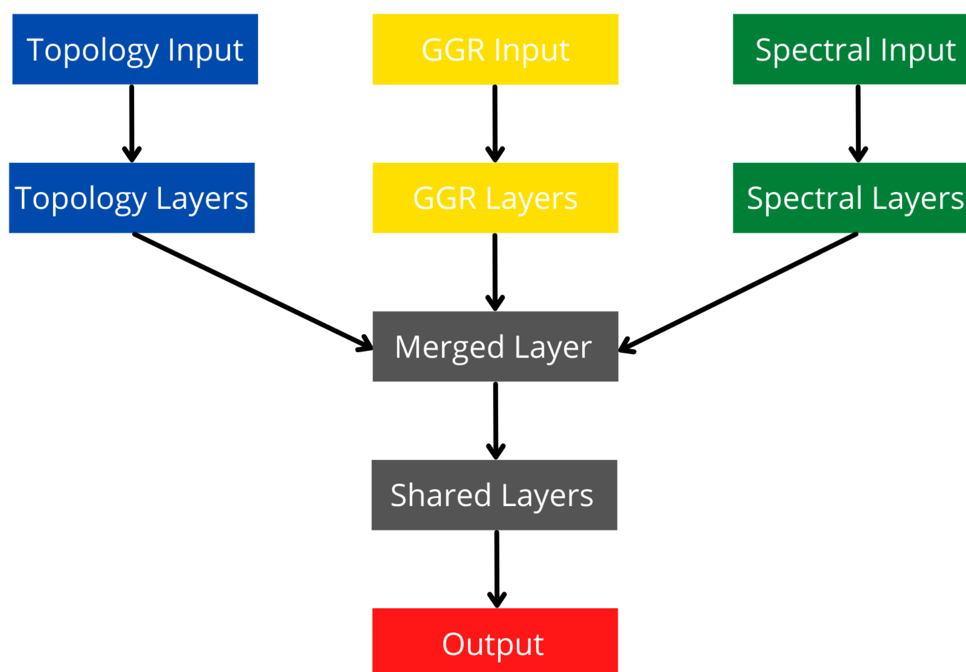


Figure 6.2: This figure shows the general structure of the comprehensive network described in this section. The geometric graph representation (GGR) layer from the AweGNN takes center stage with auxiliary topological and spectral features all merging together into one shared layer. Each set of features can have their own hidden layers to effectively process the features before merging.

In the brief set of experiments that we have using this architecture, we omit the spectral features, and the topological feature input does not have any corresponding hidden layers, nor does the GGR layer. The GGR layer output and the topological features are

concatenated from the beginning. The topological barcode data is calculated using the Ripser software [16] and the descriptors that are generated are done following the procedure as in section 4.5.5, with maximum length of 10 Åand a resolution of 0.25 Å. The GGR layer is a multi-scale layer consisting of 6 kernels types. The 3 kernel types described in section 4.4, the Lorentz, exponential, and piece-wise linear kernel, each with a distance based kernel and a electrostatics based kernel, yielding our 6 kernel types. Each kernel type yields 100 features and 200 kernel parameters, yielding a total of 600 features and 1,200 kernel parameters.

The hidden layers of our network are composed of 6 linear layers of 1024, 512, 256, 128, 64, and 32 neurons in that sequence down to the one output which gives us our prediction. After each hidden layer, we have a batch normalization layer followed by a ReLU activation layer. A constant learning rate of 0.01 with the AMSGrad version of the Adam optimizer with default PyTorch settings ($\beta_1 = 0.9$ and $\beta_2 = 0.999$), batch size of 100 samples, and τ and κ initialization ranges of 0.5-2 and 2-12 respectively were used. These hyper parameters were chosen by following intuitive guidelines and experience from past experiments to make reasonable predictions. Each model was trained for 2000 epochs and 50 models were trained per data set. The 50 predictions were averaged to get a final prediction and the results were recorded.

6.3.2 FreeSolv and ESOL Data Set

We describe 2 data sets from the MoleculeNet paper [111], FreeSolv and ESOL, which we have trained on our comprehensive neural network described in the previous section.

ESOL is a small dataset consisting of water solubility data for 1128 compounds [28]. The dataset has been used to train models that estimate solubility directly from chemical structures (as encoded in SMILES strings) [33, 111]. Although the ESOL dataset originally consisted of 2D smile strings, we used the Schrodinger software to generate mol2 files that contain the 3D structures of the molecules so we could apply our own machine

learning models which require them. The data set is split into a benchmark test set with 113 molecules, while the remaining data points are the training set consisting of 1015 molecules. we trained our models on this training set before testing on the benchmark set.

The Free Solvation Database (FreeSolv) provides experimental and calculated hydration free energy of small molecules in water [73]. A subset of the compounds in the dataset are also used in the SAMPL blind prediction challenge [75]. The calculated values are derived from alchemical free energy calculations using molecular dynamics simulations. The experimental values are included in the benchmark collection, and use calculated values for comparison [111]. This data set contains 642 molecules and has 65 molecules set aside for the benchmark test set. The remaining 577 molecules make up the training set, in which we trained our models before testing on the test set.

6.3.3 Results and Discussion

For both data sets, we consider the standard metrics of the RMSE and the Pearson correlation coefficient described in section 3.2.3. We compare these results to the benchmark test results record in the MoleculeNet paper. Below, we show the charts for the ESOL and FreeSolv datasets in figure 6.3.

For our ESOL experiments, we achieved an RMSE of 0.568 and a Pearson correlation coefficient of 0.966. For the FreeSolv experiments, we achieved an RMSE of 0.762 and a Pearson correlation coefficient of 0.978. The best results for the ESOL and FreeSolv datasets from the MoleculeNet paper are 0.58 and 1.15 RMSE respectively, both coming from their MPNN model. The MPNN mode is also referred to as the message passing neural proposed by Gilmer et al [37]. Our models outperform all of the proposed models in MoleculeNet on both datasets and so are clearly excellent models in and of themselves. The gap is especially large on the FreeSolv dataset, which is more than a 0.42 gap for RMSE. This is quite big considering the scale of the prediction evaluations, in which the

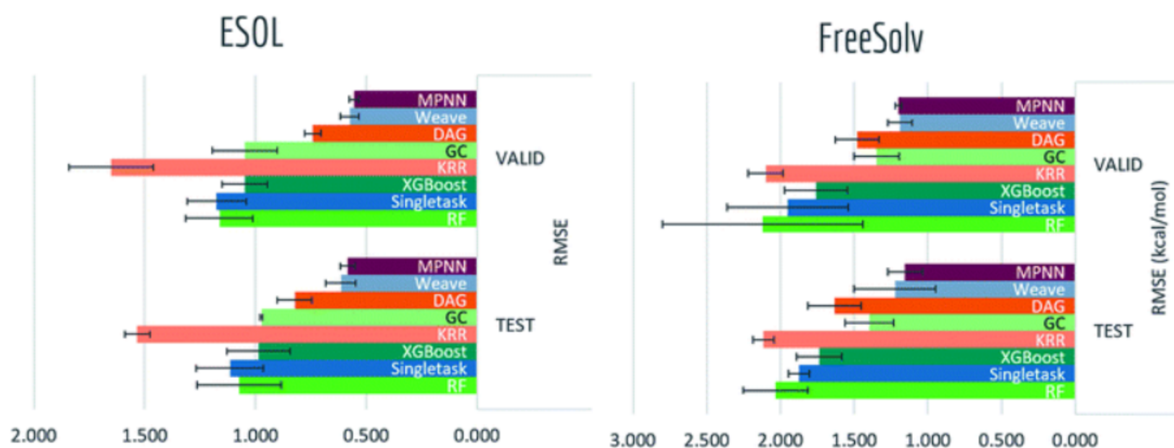


Figure 6.3: This figure displays the results gathered from the MoleculeNet paper. 8 model types were tested on a validation set and a test set and the average performance using the RMSE metric were recorded. The error bars record the standard deviation of the model results [111].

worst predictions are below 2.2 RMSE. In this case, we see another instance in which the GGR layer performs extremely well, just as we have seen stellar performance on the solvation dataset in the original AweGNN paper [100].

Although we can safely say that the comprehensive AweGNN model proposed here is a superb model in comparison, we admit that there are many improvements to be made so that the performance is optimized. First of all, the model architecture could be optimized in a more formal fashion, and there was no feature cleaning or feature importance analysis to clean up the topological data. Second, we did not apply any separated hidden layers to the auxiliary topological features and the GGR layer itself, which would provide opportunity for features to be pre-processed by the layers before being concatenated and sent down through the shared layers. Finally, we did not include the spectral auxiliary features that would give us the completion of our goal of making a truly comprehensive network.

6.4 Auto-parametrized Spectral Layer

We wish to introduce a new layer that has not been tested, but the theory is important for the future development of the technique. We want to create a layer that updates the kernel parameters of a weighted adjacency and Laplacian matrix, which extends the original idea of the AweGNN from a geometric graph layer that obtains descriptors from edge weights to an algebraic graph layer that obtains descriptors from the spectrum of Laplacian and adjacency matrices. This section applies perturbation theory to develop a formula for the derivative of an eigenvalue function with respect to the kernel parameters so that the kernel parameters can be updated throughout the training.

6.4.1 Differentiability of Eigenvalue and Eigenvector Functions

In this section, we use "Perturbation Theory for Linear Operators", by Tosio Kato [49], as a helpful resource. In particular, we have Theorem 6.1 from the book:

Theorem. *If the holomorphic family $T(\chi)$ is symmetric, the eigenvalues $\lambda_h(\chi)$ and the eigenprojections $P_h(\chi)$ are holomorphic on the real axis, whereas the eigennilpotents $D_h(\chi)$ vanish identically.*

In this theorem, the holomorphic family, $T(\chi)$, is a family of symmetric matrices parametrized by χ. The eigenvalues, $\lambda_h(\chi)$, refer to functions on some domain, $D \subset \mathbb{R}$, in $\lambda_h : \mathbb{R} \to \mathbb{R}$ is defined by mapping the value χ to an eigenvalue of $T(\chi)$. In a similar vein, the eigenprojections, $P_h(\chi)$ are functions, $P_h : D \to \mathcal{M}_n(\mathbb{R})$, where $\mathcal{M}_n(\mathbb{R})$ is the space of all $n \times n$ real matrices with the standard topology generated by the euclidean norm and n is the dimension of the matrices $T(\chi)$, in which the function maps the value χ to an eigenprojection of $T(\chi)$, which is a projection (i.e. $P^2 = P$) that projects onto the eigenspace of its corresponding eigenvalue. Since we want to work eventually with a function that maps to individual eigenvectors, we will instead consider eigenvector functions, $\mathbf{v}_h$, in place of

these eigenprojection functions. We will not bother with the definition of the eigennilpotents since these will be irrelevant to the discussion.

We notice that there are many questions that we must ask about the aforementioned eigenvalue and eigenvector functions, λ_h and v_h respectively. First, we observe that they are both multi-valued functions and must be well-defined before they can be discussed seriously and used in application. The next point is that the eigenvector functions are multi-valued in 3 ways: there is a concern about the consistency of the eigenvalue, then a concern about a multiplicity greater than one for the same eigenvalue, and finally if each eigenvalue has multiplicity 1, then we still can make an arbitrary choice for the eigenvector because any parallel vector to an eigenvector is also an eigenvector. The eigenvector functions too must be differentiable based on the differentiability of the eigenprojections. The last thing we wish to note is the how we choose the domain, D. We will answer these questions shortly, but we begin by including more definitions.

Suppose we have a function $T : D \to \mathcal{H}_n(\mathbb{R})$, where $D \subset \mathbb{R}$, and $\mathcal{H}_n(\mathbb{R})$ is the space of all real symmetric matrices. We now wish to define the eigenvalue functions on the matrix space in a way that keeps them well-defined and allows us to obtain a derivative calculation. For a given matrix $A \in \mathcal{H}_n(\mathbb{R})$, we obtain its spectrum, $\mathcal{S}(A) = \{\lambda_1, \lambda_2, ..., \lambda_n\}$, where $\lambda_i \in \mathbb{R}$ for each i. Suppose that we now let the entries of the matrix A vary, in which for any matrix M that is "close enough" to A, we have functions $\lambda_i : B \to \mathbb{R}$ with $B \subset \mathcal{H}_n(\mathbb{R})$, in which $\lambda_i(M)$ is defined to be the eigenvalue that would be in the ith place of the spectrum. We need to be careful about how we define these eigenvalue functions.

To be precise, we let $M_0 \in \mathcal{H}_n(\mathbb{R})$ such that the spectrum of M_0 contains no repeated eigenvalues. We define a ball of radius, r, centered about M_0, $B(M_0, r) = \{M \in \mathcal{H}_n(\mathbb{R}) : ||M_0 - M|| < r\}$, where $|| \cdot ||$ is the euclidean norm in which the matrices are seen as vectors and r is small enough so that the ranges, $\lambda_i(B(M_0, r)) = \{\lambda_i(M) \in \mathbb{R} : M \in B(M_0, r)\}$ are disjoint.

Because the matrices are symmetric, and so we have real-valued eigenvalues and

eigenvectors, then the spectrum of A can be ordered by the simple inequality: $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$. Thus, we may define the spectrum function as an ordered tuple, namely $\mathcal{S}(A) = (\lambda_1, \lambda_2, ..., \lambda_n)$ in which we may now define $\lambda_i(A) = (\mathcal{S}(A))_i$. Then we may change the condition to $\lambda_1(A) > \lambda_2(A) > ... > \lambda_n(A)$ for all $A \in B(M_0, r)$, so that we have a well-defined function restricted to a sufficiently small ball of symmetric matrices.

Now, we see that for the composition, $\lambda_i \circ T : D \to \mathbb{R}$, we have eigenvalue functions like those described in the beginning of the section, but now well defined when we restrict D to an interval in which $T(x) \in B(M_0, r)$ for all $x \in D$. As for the eigenvector functions, v_i, we can follow the same process by looking at the composition, $\mathbf{v}_i \circ T : D \to \mathbb{R}^n$. Now, we still have the concern that for each eigenvalue, $\lambda_i(T(x))$, I can have multiple values for $\mathbf{v}_i(T(x))$, however, since we are restricted by the domain, D, we are guaranteed to have distinct eigenvalues for each $T(x)$, thus the eigenvectors will all have multiplicity 1. This eliminates the fear that there will be eigenspaces of dimension greater than 1 for a given eigenvalue, but it still leaves the choice of an infinite family of eigenvectors lingering. This can be mended by choosing a fixed half-space of $\mathbb{R}^n$ so that we can always choose the output eigenvector to be the one normalized to unit length within the fixed half-space. There is a problem of course if the eigenvector is located on the dividing hyper-plane, but we can further restrict our domain, D, to make sure that this does not happen.

Thus, our discussion shows that it is possible to determine well-defined eigenvalue and eigenvector functions. Now, our last note is to show that the differentiability of the eigenprojections imply the differentiability of the eigenvector functions with the well-defined construction above. This follows easily by understanding that an eigenvector function is the composition of a differentiable eigenprojection function and the function that takes $n \times 1$ matrices to the unit vector in the chosen half-plane, which is also differentiable except at the intersection of the half-plane and the unit hypersphere in $\mathbb{R}^n$. This finally allows us to use the theorem stated earlier in the section to show that they are holomorphic functions, and so in this case are differentiable. In this next section, we show

how we might calculate these derivatives to then use them to automatically parametrize our kernel functions.

6.4.2 Derivatives of the Eigenvalue Functions and the Update Rule

We start with the assumption that there is a matrix, $M_0 \in \mathcal{H}(\mathbb{R})$, and a ball, $B(M_0, r))$, in which the functions, λ_i, along with the normalized eigenvector functions, $\mathbf{v}_i$, are differentiable. Then, we want to find out what the explicit derivative with respect to each component of the matrix should be. Let us consider a matrix in this domain, say A, with eigenvalue λ corresponding to the eigenvector, $\mathbf{v}$. Then we have the equation from the definition of the eigenvalue:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{6.1}$$

Then we implicitly differentiate the equation to obtain:

$$\partial A\mathbf{v} + A\partial\mathbf{v} = \partial\lambda\mathbf{v} + \lambda\partial\mathbf{v} \tag{6.2}$$

Now we will include the condition that the eigenvector, $\mathbf{v}$, is of unit length, i.e. $||\mathbf{v}|| = 1$. Then we multiply both sides by $\mathbf{v}^T$ and we get:

$$\mathbf{v}^T\partial A\mathbf{v} + \mathbf{v}^T A\partial\mathbf{v} = \mathbf{v}^T\partial\lambda\mathbf{v} + \mathbf{v}^T\lambda\partial\mathbf{v} \tag{6.3}$$

Then we notice that $\mathbf{v}^T\partial\lambda\mathbf{v} = \partial\lambda\mathbf{v}^T\mathbf{v} = \partial\lambda||\mathbf{v}||^2 = \partial\lambda$ and $\mathbf{v}^T\lambda\partial\mathbf{v} = \lambda\mathbf{v}^T\partial\mathbf{v} = \lambda\partial(\frac{1}{2}\mathbf{v}^T\mathbf{v}) = \lambda\frac{1}{2}\partial(||\mathbf{v}||^2) = 0$, all due to the condition that $||\mathbf{v}|| = 1$. Thus, the right side of the equation 6.3 is simplified, leaving us with:

$$\mathbf{v}^T\partial A\mathbf{v} + \mathbf{v}^T A\partial\mathbf{v} = \partial\lambda \tag{6.4}$$

Now, we must apply the properties of a real symmetric matrix to simplify the left side of equation 6.4. We have that $\mathbf{v}^T A\partial\mathbf{v} = \mathbf{v}^T A^T\partial\mathbf{v} = (A\mathbf{v})^T\partial\mathbf{v} = (\lambda\mathbf{v})^T\partial\mathbf{v} = \lambda\mathbf{v}^T\partial\mathbf{v} =$

$\lambda \mathbf{v}^T \partial \mathbf{v} = \lambda \partial(\frac{1}{2}||\mathbf{v}||^2) = 0$, where we use our preliminary assumptions that $A\mathbf{v} = \lambda \mathbf{v}$, $A = A^T$, and $||\mathbf{v}|| = 1$. This then yields the final simplified equation in which we will get out desired derivative:

$$\mathbf{v}^T \partial A \mathbf{v} = \partial \lambda \tag{6.5}$$

We now wish to obtain an expression for $\frac{\partial \lambda}{\partial A_{ij}}$, for any index, (i, j). The definition of the multiplication of matrices shows us that:

$$\mathbf{v}^T \partial A \mathbf{v} = \sum_{s=1}^{n} \mathbf{v}_s \sum_{t=1}^{n} \partial A_{st} \mathbf{v}_t = \partial \lambda$$

And so by dividing both sides of the second equality by ∂A_{ij}, we have that:

$$\frac{\partial \lambda}{\partial A_{ij}} = \sum_{s=1}^{n} \mathbf{v}_s \sum_{t=1}^{n} \frac{\partial A_{st}}{\partial A_{ij}} \mathbf{v}_t = \mathbf{v}_i \mathbf{v}_j$$

where our last equality comes from the fact that:

$$\frac{\partial A_{st}}{\partial A_{ij}} = \begin{cases} 1 \ : \ i = s \text{ and } j = t \\ 0 \ : \ \text{otherwise} \end{cases}$$

We then let $E = vv^T$, which has components $E_{ij} = \mathbf{v}_i \mathbf{v}_j$, so that the description of the derivatives are in the concise form, $\frac{\partial \lambda}{\partial A} = E$. Therefore, we have succeeded in finding the derivative rule for use in back propagation for any given matrix entries.

As an important note, we also can see by the description of the matrix of derivatives, E, that E is a real symmetric matrix, and so applying an update rule to the matrix A, i.e. $A := A - \alpha E$ for some learning rate $\alpha \in \mathbb{R}_{>0}$, leaves the updated matrix A as another real symmetric matrix because the space of real symmetric matrices is a vector space over the real numbers. Other update rules that treat each parameter individually will not necessarily update the matrix so that the resulting matrix is a real symmetric matrix. Thus, most

popular adaptive learning rates, like Adam, may not be applied unless there is another way to keep this from being the case.

In reality, when we want to apply this to the case of biology using our kernel methods, we would have a kernel function that generates an adjacency or a Laplacian matrix in which the kernel parameters would have to be updated. We want to showcase this in terms of an eigenvalue function, λ. For a kernel function, $\Phi(x;\eta,\kappa)$, and a weighted adjacency matrix or a Laplacian matrix, A, that represents the distances between the atoms in a group of them, we can apply Φ to each entry of the matrix, i.e. we define $\Phi(A;\eta,\kappa)$ to be the matrix with entries $\Phi(A;\eta,\kappa)_{ij} = \Phi(A_{ij};\eta,\kappa)$. In this case, if we now wish to simply update the η and κ values based on the loss of the eigenvalue function, λ, then we calculate the derivative with respects to the η and κ values by the chain rule, i.e. $\frac{\partial \lambda}{\partial x} = \frac{\partial \Phi(A)}{\partial x}\frac{\partial \lambda}{\partial \Phi(A)} = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial \Phi(A)_{ij}}{\partial x}\frac{\partial \lambda}{\partial \Phi(A)_{ij}}$, where $x = \eta$ or $x = \kappa$. The $\frac{\partial \Phi}{\partial x}$ terms are easy to calculate, and have been derived in an earlier work, while the $\frac{\partial \lambda}{\partial \Phi}$ terms have been described above with the matrix, E, so we have a full description of the derivatives an eigenvalue function with respects to the parameters of a chosen kernel function that generates an adjacency/Laplacian matrix. We note that since the updates are made on the kernel parameters, and the kernel functions generate matrices that are real symmetric, then the update rule will not cause any issues involving matrices that are outside of the real symmetric space and thus the derivatives of the eigenvalue functions with respect to the entries of the matrices will always be valid throughout the training.

6.4.3 Discussion

We can implement the above ideas in any of the previously applied fixed-kernel methods involving the use of eigenvalues. A simple idea would be to analyze the graph consisting of atoms as vertices with the edges corresponding to the covalent bonds between them, calculating the eigenvalues of the Laplacian and adjacency matrices, then record-

ing the sum (trace), average, non-zero minimum (Fiedler value), maximum, etc. as the features. This will generate a small amount of features, but may provide a useful bolster to something like the geometric graph layer of the AweGNN. If more spectral features are desired, then we may consider element specific features as in Nguyen et al. [80], but this may require a description of the derivatives of the eigenvector functions.

Although, not much can be concluded about the performance of a auto-parametrized spectral layer since it has not been tested, we can speculate that the performance will be good, as we have seen with the AweGNN [100], and the performance of the comprehensive model in section 6.3, along with the success of spectral-based representations in previous works [80, 90]. The main concern would be the computational feasibility of the layer. The computational costs of calculating eigenvalues can become exorbitantly expensive with very large matrices, and so if we encounter large graph structures, we will have large Laplacian matrices, leading to potentially infeasible calculations. GPUs or other hardware/software that can provide a boost may be necessary.

6.5 Auto-parametrized Kernel Models and Future Possibilities

We have discussed many topics so far involving mathematical theory, machine learning, biomolecular modeling, kernel-based models, the AweGNN model, a comprehensive network involving topological and spectral features, and the auto-parametrized spectral network layer. We can apply kernels to many of the techniques used in our group. Any of the features that use a kernel function with tunable parameters is a target for auto-parametrization. Geometric graphs, persistent homology, algebraic graphs, and differential geometry have seen kernel functions applied to them in different ways, all with success [78, 110, 22, 80, 81, 79, 108].

Inspired by the success of the convolutional neural networks (CNN), which updates filters that pre-processes image-like data throughout training, we developed the AweGNN. The AweGNN applies kernel functions to biological data like the CNN applies filters to

image-like data, and we automated the process of updating the kernel functions through their parameters as CNNs do with their filters. To implement our idea, we used geometric graphs with weighted edges to get features from the edge weights and some additional information, such as the partial charges. This provided an easy set of features to extract and an easy calculation for the derivatives of the kernel parameters necessary to update them during training. This new network layer was named the geometric graph representation (GGR) layer, since it produced a geometric graph representation of a molecule based on input kernel parameters.

The GGR layer performed well, but was limited in scope due to the computational cost of the calculations involved. The computational burden had to be lessened to further advance the possibilities of the GGR, and make multi-scale representations feasible. This was done with the GPU-enchanced AweGNN, in which the GGR layer was adapted to work with the GPU technology. This gave us the opportunity for a multi-scale model to be trained. To bolster the new GGR layer, we proposed using a comprehensive neural network architecture which incorporated fixed topological and spectral features, drawing out the geometric and structural information of the molecules in conjunction with the element-specific features from the GGR layer. A heavily restricted implementation of this comprehensive network had much success on two datasets, ESOL and FreeSolv, which proves that the network might be worth improving to reach for even greater success.

The theory for the Auto-parametrized spectral layer was outlined, although no layer has been built yet or experimented with. It is likely that it will do well, given previous success of previous work with spectral features. There are two more kernel-based models who have not been explored yet in terms of our auto-parametrization technique, namely the differential geometry based models and the algebraic topology-based models. Given the promising theory laid down for the Auto-parametrized spectral layer, we may have confidence that in both cases, we may find a way automate the kernel parameter selection. There will always be an issue with large data sets and computation time, but we have

improved our models before using parallel algorithms and GPUs, so it is also likely that we can tackle this problem as it comes up.

CHAPTER 7

DISSERTATION CONTRIBUTION

The contributions of this dissertation are as follows:

- In chapter 3, we propose a new kernel function, the piece-wise linear kernel, which attempts to more closely approximate the ideal low-pass filter as compared to the Lorentz and exponential kernels. This new piece-wise linear kernel has been tested in a multi-scale model including the Lorentz and exponential kernels on two datasets, ESOL and FreeSolv [111], in which the results of the experiments were recorded in section 6.3.3.

- Chapter 5 details the auto-parametrized weighted element-specific graph neural network, which was developed to address the limitations of kernel-based representations, and also to simply present a new and interesting model for experimentation. The AweGNN was tested on a series of toxicity datasets, using both single and multi-task architectures, then tested on a small dataset involving solvation free energy predictions, all with excellent results [100].

- The GPU-enchanced AweGNN was developed to alleviate the computational expense in the original AweGNN model. The new structure and algorithm is detailed in chapter 6.

- A new comprehensive architecture including, multi-scale auto-parametrized geometric graph features, and fixed topological and spectral auxiliary features is proposed also in chapter 6. Limited experiments are conducted on the ESOL and FreeSolv datasets [111], with promising results.

- In Chapter 6, we also lay a theoretical foundation for the construction of an auto-parametrized kernel-based layer that produces representations based on the spec-

trum of Laplacian and adjacency matrices.

This dissertation was centered mainly around the concepts introduced and work done in the publication:

Szocinski, Timothy & Nguyen, Duc & Wei, Guo-Wei. (2021). AweGNN: Auto-parametrized weighted element-specific graph neural networks for molecules. Computers in Biology and Medicine. 134. 104460. 10.1016/j.compbiomed.2021.104460.

BIBLIOGRAPHY

[1] S. Shivkumar A. Speranzon and R. Ghrist. “Localization Exploiting Topological Constraints”. In: *Proc. Amer. Control Conf. [ACC].* (2020).

[2] Ralph Abbey et al. *Search Engines and Data Clustering*.

[3] Kevin S Akers, Glendon D Sinks, and T Wayne Schultz. “Structure–toxicity relationships for selected halogenated aliphatic chemicals”. In: *Environ. Toxicol. Pharmacol.* 7.1 (1999), pp. 33–39.

[4] A. Sizemore et al. “"The importance of the whole: topological data analysis for the network neuroscientist"”. In: *Network Neurosci.* 3(3) (2019), pp. 656–673.

[5] Guo Wei Wei et al. “Tailoring Wavelets for Chaos Control”. In: *Phys. Rev. Lett.* 89.284103 (2002).

[6] K. L. Xia et al. “Multiscale geometric modeling of macromolecules i: Cartesian representation”. In: *Journal of Computational Physics* 275 (2014), pp. 912–936.

[7] Klenin K et al. “"Derivatives of molecular surface area and volume: Simple and exact analytical formulas"”. In: *Journal of Computational Chemistry* 32 (12) (2011), pp. 2647–2653. DOI: 10.1002/jcc.21844.

[8] R. Adamczak et al. “"Combining prediction of secondary structure and solvent accessibility in proteins"”. In: *Proteins* 59 (3) (2005), pp. 467–75. DOI: 10.1002/prot.20441.

[9] W. Rocchia et al. “Rapid grid-based construction of the molecular surface and the use of induced surface charge to calculate reaction field energies: Applications to the molecular systems and geometric objects”. In: *Journal of Computational Chemistry* 23 (2002), pp. 128–137.

[10] X. Feng et al. “Multiscale geometric modeling of macromolecules II: lagrangian representation”. In: *Journal of Computational Chemistry* 34 (2013), pp. 2100–2120.

[11] Xin Feng et al. “Geometric modeling of subcellular structures, organelles and large multiprotein complexes”. In: *Int J Numer Method Biomed Eng* 28 (2012), pp. 1198–1223.

[12] Marat Andreev et al. “Influence of Ion Solvation on the Properties of Electrolyte Solutions”. In: *The Journal of Physical Chemistry B* 122.14 (2018). PMID: 29611710, pp. 4029–4034. DOI: 10.1021/acs.jpcb.8b00518. eprint: https://doi.org/10.1021/acs.jpcb.8b00518. URL: https://doi.org/10.1021/acs.jpcb.8b00518.

[13] Pedro J Ballester, Adrian Schreyer, and Tom L Blundell. "Does a more precise chemical description of protein–ligand complexes lead to more accurate prediction of binding affinity?" In: *Journal of chemical information and modeling* 54.3 (2014), pp. 944–955.

[14] João Carlos Alves Barata and Mahir Saleh Hussein. "The Moore–Penrose Pseudoinverse: A Tutorial Review of the Theory". In: *Brazilian Journal of Physics* 42.1-2 (Dec. 2011), pp. 146–165. ISSN: 1678-4448. DOI: 10.1007/s13538-011-0052-z. URL: http://dx.doi.org/10.1007/s13538-011-0052-z.

[15] Maciej Barycki et al. "Multi-objective genetic algorithm (MOGA) as a feature selecting strategy in the development of ionic liquids' quantitative toxicity–toxicity relationship models". In: *Journal of chemical information and modeling* 58.12 (2018), pp. 2467–2476.

[16] Ulrich Bauer. "Ripser: a lean c++ code for the computation of Vietoris-Rips persistence barcodes". In: (2017). URL: https://github.%20com/Ripser/ripser.

[17] Stefan Behnel et al. "Cython: The best of both worlds". In: *Comput. Sci. Eng.* 13.2 (2010), pp. 31–39.

[18] Béla Bollobás. *Modern Graph Theory*. 1st ed. Graduate Texts in Mathematics 184. Springer-Verlag New York, 1998. ISBN: 978-0-387-98488-9,978-1-4612-0619-4.

[19] Simon Brandt et al. "Machine learning of biomolecular reaction coordinates". In: *The journal of physical chemistry letters* 9.9 (2018), pp. 2144–2150.

[20] R. Brooks. "Constructing isospectral manifolds". In: *Amer. Math. Month.* 95 (1988), pp. 823–839.

[21] Keith T Butler et al. "Machine learning for molecular and materials science". In: *Nature* 559.7715 (2018), pp. 547–555.

[22] Z. X. Cang and Guo-Wei Wei. "TopologyNet: Topology based deep convolutional and multi-task neural networks for biomolecular property predictions". In: *PLOS Computational Biology* 13(7) (2017).

[23] Rich Caruana. *Learning to Learn*. Springer, 1998, pp. 95–133.

[24] Anupam Choudhary and Ravi Kshirsagar. "Process Speech Recognition System using Artificial Intelligence Technique". En. In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.5 (Nov. 2012). Ed. by Dr. Shiv Kumar, pp. 239–242. ISSN: 2231-2307. URL: https://www.ijsce.org/wp-content/uploads/papers/v2i5/E1054102512.pdf.

[25] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, pp. 3642–3649.

[26] Steven J Darnell, Laura LeGault, and Julie C Mitchell. "KFC Server: interactive forecasting of protein interaction hot spots". In: *Nucleic acids research* 36.suppl_2 (2008), W265–W269.

[27] S. Decherchi and W. Rocchia. "A general and robust ray-casting-based algorithm for triangulating surfaces at the nanoscale". In: *PLoS ONE* 8:e59744 (2013).

[28] J.S. Delaney. 2004.

[29] Sarah Jane Delany and Pádraig Cunningham. "Ecue: A spam filter that uses machine learning to track concept drift". In: *In Proceeding of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29–September 1, 2006, Riva del Garda, Italy*. IOS Press, 2006, pp. 627–631.

[30] Li Deng, Geoffrey Hinton, and Brian Kingsbury. "New types of deep neural network learning for speech recognition and related applications: An overview". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 8599–8603.

[31] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. cite arxiv:1810.04805Comment: 13 pages. 2018. URL: http://arxiv.org/abs/1810.04805.

[32] Ashay Dharwadker and Shariefuddin Pirzada. "Applications of Graph Theory". In: *JOURNAL OF THE KOREAN SOCIETY FOR INDUSTRIAL AND APPLIED MATHEMATICS* 11.4 (2007).

[33] D. et al. Duvenaud. *arXiv preprint*. 2015. arXiv: 1509.09292 [cs.LG].

[34] Herbert Edelsbrunner and John Harer. *Computational Topology - an Introduction*. American Mathematical Society, 2010, pp. I–XII, 1–241. ISBN: 978-0-8218-4925-5.

[35] Lance Eliot and Michael Eliot. *Autonomous Vehicle Driverless Self-Driving Cars and Artificial Intelligence: Practical Advances in AI and Machine Learning*. 1st. LBE Press Publishing, 2017. ISBN: 0692051023.

[36] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299.

[37] Justin Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 1263–1272. URL: https://proceedings.mlr.press/v70/gilmer17a.html.

[38] Alexander Golbraikh et al. "Rational selection of training and test sets for the development of validated QSAR models". In: *Journal of computer-aided molecular design* 17.2 (2003), pp. 241–253.

[39] Yoav Goldberg and Omer Levy. *word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method*. cite arxiv:1402.3722. 2014. URL: http://arxiv.org/abs/1402.3722.

[40] Allen Hatcher. *Algebraic topology*. Cambridge: Cambridge University Press, 2002, pp. xii+544. ISBN: 0-521-79160-X; 0-521-79540-0.

[41] Ian Holmes, Keith Harris, and Christopher Quince. *Dirichlet Multinomial Mixtures: Generative Models for Microbial Metagenomics*.

[42] Aapo Hyvärinen and Erkki Oja. "Independent component analysis: algorithms and applications". In: *Neural Networks* 13 (2000), pp. 411–430.

[43] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[44] P. Mach J. Li and P. Koehl. "Measuring the shapes of macromolecules - and why it matters". In: *Comput Struct Biotechnol J.* 8:e201309001 (2013).

[45] Jian Jiang, Rui Wang, and Guo-Wei Wei. "GGL-Tox: Geometric Graph Learning for Toxicity Prediction". In: *Journal of Chemical Information and Modeling* 61.4 (2021). PMID: 33719422, pp. 1691–1700. DOI: 10.1021/acs.jcim.0c01294. eprint: https://doi.org/10.1021/acs.jcim.0c01294. URL: https://doi.org/10.1021/acs.jcim.0c01294.

[46] Jian Jiang et al. "Boosting tree-assisted multitask deep learning for small scientific datasets". In: *Journal of chemical information and modeling* 60.3 (2020), pp. 1235–1244.

[47] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[48] Abdul Karim et al. "Efficient toxicity prediction via simple features using shallow neural networks and decision trees". In: *Acs Omega* 4.1 (2019), pp. 1874–1888.

[49] Tosio Kato. *Perturbation theory for linear operators*. 1st ed. Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete, Bd. 132. Springer-Verlag New York, 1966.

[50] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[51] Volodymyr Kovenko and Vitalii Shevchuk. *l1 and l2 regularization*. 2020. URL: https://machine-learning-and-data-science-with-python.readthedocs.io/en/latest/assignment3_sup_ml.html (visited on 2020).

[52] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[53] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[54] J. De Leeuw. "Theory of multidimensional scaling". In: *Handbook of Statistics, volume II', North*. Holland Publishing Company, 1980.

[55] Lin Li, Chuan Li, and Emil Alexov. "On the modeling of polar component of solvation energy using smooth Gaussian-based dielectric function". In: *Journal of Theoretical and Computational Chemistry* 13.03 (2014), p. 1440002.

[56] Chuan Li Lin Li and Emil Alexov. "On the modeling of polar component of solvation energy using smooth gaussian-based dielectric function". In: *Journal of Theoretical and Computational Chemistry* 13:10.1142/S0219633614400021 (2014).

[57] Ruifeng Liu et al. "Assessing deep and shallow learning methods for quantitative prediction of acute chemical toxicity". In: *Toxicological Sciences* 164.2 (2018), pp. 512–526.

[58] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[59] P. Mach and P. Koehl. "Geometric measures of large biomolecules: Surface, volume, and pockets". In: *J. Comp. Chem.* 32 (2011), pp. 3023–3038.

[60] Alireza Makhzani et al. *Adversarial Autoencoders*. cite arxiv:1511.05644. 2015. URL: http://arxiv.org/abs/1511.05644.

[61] John Mannes. *Facebook's Artificial Intelligence Research lab releases open source fastText on GitHub*. 2018.

[62] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press, 1999. ISBN: 978-0-262-13360-9.

[63] Aleksandr V Marenich, Christopher J Cramer, and Donald G Truhlar. "Performance of SM6, SM8, and SMD on the SAMPL1 test set for the prediction of small-molecule solvation free energies". In: *The Journal of Physical Chemistry B* 113.14 (2009), pp. 4538–4543.

[64] T. M. Martin. *User's Guide for T.E.S.T. (version 4.2) (Toxicity Estimation Software Tool): A Program to Estimate Toxicity from Molecular Structure*. USEPA, 2016.

[65] Todd M Martin et al. "A hierarchical clustering methodology for the estimation of toxicity". In: *Toxicology Mechanisms and Methods* 18.2-3 (2008), pp. 251–266.

[66] Meinhard E. Mayer. "The Feynman Integral and Feynman's Operational Calculus". In: *Physics Today* 54 (8) (2001).

[67] W.S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysic* 5 (1943), pp. 115–133.

[68] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. cite arxiv:1802.03426Comment: Reference implementation available at http://github.com/lmcinnes/umap. 2018. URL: http://arxiv.org/abs/1802.03426.

[69] Zhenyu Meng et al. "Weighted persistent homology for biomolecular data analysis". In: *Scientific reports* 10.1 (2020), pp. 1–15.

[70] Z. X. Cang Menglun Wang and Guo-Wei Wei. "A topology-based network tree for the prediction of protein-protein binding free energy changes following mutation". In: *Nature Machine Intelligence* 2 (2020), pp. 116–123.

[71] S. Papert Minsky M. *An Introduction to Computational Geometry*. 1969. ISBN: 978-0-262-63022-1.

[72] Bin Tu Minxin Chen and Benzhuo Lu. "Triangulated manifold meshing method preserving molecular surface topology". In: *J. Mole. Graph. Model.* 3 (2012), pp. 411–418.

[73] D. L. et al. Mobley. 2014.

[74] David L Mobley et al. "Blind prediction of solvation free energies from the SAMPL4 challenge". In: *Journal of computer-aided molecular design* 28.3 (2014), pp. 135–150.

[75] J. P. Mobley D. L.; Guthrie. 2014.

[76] A et al. Momen-Roknabadi. “"Impact of residue accessible surface area on the prediction of protein secondary structures"”. In: *BMC Bioinformatics* 9 (2008), p. 357. DOI: 10.1186/1471-2105-9-357.

[77] Dmitriy Morozov. *Dionysus*. 2015. URL: http://www.mrzv.org/software/dionysus/.

[78] Duc D Nguyen et al. “Rigidity strengthening: A mechanism for protein–ligand binding”. In: *Journal of chemical information and modeling* 57.7 (2017), pp. 1715–1721.

[79] Duc Duy Nguyen, Zixuan Cang, and Guo-Wei Wei. “A review of mathematical representations of biomolecular data”. In: *Physical Chemistry Chemical Physics* 22.8 (2020), pp. 4343–4367.

[80] Duc Duy Nguyen and Guo-Wei Wei. “AGL-Score: Algebraic graph learning score for protein–ligand binding scoring, ranking, docking, and screening”. In: *Journal of chemical information and modeling* 59.7 (2019), pp. 3291–3304.

[81] Duc Duy Nguyen and Guo-Wei Wei. “DG-GL: Differential geometry-based geometric learning of molecular datasets”. In: *International journal for numerical methods in biomedical engineering* 35.3 (2019), e3179.

[82] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *arXiv preprint arXiv:1912.01703* (2019).

[83] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[84] P. Perona. “Vision of a Visipedia”. In: *Proceedings of the IEEE* 98.8 (Aug. 2010), pp. 1526–1534. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2049621.

[85] G. Polyá and S. Szegö. “Isoperimetric inequalities in Mathematical Physics”. In: *Annals of Math. Studies* 27 (1951).

[86] P. Perry R. Brooks and P. Yang. “Isospectral sets of conformally equivalent metrics”. In: *Duke Math J.* 58 (1989), pp. 131–150.

[87] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. “On the convergence of adam and beyond”. In: *arXiv preprint arXiv:1904.09237* (2019).

[88] Douglas A. Reynolds. “Gaussian Mixture Models”. In: *Encyclopedia of Biometrics*. 2009.

[89] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[90] Duc D Nguyen Rui Wang and Guo-Wei Wei. "Persistent spectral graph". In: *International Journal for Numerical Methods in Biomedical Engineering* 36(9) (2020).

[91] Kallu Samatha et al. "Chronic Kidney Disease Prediction using Machine Learning Algorithms". En. In: *International Journal of Preventive Medicine and Health (IJPMH)* 1.3 (July 2021). Ed. by Dr. Shiv Kumar, pp. 1–4. ISSN: 2582-7588. DOI: 10.35940/ijpmh.C1010.071321. URL: https://www.ijpmh.latticescipub.com/wp-content/uploads/papers/v1i3/C1010071321.pdf.

[92] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[93] Jeffrey Seely. *Topological Data Analysis*. 2016. URL: https://jsseely.github.io/notes/TDA/ (visited on 04/13/2016).

[94] X. Shi and P. Koehl. "Geometry and topology for modeling biomolecular surfaces". In: *Far East J. Applied Math* 50 (2011), pp. 1–34.

[95] JA. Shrake A; Rupley. ""Environment and exposure to solvent of protein atoms. Lysozyme and insulin"". In: *J Mol Biol* 79 (2) (1973). DOI: 10.1016/0022-2836(73)90011-9.

[96] Nicoleta Spinu et al. "Quantitative adverse outcome pathway (qAOP) models for toxicity prediction". In: *Archives of toxicology* 94 (2020), pp. 1497–1510.

[97] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[98] T. Sunada. "Riemannian coverings and isospectral manifolds". In: *Ann. Math.* 121 (1985), pp. 169–186.

[99] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *arXiv preprint arXiv:1409.3215* (2014).

[100] Timothy Szocinski, Duc Duy Nguyen, and Guo-Wei Wei. "AweGNN: Auto-parametrized weighted element-specific graph neural networks for molecules". In: *Computers in Biology and Medicine* 134 (2021), p. 104460. ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2021.104460. URL: https://www.sciencedirect.com/science/article/pii/S0010482521002547.

[101] The GUDHI Project. *GUDHI User and Reference Manual*. 3.4.1. GUDHI Editorial Board, 2021. URL: https://gudhi.inria.fr/doc/3.4.1/.

[102] Nenad Tomašev et al. *Assessing Game Balance with AlphaZero: Exploring Alternative Rule Sets in Chess*. 2020. arXiv: 2009.04374 [cs.AI].

[103] N. Volkmann. "Methods for segmentation and interpretation of electron tomographic reconstructions". In: 483 (2010), pp. 31–46.

[104] Bao Wang et al. "Breaking the polar-nonpolar division in solvation free energy prediction". In: *Journal of computational chemistry* 39.4 (2018), pp. 217–233.

[105] Ercheng Wang et al. "End-point binding free energy calculation with MM/PBSA and MM/GBSA: strategies and applications in drug design". In: *Chemical reviews* 119.16 (2019), pp. 9478–9508.

[106] Junmei Wang et al. "Solvation model based on weighted solvent accessible surface area". In: *The Journal of Physical Chemistry B* 105.21 (2001), pp. 5055–5067.

[107] Rui Wang, Duc Duy Nguyen, and Guo-Wei Wei. "Persistent spectral graph". In: *International journal for numerical methods in biomedical engineering* 36.9 (2020), e3376.

[108] Guo Wei Wei. "Mathematical molecular bioscience and biophysics." In: *SIAM News* 49(7) (2016).

[109] Still WC Weiser J Shenkin PS. ""Approximate atomic surfaces from linear combinations of pairwise overlaps (LCPO)"". In: *Journal of Computational Chemistry* 20 (2) (1999), pp. 217–230.

[110] Kedi Wu and Guo-Wei Wei. "Quantitative toxicity prediction using topology based multitask deep neural networks". In: *Journal of chemical information and modeling* 58.2 (2018), pp. 520–531.

[111] Zhenqin Wu et al. *MoleculeNet: A Benchmark for Molecular Machine Learning*. 2018. arXiv: 1703.00564 [cs.LG].

[112] Kelin Xia, Kristopher Opron, and Guo-Wei Wei. "Multiscale multiphysics and multidomain models—Flexibility and rigidity". In: *The Journal of chemical physics* 139.19 (2013), 11B614_1.

[113] Hao Zhu et al. "Combinatorial QSAR modeling of chemical toxicants tested against Tetrahymena pyriformis". In: *Journal of chemical information and modeling* 48.4 (2008), pp. 766–784.

[114] Hao Zhu et al. “Quantitative structure- activity relationship modeling of rat acute toxicity by oral exposure”. In: *Chemical research in toxicology* 22.12 (2009), pp. 1913–1921.