A NOVEL FRAMEWORK FOR INVARIANT NEURAL NETWORKS APPLIED TO GRAPH AND SET DATA

by

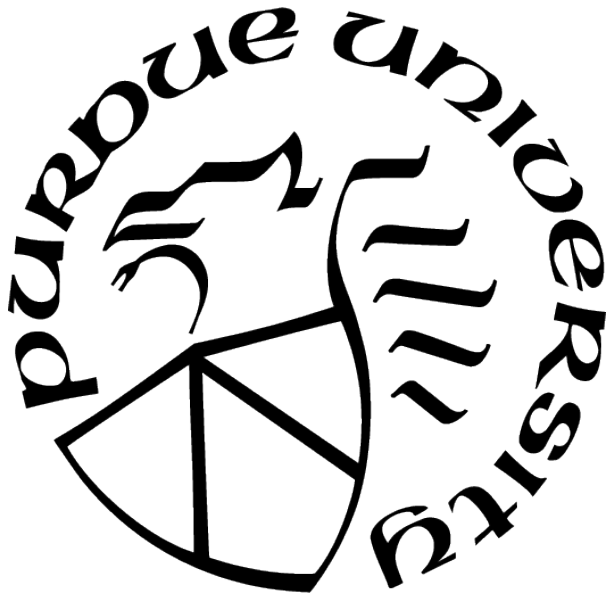
Ryan L. Murphy

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



Department of Statistics

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Vinayak Rao, Co-Chair

Department of Statistics, Department of Computer Science (Courtesy)

Dr. Bruno Ribeiro, Co-Chair

Department of Computer Science

Dr. Jennifer Neville

Department of Computer Science, Department of Statistics

Dr. Min Zhang

Department of Statistics

Approved by:

Dr. Jun Xie

Dedicated to my family, my friends, and the limitless source of energy that comes from pursuing one's dreams.

ACKNOWLEDGMENTS

First, I would like to thank my advisors, Drs. Rao and Ribeiro, for their mentorship and for helping me develop as a machine learning researcher. In our work, I learned an overwhelming amount of new theoretical ideas and programming/computational techniques. I would like to highlight Dr. Rao's Computational Statistics course, which I took in my first semester. This course reaffirmed the power of what I would learn throughout this degree and motivated me through the pre-research years. Additionally, I thank Dr. Ribeiro for the opportunity to present my work at NetSci and for the generous support.

Next, I would like to thank my main collaborators, Balasubramaniam Srinivasan and Shishang Wu. Their thoughtful discussions and work strengthened Janossy pooling and Birkhoff regularization. I also thank the students in my professors' research groups. I learned so much about programming and computation from my computer science peers, and had countless thoughtful discussions of recent research papers.

Moving past academics, it goes without saying that my parents and brother taught me the importance of hard work, perseverance, and determination. My family supported me every step of the way, not least during the incredibly refreshing holidays spent back home!

I also thank the contemporaries that I met at Purdue. I especially thank April Yue for her encouragement, support, and boundless wisdom. We grew together in countless ways through these years. Jincheng Bai, Will Eagan, Eric Gerber, Emery Goossens, Nathan Hankey, Babak Ravandi, Tim Smith, Jeremy Troisi, Qi Wang, Yumin Zhang and others offered wisdom, mentorship, and invaluable opportunities for blowing off steam. I am grateful to Farzin Shamloo, Michelle Blitzman, and James Sterritt whose selflessness helped me through a rather sudden and unexpected challenge.

Finally, I must mention Professors Steven Landsburg, Paulo Barelli, Josh Kinsler, and Michael Rizzo at my undergraduate university. They ignited my passion for pursuing higher education and its power for making a positive difference in this world. When I decided that I would not pursue this dream through economics, it was the journalists at the MIT Technology Review and similar media that ultimately shaped my decision to move towards statistics, machine learning, and computer science.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\mathbf{A}$ A tensor (multi-dimensional array)

$\boldsymbol{A}$ A matrix

π The ISO standard for the constant $3.1415\ldots$

π Usually denotes permutations

NOMENCLATURE

Encoding a Graph/Set	Storing a graph/set on a computer.
Graph	*Graph* refers to input relational data. See Network.
Labeling	A function defined on the vertex set. See Target.
Multilayer Perceptron	Feedforward neural network of affine layers, all weights free.
Network	Usually reserved for *neural networks*. See Graph.
Target	The response variable, the quantity to predict.

ABSTRACT

The generalization performance of neural networks can be improved by respecting *invariances* inherent in a problem. For instance, it is often the case that rotating an image does not change its meaning; the target variable is likely *invariant* to rotations of the input. Models for graph and set data are typically designed to be permutation-invariant, since the order of set elements and isomorphisms of graphs usually do not change their meaning, but a unified paradigm for both types of data is lacking. Moreover, existing models are either insufficiently flexible or difficult to reliably train in practice. This dissertation presents *Janossy pooling* (JP), a novel framework for training neural networks that are (approximately) invariant to a prespecified finite group of transformations (e.g., permutations), with a focus on graphs and sets. Motivated by partial exchangeability and Janossy densities, we view transformation-invariant models as averages over all transformations of the input, each passed to a transformation-sensitive function. As the set of transformations can be very large, we advance three approximation strategies: π-Stochastic Gradient Descent (π-SGD), k-ary approximations, and poly-canonical orderings. Compared to state-of-the-art approaches, JP is capable of expressing a richer class of models than Message Passing Graph Neural Networks and does not necessitate the use of highly complex and discontinuous functions for modeling sets effectively. Empirical evidence, including experiments on molecular and protein-protein-interaction datasets, supports the theory we develop and the practical benefits of JP. However, on another note, it may be unclear whether enforcing invariances is appropriate in any given task, such as in the case of time-evolving graphs. Accordingly, we propose a data-driven scheme where the extent of transformation sensitivity is determined by a regularization hyperparameter. This approach invokes the Birkhoff-von Neumann theorem, making it directly applicable to both graphs and sets, and establishes a link between existing methods and infinitely strong regularization. Finally, we verify this approach experimentally.

1. INTRODUCTION

The generalization performance of neural networks can be significantly improved by designing them to respect data invariances. Neural networks are highly flexible function approximators, which contributes to their success in numerous applications, but the same flexibility makes them prone to overfitting to nuisance variation in the data. For example, in set data, the order of elements carries no meaning; the true function of the data is invariant to permutations of the input. However, a neural network that is not designed to respect this invariance could make very different predictions for the same set with different orderings [1]. We can mitigate this by building a model that respects the permutation invariance. We say that this invariance arises due to the lack of a unique *encoding* of the data (see also Figure 3.1). Invariances also arise from physical laws governing the data-generating process [2], [3]. Similarly, it is often important for models to respect *equivariances* such as translation equivariance. For example, if a model is designed to localize an object in an image by drawing a bounding box around it, translations of the object should result in a corresponding translation of the box. We call any strategy that promotes invariance or equivariance to some input *transformations*, such as permutations, rotations, or phase shifts, *invariant modeling*. The goal of invariant modeling is to maintain the same level of flexibility while improving the generalization performance. Numerous works demonstrate that invariant models can achieve better generalization and extrapolation performance [4]–[9] and these models have been successful in applications such as speech recognition [10], [11], image classification [12], [13], and protein structure prediction [14].

Some invariant modeling strategies enforce *strict* invariance, where the model makes the same prediction for all relevant input transformations, whereas others simply encourage predictions to be *similar,* and both can be effective [9]. Examples include modifying the optimization objective with TangentProp [15], augmenting the data with transformed inputs [16], [17], and crafting functions that are invariant by design [18]–[24].

Permutation invariance is important for graph and set data, which appear widely in practice. In set applications, each input is a variable-length collection of vectors (without loss of generality) whose ordering is arbitrary. To be clear, this differs from classical linear

regression settings where every input has the same ordering (e.g., $X_{i,1}$ corresponds to height, $X_{i,2}$ to age, etc.). Set data appear in point cloud classification (Figure 3.3) [25]–[28]; multiple instance learning, which can be used in disease classification [29]–[34]; piloting drone swarms [35]; and many others [2], [3], [36]–[39]. Graph data arise whenever entities (i.e., *vertices*) are connected (i.e., by *edges*), such as by a physical connection or a less tangible relationship. These find applications in chemical property prediction and drug discovery (atoms are connected by bonds) [40]–[44]; protein-protein interactions (connections indicate an interaction between proteins) [45], [46]; knowledge bases [47], [48]; and others [49]–[53]. Beyond relationships, graph data often capture additional vertex- or edge- level features (see Figure 3.2). As we will see, permutation-invariant models designed for set inputs can also be used in graph modeling as an *aggregator* over features in each vertex neighborhood [45], [54]–[56].

A state-of-the-art approach to modeling sets is *pooling in latent space* [25], [26]. In these models, each element of a set is first passed through a neural network g, then the outputs are aggregated with a simple permutation-invariant function like summation, and the result is passed through another network r. It has been shown that these models can approximate any permutation-invariant function arbitrarily well given suitable choices for g and r [25], [26], [57]–[59]. However, independently processing each element in the input causes difficultly in capturing relationships among set elements, as we described in [54] and was also pointed out by [30], [32], [60], [61]. Consequently, the networks g and r must be very complex and contain a large number of parameters, making modeling difficult in practice [62]. Regarding graphs, a state-of-the-art approach is Message Passing Graph Neural Networks (MPGNNs) [40], [44], [45], [56], [63]–[67], which recursively aggregate feature vectors in the neighborhood of each vertex. Unfortunately, MPGNNs are not flexible enough to model any graph function, an expressiveness limitation not shared by neural network models applied to other types of data [68]–[73]. In particular, MPGNNs predict the same value for some pairs of nonisomorphic graphs, since passing messages over neighborhoods cannot distinguish complex cycles [74]. A third approach, applicable to both sets and graphs, is to apply an algorithm that *orders* the input [5], [75]–[77], which renders the original ordering irrelevant. Unfortunately, there

is no guarantee that the ordering will be effective, as observed empirically in [55]. Moreover, these methods often rely on heuristics and do not provide guidance for general applicability.

We advance Janossy pooling (JP) to address these challenges, which stem from the difficulty of specifying a permutation-invariant *model* that is flexible enough to express all permutation-invariant *functions* without relying on the ability to model complex functional forms. Instead, in JP, we use permutation-sensitive functions $\vec{f}$ to build a permutation-invariant model. In particular, *full* JP is defined by averaging the values of $\vec{f}$ over all permutations of the input. In practice, we must *approximate* full JP, thus constituting an *approximate invariant* modeling approach like TangentProp or data augmentation. We demonstrate that modeling $\vec{f}$ with a Recurrent Neural Network better captures relationships in the input than latent space pooling, and we propose a special $\vec{f}$ called RPGNN that is more flexible than MPGNNs. We then discuss and justify three approximation schemes: π-SGD, k-ary dependencies, and poly-canonical orderings. In parallel, we show that many prior methods can be seen as approximations to JP, which further clarifies limitations of these works and motivates simple yet effective improvements thereof. For instance, our Circulant Skip Links dataset (Section 4.4.1), which highlights limitations of MPGNNs, has been adopted to test new graph neural networks [8], [78]–[80]. We demonstrate the benefits of JP on protein-protein-interaction and molecular graph datasets, and explore our theoretical predictions on these and other synthetic data.

We proposed JP in [54] and elaborated on its applications to graph data in [81], under the name Relational Pooling (RP). However, JP can be extended to building models invariant to transformations other than permutations. Accordingly, we will write JP regardless of whether the input is a set or graph and touch on its applicability to invariant modeling at large. Indeed, a key contribution here is to provide one framework for modeling disparate data types such as sets and graphs, whereas previous literature focused on separate approaches for each.

Along another direction, we observe that the appropriateness of permutation-invariant modeling in a given task may be difficult to determine a-priori. While permutation invariance is indeed a sensible condition, due to the nonunique encoding of graphs and sets, it does not imply that the order never conveys meaning in the data-generating process. For instance,

in graphs that arise from preferential attachment and fitness mechanisms [82], knowledge of the temporal ordering can help distinguish between "fit" and "early" vertices (although these types are not necessarily mutually exclusive). We introduce Birkhoff regularization (BReg), a permutation sensitivity *penalty* that we include in the optimization objective, and train any arbitrary permutation-sensitive function. Tuning the regularization strength serves a data-driven approach for identifying whether invariances are appropriate for the task. Our regularization is inspired by TangentProp [15], and penalizes model fluctuations along tangent vectors in the direction of permutations. We experimentally demonstrate that training with BReg can indeed adapt to the appropriateness of invariance in a task.

Finally, we make deeper connections between existing methods, probability, and optimization. Studying exchangeable distributions provides a motivation for JP, which is in fact named after Janossy densities [83], [84], and shows that DeepSets corresponds to the strong and restrictive assumption of infinite exchangeability. We also show that sum pooling models (e.g., DeepSets) arise from infinitely strong Birkhoff regularization.

Terminology. This work exists at the intersection of numerous fields that sometimes use different terminology. The nomenclature above clarifies the most common sources of confusion. For instance, *graph* and *network* will refer to data and models, respectively. Graph *size* refers to the number of vertices. Also, while *sets* cannot have duplicates by definition, this term is commonly used in the literature even when duplicates are allowed.

Outline. In Chapter 2, we motivate artificial neural networks for modeling graphs and sets, and introduce invariant modeling. In Chapter 3, we carefully explain graph and set data, and the importance of permutation invariance. We then summarize the most important literature and highlight limitations. In Chapter 4, we introduce JP and its dual role as a more flexible approach to invariant modeling and as a unifying framework. We introduce and theoretically justify the approximation schemes, which we then explore and validate in experiments. We also see the probabilistic motivations behind JP, revealing that DeepSets makes strong assumptions. In Chapter 5, we propose BReg as an approach for enforcing the correct amount of invariance in the model, as dictated by the data. We demonstrate its success experimentally and draw connections between BReg and existing sum pooling methods.

2. BACKGROUND: INVARIANCES IN NEURAL NETWORKS

In this Chapter, we establish the necessary background of neural networks and invariances before engaging with set and graph data in Chapter 3. We (1) review neural network methodology and justify its use for graph and set data as opposed to other statistical approaches; (2) define invariant modeling of neural networks and explain its importance; (3) introduce a mathematical framework that facilitates a unified discussion of graphs and sets; and (4) review the broader literature in neural network invariances to streamline and better appreciate the related work in graphs and sets. Furthermore, while graphs and sets will be our focus, our proposed Janossy pooling is not conceptually limited to graph and set data. Discussing invariances in general opens up avenues for future exploration.

We will largely restrict the scope here and in Chapters 3 and 4 to supervised learning.

2.1 Artificial Neural Networks in Modeling Graphs and Sets

Our contributions to modeling graph and set data are largely formulated with *artificial neural networks*, henceforth simply neural networks (NNs). I refrain from writing "Deep Learning" as many successful models – certainly for Graph Neural Networks – are not "deep" models in the usual sense (see Section 3.2.2). In this section, we will fix notation and terminology, review NNs, and briefly review optimization techniques that are used to estimate NNs and Janossy pooling models. We then review properties of NNs that make them useful for modeling graphs and sets. Many references treat these ideas in greater depth, including [24], [85], [86].

2.1.1 Neural Networks and Optimization

We begin with Multilayer Perceptrons (MLPs). For this dissertation,[1] MLP refers to the class of feedforward neural networks where each layer is fully connected and all parameters are free. Specifically, let $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ denote n observations of p-dimensional data vectors and

[1]MLP may refer to slightly different models throughout the literature. Our definition helps distinguish from other neural networks.

consider first a regression problem so that $\boldsymbol{Y} \in \mathbb{R}^{n\times 1}$ are the associated targets. An MLP computes predictions $\hat{\boldsymbol{Y}}$ by first setting $\boldsymbol{H}^{(0)} = \boldsymbol{X}$, computing the following for $l = 1, \ldots, L$, $L \in \mathbb{Z}_{\geq 2}$,

$$\boldsymbol{H}^{(l)} = \psi^{(l)}\left(\boldsymbol{B}^{(l-1)} + \boldsymbol{H}^{(l-1)}\boldsymbol{W}^{(l-1)}\right), \tag{2.1}$$

and finally setting $\hat{\boldsymbol{Y}} = \boldsymbol{H}^{(L)} \in \mathbb{R}^{n\times 1}$, where $\psi^{(l)} : \mathbb{R} \to \mathbb{R}$ are nonlinear functions applied elementwise, $\psi^{(L)}$ is the identity mapping $\psi^{(L)}(x) = x$, and $\boldsymbol{B}^{(0)}, \ldots, \boldsymbol{B}^{(L-1)}$, $\boldsymbol{W}^{(0)}, \ldots, \boldsymbol{W}^{(L-1)}$ are parameter matrices of real numbers that must be estimated from data. All parameters in $\{\boldsymbol{W}^{(l)}\}_{l=0}^{L-1}$ are *free*, so we can have $(\boldsymbol{W}^{(l)})_{ij} \neq (\boldsymbol{W}^{(l)})_{km}$ for all i, j, k, m and all l. In the *bias* matrices $\boldsymbol{B}^{(l)}$, every value in a column takes the same value. Hence, each *layer* l computes one or more affine functions of its input, with $\boldsymbol{B}^{(l)}$ serving as the translation terms, followed by an elementwise nonlinear function. The modeler specifies the number of columns $d^{(0)}, \ldots, d^{(L-2)} \in \mathbb{Z}_{\geq 1}$ of $\boldsymbol{W}^{(0)}, \ldots, \boldsymbol{W}^{(L-2)}$ and $\boldsymbol{B}^{(0)}, \ldots, \boldsymbol{B}^{(L-2)}$ as hyperparameters; the remaining row and column dimensions are determined by $\boldsymbol{X}$ and $\boldsymbol{Y}$. We are also free to specify $L \in \mathbb{Z}_{\geq 2}$ and the nonlinear *activation functions* $\psi^{(l)}$, for $l < L$, such as the sigmoid $\sigma(x) = \frac{\exp(x)}{\exp(x)+1}$ or $\mathrm{ReLU}(x) = \max(0, x)$. These hyperparameters constitute the *architecture* of the network and are tuned with standard methods. L is termed the *depth* or the *number of layers*; the quantities $d^{(l)}$ are referred to as the *number of neurons*, number of *units*, or the *width* of a layer. Since $\{\boldsymbol{H}^{(l)}\}_{0<l<L}$ are not observed in the data or as predictions, we say they are *hidden*.

Next, to estimate (i.e., *train*) the parameters (i.e., *weights*), we minimize an objective function using variants of (Stochastic) Gradient Descent [24]. For regression problems, it is common to use the ℓ_1 or squared ℓ_2 loss. Given true targets $\boldsymbol{Y}$ and predictions $\hat{\boldsymbol{Y}}$, the loss function $\mathcal{L}$ satisfies $\mathcal{L}\left(\boldsymbol{Y}, \hat{\boldsymbol{Y}}\right) \in \mathbb{R}^n_{\geq 0}$ and is defined (in the case of ℓ_1) by $\mathcal{L}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})_i = |\boldsymbol{Y}_i - \hat{\boldsymbol{Y}}_i|$. Specifically, for a given choice of fixed hyperparameters, let $\boldsymbol{\Theta}$ denote a vector of all parameters $\left(\boldsymbol{W}^{(l)}, \boldsymbol{B}^{(l)}\right)_{l=0}^{L-1}$ of a neural network $f(\cdot; \boldsymbol{\Theta})$.[2] To optimize the parameters $\boldsymbol{\Theta}$ with

[2]We write $f(\cdot;\cdot)$ as the function defined by fixing the terms appearing after ";" e.g., $\boldsymbol{\Theta}$.

Gradient Descent, we first randomly initialize $\boldsymbol{\Theta}(0)$ (e.g., using Xavier initialization [87]) and iteratively compute

$$\boldsymbol{\Theta}(t+1) = \boldsymbol{\Theta}(t) - \eta(t)\nabla_{\boldsymbol{\Theta}(t)} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\Big(\boldsymbol{Y}, f\Big(\boldsymbol{X}; \boldsymbol{\Theta}(t)\Big)\Big)_i \tag{2.2}$$

for $t \in \mathbb{Z}_{\geq 0}$ and some *learning rate* $\eta(t) > 0$ until a termination criterion has been satisfied. We write $\eta(t)$ as a function so that it may vary according to a *schedule* during training, often decreasing in t [88]. The total number of iterations is termed the number of *epochs*. *Stochastic* Gradient Descent (SGD) samples and averages over subsets from $\{1, \ldots, n\}$ when computing Equation 2.2, and the parameters are updated using the gradients from each subset. SGD has been shown to converge with probability 1 given appropriate conditions [88]–[90].[3] The objective function is non-convex, with many local minima. Many variants of SGD, such as Adam [91], have been proposed and often result in better performance.

To extend beyond the regression case, we modify the final layer and the loss function. For binary classification, define $\psi^{(L)}(h) = \frac{\exp(h)}{\exp(h)+1}$, $h \in \mathbb{R}$ so that the outputs of the network are values in $(0, 1)$. The resulting probability is used to compute the cross-entropy loss. For prediction, we can threshold probabilities as in logistic regression. For K-class classification, we can define the last layer to yield K-dimensional values ($\hat{\boldsymbol{Y}} \in \mathbb{R}^{n \times K}$) and compute the row-wise softmax : $\mathbb{R}^K \to (0, 1)^K$, defined by $\text{softmax}(\boldsymbol{h})_k = \frac{\exp(h_k)}{\sum_{j=1}^{K} \exp(h_j)}$, so that the output is a probability vector. In this case, we again use cross-entropy. Other loss functions such as the hinge loss may be used. Estimation again proceeds with Gradient Descent or Stochastic Gradient Descent.

Other Neural Networks

It is instructive to contrast Equation 2.1 with other NNs (and layers thereof) such as convolutional layers [23], [92]. Such methods also compose affine functions and elementwise nonlinearities, train with SGD or its variants, and have numerous hyperparameters. The key difference for us is that all parameters of an MLP are free whereas weights in a CNNs are *shared*, or *tied,* as discussed more in Section 2.2.

[3] More details on the convergence of SGD are provided in later sections.

Another note to highlight is that Equation 2.1 assumes fixed-sized vectors as inputs. In contrast, Recurrent Neural Networks (RNNs) [93], [94] can process variable-length sequences, which is appropriate for natural language tasks such as translating sequences from English to French.

Affine layers in Equation 2.1 need not constitute an entire architecture, but can be used in conjunction with convolutional and other neural network layers. This observation is used throughout graph and set literature.

2.1.2 Benefits and Limitations of Various Neural Network Approaches

In this section, we will examine the benefits of using neural networks for tasks defined on sets and graphs. We then review the limitations of MLPs that motivate invariances.

Automating Feature Engineering: an Asset for Modeling Graph and Set Data

Not all applied problems call for neural network solutions. Yet, their ability to *automate feature engineering* makes them very useful for graph and set data [23], [95]–[97]. To motivate with a classical example, the prevailing approach to Computer Vision was for decades a two-stage procedure of first extracting hand-crafted features of images then treating them as input to a statistical model [13], [98], [99]. The advent of better computing resources, data availability, and better methodological understanding catalyzed a shift away from this approach to Convolutional Neural Networks [100]–[103]. These models can be applied directly to the raw input (e.g., a matrix of nonnegative real numbers representing pixel intensities of a grayscale image) and are trained *end-to-end,* rather than in a two-stage procedure. Without predefined features, the optimization process allows the model to "learn" the best function for the specific task. The statistical understanding behind "automatic" feature engineering is provided by, for example, [85].

To elaborate, we start with an observation that linear functions are often unlikely to capture the true underlying function of the data. One solution is to compute a *basis expansion*

$$f(\boldsymbol{x}; \boldsymbol{w}) = \sum_{m=1}^{M} w_m h_m(\boldsymbol{x})$$

where $\boldsymbol{x} \in \mathbb{R}^p$, $h_m : \mathbb{R}^p \to \mathbb{R}$ are M nonlinear transformations, and $\boldsymbol{w} \in \mathbb{R}^M$ is a parameter vector. Thus we define a nonlinear function of the data by computing transformations (such as square terms or interactions), which can express more complex functions. Now, if $L = 2$ in Equation 2.1, we can view $\boldsymbol{H}^{(1)}$ as the result of applying nonlinear transformations of the original inputs $\boldsymbol{X}$. Thus, the MLP is analogous to a basis expansion models except the transformations are data driven and learned through optimization. In other words, the intermediate layers of MLPs are effectively *derived features*. Moreover, when $L > 2$, each layer extracts hierarchical features at different levels of resolution [104]–[106]. We say that these models *learn a representation for the data*. When enough data are available to fit more complex neural network models, this is often a preferable approach.

Now, for graph regression and classification,[4] a similar trend has emerged [4], [107]–[110]. Modelers are beginning to apply specialized neural networks directly on graph data rather than on graph-specific precomputed features. This has led to performance improvements in applications such as chemistry [40], [44], [111] and neuroscience [96], [112], [113]. More broadly, defining graph features and passing them to (kernelized) models describes the approach of *graph kernels* [114]–[116]. While graph kernels remain an actively researched approach, they have been outperformed by neural networks when data and computational resources are sufficient [75], [117]. Indeed, this reflects a broader trend where neural networks, which are said to *learn a kernel*, are gaining popularity over kernel methods [95], [118].

The literature on set-like data has also followed this trend. Examples can be found in applications to point clouds (see Chapter 3) [25], [119]–[121] and in multiple instance learning [34], [122], [123].

More methods and related work are discussed at length in subsequent sections. We turn to *universal approximation*, and ultimately motivate invariances in neural networks.

Universal Approximation: A Beneficial Property but no Panacea

Multilayer Perceptrons have been shown to be *universal approximators* [24], [71], [124]–[126]. For our purposes, such results state that functions we are likely to encounter in practice

[4]Graph classification and regression refer to predicting a single property of an entire graph.

for vector data (i.e., continuous functions on finite-dimensional spaces) can be approximated arbitrarily well by some function in the class of MLPs defined in Equation 2.1. This holds for a broad choice of hyperparameters, but may require a very large number of parameters. The relationship between the depth and width is conceptually related to learning representations. Shallower networks may have difficulty representing any function without an impractically large width. This is intuitive given that subsequent layers derive effective representations of the data [104], [105].[5]

While it is encouraging that this class of models is capable of mathematically *expressing* most functions, it does not imply we can specify the correct architecture or estimate the correct parameters in practice. First, we may lack sufficient time to comprehensively tune hyperparameters. Second, we may not have enough data to estimate a model with a large number of parameters that generalizes effectively. Third, the objective is non-convex, containing many local minima, and highly complex; the estimated parameters at the end of training may not be the best for generalization [24], [103], [127], [128].

To reconcile these ideas, it is often remarked that MLPs can learn the correct function with infinite training time and data [15]. However, we have access to neither, so indeed MLPs cannot simply be applied to any task. *Invariances/Equivariances* and *inductive biases* in NNs facilitate learning models that generalize better given only finite resources, and are central to learning on sets and graphs.

2.2 Invariances and Equivariances in Neural Networks

We leverage invariances to train effective models when data and resources are finite. Invariances capture the notion that the target value for some input is unchanged by, or invariant to, transformations of the input. Prior knowledge of the invariances can make neural networks generalize (and extrapolate) better given limited data and computational resources. Invariances are also called *symmetries:* by definition, "a symmetry is a property of an object which causes it to remain invariant under certain classes of transformations" [129]. Since MLPs do not in general enforce invariances – they are defined with affine functions

[5]This intuition cannot be applied naively to graph data, as we will explain below.

in which all parameters are free – these methods introduce means for promoting invariant predictions.

As noted by Lyle, Wilk, Kwiatkowska, *et al.* [9], "invariant training" itself has many meanings in the literature, from enforcing exact invariance, promoting "insensitivity" to transformations, "approximate invariance", etc. Sometimes it is written that a training approach "encourages" invariance. We will use "invariance" broadly as all such interpretations share similar motivations and capture the expected benefits of invariant training. We will see that a highly useful paradigm for leveraging the benefits of invariant training is to specify an exactly invariant model and estimate it, thus encouraging invariance.

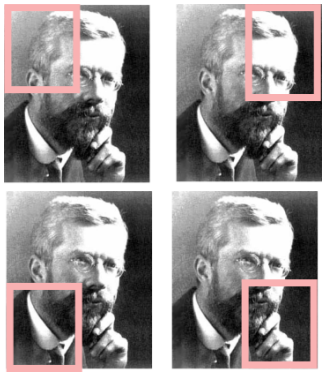
This section presents ideas in a degree of generality just above the minimum needed for modeling graph and set data. This will enable us to unify the treatment of sets and graphs, better highlight the contributions of Janossy pooling, and lay the foundation for future applications thereof.

2.2.1 Motivating Invariances with Convolutional Neural Networks

Convolutional Neural Networks (CNNs) serve as a classic example of invariances and equivariances in NNs. Beyond serving as a familiar example, the *convolution* is a fundamental invariant operation – indeed, appearing in Linear Time Invariant theory [130], [131] – and has been generalized to several domains/applications [132]–[136]. Furthermore, convolutions are appropriate for variable-sized inputs, a characteristic of graph and set data. We introduce the key ideas for CNNs applied to images here; more detailed accounts can be found in [12], [13], [24], [101], [137].

Typically, image classifiers should be invariant to local translations. If asked to identify the animal shown in an image, our answer would not depend on its exact position against the background.[6] Accordingly, the CNN model does not associate a unique free parameter to every pixel in the input image. Rather, we apply the same parametric affine function over different parts of the image, which is thus less sensitive to the animal's exact location on the image. This *conv* layer returns a hidden output $\boldsymbol{H}$. Next, in the *pooling layer*, a symmetric function such as max combines adjacent hidden representations (e.g.,

[6] We ignore pathological cases such as when the object of interest (e.g., dog, cat) slides off the image.



(a) Input & conv filters

$$\begin{pmatrix} h_1 & h_2 \\ h_3 & h_4 \end{pmatrix}$$

(b) Hidden

$$\max(h_1, h_2, h_3, h_4)$$

(c) Pooling

Figure 2.1. Simplified Illustration of Conv Operation. (a) Illustration of a conv layer on a grayscale image input (a matrix of nonnegative numbers). The pink rectangles depict inputs to an affine transformation with a small – relative to the size of the image – parameter matrix (or *filter)*. The filter slides over the image, shown by repeating the image and drawing the filter at four locations. (b) Shows the output of the conv. The value h_1 is the result of applying an affine function and activation function ψ over the top-left of the image. h_2 is the output of the same operation on the top-right, and so on. (c) Shows a max pooling step. If our task was to classify whether a tie was present in the image, the tie would likely be "detected" even if it were translated to different corners of the image.

$\max(\boldsymbol{H}_{1,1}, \boldsymbol{H}_{1,2}, \boldsymbol{H}_{2,1}, \boldsymbol{H}_{2,2})$). Together, these approximately achieve local translation invariance. An illustration is provided in Figure 2.1. The *conv* layer described performs a discrete convolution between a parameter matrix and an input with a finite number of nonzero elements (the image), hence its name.

Remark 1 *In this example, an observation about the invariances inherent in image classification motivated an architecture with fewer parameters. If we were to apply an MLP on the (vectorized) image input, there would have been a distinct free parameter for every coordinate (pixel) of the input. Additionally, the result would not be invariant in general. In contrast, the CNN architecture reuses, or shares, parameters in its operations on different parts of the image. This can mitigate overfitting. Moreover, CNNs admit computationally faster implementations, allowing for more thorough optimization and hyperparameter tuning [24], [134].*

Models that enforce invariances will not in general be capable of expressing the same class of functions as MLPs. Fortunately, CNNs have universality guarantees for the restricted class of true functions exhibiting these invariances [68]–[70]. In support of the theory, CNNs have repeatedly achieved strong empirical performance [138]–[141].

Remark 2 *Invariant models are less expressive than those that do not encode invariances. It is important to characterize their expressiveness; ideally, they should be universal within their function class so long as they perform well in practice.*

Since convolutional networks enforce translation invariance and can operate on variable-sized inputs, they are applied to other domains such as audio and graphs [10], [24], [142], [143]. Last, note that additional factors explain the success of CNNs [24], [134], [144], but this short section has served to introduce invariances in neural networks.

2.2.2 Formalizing Invariance and Equivariance with Group Theory

In his dissertation [145], Risi Kondor wrote "Since groups ultimately capture symmetries, the study of invariants has been inseparable from group theory from the start". Figure 2.2 illustrates that the symmetries of a square are described by a group of transformations; loosely speaking, the square is *invariant* to these operations. Basic group theory will facilitate a unified treatment of sets and graphs. The key idea is the *action of the symmetric group*. Simply put, this describes permutations of an arbitrary object. These definitions are standard and the reader can refer to [145]–[149] for additional background. Also note that important methodological and theoretical contributions do not *rely* on group structure, but it does provide a streamlined theoretical tool for characterizing invariances in general [9].

Definition 1 ***(Group)*** *Let $\mathbb{G} \neq \emptyset$ be a set and $\cdot : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$ be a binary operation on $\mathbb{G}$. We say $\mathbb{G}$ is a* group[7] *with the* group operation $\cdot$ *if the following properties hold: (1) (*Associative*) $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in \mathbb{G}$; (2) (*Identity*) there exists an element $E \in \mathbb{G}$ such that $a \cdot E = a = E \cdot a$ for all $a \in \mathbb{G}$, and (3) (*Inverses*): For all $a \in \mathbb{G}$, there exists $a^{-1} \in \mathbb{G}$ such that $a \cdot a^{-1} = a^{-1} \cdot a = E$.*

[7] We use $\mathbb{G}$ to denote groups and $\mathcal{G}$ to denote graphs.

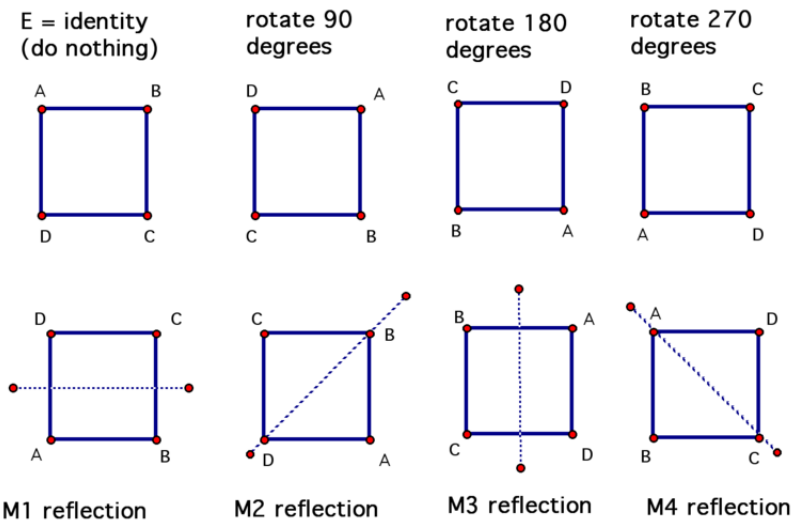


Figure 2.2. Invariances (Isometries) of a Square. This figure shows the *symmetry group* of a square, i.e., the dihedral group D_4. Intuitively, we know that rotating a square in a plane by 90° does not change its appearance on the page. Formally, the distance-preserving bijections of the square into itself form a group whose operation is the composition of functions. These transformations are precisely those that the square is *invariant* to. Note that the symmetry group is distinct from the *symmetric group*, the group of permutations. The figure is from the open project [150].

To verify that $(\mathbb{G}, \cdot)$ is a well-defined group, one often verifies that $\mathbb{G}$ is closed under $\cdot$. Some definitions (as above) let this be implicit in the fact that $\cdot$ is a well-defined binary operation with codomain $\mathbb{G}$. As an example, Figure 2.2 shows the symmetry group (which is different from the symmetric group) of a square. $\mathbb{G}$ is the set of transformations shown and $\cdot$ denotes composition of these functions. We can verify its group properties; for instance, function composition is a well-defined binary operation on these transformations, the identity mapping is present, all reflections are their own inverses, and so on.

The most important group for this dissertation is the symmetric group.

Definition 2 *(**Permutations and the Symmetric Group**) The* permutations *of a finite set A are the bijective maps of A onto itself. The set $\mathbb{S}_A = \{\pi : A \to A : \pi$ is a permutation$\}$ is a group under the operation of function composition, called the* symmetric group.[8] *$\mathbb{S}_n$ denotes the symmetric group on $\{1, 2, \ldots, n\}$, or the set of permutations of n objects.*

[8]We usually let π denote a permutation and π the constant.

For example, $\mathbb{S}_2 = \{\pi_1, \pi_2\}$, where $\pi_1(i) = i$ for $i \in \{1, 2\}$ is the identity permutation and $\pi_2(1) = 2$ and $\pi_2(2) = 1$ is the permutation that swaps indices. Next, *group actions* help to formalize invariant functions. This more general notation facilitates a unified treatment of sets and graphs.

Definition 3 *(**Group action**) Let A be a set and $\mathbb{G}$ be a group with operation $\cdot$ and identity element E. A* left action*, or simply* action*, of $\mathbb{G}$ on A is a function $\gamma : \mathbb{G} \times A \to A$ such that $\gamma(E, a) = a$ and $\gamma\Big(g, \gamma(h, a)\Big) = \gamma(g \cdot h, a)$ for all $g, h \in \mathbb{G}$ and $a \in A$. It is common to use the shorthand $g \cdot a$ rather than $\gamma(g, a)$.*

Intuitively, a group action on a set A takes in a group element and "returns" a function that then operates on the original set A. The first property states that a group action associates the group identity $E \in \mathbb{G}$ with the identity function on A. The symmetric group $\mathbb{S}_n$ acts on $\mathbb{R}^n$ by permuting vectors: for $\boldsymbol{x} \in \mathbb{R}^n$ and $\pi \in \mathbb{S}_n$, the action is defined to be[9]

$$\pi \cdot \boldsymbol{x} = \Big(x_{\pi^{-1}(1)}, \ldots, x_{\pi^{-1}(n)}\Big).$$

For example, taking $\pi \in \mathbb{S}_3$ defined by $\pi(1) = 2$, $\pi(2) = 3$, and $\pi(3) = 1$,

$$\pi \cdot (x_1, x_2, x_3) = (x_{\pi^{-1}(1)}, x_{\pi^{-1}(2)}, x_{\pi^{-1}(3)}) = (x_3, x_1, x_2).$$

More generally, consider an order $(d+1)$ array (or *tensor*), $\mathbf{X} \in \mathbb{R}^{n \times \cdots \times n \times p}$ with $(\mathbf{X})_{i_1, \ldots, i_d} \in \mathbb{R}^p$. This data structure associates tuples of d indices $(i_1, \ldots, i_d)$ with a p-dimensional data vector. Beyond being suitable for encoding graphs and sets (see Chapter 3), tensors can be used to study the association between human subjects (axis one), words (axis two), and brain activity at different voxels (axis three), as well as product-item interactions [152]–[154]. While these cases would be described by tensors of shape $n_1 \times n_2 \times p$, we can assume that all axes but the last have the same shape for our purposes (i.e., $n_1 = n_2$). Thus, following

[9]Keith Conrad gives a detailed explanation of why we do not define $\pi \cdot \boldsymbol{x} = \big(x_{\pi(1)}, \ldots, x_{\pi(n)}\big)$ in [151].

Maron, Fetaya, Segol, *et al.* [149], we can define the action of the symmetric group on arrays $\mathbb{R}^{n\times\cdots\times n\times p}$ by permuting all the indices of the array except for the last,

$$(\pi\cdot\mathbf{X})_{i_1,\ldots,i_d,j}=\mathbf{X}_{\pi^{-1}(i_1),\ldots,\pi^{-1}(i_d),j},\quad \pi\in\mathbb{S}_n,\mathbf{X}\in\mathbb{R}^{n\times\cdots\times n\times p}. \tag{2.3}$$

There are many other groups and possible actions, but these will be sufficient for this dissertation. Now we can formally define an invariant function. In what follows, sets $\mathcal{X}$ and $\mathcal{Y}$ can be thought of as the input feature space and the target space in some modeling task, respectively.

Definition 4 *($\mathbb{G}$-**invariant function**) Let $\mathbb{G}$ be a group with binary relation $\cdot$ that acts on the set $\mathcal{X}$. Let $\mathcal{Y}$ be a set and $f:\mathcal{X}\to\mathcal{Y}$ a function. We say f is* $\mathbb{G}$-invariant *if*

$$f(g\cdot\boldsymbol{x})=f(\boldsymbol{x})\quad\forall\boldsymbol{x}\in\mathcal{X},\forall g\in\mathbb{G}.$$

If $\mathcal{X}=\mathbb{R}^{n^d\times p}$ for some $d,n\in\mathbb{Z}_{\geq 1}$, we say a function f is *permutation-invariant* if $f(\pi\cdot\boldsymbol{x})=f(\boldsymbol{x})$ for all $\pi\in\mathbb{S}_n$ and $\boldsymbol{x}\in\mathcal{X}$. An example is $f(\mathbf{X})=\sum_{i_1}\cdots\sum_{i_d}\sum_{i_{d+1}}(\mathbf{X})_{i_1,\ldots,i_d,i_{d+1}}$. Next we define the more general notion of equivariance. For this definition, $\mathcal{Y}$ can be thought of as the space of hidden variables or output space of a neural network.[10]

Definition 5 *($\mathbb{G}$-**equivariant function**) Let $\mathbb{G}$, $\cdot$, $\mathcal{X}$, $\mathcal{Y}$ and f be defined as in Definition 4. Assume that $\mathbb{G}$ acts on both $\mathcal{X}$ and $\mathcal{Y}$. We say that f is $\mathbb{G}$-equivariant if*

$$f(g\cdot\boldsymbol{x})=g\cdot f(\boldsymbol{x})\quad\forall\boldsymbol{x}\in\mathcal{X},\forall g\in\mathbb{G}.$$

Note that $\mathcal{X}$ need not equal $\mathcal{Y}$ but the group action $\mathbb{G}$ should be well-defined on both. For instance, if $\mathcal{X}=\mathbb{R}^{n\times p}$, $\mathcal{Y}=\mathbb{R}^p$, and $\mathbb{G}=\mathbb{S}_n$, the action is still well-defined. Consider for example row summation $f(\boldsymbol{X})_i=\sum_{j=1}^p(\boldsymbol{X})_{i,j}$ for all $\boldsymbol{X}\in\mathbb{R}^{n\times p}$. Permuting the rows does not change the sums, only their ordering in the output matrix: $f(\pi\cdot\boldsymbol{X})=\pi\cdot f(\boldsymbol{X})$. However, if we put $\mathcal{X}=\mathbb{R}^{n\times p}$ and $\mathcal{Y}=\mathbb{R}^{m\times p}$ for $n\neq m$, then we cannot say $\mathbb{S}_n$ acts on both. For the present dissertation, we do not need a more general definition.

[10] This can be generalized to $\mathbb{G}$,$\mathbb{H}$-equivariance but is not necessary for the main discussion of Janossy pooling.

Fact 1 *A function $f = f_L \circ f_{L-1} \circ \cdots \circ f_1$ is $\mathbb{G}$-invariant if $f_1, \ldots, f_{L-1}$ are $\mathbb{G}$-equivariant and f_L is $\mathbb{G}$-invariant. The converse is not true.*

Finally, permutation matrices [155] will be convenient in calculations.

Definition 6 *A matrix $\boldsymbol{P} \in \{0,1\}^{n \times n}$ is a* permutation matrix *if it is obtained by permuting the rows of the $n \times n$ identity matrix $\boldsymbol{I}_n$.*

Every row and column of a permutation matrix has exactly one 1, the rest 0. Permutation matrices can affect the action of $\mathbb{S}_n$ on $\mathbb{R}^{n \times p}$; $\boldsymbol{PX}$ permutes the rows of a matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$. An example is shown in the appendix. Another useful property is that the inverse of a permutation matrix is its transpose.

Fact 2 *If $\boldsymbol{P}$ is any $n \times n$ permutation matrix, then $\boldsymbol{P}\boldsymbol{P}^T = \boldsymbol{P}^T\boldsymbol{P} = \boldsymbol{I}_n$, where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix.*

2.2.3 Enforcing Invariances to Improve Performance

In this section, we first provide an overview of common invariant modeling methods then review why they improve (generalization) performance. We conclude with a summary of recent literature invoking invariances to improve *extrapolation* performance to provide further insights into invariances as well as to provide hints of opportunities for future work.

Weight Sharing and Convolutions

An MLP $f(\cdot; \boldsymbol{\Theta}) : \mathcal{X} \to \mathcal{Y}$ cannot be $\mathbb{G}$-invariant in general if all its weights are free. Some must be tied (i.e., *shared*) [18], [19], [148], [156]. Ravanbakhsh, Schneider, and Poczos [19] demonstrate a method for tying the parameters of layers in Equation 2.1 to achieve a desired equivariance *uniquely*. Here, *uniquely* refers to the fact that unwanted invariances are not enforced; functions with a singleton image are invariant to every transformation but are not very useful. A similar analysis in a slightly less general context was performed earlier by Shawe-Taylor [18]. In Hartford, Graham, Leyton-Brown, *et al.* [154] and Maron, Ben-Hamu, Shamir, *et al.* [157], the authors take a different approach and derive a basis for the space

of invariant/equivariant functions. In these approaches, the only parameters are the scalars associated with each basis component in a linear combination, and there are at most 15.

We saw that convolutions reduce the number of parameters and are a fundamental operation for invariances. These have been generalized to several domains. Spherical CNNs [21] and steerable filters [22] were developed for rotation invariance. Cohen and Welling [135] develop G-CNNs ("G" stands for group-equivariant) which use fewer free parameters than the standard conv layer.

Special Invariant Layers

Beyond tying weights, specialized layers can help to achieve invariance. We have already seen pooling functions in CNNs that facilitate local translation invariance. Recent works have proposed new pooling operations for CNNs to achieve a wider scope of invariances [20], [158], [159]. Instance normalization is an averaging method proposed to improve contrast invariance in image stylization [160]. Pyramidal layers have been proposed to improve scale invariance in computer vision tasks [161], [162]. More generally, an entire literature exists on *spiking neurons*, which rethink the operations and training procedure of artificial neural networks, and some of their promise has been linked to shift-invariant and scale-invariant properties [163]–[165]. Interestingly, pooling operations have also appeared outside of neural networks; the *symmetrized* kernel, which bears similarity to the Janossy pooling framework [145], is a group-invariant kernel. We will see that specialized architectures are popular in modeling graphs and sets.

Variations on Training Strategies

Another approach to training approximately invariant models is to modify the training strategy instead of the model architecture. Unlike weight sharing, these methods do not enforce exact invariance. Rather, they encourage models to be less sensitive to given transformations [9], but have been used successfully to improve the performance of neural networks. The following have close parallels to Janossy pooling.

Data augmentation [16] is considered one of the simplest approaches [19] and is still quite effective [166]. To learn models whose predictions are less sensitive to some class of transformations, we apply those transformations to inputs in the training data (without changing the target) and train in the usual way. For example, rotating images in the training data can be used to promote rotation invariance. Empirically, training with data augmentation can lead to models that are "more invariant" and improve generalization [17], [167]. However, Lyle, Wilk, Kwiatkowska, *et al.* [9] point out that data augmentation may only reduce transformation sensitivity over in-distribution, but not out-of-distribution, data. There have been variations of data augmentation. For instance, claiming that data augmentation still models nuisance variation, Laptev, Savinov, Buhmann, *et al.* [166] apply multiple transformations to the same input, pass each through the model, and predict the elementwise maximum of the output vectors.

TangentProp [15] is motivated by the observation that if the true function is invariant to some transformation, then its directional derivatives are zero along the direction of the transformation. Accordingly, TangentProp penalizes the norm of the directional derivatives during optimization. In particular, let $\boldsymbol{x} \in \mathbb{R}^p$ and $s_i : \mathbb{R}^p \times \Omega_i \to \mathbb{R}^p$, $i = 1, \ldots, m$ be $m \in \mathbb{Z}_{\geq 1}$ transformations of an input $\boldsymbol{x}$. For example, $s(\cdot, \omega)$ may rotate a vector by angle $\omega \in [0, 2\pi)$.[11] To be clear, the quantities ω are not parameters to be estimated, but instead describe transformations of the input. Then, the TangentProp penalty term is

$$\sum_{i=1}^{m} \left\| f'(\boldsymbol{x}) \cdot \frac{\partial s_i(\boldsymbol{x}, \omega)}{\partial \omega}\Big|_{\omega=0} \right\|_2^2, \tag{2.4}$$

where f is the neural network model, $f'(\boldsymbol{x})$ is the Jacobian (or gradient) of f with respect to its input, and $\cdot|_{\omega=0}$ denotes the quantity evaluated at $\omega = 0$. The tangent vectors are approximated by finite differences. Interestingly, the authors invoke group theory [168] to argue that invariance to m directional derivatives will induce invariance to linear combinations thereof. The authors argue that TangentProp is more data efficient than data augmentation since, in the case of data augmentation, transformed inputs are highly correlated with the originals. This method is part of a wider class of approaches that invoke the *manifold hy-*

[11] Some operations, such as rotations of digital images, need to be smoothed [15].

pothesis. It is thought that data points from a similar class concentrate close together on a lower-dimensional manifold of the data, and penalizing tangent vectors prevents the function from assigning different classes to those data [24].

Strengths and Weaknesses

Weight sharing can mitigate overfitting, improve sample complexity, and tighten generalization bounds by reducing the number of parameters [9], [22], [85], [135], [169]–[171]. VC theory [85], [169], [172], [173] provides insights into this general result and in particular some of the weight sharing schemes for sets and graphs, despite its possible limitations to studying general neural networks.[12] Weight sharing reduces the hypothesis space (for a fixed architecture), thereby the VC dimension, and consequently the sample size needed to fit the model. The assumption of invariances implies a partition of the input space $\mathcal{X}$ into sets $\text{orbit}(\boldsymbol{x}) = \{g \cdot \boldsymbol{x} : g \in \mathbb{G}\}$ for $\boldsymbol{x} \in \mathcal{X}$, where $\mathbb{G}$ is a group acting on $\mathcal{X}$; a coarser partition implies a greater reduction in VC dimension. It is interesting to point out that reducing the input space is a motivation for symmetrized kernels, which average over the action of a group, an approach similar to Janossy pooling [145].

Weight sharing and convolutions can be considered as placing an "infinitely strong prior" on the parameters of an MLP that forces the model to respect desired invariances. In the context of Convolutional Neural Networks, Goodfellow, Bengio, and Courville [24] write: "This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor but shifted in space." We know that a strong prior, when appropriate for the problem, effectively increases the sample size and affects better inference [175]. However, just as poorly specified priors inhibit accurate inference, strictly enforcing invariances will hurt predictive performance when the true task does not exhibit those invariances. Indeed, this motivates some approaches that try to "learn" the correct type or extent of invariance from the data [176]–[180]. Other examples in which incorrectly enforcing invariance hurts performance are discussed in [181], [182].

[12]There has been some recent progress in studying the VC dimension of neural networks [174].

Many invariant models are universal approximators among "true" functions respecting those invariances [58], [68], [69], [149], [183]–[186]. However, some invoke constructions that are not practical, as is the case, for example, in [149], [183] (cf. [187]).

There are a plethora of examples demonstrating that data augmentation can improve generalization performance, as described in [24], [188] and references therein. It can be applied generally and does not confine modelers to using predefined architectures to achieve invariance. One interesting theoretical justification is that data augmentation provides lower-variance estimates of an *augmented* risk and its gradients, compared to the estimates of risk from "standard" training, if the group of invariances is small enough to feasibly apply all transformations during training [9]. However, if the group of transformations is large (i.e., continuous or a large discrete set) then the variance in the risk estimation could overwhelm this benefit. Additionally, this approach may be inefficient due to correlations between the original image and its transformations, as well as the added overhead [15]. To the best of the author's knowledge, most of the community's understanding of the benefits of data augmentation and the merits of its various forms are driven by empirical rather than theoretical work (cf. [188]).

2.2.4 Inductive Biases and Extrapolation

We will conclude this section by briefly discussing benefits that invariances may furnish to good *extrapolation* performance. Humans do not require observing cars at all angles, forms of ambient lighting, colors, etc. in order to recognize a car in the future. Humans can extrapolate. In contrast, many existing approaches suffer from what Mouli and Ribeiro [179] call an "unseen-is-underspecified" hypothesis. It is believed that intelligent beings exploit symmetries in their mental representations, pointing towards the need for invariant models [50], [179]. Anselmi, Leibo, Rosasco, *et al.* [170] write that "[this is] is supported by previous theoretical work showing that almost all the complexity in recognition tasks is often due to the viewpoint and illumination nuisances that swamp the intrinsic characteristics of the object", citing in particular [189]. The review paper by Battaglia, Hamrick, Bapst, *et al.* [50] argues – among other points – that the community should encode *inductive biases* into models, which refers broadly to techniques and architectures that "bias" towards estimating

invariant models.[13] Interestingly, they argue that models of *graphs*, which capture relationships among entities, will be a key methodology going forward. Mouli and Ribeiro [179] advance a framework that improves extrapolation performance by leveraging invariances. Along a slightly different direction, Arjovsky, Bottou, Gulrajani, *et al.* [190] propose a new paradigm that enforces invariance across different *training environments* to improve generalization and extrapolation capabilities. Additional literature on the role of invariances in extrapolations can be found in [191] and the references therein.

[13] In this usage, bias refers to neither the "intercept" term in neural networks nor the usual statistical definition of "bias" when describing estimators.

3. GRAPHS AND SETS

With the broader context of invariances and neural networks established, we introduce neural network approaches for graph and set data. The reader may find the review of such literature slightly disjointed, which in fact highlights a major contribution of Janossy pooling (JP); JP provides a theoretical and methodological framework applicable to both. Of course, there are unavoidable nuances of both data types, which we will highlight. We focus on supervised learning tasks, undirected graphs, and *joint* permutation invariance (as opposed to *separate* permutation invariance) to simplify the discussion. Extensions will be deferred to Section 4.3.

3.1 Permutation Invariance in Graphs and Sets

We begin by illustrating the appropriate invariance for graphs and sets and treating the nuances of both. Then we introduce a unified notation for discussing both abstractly. Specifically, we will ultimately write $\underline{\mathbf{x}}$ for either sets or graphs and $\pi \cdot \underline{\mathbf{x}}$ for permutations thereof.

3.1.1 Motivation and Basic Definitions

There are two common properties between graphs and sets. Models thereof are often permutation-invariant, and datasets can contain variable-size inputs.[1]

By definition, sets do not have order: $\{1, 2, 3\} = \{3, 2, 1\} = \{2, 1, 3\}$. However, to store a set on a computer, we cannot escape assigning some ordering. Thus, we say that sets are stored (i.e., *encoded*[2]) as *sequences,* using the term "sequence" rather than "vector" to highlight their variable-size nature. Concretely, two valid encodings of the set $\{1, 2, 3\}$ are $(1, 2, 3)$ and $(2, 3, 1)$ yet $(1, 2, 3) \neq (2, 3, 1)$. To encode a dataset of N sets where the longest has length p, we can specify some ordering for each and create a matrix $\boldsymbol{X} \in \mathbb{R}^{N \times p}$. Sequences shorter than p are padded with a special character, as is typical in the field. Whenever we

[1] In graph theory, *size* refers to the number of edges and *order* the number of vertices. However, it would be terribly confusing to refer to graph order in the context of invariances, so we follow the convention in the literature and write *size* as the number of vertices.

[2] "Encode" here is unrelated to its appearance in "auto*encoder*", and is chosen to avoid the special word "representation".

write $\boldsymbol{X}$, we implicitly suppose some arbitrary ordering has been chosen. Once the sets have been encoded as a matrix, we may be tempted to pass $\boldsymbol{X}$ to an MLP (Equation 2.1) and inherit its automatic feature engineering and universal approximation capabilities, but we know that this would fail to leverage the theoretical and empirical benefits of invariant models (Chapter 2). Rather, our models should respect the invariance inherent in the problem; all permutations of the input set should yield the same prediction.

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a pair containing a *vertex set* $\mathcal{V}$ and an *edge set* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$.[3] For simplicity, we assume graphs are *undirected*: $(u, v) \in \mathcal{E} \iff (v, u) \in \mathcal{E}$. Figure 3.1 shows an example of encoding a graph. First we order (i.e., *label*) the vertices, then create an *adjacency matrix*. An *adjacency matrix* is a square matrix s.t. $(\boldsymbol{A})_{(u,v)} = 1$ if $(u, v) \in \mathcal{E}$ and $(\boldsymbol{A})_{(u,v)} = 0$ otherwise. There are multiple valid orderings, leading to possibly different adjacency matrices, so the encoding is ambiguous.

Remark 3 *The encoding of sets and graphs is ambiguous, with $n!$ possible orderings. These reorderings are permutations, so graph and set models should be permutation-invariant. An imprecise (for now) but intuitive condition we would like to satisfy is: if $\underline{\boldsymbol{x}}$ is any set or graph then $f(\pi \cdot \underline{\boldsymbol{x}}) = f(\underline{\boldsymbol{x}})$ for any permutation π. To be precise, we define $\pi \cdot \underline{\boldsymbol{x}}$ on both sets and graphs. A summary without group theory is provided in Section 3.1.4.*

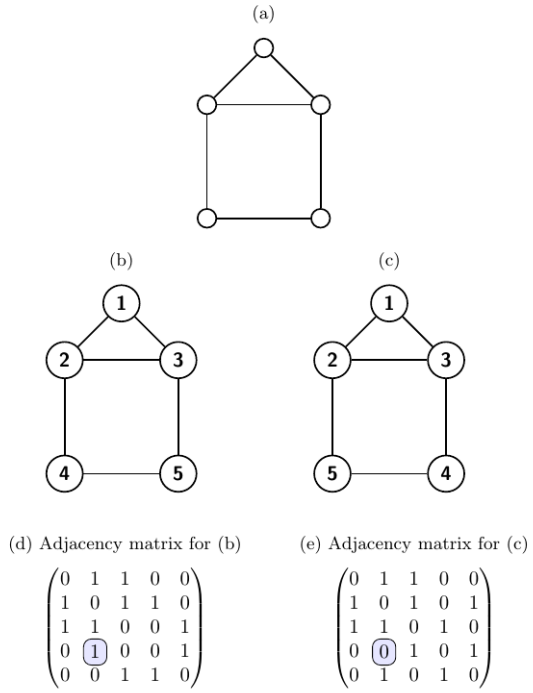
To better formalize this, we will need to introduce notation.

Definition 7 *We use colon notation to facilitate indexing matrices and tensors. Given integers a, b such that $b > a$, we write $a : b = (a, a+1, \ldots, b-1, b)$. The notation $a :$ denotes $(a, a+1, \ldots, n)$ where $n \geq a$ is the total number of possible values in the given axis. Simply writing $:$ denotes all entries for the given axis. Indexing starts at 1 and the bounds a, b are always included.*

3.1.2 Definitions for and Nuances of Graphs

In most graph applications, we do not use solely the adjacency matrix, which only captures relationships between entities (i.e., *topological* information). In applications, graphs

[3] $\mathbb{G}$ denotes a group and $\mathcal{G}$ denotes a graph.



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & \boxed{1} & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & \boxed{0} & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 3.1. Graphs and Orderings. (a) An example of an abstract graph. (b) and (c) show two equally valid orderings thereof, by drawing numbers in $\mathcal{V} = \{1, \ldots, 5\}$ on the vertices. The two ordered graphs differ by a swap of vertices "4" and "5". (d) and (e) show the corresponding adjacency matrices. Notice that these matrices are not equivalent, and we have highlighted an element in row 4 where they differ.

also contain vertex- and edge-level information. An example lies in using graphs to predict molecular properties [40], [41], [44], [192]. Figure 3.2 shows a simple example encoding a molecule, using feature vectors to capture atom type and the number of bonds (single, double, etc.). In practice, one may encode many additional vertex- and edge-level features, such as charge and aromaticity [193].

Accordingly, we encode a graph of size n with a *vertex feature matrix* $\boldsymbol{F} \in \mathbb{R}^{n \times d_v}$, $d_v \geq 1$, and an *adjacency tensor* $\mathbf{A} \in \mathbb{R}^{n \times n \times d_e}$, $d_e \geq 1$. Let $\pi : \mathcal{V} \to \{1, \ldots, n\}$ denote an ordering and define $(\boldsymbol{F})_{\pi(v),:}$ to be the vector of vertex features associated with vertex v. If the raw data do not provide vertex features, it is common to precompute simple graph statistics for each vertex or simply let $\boldsymbol{F} = (1, \ldots, 1)^{\mathrm{T}}$ [56], [194]. Thus, the vertex feature dimension always satisfies $d_v \geq 1$. Next, for every pair $(u, v) \in \mathcal{V}$, we define $(\mathbf{A})_{\pi(u),\pi(v),1} = 1$ if $(u, v) \in \mathcal{E}$ and $(\mathbf{A})_{\pi(u),\pi(v),1} = 0$ otherwise, so the first frontal slice of $\mathbf{A}$ encodes the usual *adjacency matrix*. The remaining dimensions $(\mathbf{A})_{\pi(u),\pi(v),2:}$ contain any additional edge features such as bond type and distance.[4] Notice that $d_e \geq 1$. Henceforth we simply write $\mathbf{A}$ and $\boldsymbol{F}$ and implicitly assume an ordering has been defined. This does not imply the ordering is the "true" ordering.

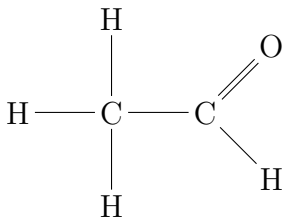
Molecules have varying numbers of atoms, so datasets contain variable-size graphs. However, the feature dimensions (e.g., d_v) do not typically vary from graph to graph. All definitions below apply to variable-size graphs.

Definition 8 *A* graph data input *of arbitrary size $n \in \mathbb{Z}_{\geq 1}$ with $d_v \in \mathbb{Z}_{\geq 1}$ dimensional vertex features and $d_e \in \mathbb{Z}_{\geq 1}$ dimensional edge features is a pair $\mathcal{G} = (\mathbf{A}, \boldsymbol{F})$ of an adjacency tensor $\mathbf{A} \in \mathbb{R}^{n \times n \times d_e}$ and vertex feature matrix $\boldsymbol{F} \in \mathbb{R}^{n \times d_v}$. We may also write $\underline{\boldsymbol{x}} = (\mathbf{A}, \boldsymbol{F})$ to unify treatment with sets. It is understood that some arbitrary ordering was specified to define the rows of $\boldsymbol{F}$ and $\mathbf{A}$. $|\mathcal{G}|$ denotes the number of vertices in the graph.*

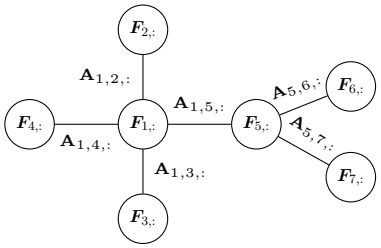
We saw in Figure 3.1 and Remark 3 that graph models should be permutation-invariant. To make this precise, we define the action of the symmetric group $\mathbb{S}_n$ on the set of all graphs with features. Intuitively, different orderings result in permutations of the rows (but not the columns) of $\boldsymbol{F}$ and of the first two modes of $\mathbf{A}$. Vertex/edge feature vectors (e.g., columns of

[4]Please refer to Definition 7 for the meaning of colon notation.

(a) Molecule



(b) Graph with Features



(c) Vertex Feature Matrix

$$\boldsymbol{F} = \begin{matrix}(\mathrm{C})\\(\mathrm{H})\\(\mathrm{H})\\(\mathrm{H})\\(\mathrm{C})\\(\mathrm{O})\\(\mathrm{H})\end{matrix}\begin{pmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 1 & 0\\0 & 1 & 0\\1 & 0 & 0\\0 & 0 & 1\\0 & 1 & 0\end{pmatrix}$$

(d) Part of Adjacency Tensor

$\mathbf{A}_{5:7,5:7,:} =$

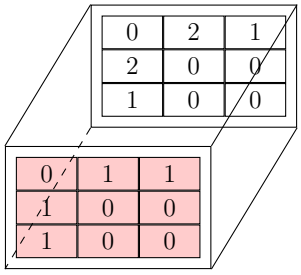


Figure 3.2. Encoding a Graph with Features. (a) displays a molecule and (b) illustrates at a high-level a corresponding graph with features, for some vertex ordering indicated by the first index of $\boldsymbol{F}$. The layout of the drawing mimics that of the molecule to ease understanding, but in general the positions of vertices on the page carry no meaning. We can see that vertices encode atoms and edges encode bonds. While in reality each pair of vertices corresponds to a fiber of the tensor $\mathbf{A}$, regardless of whether an edge is present, we only draw $\mathbf{A}$ on the edges to avoid clutter. (c) shows that atom types are encoded by one-hot vectors in rows $\boldsymbol{F}_{v,:}$, where $v \in \{1, 2, \ldots, 7\}$ is the index associated with a vertex in the ordering. For example, $(\boldsymbol{F})_{1,:} = (\boldsymbol{F})_{5,:} = (1, 0, 0)$ represents carbon and $(\boldsymbol{F})_{6,:} = (0, 0, 1)$ represents oxygen. (d) Each pair $(v, u) \in \{1, \ldots, 7\}^2$ is associated with a vector $(\mathbf{A})_{v,u,:}$ where $(\mathbf{A})_{v,u,1}$ indicates absence/presence of an edge and $(\mathbf{A})_{v,u,2:}$ holds additional edge features (not shown). We show only a sub-tensor of $\mathbf{A}$, for space considerations, corresponding to vertices 5, 6, and 7 (Carbon, Oxygen, and Hydrogen on the right). The closer slice with shaded cells is the submatrix of the standard adjacency matrix for the selected vertices, with 1 denoting edge presence. The farther slice indicates whether the bond, if present, is single or double; for purely illustrative purposes, we use a "continuous" variable to count the number of edges as opposed to a one-hot encoding for single-, double-, and triple-bonds. We write this graph as $\mathcal{G} = (\mathbf{A}, \boldsymbol{F})$ where $\mathbf{A}$ is $7 \times 7 \times 2$ and $\boldsymbol{F}$ is as shown.

$\boldsymbol{F}$) are never permuted. We decide in advance which coordinates of the vectors correspond to which features, as we would with linear regression, and it does not depend on the vertex ordering.

Definition 9 *Fix $n, d_v, d_e \in \mathbb{Z}_{\geq 1}$ and denote by $\mathscr{G}_{n,d_v,d_e} = \left\{(\mathbf{A}, \boldsymbol{F}) : \mathbf{A} \in \mathbb{R}^{n\times n\times d_e}, \boldsymbol{F} \in \mathbb{R}^{n\times d_v}\right\}$ all graphs with n vertices of given feature dimensions. The set of variable-size graphs is $\mathscr{G}_{d_v,d_e} = \bigcup_{n\in\mathbb{Z}_{\geq 1}} \mathscr{G}_{n,d_v,d_e}$. With a slight abuse of notation, we define the action of the symmetric group $\mathbb{S}_n$ on $\mathscr{G}_{d_v,d_e}$ as*

$$\pi \cdot \mathcal{G} \stackrel{def}{=} (\pi \cdot \mathbf{A}, \pi \cdot \boldsymbol{F}), \quad \text{for any } \mathcal{G} = (\mathbf{A}, \boldsymbol{F}) \in \mathscr{G}_{d_v,d_e} \text{ and } \pi \in \mathbb{S}_{|\mathcal{G}|},$$

where the operation $\pi \cdot \mathbf{A}$ is applying the action on $\mathbb{R}^{n\times n\times d_e}$ and $\pi \cdot \boldsymbol{F}$ the action on $\mathbb{R}^{n\times d_v}$ defined in Section 2.2.2 (see Equation 2.3). Notice that, by definition, this action does not permute the last dimension corresponding to vertex and edge features.

This definition satisfies the conditions of a group action. We say that two graphs $\mathcal{G}_1, \mathcal{G}_2$ are *isomorphic* if there exists some $\pi \in \mathbb{S}_{|\mathcal{G}|}$ such that $\pi \cdot \mathcal{G}_1 = \mathcal{G}_2$. We say π is an *isomorphism.*

Definition 10 *(**Graph permutation invariance and equivariance**) Let $d_v, d_e \in \mathbb{Z}_{\geq 1}$ be arbitrary feature vector dimensions and $\mathcal{Y}$ be a task-appropriate target space. We say a function $f : \mathscr{G}_{d_v,d_e} \to \mathcal{Y}$ is* permutation-invariant *or* isomorphism-invariant *if for all $\underline{\boldsymbol{x}} = \mathcal{G} \in \mathscr{G}_{d_v,d_e}$,*

$$f(\pi \cdot \underline{\boldsymbol{x}}) = f(\underline{\boldsymbol{x}}),$$

where we write graphs as $\underline{\boldsymbol{x}}$ to help unify notation for sets and graphs. A similar definition exists for equivariance (see Definition 5 with $\mathbb{G}$ the symmetric group).

Example 1 *(**Permutation-sensitive graph function**) Define $f : \mathscr{G}_{d_v,d_e} \to \mathbb{R}$ by $f(\mathcal{G}) = f(\mathbf{A}, \boldsymbol{F}) = \sum_{j=1}^{|\mathcal{G}|} (\mathbf{A})_{1,j,1}$. This is the sum of the first row of the adjacency matrix, or the* degree *of the "first" vertex in some implicit ordering. This is not a permutation-invariant function since different vertices typically have different degrees. Thus, the value of the function depends on whichever vertex is first in the ordering.*

The following example is important for Message Passing Graph Neural Networks.

Example 2 *(**Permutation-invariant graph function**) Suppose $d_v = d_e = 1$, so that the adjacency tensor reduces to an adjacency matrix $\boldsymbol{A}$, and $\boldsymbol{F}$ is a column vector. Define the function $f(\mathcal{G}) = f(\boldsymbol{A}, \boldsymbol{F}) = \boldsymbol{AF}$. We will show that this is a permutation-equivariant function. Written in terms of permutation matrices (Definition 6), the permutation of a graph $(\boldsymbol{A}, \boldsymbol{F})$ is carried out as $(\boldsymbol{PAP}^T, \boldsymbol{PF})$. By Fact 2, we can write*

$$f(\boldsymbol{PAP}^T, \boldsymbol{PF}) = (\boldsymbol{PAP}^T)(\boldsymbol{PF}) = \boldsymbol{PAF} = \boldsymbol{P}f(\boldsymbol{A}, \boldsymbol{F}),$$

for any $|\mathcal{G}| \times |\mathcal{G}|$ permutation matrix. In other words, $f(\pi \cdot \mathcal{G}) = \pi \cdot f(\mathcal{G})$, so f is permutation-equivariant. Now, if g is any permutation-invariant function such as sum or max, then $g \circ f$ is a permutation-invariant graph function.

Remark 4 *In Example 2, the function $\boldsymbol{AF}$ computes the sum over vertex* neighborhoods. *In an undirected graph with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, the neighborhood $\mathcal{N}(v) = \{u \in \mathcal{V} : (u, v) \in \mathcal{E}\}$ of v is the set of vertices connected to v by an edge. This example suggests that we can construct permutation-invariant or -equivariant functions of graphs by defining permutation-invariant functions of vertex neighborhoods; the function should be permutation-invariant as the neighborhood set will be encoded as a sequence whose order depends on the vertex ordering. The action of permutations $\pi \cdot \mathcal{G}$ changes the order in which neighboring vertices are "visited".*

3.1.3 Definitions for and Nuances of Sets

Set data contain a variable number of elements of arbitrary complexity and the ordering does not matter. Mathematically, sets should not contain duplicates. However, consistent with the literature and for brevity, we will write *set* instead of *multiset* even when duplicates are present. Existing methods like DeepSets [26] can process multisets,[5] as can Janossy pooling.

A classic set task is point cloud classification [25]–[28] where an object (e.g., a chair or table) is depicted by a set of three dimensional coordinates tracing out the object, as in Figure 3.3. Clearly, reordering the same points (in computer storage) without changing the

[5]While some proofs in the DeepSets paper assumed sets, later works extended it to multisets.

Figure 3.3. Example Set Data: a Point Cloud. This point cloud is a set of three dimensional coordinates $\{(x_i, y_i, z_i)\}_{i=1}^n$ tracing out an object. We would encode this set by pre-specifying some ordering and arranging the coordinates as vectors in a matrix $\boldsymbol{S} \in \mathbb{R}^{n\times 3}$. The point cloud shown is from Meng, Yang, Ribeiro, *et al.* [27].

coordinates does not change the shape of the object in question. Moreover, point clouds are often subsampled prior to analysis, adding additional ambiguity to their ordering. Another example set task is in multiple instance learning for high resolution medical images. To overcome the sheer size of these images, modelers break them into sets of image patches and predict whether the set of patches represents a healthy or afflicted patient [32]–[34].

Abstractly, a set of vectors with some implicit ordering is a sequence $(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)})$, for $\boldsymbol{x}^{(i)} \in \mathbb{R}^{d_s}$ and some $d_s \geq 1$, that we encode as a matrix $\boldsymbol{S} \in \mathbb{R}^{n\times d_s}$ such that $(\boldsymbol{S})_{i,:} = \boldsymbol{x}^{(i)}$. In practice, we are not limited to modeling sets of vectors and can process sets of images or other structures. However, it is more convenient to write elements as vectors. Theoretically, we can suppose images are vectorized, and similarly for other data types.

To define the action of the symmetric group on set data, let $d_s \in \mathbb{Z}_{\geq 1}$, and $\mathscr{S}_{n,d_s} = \mathbb{R}^{n\times d_s}$ denote the set of all sequences of length n whose vectors are of dimension d_s. Let $\mathscr{S}_{d_s} = \bigcup_n \mathscr{S}_{n,d_s}$ denote variable-length inputs. For $\boldsymbol{S} \in \mathscr{S}_{n,d_s}$, we will write $|\boldsymbol{S}|$ as its number of rows, equally the cardinality of the set being encoded. The action of the symmetric group on $\mathscr{S}_{n,d_s}$ is the one defined in Section 2.2.2 and Equation 2.3. That is, $\pi \cdot \boldsymbol{S}$ permutes the rows of $\boldsymbol{S}$ according to $\pi \in \mathbb{S}_{|\boldsymbol{S}|}$. We may also write $\underline{\mathbf{x}} = \boldsymbol{S} \in \mathscr{S}_{d_s}$ to unify with graphs. Now we can define a permutation-invariant and permutation-equivariant set function.

Definition 11 *Given a set task with appropriate target space $\mathcal{Y}$ and feature dimension $d_s \geq 1$, let $f : \mathscr{S}_{d_s} \to \mathcal{Y}$ be a function over variable sized sets. f is permutation-invariant if for*

Table 3.1. Non Group-theoretic Summary. We show the encoding and describe the permutation action for sets and graphs.

Input Type	Encoding	Permutation
Set, size n	$\underline{\mathbf{x}} = \boldsymbol{S} \in \mathbb{R}^{n \times d_s}$	Permute the rows of $\boldsymbol{S}$
Graph, size n	$\underline{\mathbf{x}} = \mathcal{G} = (\mathbf{A}, \boldsymbol{F}) \in \mathscr{G}_{n,d_v,d_e}$	Permute first/second mode of $\mathbf{A}$ and rows of $\boldsymbol{F}$

any $n \in \mathbb{Z}_{\geq 1}$, it is $\mathbb{G}$-invariant to the action of group $\mathbb{G} = \mathbb{S}_n$ on $\mathscr{S}_{n,d_s} = \mathbb{R}^{n \times d_s}$ defined in Equation 2.3. Permutation equivariance is defined similarly.

Remark 5 *To simplify exposition, we have considered the input set $\boldsymbol{S}$ to be an encoding of the "raw" input data, but this need not be the case. We can also use learnable permutation-invariant layers on sets of hidden representations in a broader neural network architecture. One eminently important example is in Message Passing Graph Neural Networks (described below).*

3.1.4 Summary Without Group-Theoretic Terms

A summary of the encodings and permutations of sets/graphs is provided in Table 3.1. When discussing a framework in which the input can be a set or a graph (as in Janossy pooling), we will write $\underline{\mathbf{x}}$ to denote either.

Pairwise information in a graph is encoded in a 3-mode tensor $\mathbf{A}$. The first two modes of $\mathbf{A}$ correspond to vertices and the third to feature vectors for pairs thereof. Vertex attributes are encoded in a matrix $\boldsymbol{F}$ whose rows are vertex features. Thus, reordering a graph permutes the first two modes of $\mathbf{A}$ and the rows of $\boldsymbol{F}$.

Set inputs with vector elements are encoded as a matrix, with elements stored as rows. Permuting a set thus permutes the rows of this matrix. To ease notation, more abstract objects like images are assumed to be vectorized, although in practice this need not be the case.

3.2 Existing Work

Having introduced graphs and sets as well as the importance of permutation invariance, we turn to existing work. In particular, we expose key contributions of JP, which (1) provides a unified theoretical framework for models of graphs and sets, (2) generalizes existing methods, (3) improves the expressiveness of state-of-the-art graph models, and (4) improves finite-sample learning for sets.

3.2.1 Methods Applicable to Both Graphs and Sets

Two methods, ordering data and weight sharing, are broadly applicable to graphs and sets.

Ordering the Graph or Set

One approach is to define a *canonical ordering* and reorder graphs or sets accordingly before passing to permutation-sensitive layers. This renders the original arbitrary ordering irrelevant. Typically, the canonical reordering function does not contain learnable parameters, an example being sorting by some characteristic. Niepert, Ahmed, and Kutzkov [75] propose to sort vertices in a graph by some user-specified vertex statistic like betweenness centrality before passing to a specially designed architecture. Montavon, Hansen, Fazli, *et al.* [195] sort molecules according to the norms of the vertex features.

A canonical ordering is only useful if it is relevant to the task at hand; unwittingly sorting by some characteristic orthogonal to the task could result in failure. Concretely, Moore and Neville [55] observed that ordering vertices by Personalized Page Rank [196], [197] ultimately resulted in lower accuracy than simply randomizing the ordering.

Rather than predefining a canonical order, one can try to *learn* an ordering. PointCNN [76] uses an MLP (Equation 2.1) to learn a special vector that is used to orient a point cloud "like an image" which is then passed to convolutional layers. Mena, Belanger, Linderman, *et al.* [198] propose to incorporate permutations of the input into the architecture and optimize them during end-to-end training by way of a differentiable tempered Sinkhorn operator. Building on this, Zhang, Hare, and Prügel-Bennett [199] propose a Permutation-

Optimization module that better incorporates pairwise interactions in a set. Note that permutation matrices are discrete but the elements of the learned matrices are parameters in $\mathbb{R}$, so these methods each propose additional steps to recover an actual permutation of the input. For instance, a "proper" permutation matrix is recovered when the temperature of the Sinkhorn approaches zero. Another approach is used in NeuralSort [5] where changes in the training scheme allow a simple row-wise argmax to provably recover a permutation matrix. These were inspired by Vinyals, Bengio, and Kudlur [77], who use ancestral sampling to sample permutations that optimize performance in an otherwise permutation-sensitive architecture. Overall, we are unaware of a careful theoretical or empirical exploration of heuristics such as those used in [77]. Consequently, it is difficult to apply in new tasks. Finally, in another direction, Linderman, Mena, Cooper, *et al.* [200] use variational inference and the reparameterization trick [201], [202] to estimate optimal permutations of the input data. The authors apply the method to a Bayesian regression analysis where the ordering of the predictors $\{x_1, \ldots, x_p\}$ is unknown due to measurement ambiguity. Interestingly, many of these methods invoke continuous relaxations to enable SGD optimization over a discrete object, the permutation matrix, which we return to in Chapter 5.

Optimizing permutations is rooted in a different philosophy than training (approximately) invariant models. Fundamentally, as described in Chapter 2, modelers choose invariant architectures or training schemes to achieve better generalization (or even extrapolation) capabilities, often believing that the ordering is simply nuisance variation. In contrast, learning a permutation assumes that some permutations are more advantageous than others given some architecture and task. We discuss this more in Chapter 5.

Weight Sharing and Beyond

Broadly, the weight sharing approaches discussed in Chapter 2 can be used for sets and graphs (e.g., [18], [19], [154], [157]). However, an affine layer (see Equation 2.1) is permutation-equivariant if and only if its weight matrix takes the form $\theta \boldsymbol{I} + \vartheta \mathbf{1}\mathbf{1}^{\mathrm{T}}$ for parameters $\theta, \vartheta \in \mathbb{R}$, where $\boldsymbol{I}$ is the identity matrix and $\mathbf{1}\mathbf{1}^{\mathrm{T}}$ is the matrix where every element is 1 [19], [26]. In particular, there are only two free parameters. This is related to our discussion in Section 2.2.3; the sheer size of the symmetric group restricts the degrees of freedom.

Recently, Haan, Cohen, and Welling [203] argue to rethink permutation equivariance in favor of a new paradigm that is more flexible. While exciting, it is beyond the scope of this dissertation.

3.2.2 Existing Graph-Focused Approaches

We focus our discussion on neural network models for graph data. A brief discussion of graph kernels is provided in Section 2.1.2.

Message Passing Graph Neural Networks

Graph Neural Networks with message passing form one of the most popular classes of methods [40], [44], [45], [56], [63]–[67]. These were developed in different communities and with different approaches: *convolutional signal-processing* approaches, *probabilistic models*, and the *Weisfeiler-Lehman test.* In [132], William Hamilton writes: "the convergence of these three disparate areas into a single algorithm framework is remarkable". Despite their disparate theoretical underpinnings, the practical methods are so similar that it is possible to implement methods from all three perspectives into one common software framework, an example being PyTorch Geometric [204]–[206]. While each perspective has a rich literature and a clear impact on researchers' motivations (cf. [56], [66]), the best exposition of Janossy pooling comes from treating them as one.

Recall that Example 2 and Remark 4 suggest that permutation-invariant graph functions can be built by constructing permutation-invariant functions of neighborhoods. This is the approach of Message Passing Graph Neural Networks (MPGNNs). In particular, suppose $\mathcal{G} = (\mathbf{A}, \boldsymbol{F})$ is an input data graph (in some implicit ordering). Let $\mathcal{V} = \{1, 2, \ldots, |\mathcal{G}|\}$ denote its vertices, $\mathcal{E}$ its edges, and $\mathcal{N}(v) = \{u \in \mathcal{V} : (u, v) \in \mathcal{E}\}$ denote the neighborhood of $v \in \mathcal{V}$. Putting aside edge attributes momentarily, a general MPGNN initializes $\boldsymbol{H}^{(0)} = \boldsymbol{F}$ and computes the following recursion for all vertices $v \in \mathcal{V}$, for layers $l = 1, \ldots, L$:

$$\boldsymbol{H}_{v,:}^{(l)} = g\left(\boldsymbol{H}_{v,:}^{(l-1)}, f\left(\left(\boldsymbol{H}_{u,:}^{(l-1)}\right)_{u \in \mathcal{N}(v)}; \boldsymbol{\Theta}_f^{(l)}\right); \boldsymbol{\Theta}_g^{(l)}\right), \tag{3.1}$$

where $g(\cdot, \cdot; \boldsymbol{\Theta}_g^{(l)})$ is a nonlinear function with learnable parameters $\boldsymbol{\Theta}_g^{(l)}$ that are typically distinct at each layer l, $f(\cdot; \boldsymbol{\Theta}_f^{(l)})$ is a permutation-invariant learnable function of variable-length inputs called the *aggregator*, and $\left(\boldsymbol{H}_{u,:}^{(l-1)}\right)_{u \in \mathcal{N}(v)}$ denotes the sequence of (hidden or visible) features in the neighborhood of v at layer l. Notice that the features of a vertex $v \in \mathcal{V}$ constitute one argument of g and the aggregated features as a second, thus treating "self" features specially. Ultimately, this yields hidden representation matrices $\boldsymbol{H}^{(l)} \in \mathbb{R}^{|\mathcal{G}| \times d_l}$, $l = 1, \ldots, L$ (whose column dimensions are hyperparameters). To map these to a single prediction of the entire graph, we may apply a permutation-invariant *readout* over the rows of $\boldsymbol{H}^{(L)}$, obtaining a prediction $r(\boldsymbol{H}^{(L)}) = r\Big((\boldsymbol{H}_{1,:}^{(L)}, \boldsymbol{H}_{2,:}^{(L)}, \ldots, \boldsymbol{H}_{|\mathcal{G}|,:}^{(L)})\Big)$. A common example is summation, $r(\boldsymbol{H}^{(L)}) = \mathbf{1}^{\mathrm{T}} \boldsymbol{H}^{(L)} = \sum_{v=1}^{|\mathcal{G}|} \boldsymbol{H}_{v,:}^{(L)}$. It is also common to apply the readout to all intermediate layers, $r(\boldsymbol{F} \bowtie \boldsymbol{H}^{(1)} \bowtie \cdots \bowtie \boldsymbol{H}^{(L)})$ where $\bowtie \colon \mathbb{R}^{m \times p} \times \mathbb{R}^{m \times q} \to \mathbb{R}^{m \times (p+q)}$ denotes concatenating the rows of two matrices [207]. We ensure that the entire GNN is (sub)differentiable in its parameters so that the function can be learned "end-to-end". Finally, note that in practice MPGNNs can be computed with matrix multiplications: Example 2 shows a simple case.

A popular MPGNN is the Graph Isomorphism Network (GIN) [56] which defines $g(\boldsymbol{a}, \boldsymbol{b}; \boldsymbol{\Theta}_g^{(l)}) = \mathrm{MLP}(\boldsymbol{a} + \boldsymbol{b}; \boldsymbol{\Theta}_g^{(l)})$ and f as a summation without parameters. In particular, one variation of GIN defines the recursion

$$\boldsymbol{H}_{v,:}^{(l)} = \mathrm{MLP}\left(\boldsymbol{H}_{v,:}^{(l-1)} + \sum_{u \in \mathcal{N}(v)} \boldsymbol{H}_{u,:}^{(l-1)}; \boldsymbol{\Theta}_g^{(l)}\right) \tag{3.2}$$

and uses summation for the readout $r(\boldsymbol{F} \bowtie \boldsymbol{H}^{(1)} \bowtie \ldots \bowtie \boldsymbol{H}^{(L)})$. There is also a variation of GIN that premultiplies $\boldsymbol{H}_{v,:}^{(l-1)}$ with $(1 + \vartheta^{(l)})$, for learnable scalars $\vartheta^{(l)}$. Our experiments use the version of GIN with these ϑ terms.

To model edge features, a general approach is to introduce the edge feature vectors $(\mathbf{A}_{v,u,2:})_{u \in \mathcal{N}(v)}$ as input to the aggregator f. Examples that appear in the literature are described in [8]. Additional variations use attention mechanisms [64] and incorporate neighborhood size in the function f [63]. In general, any learnable permutation-invariant function of variable-length inputs can be used as the aggregator. Janossy pooling serves as a general-purpose approach to learning the aggregator, and our experiments show that it can improve

upon existing methods. Interestingly, along a different direction, Chen, Bian, and Sun [194] demonstrate that learning a set function on the vertex features, ignoring graph structure, can sometimes achieve decent performance. The authors argue for using this approach as a simple baseline.

Limitations and the WL Test. Message Passing Graph Neural Networks (MPGNNs) do not enjoy universal approximation guarantees. There are pairs of nonisomorphic graphs that MPGNNs cannot distinguish; that is, there exist graphs $\mathcal{G}_1, \mathcal{G}_2$ such that $\pi \cdot \mathcal{G}_1 \neq \mathcal{G}_2$ for all permutations $\pi \in \mathbb{S}_{|\mathcal{G}_1|}$ yet $\text{MPGNN}(\mathcal{G}_1; \boldsymbol{\Theta}) = \text{MPGNN}(\mathcal{G}_2; \boldsymbol{\Theta})$ for any Message Passing GNN and any parameters $\boldsymbol{\Theta}$. Consequently, any true permutation-invariant function of graph data ϱ such that $\varrho(\mathcal{G}_1) \neq \varrho(\mathcal{G}_2)$ cannot be approximated by any MPGNN. This was demonstrated by the parallel works of Xu, Hu, Leskovec, *et al.* [56] and Morris, Ritzert, Fey, *et al.* [208].

Specifically, MPGNNs cannot distinguish pairs of nonisomorphic featureless graphs that the Weisfeiler-Lehman (WL) graph isomorphism testing heuristic (also known as color refinement) [209]–[211] cannot distinguish.[6] Intuitively, this follows from the fact that the WL test follows the same recursion as Equation 3.1, replacing a learnable neural network layer with an injective "hash" function. This "hash" is designed specifically to distinguish graphs, so learnable layers in an MPGNN cannot perform better in this regard. Hence, we say "MPGNNs are no more *powerful* than the WL test". Indeed, sometimes GNNs in Equation 3.1 are called WLGNNs to highlight the parallel, but this has become increasingly confusing as new researchers suggest new methods related to the k-dimensional higher-order versions of the Weisfeiler-Lehman test (WL[k]) [6], [208], [212].

Several papers propose solutions to this limitation. For example, models that use high-order tensors can achieve universality but are computationally intractable [149], [187], [213]. In contrast, the Janossy pooling framework provides simple and effective strategies to make MPGNNs more powerful.

[6]There are "higher-order" WL algorithms and different variants. When we write WL, we refer to color refinement, which is described in the text.

Other Noteworthy Graph-Focused Literature

There is an abundance of work aiming to improve different aspects of GNNs; comprehensive reviews are provided in [214], [215]. Many are not our focus, but JP can incorporate several of the ideas directly. Rather than summarize the entire literature, we briefly summarize a few interesting directions. Several papers explain that using too many recursion layers ("deep" MPGNNs) can actually result in *loss* of information [216]–[218] and some solutions are described in [207], [219]. Indeed, GRAPH-BERT, a new neural network on graphs, cites this depth problem as a motivation [220]. We had made some theoretical progress towards a description of this phenomenon but others published before we did. Hyperbolic message passing was designed to better capture graph structure [221], [222]. Some works aim to make GNNs faster, perhaps a leading factor in the widespread adoption of new methods [223], [224]. These include Hierarchical Aggregation, which reduces redundant calculations in GNN architectures; clever neighborhood sampling that reduces the computational burden of aggregation while keeping variance low [45], [225]; and using a linear model rather than a neural network as a simple and fast baseline that can achieve adequate performance on some tasks [226].

Convolution, a fundamental operation in invariances, has been generalized to the graph domain [63], [133], [227]–[229]. These treat the vertex feature matrix $\boldsymbol{F}$ as a graph signal and proceed by the graph Fourier transform. Exact convolutions are computationally intensive and require approximations that effectively become recursive functions of the adjacency matrix (i.e., MPGNNs) [63], [66]. Moreover, the proper convolutions are not suited for graph-wide classification/regression tasks (only vertex-level tasks on one graph).

3.2.3 Existing Set-Focused Literature

Permutation-invariant models for set data (encoded as sequences) take a few broad approaches.

Pooling in Latent Space

We have seen that pooling is a common operation in invariant models. *Pooling in latent space* has been one of the most popular neural network models for sets, including applications in controlling drone swarms [35], reinforcement learning with natural language [230], detecting cancerous regions in histopathology images [34], and point clouds [25], [26]. This approach became popular when Zaheer, Kottur, Ravanbakhsh, *et al.* [26] and Qi, Su, Mo, *et al.* [25] described the method, demonstrated its empirical success, and characterized its universality properties. The idea is to (1) map every vector in a set to a hidden space with neural network layers, (2) apply a simple symmetric function such as summation, and (3) pass the resulting vector through another neural network to obtain a prediction. Formally, as in Definition 11, let $d_s \in \mathbb{Z}_{\geq 1}$ be a fixed vector dimension,[7] $\mathcal{Y}$ denote the target space, and $\mathscr{S}_{d_s} = \bigcup_{n \in \mathbb{Z}_{\geq 1}} \mathscr{S}_{n,d_s} = \bigcup_{n \in \mathbb{Z}_{\geq 1}} \mathbb{R}^{n \times d_s}$ be the space of input sets – encoded as sequences of vectors (i.e., matrices). Recall from Remark 5 that $\boldsymbol{S}$ can also denote a set of *hidden* outputs in some broader neural network architecture. Then, the DeepSets [26] model $f : \mathscr{S}_{d_s} \to \mathcal{Y}$ is defined by

$$f(\boldsymbol{S}) = r\Big(\Big(\sum_{i=1}^{|\boldsymbol{S}|} g\big(\boldsymbol{S}_{i,:}; \boldsymbol{\Theta}_g\big)\Big); \boldsymbol{\Theta}_r\Big), \quad \forall \boldsymbol{S} \in \mathscr{S}_{d_s}, \tag{3.3}$$

where $g(\cdot; \boldsymbol{\Theta}_g) : \mathbb{R}^{d_s} \to \mathbb{R}^{d_h}$ is a learnable mapping with parameters $\boldsymbol{\Theta}_g$, d_h is a hyperparameter, and $r(\cdot; \boldsymbol{\Theta}_r) : \mathbb{R}^{d_h} \to \mathcal{Y}$ is another learnable mapping to the target space. The function g can be an MLP like Equation 2.1 or other appropriate neural network layers. We ensure that f is (sub)differentiable in its parameters so that all the parameters can be trained end-to-end. PointNet of Qi, Su, Mo, *et al.* [25] replaces summation with max and Ilse, Tomczak, and Welling [34] add attention mechanisms to affect a weighted summation. Learnable Aggregation Functions [231] is a family of aggregation functions (including sum and max) and that provides the flexibility to *learn* the most appropriate aggregation function. Indeed, numerous differentiable/learnable pooling functions on the latent space (i.e., after g) have been proposed [29], [34], [232], [233], but summation and maximum have been the most common in practice.

[7] We reiterate that DeepSets can handle arbitrary vector dimensions, and fixing the dimension is analogous to assuming a fixed number p of covariates in linear regression. The number of vectors in the input set may vary.

Latent space pooling methods like DeepSets and PointNet are universal approximators of set functions under fairly general assumptions [25], [26], [57]–[59]. However, we know that mathematical universality guarantees do not always ensure good real-world performance (cf. Chapter 2). The forms of g and r in Equation 3.3 must often be very complex or discontinuous, the number of parameters must grow with the cardinality of the set, and these approaches struggle to handle tasks where inputs are variable-length [62], [231]. Thus, it is difficult to find the correct parameters of the model through standard optimization routines.

Moreover, latent space pooling reduces all latent representations $g(\boldsymbol{S}_{i,:})$ into a single vector via a simple function such as summation. This can present challenges in capturing the relationships between elements in the input set. Consider learning the range of a set of scalars, $\text{range}(\boldsymbol{S}) = \max_i(\boldsymbol{S}_{i,:}) - \min_i(\boldsymbol{S}_{i,:}), \forall \boldsymbol{S} \in \mathscr{S}_1$. This requires identifying the largest and smallest elements – by definition relative concepts – and taking their difference. These relationships among set elements may be lost in summation, so the outer function r must somehow capture that lost signal. This can only be achieved by making the output dimension of g very large or making r complex (not smooth), as supported by the theory in [62]. Our Janossy pooling paper was an early paper to discuss this issue and propose solutions, but it was also discussed in narrower contexts at a similar time, (e.g., [60], [234]).

Higher-Order Methods

Whereas the first layer g in latent space pooling takes each individual element of $\boldsymbol{S}$ as input, other approaches strive to capture relationships among elements in the set more directly. PointNet++ [235] partitions point clouds into spatial regions and models distances to regional centroids. This method also uses geodesics to capture intrinsic structure within a point cloud on a Non-Euclidean space. SetTransformer [60] (inspired by [236]) uses several permutation-equivariant Multi-head Attention Blocks that each capture pairwise interactions in variable-size inputs. Additional recent approaches leverage self-attention layers to model point clouds [237], [238]. Relation Network [51] is similar to latent space pooling (Equation 3.3) but learns representations for *pairs* of inputs, replacing the input to r with

$\sum_{i,j} g(\boldsymbol{S}_{i,:}, \boldsymbol{S}_{j,:}; \boldsymbol{\Theta}_g)$. Finally, Meng, Yang, Ribeiro, *et al.* [27] consider sets of sets – where the "elements" of a set are themselves sets.

Higher-order methods take an important step towards capturing relationships among elements in the input sequence, which Janossy pooling generalizes. Many of the approaches discussed above, including Relation Network [51] and SetTransformer [60] are special cases thereof.

Other Noteworthy Set-Focused Methods

Some authors use Recurrent Neural Networks [93], [94] to process variable-size sets. However, RNNs are permutation-sensitive models. To overcome this, one approach that has been used is to randomly permute the input sequence at each iteration [45], [55]. This approach enjoyed empirical success but lacked the theoretical justification we later provided in Janossy pooling [54], as discussed in Chapter 4.

4. JANOSSY POOLING FOR INVARIANT MODELS

To preserve permutation invariance in modeling sets and graphs, the literature advances generic methods applicable to both types of data (weight sharing and ordering) as well as those specific to each. As discussed, the general methods are limited in their capacity and modeling flexibility. Yet, we also saw that the set- and graph-specific state-of-the-art models such as DeepSets and Message Passing Graph Neural Networks face challenges as well. DeepSets models rely on neural networks with a very large number of parameters to compensate for processing set elements independently, and MPGNNs are unable to express all permutation-invariant graph functions. The Janossy pooling (JP) framework simultaneously unifies models for sets and graphs and can overcome limitations of the state-of-the-art methods specific to each. Moreover, this unified framework provides a new perspective of and theoretical justification for existing approaches.

A key observation is that JP can be incorporated into graph models in two distinct ways. First, as a model for variable-length sets, it can be used as a *neighborhood aggregator* in MPGNNs (Equation 3.1). Second, we can apply JP to a graph directly to learn a permutation-invariant representation thereof (see Definition 9). In Murphy, Srinivasan, Rao, *et al.* [81], we referred to JP for learning graph representations as relational pooling (RP).

The original motivation for JP came from graph and set data. Thus, most of our discussion here, which appeared in our publications [54], [81], focus on such data. However, we will see that JP can be generalized to other invariances and note avenues for future exploration. First, JP involves a sum over the symmetric group of permutations, but other transformation sets could be considered. Second, while we mostly focus on the form of permutation-*invariant* models for graphs defined in Chapter 3 – *joint* invariance – we will also briefly mention possible extensions to *separate* invariance as well as $\mathbb{G}$ *equivariance*. Indeed, since publishing, other scholars have advanced new methods based on our ideas and refined the theory. We will mention these works throughout the discussion.

We begin by introducing the JP framework in the context of invariant neural networks and explain its ability to improve on existing methods (Section 4.1). Next, since it is computationally expensive to compute full JP, we advance three approximation strategies (Section 4.1.2).

Then, we discuss a parallel motivation for JP with the lens of probability (Section 4.2) and extensions (Section 4.3). We empirically investigate JP and demonstrate its benefits on several graph and set tasks in Section 4.4 before concluding with a summary of the impact of JP on the literature in Section 4.5.

4.1 Janossy Pooling

It can be challenging to directly specify a permutation-invariant model of variable-length inputs. By summing over latent representations of the input, pooling in latent space (e.g., PointNet, DeepSets [25], [26]) can satisfy both requirements, but presents challenges in estimation. As discussed in Section 3.2.3, a key problem is that processing inputs individually loses much of the signal, so highly complex and high-dimensional functions are required to compensate. MPGNNs invoke set functions on graph neighborhoods but are no more powerful than the WL test and fail to distinguish pairs of nonisomorphic graphs (Section 3.2.2). We also saw that the size of the symmetric group implies that weight sharing techniques can only use two free parameters per layer (Section 3.2.1).

Motivated by this, we propose JP, which allows modelers to invoke permutation-*sensitive* functions in approximating a permutation-invariant function. For instance, JP allows one to use Recurrent Neural Networks [93], [94], which are successful in modeling the complexities of language, to capture relationships in a variable-length sequence. In principle, without the burden of capturing relationships, downstream layers need not be as complex as those in DeepSets. More concretely, we will see that permutation-sensitive graph models give rise to a simple and effective strategy for making MPGNNs "more powerful" than the WL test.

To obtain an invariant model from permutation-sensitive functions, we can sum over permutations of the input. This is expensive, so we must use approximations. This is in line with existing training schemes that promote "approximate" invariances (see Chapter 2) and is justified by our empirical results. This idea is not totally new. Summing over transformations has been used in the symmetrized kernel [145] and is related to the Reynolds operator in algebra [239]. We set ourselves apart by proposing a variety JP models that provably fill gaps in existing literature on graphs and sets, proposing approximation schemes and developing

supporting theory thereof, and posing JP as a unifying framework under which many existing methods can be understood.

Formally, let $\mathcal{X}$ denote the space of inputs to the JP layer, such as variable-size sets or graphs, depending on the application. $\mathcal{X}$ can either denote "raw" data such as point clouds or molecules, or a set of hidden representations from preceding neural network layers. Specifically, assume either $\mathcal{X} = \mathscr{G}_{d_v,d_e}$ or $\mathcal{X} = \mathscr{S}_{d_s}$ for some graph feature dimensions $d_v, d_e \in \mathbb{Z}_{\geq 1}$ or set feature dimension $d_s \in \mathbb{Z}_{\geq 1}$ (see Definitions 9 and 11). Since we have defined the actions of the symmetric groups on $\mathscr{G}_{d_v,d_e}$ and $\mathscr{S}_{d_s}$, we can simply write $\pi \cdot \underline{\mathbf{x}}$ to refer to permutations of a set or graph, where $\underline{\mathbf{x}} \in \mathcal{X}$, $\pi \in \mathbb{S}_{|\underline{\mathbf{x}}|}$ is a permutation, and $|\underline{\mathbf{x}}|$ the size of $\underline{\mathbf{x}}$. A review of these operations is provided in Table 3.1. Letting $\vec{f}$ denote any (possibly permutation-sensitive) function on $\mathcal{X}$, with parameter matrix $\mathbf{\Theta}$, JP defines a permutation-invariant function by

$$\overline{\vec{f}}(\underline{\mathbf{x}}; \mathbf{\Theta}) = \frac{1}{|\underline{\mathbf{x}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{x}}|}} \vec{f}(\pi \cdot \underline{\mathbf{x}}; \mathbf{\Theta}). \tag{4.1}$$

$\overline{\vec{f}}$ is called the *Janossy function* associated with $\vec{f}$, and is permutation-invariant. Specifically, $\vec{f}$ is any composition of neural network layers, (sub)differentiable in $\mathbf{\Theta}$. $\overline{\vec{f}}$ can be a layer in a broader neural network architecture trained end-to-end, in which case we call it a Janossy *layer*, or the entire model itself. Notice that, while we have defined JP as an average, it can equivalently be defined with summation, letting $\vec{f}(\underline{\mathbf{x}}; \mathbf{\Theta}) = |\underline{\mathbf{x}}|! \vec{g}(\underline{\mathbf{x}}; \mathbf{\Theta})$ for some function $\vec{g}$ on $\mathcal{X}$. Although the merits of sum pooling versus average pooling are still debated [56], [240], writing as an average exposes the equivalent form

$$\overline{\vec{f}}(\underline{\mathbf{x}}; \mathbf{\Theta}) = \mathbb{E}_{\pi \sim \mathrm{Unif}(\mathbb{S}_{|\underline{\mathbf{x}}|})}[\vec{f}(\pi \cdot \underline{\mathbf{x}}; \mathbf{\Theta})], \tag{4.2}$$

where $\mathbb{E}$ denotes an expectation and Unif denotes the discrete uniform distribution. Thus, training with JP can be regarded as approximating a population mean, the "true" Janossy function. Figure 4.1 illustrates a JP layer for the special case of a sequence input.

We reiterate that JP will often be just one component in an architecture. In many such cases, the output of $\overline{\vec{f}}$ will not be the dimension of the target (especially for graphs), and downstream layers can be used to map to the prediction. Additionally, as we have seen,

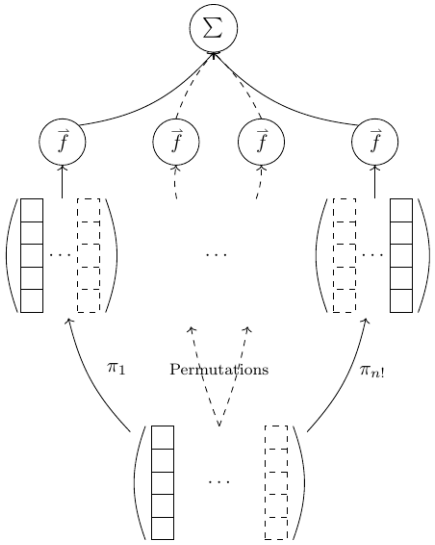


Figure 4.1. Janossy Pooling Layer for a Sequence Input. A variable-length input set of vectors – encoded as a sequence – is shown at the bottom. The stacks of solid and dashed blocks indicate two different vectors in the sequence. Full JP (which we approximate) permutes the length-n input sequence with all $n!$ permutations $\pi_1, \pi_2, \ldots, \pi_{n!} \in \mathbb{S}_n$. Two such permutations, π_1 and $\pi_{n!}$, are shown; notice that the solid and dashed vectors are swapped on the right. We compute a permutation-sensitive $\vec{f}$ on all permutations of the sequence and sum (or average) the outputs. This can be applied to latent representations or the raw input.

adding more downstream layers can facilitate learning a more expressive function, especially when capacity is lost in *approximating* $\overline{\overline{f}}$. Thus, letting $y(\cdot; \boldsymbol{\Theta}_f, \boldsymbol{\Theta}_r) : \mathcal{X} \to \mathcal{Y}$ denote the full model, we may write

$$y(\underline{\mathbf{x}}; \boldsymbol{\Theta}_f, \boldsymbol{\Theta}_r) = r\left(\overline{\overline{f}}(\underline{\mathbf{x}}; \boldsymbol{\Theta}_f); \boldsymbol{\Theta}_r\right) \tag{4.3}$$

to emphasize the downstream *readout* layer $r(\cdot; \boldsymbol{\Theta}_r)$. Note that the sequence or graph structure in $\underline{\mathbf{x}}$ will be "reduced" by the operation $\overline{\overline{f}}$, so the input to r will be a vector, not a sequence or graph. Hence it can simply be one or more affine layers (Equation 2.1). We will call $r \circ \overline{\overline{f}}$ a Janossy *model*. Observe that $\overline{\overline{f}}$ itself can be a full model by letting r be the identity map.

More generally, we can define JP for other invariances by replacing $\mathbb{S}_{|\underline{\mathbf{x}}|}$ with another group $\mathbb{G}$, defining the action of $\mathbb{G}$ on $\mathcal{X}$ (that is, defining $\pi \cdot \underline{\mathbf{x}}$), and specifying an appropriate $\vec{f}$. Examples of transformations of images include the set of 90 degree rotations, channel

permutations (RGB), and vertical flips [179]. In fact, many JP approaches need not rely on a proper group structure. Rotation by angles in $[-\pi/2, \pi/2]$ is not closed under composition, and therefore does not form a group, but we could still contemplate a JP approach by averaging over these rotations.

Later, Lyle, Wilk, Kwiatkowska, *et al.* [9] studied Equation 4.2 from a more theoretical standpoint. They extend the analysis to arbitrary compact groups and the Haar measure. The main similarity with our work is that both unify and characterize a broad range of existing methods. The main difference is that our work presents a methodological approach with concrete instantiations that improve existing models of sets and graphs. Interestingly, though, their work provides further justification and understanding of JP, which we will highlight in Section 4.1.2.

Before discussing approximation strategies, we study the expressive power of Janossy pooling and choices for $\vec{f}$.

4.1.1 Expressive Power and Choices for $\vec{f}$

In this section, we establish that JP is a most-powerful framework for learning on sets and graphs. Then we highlight some specific definitions of $\vec{f}$ that realize improvements over existing methods.

We desire a framework for constructing permutation-invariant models $\overline{\overline{f}}$ (or $r \circ \overline{\overline{f}}$ to emphasize the downstream layer) that can approximate any (well-behaved) permutation-invariant function arbitrarily well. A necessary condition is:[1] for all $\underline{\mathbf{x}}, \underline{\mathbf{x}}' \in \mathcal{X}$, if $\underline{\mathbf{x}} \neq \pi \cdot \underline{\mathbf{x}}'$ for any $\pi \in \mathbb{S}_{|\underline{\mathbf{x}}|}$, then $\overline{\overline{f}}(\underline{\mathbf{x}}) \neq \overline{\overline{f}}(\underline{\mathbf{x}}')$. The following theorem characterizes the expressive power of JP, that is, the permutation-invariant functions that JP layers can approximate arbitrarily well. Note that the result would be trivial for arbitrary $\vec{f}$ and r, but we are restricting them to be expressible by neural network models.

Theorem 1 ***(JP is most expressive)*** *Let $\mathcal{Y}$ denote a space of targets in some task. (1) For set tasks, let $\varrho : \mathcal{S}_{d_s} \to \mathcal{Y}$ denote any "true" permutation-invariant function of variable-length inputs with feature dimension $d_s \in \mathbb{Z}_{\geq 1}$. Then, there exist functions $\vec{f}$ and*

[1] Recall that $\underline{\mathbf{x}}$ is either encoded as a matrix or a pair of tensors, so we can write standard equality $=$, as opposed to $\simeq$. Even if graphs are isomorphic, they may not be equal once encoded as tensors.

r, composed of learnable neural network layers, such that $r \circ \overline{\overrightarrow{f}}$ approximates ϱ arbitrarily well for some choice of parameters, where $\overline{\overrightarrow{f}}$ is the Janossy function associated with $\overrightarrow{f}$. (2) Assume graphs and their features come from a finite subset $\mathscr{H}_{d_v,d_e} \subset \mathscr{G}_{d_v,d_e}$. That is, for all $\mathcal{G} = (\mathbf{A}, \boldsymbol{F}) \in \mathscr{H}_{d_v,d_e}$, $\mathbf{A}$ and $\boldsymbol{F}$ take elements from a finite set (i.e., features are discrete) and there exists some $M \in \mathbb{Z}_{\geq 1}$ such that $|\mathcal{G}| \leq M$. Then, for any "true" permutation-invariant function $\varrho : \mathscr{H}_{d_v,d_e} \to \mathcal{Y}$, there is a choice of neural network layers $\overrightarrow{f}$, r and parameters such that $r \circ \overline{\overrightarrow{f}}$ approximates ϱ arbitrarily well.

The proof uses the following lemma.

Lemma 1 *Any function that can be expressed using the DeepSets model (Equation 3.3) can be expressed using a JP model of the form written in Equation 4.3, for an appropriate choice of $\overrightarrow{f}$ and r.*

We defer the proof of this lemma to Section 4.1.2, as it uses ideas from our k-ary approximations. For now, we proceed with the proof of Theorem 1.

Proof 1 (1) DeepSets is a universal approximator for permutation-invariant functions of variable-length input sets [26], [58], [59].[2] By Lemma 1, we can find a Janossy model that is equivalent to DeepSets to finish this proof. (2) For graphs, we first construct a one-hot encoding. Denote $N = |\mathscr{H}_{d_v,d_e}| < \infty$ and define any bijective mapping $\iota : \mathscr{H}_{d_v,d_e} \to \{1, 2, \ldots, |N|\}$. Then, define $\overrightarrow{g} : \mathscr{H}_{d_v,d_e} \to \{0,1\}^N$ by $\left(\overrightarrow{g}(\mathcal{G})\right)_j = \begin{cases} 1 & \iota(\mathcal{G}) = j \\ 0 & \text{otherwise} \end{cases}$, for $j = 1, \ldots, |N|$, $\mathcal{G} \in \mathscr{H}_{d_v,d_e}$ (recall that isomorphic graphs, encoded as tensors, are different in $\mathscr{H}_{d_v,d_e}$). Next, let $\overrightarrow{f}$ be an MLP that approximates $\overrightarrow{g}$ arbitrarily well for some choice of parameters, which is guaranteed to exist [24], [71], [124]–[126]. Drop the parameters in

[2]Technically, for universal guarantees, DeepSets should be redefined to average pooling, but this can also be captured by JP.

writing $\vec{f}$ to simplify notation, and let $\text{orbit}(\mathcal{G}) = \{\pi \cdot \mathcal{G} : \pi \in \mathbb{S}_{|\mathcal{G}|}\}$ for all $\mathcal{G} \in \mathscr{H}_{d_v,d_h}$. Let $\mathcal{G}' \in \text{orbit}(\mathcal{G})$ and notice $\text{orbit}(\mathcal{G}) = \text{orbit}(\mathcal{G}')$, so we can verify

$$\begin{aligned}\overline{\vec{f}}(\mathcal{G}) &= \frac{1}{|\mathcal{G}|!}\sum_{\pi\in\mathbb{S}_{|\mathcal{G}|}}\vec{f}(\pi\cdot\mathcal{G}) = \frac{1}{|\mathcal{G}|!}\sum_{G\in\text{orbit}(\mathcal{G})}\vec{f}(G)\\ &= \frac{1}{|\mathcal{G}'|!}\sum_{G\in\text{orbit}(\mathcal{G}')}\vec{f}(G) = \frac{1}{|\mathcal{G}'|!}\sum_{\pi\in\mathbb{S}_{|\mathcal{G}'|}}\vec{f}(\pi\cdot\mathcal{G}') = \overline{\vec{f}}(\mathcal{G}'). \qquad (4.4)\end{aligned}$$

Hence, the output of the JP layer is a "fingerprint", a uniquely identifying value, for the orbit. The readout function r will operate on these fingerprint vectors. The universal approximation theorems guarantee that there exists an MLP r, with some choice of parameters, that agrees with ϱ arbitrarily well on the fingerprints of the orbits in $\mathscr{H}_{d_v,d_h}$. Thus $r \circ \overline{\vec{f}}$ approximates ϱ arbitrarily well, completing the proof. ■

Theorem 1 implies that, if $\vec{f}$ and r are modeled as neural network layers, then it is possible to estimate a Janossy model that can approximate any permutation-invariant function of sequences or (finite feature) graphs arbitrarily well. As we have discussed, universal approximation results ensure that the model class is flexible enough. Hence, this theorem shows that JP is a good starting point for developing permutation-invariant models, including graph models that exceed the expressive capacity of MPGNNs (see Section 3.2.2). Regarding assumptions, it is generally true that graphs are bounded in size. The stronger assumption is that of discrete features. We will see that there are benefits to this framework even when this assumption is violated.

However, as we have discussed, the usefulness of universal approximation results is limited if it is too difficult to learn effective models in practice. We still need to propose successful $\vec{f}$ and approximations for JP since summing over all permutations is typically infeasible. We start with specific functions for $\vec{f}$ that are theoretically justified and effective in practice.

RPGNN $\vec{f}$

In this section, we consider a special $\vec{f}$ for graph models motivated by the observation that MPGNNs are no more powerful than the WL test (i.e., color refinement) in distinguishing

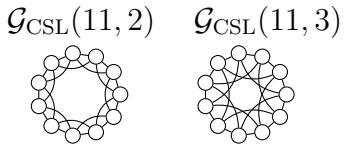


Figure 4.2. CSL Graphs. These nonisomorphic graphs cannot be distinguished by the WL test (color refinement).

pairs of graphs [208], [209], [211], [241]. We call it Relational Pooling Graph Neural Network as JP can be called relational pooling when applied to graphs [81].

Motivation. The first MPGNN recursion (Equation 3.1) does not yield hidden representations capable of distinguishing whether two vertices have the *same neighbor* or *distinct neighbors with the same features*. Successive layers of the MPGNN update vertex representations with the hope that vertices eventually get unique representations (up to isomorphisms), but this may not occur when graphs have complex cycles. This limits the ability of MPGNNs to learn an expressive graph representation.

In particular, recall that MPGNNs are no more powerful than the WL test (Section 3.2.2). Figure 4.2 shows an example of nonisomorphic graphs that cannot be distinguished by the WL test [78], [209], [211], [241]. In general, WL cannot distinguish degree-regular graphs with the same degree and number of vertices [56], [208], [241].[3] This pair of graphs may serve as an extreme example, but it is useful for illustrative purposes. We call these Circulant Skip Links (CSL) graphs, a type of circulant graph [242], and define them below.

Definition 12 ***(CSL graphs)*** *Let a and n denote co-prime natural numbers with $n-1 > a$. That is, the only common factor between a and n is 1. Then, $\mathcal{G}_{\text{CSL}}(n,a) = (\mathcal{V}, \mathcal{E})$ denotes an undirected 4-regular graph with vertices $\mathcal{V} = \{0, 1, \dots n-1\}$ whose edges form a cycle and have skip links. That is, for the cycle, $\{j, j+1\} \in \mathcal{E}$ for $j \in \{0, \dots, n-2\}$ and $\{n-1, 0\} \in \mathcal{E}$. For the skip links, recursively define the sequence $s_1 = 0$, $s_{i+1} = (s_i + a) \bmod n$ and let $\{s_i, s_{i+1}\} \in \mathcal{E}$ for every $i \in \mathbb{Z}_{\geq 1}$.*

Two CSL graphs $\mathcal{G}_{\text{CSL}}(n,a)$ and $\mathcal{G}_{\text{CSL}}(n',a')$ are not isomorphic unless $n = n'$ and $a \equiv \pm a' \mod n$ [78]. Thus, while the two graphs in Figure 4.2 are indistinguishable by the WL

[3] An undirected graph is d-regular if all vertices have d edges.

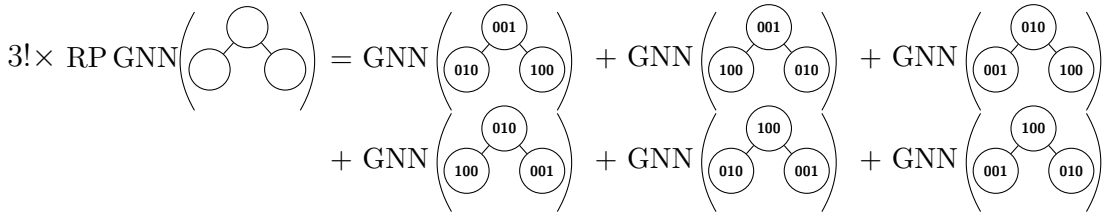


Figure 4.3. An illustration of full RPGNN. We append one-hot ID features to every vertex before passing to a GNN. Repeating this for all permutations of the graph amounts to all 3! ID assignments. (Note, we multiply by 3! on the left-hand side since JP is defined as an *average*). We see the power of RPGNN to break symmetries; the leaves have identical neighborhoods (the root) but their IDs make them distinguishable.

test, they are nonisomorphic. Thus, an *MPGNN would always fail on a task that requires classifying these graphs differently.*

This toy example was for illustration, but there are real-world parallels. Sato [243] shows two pairs of molecules that WL cannot distinguish: Decalin/Bicylopentyl and Decaprismane/Dodecahedrane.

Naturally, this challenge can be mitigated if vertex or edge features are available. Yet, the molecules discussed by Sato [243] do have features, the atom type. However, they are highly cyclical with only C and H atoms, so successive recursions did not result in differing hidden representations. A more robust solution is to *add unique vertex features* to break graph symmetries and yield distinct representations. Such intuition leads to RPGNNs.

Defining RPGNN. We will append unique one-hot encoding vectors of the vertices in a graph $\mathcal{G}$ – the standard basis vectors of $\mathbb{R}^{|\mathcal{G}|}$ – to the vertex features. In statistical terminology, we are defining a new categorical variable such that each vertex has its own level. After appending these features, we pass the augmented graph to an MPGNN. Note that these unique identifier features (IDs) will depend on the implicit ordering of the graph, e.g., the "first" vertex will have ID $(1, 0, \ldots, 0)$. Hence, appending IDs is a permutation-sensitive operation. To respect permutation invariance, we average the result over permutations, per the JP framework. Figure 4.3 shows an example.

Formally, let $\mathcal{G} = (\mathbf{A}, \boldsymbol{F}) \in \mathscr{G}_{d_v,d_e}$ be a graph with vertex feature matrix $\boldsymbol{F}$, or $\boldsymbol{F} = \mathbf{1}^{\mathrm{T}}$ if no features are available. To add unique one-hot IDs, we concatenate $\boldsymbol{F}$ with the identity matrix. Let $\mathrm{GNN}(\cdot;\boldsymbol{\Theta}) : \mathscr{G}_{d_v,d_e} \to \mathcal{Y}$ denote an MPGNN, $\bowtie$ denote row concatenation of two matrices as defined in Section 3.2.2 (i.e., leading to a "wide" matrix), and define $\vec{f}(\mathbf{A}, \boldsymbol{F}; \boldsymbol{\Theta}) = \mathrm{GNN}(\mathbf{A}, \boldsymbol{F} \bowtie \boldsymbol{I}_{|\mathcal{G}|}; \boldsymbol{\Theta})$. Note that $\vec{f}(\pi \cdot \mathbf{A}, \pi \cdot \boldsymbol{F}; \boldsymbol{\Theta}) = \mathrm{GNN}(\pi \cdot \mathbf{A}, (\pi \cdot \boldsymbol{F}) \bowtie \boldsymbol{I}_{|\mathcal{G}|}; \boldsymbol{\Theta})$. That is, $\boldsymbol{I}_{|\mathcal{G}|}$ is not permuted. Thus, RPGNN is defined by

$$\overline{\vec{f}}(\mathcal{G}; \boldsymbol{\Theta}) = \overline{\vec{f}}(\mathbf{A}, \boldsymbol{F}; \boldsymbol{\Theta}) = \frac{1}{|\mathcal{G}|!} \sum_{\pi \in \mathbb{S}_{|\mathcal{G}|}} \mathrm{GNN}\Big(\pi \cdot \mathbf{A}, (\pi \cdot \boldsymbol{F}) \bowtie \boldsymbol{I}_{|\mathcal{G}|}; \boldsymbol{\Theta}\Big). \tag{4.5}$$

RPGNN can also be defined as a model with readout, $r \circ \overline{\vec{f}}$ (see Equation 4.3). One implication is that the dimension of the augmented vertex features passed to the GNN depends on the graph size. In Remark 6 below, we discuss modeling variable-size graphs with RPGNN.

The following proposition is useful for understanding and implementing RPGNN. It shows that we only need to permute $\boldsymbol{I}_{|\mathcal{G}|}$, not $\mathbf{A}$ and $\boldsymbol{F}$. Further improvements to the computational aspects of RPGNN will be discussed after establishing its expressiveness in Theorem 2.

Proposition 1 *Equation 4.5 can be written equivalently as*

$$\overline{\vec{f}}(\mathcal{G}; \boldsymbol{\Theta}) = \overline{\vec{f}}(\mathbf{A}, \boldsymbol{F}; \boldsymbol{\Theta}) = \frac{1}{|\mathcal{G}|!} \sum_{\pi \in \mathbb{S}_{|\mathcal{G}|}} \mathrm{GNN}\Big(\mathbf{A}, \boldsymbol{F} \bowtie (\pi \cdot \boldsymbol{I}_{|\mathcal{G}|}); \boldsymbol{\Theta}\Big).$$

Proof 2 We use the facts that GNNs yield permutation-invariant graph representations and that all groups (including the symmetric group of permutations) have a unique inverse for each element [146]. For any $\pi \in \mathbb{S}_{|\mathcal{G}|}$, we can write

$$\begin{aligned}
\vec{f}(\pi \cdot \mathbf{A}, \pi \cdot \boldsymbol{F}; \boldsymbol{\Theta}) &\overset{(1)}{=} \mathrm{GNN}\Big(\pi \cdot \mathbf{A}, (\pi \cdot \boldsymbol{F}) \bowtie \boldsymbol{I}_{|\mathcal{G}|}; \boldsymbol{\Theta}\Big) \\
&\overset{(2)}{=} \mathrm{GNN}\Big(\pi \cdot \mathbf{A}, (\tilde{\boldsymbol{F}}); \boldsymbol{\Theta}\Big) \\
&\overset{(3)}{=} \mathrm{GNN}\Big(\pi^{-1} \cdot (\pi \cdot \mathbf{A}), \pi^{-1} \cdot (\tilde{\boldsymbol{F}}); \boldsymbol{\Theta}\Big) \\
&\overset{(4)}{=} \mathrm{GNN}\Big(\mathbf{A}, \pi^{-1} \cdot \Big((\pi \cdot \boldsymbol{F}) \bowtie \boldsymbol{I}_{|\mathcal{G}|}\Big); \boldsymbol{\Theta}\Big) \\
&\overset{(5)}{=} \mathrm{GNN}\Big(\mathbf{A}, \boldsymbol{F} \bowtie (\pi^{-1} \cdot \boldsymbol{I}_{|\mathcal{G}|}); \boldsymbol{\Theta}\Big),
\end{aligned}$$

where (1) follows by definition; (2) follows by putting $\tilde{\boldsymbol{F}} = (\pi \cdot \boldsymbol{F}) \bowtie \boldsymbol{I}_{|\mathcal{G}|}$; (3) follows since GNN is permutation-invariant and that all permutations have an inverse; (4) follows from the definition of group actions and expanding $\tilde{\boldsymbol{F}}$; (5) from the fact that if $\boldsymbol{A} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times p}$, then $\pi \cdot (\boldsymbol{A} \bowtie \boldsymbol{B}) = (\pi \cdot \boldsymbol{A}) \bowtie (\pi \cdot \boldsymbol{B})$. In detail, point (5) simply says that we can permute the rows of two matrices before binding them, or bind them and then permute, without changing the result. Therefore, we can write Equation 4.5 as

$$\begin{aligned}
\overline{\overline{f}}(\mathcal{G}; \boldsymbol{\Theta}) = \overline{\overline{f}}(\mathbf{A}, \boldsymbol{F}; \boldsymbol{\Theta}) &= \frac{1}{|\mathcal{G}|!} \sum_{\pi \in \mathbb{S}_{|\mathcal{G}|}} \vec{f}(\pi \cdot \mathbf{A}, \pi \cdot \boldsymbol{F}; \boldsymbol{\Theta}) \\
&= \frac{1}{|\mathcal{G}|!} \sum_{\pi \in \mathbb{S}_{|\mathcal{G}|}} \mathrm{GNN}\Big(\mathbf{A}, \boldsymbol{F} \bowtie (\pi^{-1} \cdot \boldsymbol{I}_{|\mathcal{G}|}); \boldsymbol{\Theta}\Big) \\
&\overset{(6)}{=} \frac{1}{|\mathcal{G}|!} \sum_{\pi \in \mathbb{S}_{|\mathcal{G}|}} \mathrm{GNN}\Big(\mathbf{A}, \boldsymbol{F} \bowtie (\pi \cdot \boldsymbol{I}_{|\mathcal{G}|}); \boldsymbol{\Theta}\Big),
\end{aligned}$$

where (6) follows because each $\pi \in \mathbb{S}_{|\mathcal{G}|}$ has a unique inverse $\pi^{-1} \in \mathbb{S}_{|\mathcal{G}|}$ and summation is commutative. ■

The following theorem establishes that RPGNNs are more powerful than MPGNNs. To avoid confusion, note that this theorem characterizes the set of functions that can be expressed by MPGNN versus RPGNN, not approximation. Thus, we drop the parameter matrices $\boldsymbol{\Theta}$.

Theorem 2 ***(Expressiveness of RPGNN)*** *Let $\mathcal{F}$ be the set of permutation-invariant graph functions and $\mathcal{F}_{\mathrm{GNN}}$ denote the set of functions that can be expressed (exactly) by MPGNNs for some architecture and choice of parameters. Finally, define $\mathcal{F}_{\mathrm{RPGNN}}$ similarly with the RPGNN models defined in Equation 4.5. Then, $\mathcal{F}_{\mathrm{GNN}} \subset \mathcal{F}_{\mathrm{RPGNN}} \subseteq \mathcal{F}$. In particular, RPGNNs represent a more expressive model class than MPGNNs.*

Proof 3 Since MPGNNs are permutation-invariant (i.e., isomorphism-invariant), $\mathcal{F}_{\mathrm{GNN}} \subseteq \mathcal{F}$. Similarly, $\mathcal{F}_{\mathrm{RPGNN}} \subseteq \mathcal{F}$ since RPGNNs are a special case of JP (see also Equation 4.4). To complete the proof, we need to show: (1) $\forall f_G \in \mathcal{F}_{\mathrm{GNN}}, \exists f_R \in \mathcal{F}_{\mathrm{RPGNN}}$ such that $f_G(\mathcal{G}) = f_R(\mathcal{G})$ for all $\mathcal{G} \in \mathscr{G}_{d_v, d_e}$, (2) $\exists f_R \in \mathcal{F}_{\mathrm{RPGNN}}$ such that $\forall f_G \in \mathcal{F}_{\mathrm{GNN}}$, $f_G(\mathcal{G}) \neq f_R(\mathcal{G})$ for some graph $\mathcal{G}$. Note that, when we say an RPGNN is equal to an MPGNN, we do not necessarily mean they take the same functional form or parameters. Rather, they express the same mapping from input space to output space.

(1) Let $f_G \in \mathcal{F}_{\text{GNN}}$ be arbitrary. By definition, f_G computes the recursions and readout function described in Equation 3.1 and the surrounding text in Section 3.2.2. In particular, let $\mathcal{G}$ be any graph with vertex set $\mathcal{V}$, neighborhoods denoted by $\mathcal{N}(v) \subseteq \mathcal{V}$ for $v \in \mathcal{V}$, and vertex feature matrix $\boldsymbol{F}$. We set $\boldsymbol{H}^{(0)} = \boldsymbol{F}$ and compute the recursions

$$\boldsymbol{H}_{v,:}^{(l)} = c^{(l)}\left(\boldsymbol{H}_{v,:}^{(l-1)}, a^{(l)}\left(\left(\boldsymbol{H}_{u,:}^{(l-1)}\right)_{u\in\mathcal{N}(v)}\right)\right) \quad \text{(recursions of } f_G\text{)},$$

for all $v \in \mathcal{V}$ and $l \in \{1, 2, \ldots, L\}$, where we have renamed the functions to $a^{(l)}$ and $c^{(l)}$ *(aggregate* and *combine)* to avoid confusion. By definition, an RPGNN defines a function $\vec{f}$ by similar recursions, with the first being

$$\boldsymbol{H}_{v,:}^{(1)} = c_R^{(1)}\left(\boldsymbol{H}_{v,:}^{(0)} \bowtie \boldsymbol{I}_{v,:}, a_R^{(1)}\left(\left(\boldsymbol{H}_{u,:}^{(0)} \bowtie \boldsymbol{I}_{u,:}\right)_{u\in\mathcal{N}(v)}\right)\right) \quad \text{(first recursion in RPGNN)}, \quad (4.6)$$

where $\boldsymbol{I}$ denotes the $|\mathcal{G}| \times |\mathcal{G}|$ identity matrix and $\boldsymbol{I}_{u,:}$ denotes row u (the one-hot IDs). The other recursions (layers $l \in \{2, \ldots, L\}$) are defined similarly, but do not append IDs $\boldsymbol{I}$. Like the GNN, this RPGNN $\vec{f}$ is followed by a permutation-invariant readout function.

Now, we can define $a_R^{(1)}$ and $c_R^{(1)}$ to ignore the dimensions in their input corresponding to the IDs. Thus, it is not difficult to specify aggregate and combine functions such that $a_R^{(l)}$ agrees with $a^{(l)}$ and $c_R^{(l)}$ agrees with $c^{(l)}$ for all l. That is, they return the same value when presented the same input graph. Similarly, we can set the readout functions of f_G and $\vec{f}$ to be the same. This implies that $\vec{f}$ is permutation-invariant, whence its associated Janossy function satisfies $\overline{\vec{f}} = \vec{f}$. Therefore, defining $f_R := \overline{\vec{f}} \in \mathcal{F}_{\text{RPGNN}}$, we have $f_G(\mathcal{G}) = \vec{f}(\mathcal{G}) = \overline{\vec{f}}(\mathcal{G}) = f_R(\mathcal{G})$, for all graphs $\mathcal{G}$. Thus $\mathcal{F}_{\text{GNN}}$ is a subset of $\mathcal{F}_{\text{RPGNN}}$.

(2) To be concrete, we consider $\mathcal{G}_1 := \mathcal{G}_{\text{CSL}}(11, 2)$ and $\mathcal{G}_2 := \mathcal{G}_{\text{CSL}}(11, 3)$ shown in Figure 4.2. Let $\mathcal{G}_{\text{CSL}}(11, 2) = (\boldsymbol{A}_1, \boldsymbol{1})$ and $\mathcal{G}_{\text{CSL}}(11, 3) = (\boldsymbol{A}_2, \boldsymbol{1})$ be their respective encodings into an adjacency matrix and constant features (no additional edge/vertex features are present). We have seen that the WL (color refinement) algorithm cannot distinguish these nonisomorphic graphs [56], [208], [241] and hence for all $f_G \in \mathcal{F}_{\text{GNN}}$, $f_G(\boldsymbol{A}_1, \boldsymbol{1}) = f_G(\boldsymbol{A}_2, \boldsymbol{1})$. Note, we do not need to show this for all permutations, since f_G is already permutation-

invariant. We will demonstrate that $\exists\ \overline{\overline{f}}_R \in \mathcal{F}_{\text{RPGNN}}$, an RPGNN function, such that $\overline{\overline{f}}_R(\boldsymbol{A}_1, \mathbf{1}) \neq \overline{\overline{f}}_R(\boldsymbol{A}_2, \mathbf{1})$.

To make the discussion more concrete, we define a specific GNN inspired by GIN [56]. Given a graph $(\boldsymbol{A}, \boldsymbol{F})$, we define

$$\text{GNN}(\boldsymbol{A}, \boldsymbol{F}) = r\Big((\boldsymbol{A} + \boldsymbol{I})\boldsymbol{F}\Big),$$

where $\boldsymbol{I}$ is the identity matrix of appropriate dimension and r is a permutation-invariant function (defined by neural network layers) over the rows of its input. In other words, $(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{F}$ computes a vertex representation for every vertex, and r is a permutation-invariant function defined over them. This is a slight modification of the function shown in Example 2 (see also Remark 4).

Next, recall that we can write permutations of a graph $(\boldsymbol{A}, \boldsymbol{F})$ as $(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}, \boldsymbol{P}\boldsymbol{F})$ where $\boldsymbol{P}$ is a permutation matrix. Hence, the corresponding RPGNN taking one of the CSL graphs with adjacency matrix $\boldsymbol{A}$ and constant vertex features $\mathbf{1}$ is computed by

$$\begin{aligned}\sum_{\boldsymbol{P}} \text{GNN}(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}, (\boldsymbol{P}\mathbf{1}) \bowtie \boldsymbol{I}) &= \sum_{\boldsymbol{P}} r\Big((\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}} + \boldsymbol{I})(\mathbf{1} \bowtie \boldsymbol{I})\Big) \\ &= \sum_{\boldsymbol{P}} r\Big(5\mathbf{1} \bowtie (\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}} + \boldsymbol{I})\Big),\end{aligned}$$

where the sum is over all permutation matrices $\boldsymbol{P}$ with the same number of rows as $\boldsymbol{A}$. The term $5\mathbf{1} = (5, \ldots, 5)^{\mathrm{T}}$ appears since all rows in $\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}} + \boldsymbol{I}$ have five entries equal to 1 and the rest 0. Denote $\boldsymbol{C}_{\boldsymbol{A},\boldsymbol{P}} := 5\mathbf{1} \bowtie (\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}} + \boldsymbol{I})$.

Next, observe that $\pi \cdot \mathcal{G}_1 \neq \sigma \cdot \mathcal{G}_2$ for all $\pi, \sigma \in \mathbb{S}_{|\mathcal{G}_1|}$, since the graphs are nonisomorphic. Equivalently, for all $|\mathcal{G}_1| \times |\mathcal{G}_1|$ permutation matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$, it holds that $\boldsymbol{P}\boldsymbol{A}_1\boldsymbol{P}^{\mathrm{T}} \neq \boldsymbol{Q}\boldsymbol{A}_2\boldsymbol{Q}^{\mathrm{T}}$. Roughly speaking, this implies that the 'sets' of vertex representations passed to r are different between the two graphs. More precisely, if we let r be an injective permutation-invariant neural network [26], [56],[4] then

$$\Big\{\!\Big\{r\Big(\boldsymbol{C}_{\boldsymbol{A}_1,\boldsymbol{P}}\Big)\Big\}\!\Big\}_{\boldsymbol{P}} \neq \Big\{\!\Big\{r\Big(\boldsymbol{C}_{\boldsymbol{A}_2,\boldsymbol{P}}\Big)\Big\}\!\Big\}_{\boldsymbol{P}},$$

[4]An injective function over sets is injective over canonical representations of every set (i.e., injective over the space mod orbits).

where double brackets denote a multiset, and they run over all permutation matrices. Therefore, we know [26], [56] there exist neural networks ρ and ϕ such that

$$\rho\left(\sum_{\boldsymbol{P}} \phi\Big(r\big(\boldsymbol{C}_{\boldsymbol{A}_1,\boldsymbol{P}}\big)\Big)\right) \neq \rho\left(\sum_{\boldsymbol{P}} \phi\Big(r\big(\boldsymbol{C}_{\boldsymbol{A}_2,\boldsymbol{P}}\big)\Big)\right).$$

Hence, if we define

$$\overline{\overline{f}}_R(\mathcal{G}) = \overline{\overline{f}}_R(\boldsymbol{A},\boldsymbol{F}) = \rho\left(\frac{1}{|\mathcal{G}|!}\sum_{\boldsymbol{P}} \phi\Big(\mathrm{GNN}(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}, (\boldsymbol{P}\boldsymbol{F}) \bowtie \boldsymbol{I})\Big)\right),$$

where we use the GNN defined above and an injective neural network r, then $\overline{\overline{f}}_R(\boldsymbol{A}_1,\boldsymbol{1}) \neq \overline{\overline{f}}_R(\boldsymbol{A}_2,\boldsymbol{1})$. The functions ϕ and r can fuse into one function, but we have written it in this way for clarity of exposition.

■

On Vertex Identifiers. We have already established that full JP (and thus RPGNN) is computationally expensive and requires approximations, but here we provide strategies specific to RPGNN that further reduce its computational burden. These can be used alongside the approximations. As written, we append $|\mathcal{G}|$-dimensional ID vectors to the feature matrix. Even for moderate-sized graphs (e.g., $|\mathcal{G}| \geq 100$), this substantially increases the cost of each operation and the number of parameters. It also presents an additional challenge in implementing RPGNN for variable-size graphs. We propose two strategies to limit the dimension of the unique IDs.

First, the increased expressiveness of RPGNNs results from the fact that unique vertex features result in unique vertex neighborhoods. However, we may not need to define a new unique ID for each vertex to achieve this. If the graph already has vertex features ($d_v > 1$), we can preprocess graphs, inspecting them to determine a smaller ID dimension that still affects unique neighborhoods. For example, we can use the atom type in molecules; in CH_2O_2, the augmented RPGNN vertex features $(C,0,1)$, $(H,0,1)$, $(H,1,0)$, $(O,1,0)$, and $(O,0,1)$ render every vertex unique. As an added benefit, this reduces the number of considered permutations from $5! = 120$ to $(1!)(2!)(2!) = 4$.

Second, we can reduce the computation and parameter count by limiting the number of new IDs to $m \in \mathbb{Z}_{\geq 1}$. In particular, for each graph $\mathcal{G}$, define the ID function $\mathrm{id}_m : \{1, \ldots, |\mathcal{G}|\} \to \{0,1\}^m$ by

$$(\mathrm{id}_m(v))_j = \begin{cases} 1, & (v-1 \mod m) = j-1 \\ 0, & \text{otherwise} \end{cases}, \quad j = 1, \ldots, m,\ v \in \{1, \ldots, |\mathcal{G}|\}.$$

We can then replace $\boldsymbol{F} \bowtie \boldsymbol{I}$ in Equation 4.5 with a new matrix $\boldsymbol{F}_{\mathrm{ID}}$ defined by $(\boldsymbol{F}_{\mathrm{ID}})_{v,:} = \boldsymbol{F}_{v,:} \bowtie \mathrm{id}_{\mathrm{m}}(v)$. Of course, this does limit the extent to which IDs can break symmetries. Thus, there is a tradeoff between computation and model flexibility, with $m = 1$ collapsing to the standard MPGNN on one end and $m = |\mathcal{G}|$ providing the most expressive yet most computationally intensive RPGNN on the other. Notice this strategy is applicable even when vertex features are not available.

Remark 6 ***(RPGNN with variable-size graphs)*** *With the* mod *strategy for defining RPGNN IDs, it is straightforward to handle variable-size graphs. In particular, the dimension of the augment vertex feature matrix does not vary from graph to graph.*

Finally, one may inquire about the choice of one-hot IDs rather than continuous IDs for breaking symmetries. A benefit of one-hot IDs is that they encode neighborhood structure with a bit vector. Moreover, discrete IDs do not have a natural order, a case for which one-hot vectors is a standard encoding choice (cf. [244]).

After we published [81], additional scholars proposed to use IDs to overcome limitations of MPGNNs. Vignac, Loukas, and Frossard [245] propagate one-hot encodings to learn the local context of vertices in a directly permutation-equivariant manner. Sato, Yamada, and Kashima [187] sample *random* IDs and pass to GNNs to break symmetries in cycle structures. They show that this method can return solutions of graph algorithm problems, such as the minimum dominating set, that are close to the true solution with high probability. They write that their "analysis provides another justification of [RPGNN] because the approximation ratios of RPGNNs" can be proved similarly when using the π-SGD approximation scheme discussed below. Abboud, Ceylan, Grohe, *et al.* [246] sample random IDs from Unif$[0,1]$ before passing to a GNN and show that this can approximate invariant graph functions

arbitrarily closely with large probability (i.e., is an (ϵ, δ) approximation) for the correct choice of hyperparameters. The authors completed this proof using the *logic-based* analysis framework proposed in [247]. In contrast to previous results on universality that leveraged computationally intractable constructions, this method is relatively cheap to compute. A similar idea was presented in [248]. Interestingly, these methods invoke randomness to extend universality results beyond the finite case we considered in the theorems above.

Padded MLPs

We have seen that MLPs are permutation-sensitive and powerful function approximators, making them a possible choice for $\vec{f}$. Since these models are defined for vector inputs, we must *vectorize* the input with $\text{vec} : \mathcal{X} \to \mathbb{R}^p$, $p \in \mathbb{Z}_{\geq 1}$. We define vec precisely for graphs and sequences in Section A.1.2 of the appendix. Another challenge is that affine layers do not naturally accept variable-length inputs. To solve this, we can *pad* the input. Letting $M \in \mathbb{Z}_{>1}$ be the largest input we expect to encounter in the data, zero padding is defined by

$$\text{paddedvec}(\underline{\mathbf{x}}; 0) := \Big(\text{vec}(\underline{\mathbf{x}}), \underbrace{0, \ldots, 0}_{M - |\underline{\mathbf{x}}|}\Big).$$

Thus we can define $\vec{f}(\underline{\mathbf{x}}; \boldsymbol{\Theta}) = \text{MLP}(\text{paddedvec}(\underline{\mathbf{x}}); \boldsymbol{\Theta})$. Since vec is defined (i.e., "overloaded") for both graphs and sequences, this describes the approach for both such inputs. However, recurrent and convolutional layers arguably provide a more principled and flexible approach that we will discuss next.

RNNs and CNNs

Any function (i.e., layer) that is (sub)differentiable in its parameters and accepts variable-size inputs can be considered for $\vec{f}$. For sequences, recurrent layers [93], [94] and one-dimensional convolutional layers [24] are two popular layers that meet these criteria. One-dimensional convolutions pass a filter along the "time" dimension of the sequence, similar to Figure 2.1, an operation that naturally accepts variable-length inputs. Recurrent layers are popular in language precisely because they can capture rich relationships among elements

(words) in the sequence (sentence), and their recurrent nature makes them permutation sensitive. Thus, using RNNs as $\vec{f}$ in JP should better capture relationships than a latent space pooling approach like DeepSets or PointNet. More recently, *Transformer*-inspired models [236], [249]–[251] have become a popular alternative to recurrent approaches. These mostly consist of permutation-equivariant attention layers but use a positional encoding that renders the model permutation-sensitive. We will characterize the attention layers in these models as a type of approximation to JP in Section 4.1.2.

For graphs $\mathcal{G} = (\mathbf{A}, \boldsymbol{F}) \in \mathscr{G}_{d_v,d_e}$, we can define CNN to be a 2D CNN if $d_e = 1$ (no additional edge features) or 3D CNN [252] if $d_e > 1$, with pooling and affine layers succeeding convolutions to output a vector. Then we can define

$$\vec{f}(\mathbf{A}, \boldsymbol{F}; \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2) = \text{CNN}(\mathbf{A}; \boldsymbol{\Theta}_1) \bowtie \text{MLP}(\text{paddedvec}(\boldsymbol{F}; 0); \boldsymbol{\Theta}_2), \tag{4.7}$$

where MLP outputs a vector and $\bowtie$ denotes concatenation. In essence, the CNN treats the adjacency tensor as if it were an image (see Figure 2.1). As we have discussed, convolutions can be designed for variable-size inputs [253], but the vertex feature matrix needs padding. Overall, this returns a graph-wide representation vector. This is just one possible architecture. We could define a more complex $\vec{f}$ that passes the graph representation to additional affine layers. Additionally, we can define a similar model that replaces CNN with an RNN that treats $\mathbf{A}_{v,:,:}$ – the edge features associated with vertex $v \in \mathcal{V}$ – as a sequence. We will investigate these architectures in our experiments.

4.1.2 Approximations

In most cases, computing a full JP layer will be impractical as it requires a factorial number of gradient computations at each training iteration (see Equations 2.2, 4.1). Thus, we propose three general strategies to tractably approximate full JP: π-SGD, k-ary dependencies, and (poly-) canonical orderings. We will also see that these methods can be used in tandem.

Tractability with π-SGD

Janossy pooling involves an expected value over all permutations in the symmetric group (Equation 4.2), so we can *estimate* JP by *sampling* permutations at each training iteration. This procedure, shown in Algorithm 1, reduces computation tremendously. The name π-SGD derives from the fact that we sample permutations π in the context of SGD training. Despite the name, the idea could be generalized to actions of another finite group on an input space.

We will formalize π-SGD, then justify it theoretically by showing that π-SGD: (1) minimizes an upper bound to the original objective, (2) can improve generalization properties compared to standard training of a permutation-sensitive function $\vec{f}$, and (3) converges to the optimal solution almost surely under similar conditions as typical SGD. Moreover, we discuss how π-SGD training of JP provides a theoretical explanation for the successes of existing methods.

Review of Optimization. Let us expand on the review of neural network optimization in Section 2.1.1. Denote a training set by $\mathcal{D}^{(\text{tr})} = \left\{(\underline{\mathbf{x}}^{(1)}, \boldsymbol{y}^{(1)}), (\underline{\mathbf{x}}^{(2)}, \boldsymbol{y}^{(2)}), \ldots, (\underline{\mathbf{x}}^{(N_{\text{tr}})}, \boldsymbol{y}^{(N_{\text{tr}})})\right\}$, where $N_{\text{tr}} \in \mathbb{Z}_{\geq 1}$ is the number of observations in training, $\underline{\mathbf{x}}^{(i)} \in \mathcal{X}$ are inputs (e.g., sequences or graphs), and $\boldsymbol{y}^{(i)} \in \mathcal{Y}$ are associated targets, for $i = 1, \ldots, N_{\text{tr}}$. We use superscripts $\cdot^{(i)}$ for indexing examples to avoid confusion with elements of a vector or matrix. To optimize with Stochastic Gradient Descent, we train with *mini-batches* of the training data. In particular, we sample subsets $\mathcal{B}^{(1)}, \ldots, \mathcal{B}^{(\lceil N_{\text{tr}}/b \rceil)}$ without replacement from $\mathcal{D}^{(\text{tr})}$ such that all batches (except possibly the last) have b input-target pairs,[5] then perform gradient updates on $\mathcal{B}^{(j)}$ according to Equation 2.2, $j = 1, \ldots, \lceil N_{\text{tr}}/b \rceil$. This is one *epoch,* and we repeat for multiple epochs. In this section, we write the loss $\mathcal{L}$ as a scalar-valued function of one prediction/target pair.

When optimizing neural networks, we also maintain a validation data set $\mathcal{D}^{(\text{vl})} \subset \mathcal{X} \times \mathcal{Y}$, often disjoint from $\mathcal{D}^{(\text{tr})}$, that is used to estimate generalization performance of the model. The ultimate goal is to attain good performance on a test dataset that becomes available after model fitting. Computations on $\mathcal{D}^{(\text{tr})}$, such as computing gradients and updating weights,

[5] In practice, some discard the last batch if it has fewer than b samples.

Algorithm 1 π-SGD Step

Input 1: Minibatch $\mathcal{B} = \left(\underline{\mathbf{x}}^{(i)}, \boldsymbol{y}^{(i)}\right)_{i=1}^{b}$
2: Step size $\eta > 0$
3: Functions $\vec{f}$, r with parameters $\boldsymbol{\Theta}_f$, $\boldsymbol{\Theta}_r$, respectively
4: Loss function $\mathcal{L}$
Learnable Parameters: $\boldsymbol{\Theta} \equiv \boldsymbol{\Theta}_f, \boldsymbol{\Theta}_r$

$\mathbf{Z} \leftarrow \mathbf{0}$ ▷ Initialize random gradient
for $i \leftarrow 1, \ldots, b$ **do** ▷ For loop (for illustration) to compute batch gradients
  Sample $\pi \sim \mathrm{Unif}(\mathbb{S}_{|\underline{\mathbf{x}}^{(i)}|})$ ▷ Sample a random permutation
  $\underline{\mathbf{x}}_\pi \leftarrow \pi \cdot \underline{\mathbf{x}}^{(i)}$ ▷ Permute set or graph input
  $\hat{\boldsymbol{y}} \leftarrow r\left(\vec{f}(\underline{\mathbf{x}}_\pi; \boldsymbol{\Theta}_f); \boldsymbol{\Theta}_r\right)$ ▷ Compute a prediction
  $\mathbf{Z} \leftarrow \mathbf{Z} + \frac{1}{b}\nabla_{\boldsymbol{\Theta}}\mathcal{L}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}})$ ▷ Differentiate loss, increment batch gradient
end for
$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \eta\mathbf{Z}$ ▷ Update parameters

return $\boldsymbol{\Theta}$

An optimization step according to π-SGD, as a subroutine in a broader architecture. This update follows SGD, but in practice other updates like Adam [91] can be used.

are called *training time computations,* whereas computations on $\mathcal{D}^{(\mathrm{vl})}$ or test data are called *inference time* computations. With π-SGD, we will see that there are differences between training and inference times with regards to the number of permutations sampled.

π-SGD. We begin by defining a single π-SGD step given an input mini-batch, which is also shown in Algorithm 1.

Definition 13 ***(π-SGD)*** *Let $\mathcal{B} \subseteq \mathcal{D}^{(tr)} \subset \mathcal{X} \times \mathcal{Y}$ be a mini-batch of size $b := |\mathcal{B}| \in \mathbb{Z}_{\geq 1}$. Without loss of generality, write $\mathcal{B}$ as a sequence $\mathcal{B} = \left(\underline{\boldsymbol{x}}^{(i)}, \boldsymbol{y}^{(i)}\right)_{i=1}^{b}$. Let $\eta > 0$ be some step size (learning rate), and r and $\vec{f}$ be the readout and permutation-sensitive functions of a JP model (Equation 4.3). Compose them to define the parameterized function $r\left(\vec{f}\left(\cdot; \boldsymbol{\Theta}_f\right); \boldsymbol{\Theta}_r\right) : \mathcal{X} \to \mathcal{Y}$ where $\boldsymbol{\Theta} \equiv (\boldsymbol{\Theta}_f, \boldsymbol{\Theta}_r)$ is either randomly initialized or from previous optimization steps. To compute a parameter update, sample independent permutations $\pi^{(i)} \sim \mathrm{Unif}\left(\mathbb{S}_{|\underline{\boldsymbol{x}}^{(i)}|}\right)$, for $i = 1, \ldots, b$, and compute the random gradient*

$$\mathbf{Z} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\boldsymbol{\Theta}}\mathcal{L}\left(\boldsymbol{y}^{(i)}, r\left(\vec{f}\left(\pi^{(i)} \cdot \underline{\boldsymbol{x}}; \boldsymbol{\Theta}_f\right); \boldsymbol{\Theta}_r\right)\right), \tag{4.8}$$

Algorithm 2 Full Training Loop with π-SGD

Input 1: Training set $\mathcal{D}^{(\text{tr})} = \left(\underline{\mathbf{x}}^{(i)}, \boldsymbol{y}^{(i)}\right)_{i=1}^{N_{\text{tr}}}$, batch size $1 \leq b \leq N_{\text{tr}}$
2: Learning rates following a schedule $\eta_1, \eta_2, \ldots$
3: Functions $\vec{f}$, r parameterized by $\boldsymbol{\Theta}_f$, $\boldsymbol{\Theta}_r$, respectively
4: Loss function $\mathcal{L}$
Learnable Parameters: $\boldsymbol{\Theta} \equiv \boldsymbol{\Theta}_f, \boldsymbol{\Theta}_r$

Initialize $\boldsymbol{\Theta}$ ▷ Initialize parameters of $\vec{f}$ and r
$t \leftarrow 1$
while Stopping criterion not met **do**
  Shuffle $\mathcal{D}^{(\text{tr})}$
  Partition $\mathcal{D}^{(\text{tr})}$ into mini-batches $\mathcal{B}_1, \ldots, \mathcal{B}_{\lceil N_{\text{tr}}/b \rceil}$
  for $i \leftarrow 1, \ldots, \lceil N_{\text{tr}}/b \rceil$ **do**
    $\boldsymbol{\Theta} \leftarrow \pi SGD(\mathcal{B}_i, \eta_t, \vec{f}, r, \mathcal{L}, \boldsymbol{\Theta})$ ▷ Update with Algorithm 1
    $t \leftarrow t + 1$
  end for
end while
Return $\boldsymbol{\Theta}$

A full optimization loop using the π-SGD "subroutine" of Algorithm 1. This closely follows Algorithm 8.1 of [24].

where $\mathcal{L}$ is the loss function. Then, we can compute the π-SGD update

$$\boldsymbol{\Theta}_{new} \leftarrow \boldsymbol{\Theta} - \eta \mathbf{Z}.$$

Note that the readout and transformation-sensitive functions, r and $\vec{f}$, are "fused" together into a single function when we train by π-SGD. This is a consequence of replacing the pooling operation – an average over multiple permutations – with just one permutation. One could generalize the definition of π-SGD training to include sampling more permutations, which we will discuss more below. Now, Definition 13 describes the update for just one batch; the full training loop is shown in Algorithm 2. One important aspect of the full training procedure is the *learning rate scheduler*. Intuitively, it is helpful to use a large learning rate initially, to explore the loss landscape efficiently, and a smaller one later on to settle into an optimum. An important type of scheduler follows the Robbins-Monro conditions [88], [254].

Definition 14 ***(Robbins-Monro scheduler)*** *Let $\eta(t)$ for $t = 1, 2, \ldots$ denote a sequence of learning rates. We will say the sequence follows a* Robbins-Monro *schedule, or the* Robbins-Monro conditions *if (1) $\sum_{t=1}^{\infty} \eta(t) = \infty$; (2) $\sum_{t=1}^{\infty} \eta(t)^2 < \infty$.*

Now we focus on the theoretical justifications for π-SGD.

Minimizes an Upper Bound. Proposition 2 below states the first "punchline", in simple terms, which we will then formalize and prove. Note that the assumption that r is an identity mapping is largely for the clarity of the mathematical exposition. Since r and $\vec{f}$ are "fused" when sampling one permutation in π-SGD training, it carries less significance in practice. Further, assuming a convex loss function is not unrealistic. Recall that the loss $\mathcal{L}$ refers to a mapping from targets and predictions to $\mathbb{R}_{\geq 0}$, but the *objective* is the full quantity we minimize during training. Although the *objective* function of neural network models is typically not convex, we only need $\mathcal{L}$ to be convex in its second argument, the prediction. That is, we only need $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to be convex in $\hat{\boldsymbol{y}}$, which is satisfied for many common loss functions.

Proposition 2 *Let $r \circ \overline{\overline{f}}$ denote a Janossy model. Assume $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ is a convex loss function and that r is an identity function. Then, π-SGD training minimizes an upper bound of the original training objective.*

To make this precise, we define the objective functions in question. When the readout function r is an identity map, the original optimization objective for a single input-target pair $(\underline{\mathbf{x}}, \boldsymbol{y}) \in \mathcal{X} \times \mathcal{Y}$ is

$$J(\underline{\mathbf{x}}, \boldsymbol{y}, \boldsymbol{\Theta}) = \mathcal{L}\Big(\boldsymbol{y}, \overline{\overline{f}}(\underline{\mathbf{x}}; \boldsymbol{\Theta})\Big) = \mathcal{L}\Big(\boldsymbol{y}, \mathbb{E}_{\pi \sim \mathrm{Unif}(\mathbb{S}_{|\underline{\mathbf{x}}|})}[\vec{f}(\pi \cdot \underline{\mathbf{x}}; \boldsymbol{\Theta})]\Big), \tag{4.9}$$

where we have used the form of JP shown in Equation 4.2. Training with π-SGD estimates an alternative objective function in which the expectation comes out of loss,

$$\tilde{J}(\underline{\mathbf{x}}, \boldsymbol{y}, \boldsymbol{\Theta}) = \mathbb{E}_{\pi \sim \mathrm{Unif}(\mathbb{S}_{|\underline{\mathbf{x}}|})}\Big[\mathcal{L}\Big(\boldsymbol{y}, \vec{f}(\pi \cdot \underline{\mathbf{x}}; \boldsymbol{\Theta})\Big)\Big]. \tag{4.10}$$

Intuitively, this follows since we sample permutations before computing the loss (see Algorithm 1). We can thus prove Proposition 2 with Jensen's inequality.

Proof 4 (Proposition 2) Assume $\mathcal{B} \subseteq \mathcal{D}^{(\text{tr})} \subset \mathcal{X} \times \mathcal{Y}$ is a batch of training data and consider the full objective. By convexity of $\mathcal{L}$, the fact that J and $\tilde{J}$ differ by swapping the order of expectation in Equations 4.9 and 4.10, and Jensen's inequality,

$$\sum_{(\underline{\mathbf{X}},\boldsymbol{y})\in\mathcal{B}} \mathcal{L}\Big(\boldsymbol{y}, \mathbb{E}_{\pi\sim\text{Unif}(\mathbb{S}_{|\underline{\mathbf{X}}|})}[\vec{f}(\pi\cdot\underline{\mathbf{X}};\boldsymbol{\Theta})]\Big) \leq \sum_{(\underline{\mathbf{X}},\boldsymbol{y})\in\mathcal{B}} \mathbb{E}_{\pi\sim\text{Unif}(\mathbb{S}_{|\underline{\mathbf{X}}|})}\Big[\mathcal{L}\Big(\boldsymbol{y}, \vec{f}(\pi\cdot\underline{\mathbf{X}};\boldsymbol{\Theta})\Big)\Big]. \quad (4.11)$$

That is, π-SGD training minimizes an upper bound of the original training objective function. ■

Observe that the inequality in Equation 4.11 becomes an equality if $\vec{f}$ is a permutation-invariant function. That is, in this case, the π-SGD objective reaches its lower bound. Consequently, minimizing the objective with π-SGD can be understood as regularizing $\vec{f}$ towards permutation invariance. Thus, we would expect it to result in a model that has better generalization performance when the true task is invariant, even if it does not train a strictly invariant model. We demonstrate this experimentally. Hence, our approach joins the large literature of neural network training schemes that *promote*, rather than *enforce*, an approximate invariance and achieve empirical success.

Before proceeding, consider the case that r is not the identity function, but rather another sequence of neural network layers. Then the original objective becomes

$$J(\underline{\mathbf{x}},\boldsymbol{y},\boldsymbol{\Theta}) = \mathcal{L}\Big(\boldsymbol{y}, r\Big(\mathbb{E}_{\pi\sim\text{Unif}(\mathbb{S}_{|\underline{\mathbf{X}}|})}[\vec{f}(\pi\cdot\underline{\mathbf{x}};\boldsymbol{\Theta})]\Big)\Big),$$

and the order of the expectation and loss cannot be "flipped" in general. Thus, Jensen's inequality is not applicable. In practice, however, our scheme is to sample a single permutation, forward the permuted input, and compute a loss. We can thus fuse the functions to define $\vec{f}_{\text{new}} = r \circ \vec{f}$, and then the result still holds if we follow Algorithm 2 with $\vec{f}_{\text{new}}$.

Generalization Analysis and Data Augmentation. After the publication of JP [54], Lyle, Wilk, Kwiatkowska, *et al.* [9] conducted a theoretical analysis of the approach, calling it Feature Averaging. They derive PAC-Bayesian bounds [255] that provide an understanding of the generalization performance of invariant training. We can directly inherit their results, which we summarize here, writing JP instead of Feature Averaging. Note that these results

do not confine to the case that r is the identity mapping, even when more permutations are sampled.

Assume that the joint distribution $\mathbb{P}_{X,Y}$ on $\mathcal{X} \times \mathcal{Y}$ is truly $\mathbb{G}$-invariant, in the sense that $\mathbb{P}_{X,Y}(X, Y) = \mathbb{P}_{X,Y}(G \cdot X, Y)$ for all $G \in \mathbb{G}$. Then, standard (non-invariant) training would minimize the empirical risk (objective function) $\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\Big(f(X^{(i)}), Y^{(i)}\Big)$, where $(X^{(i)}, Y^{(i)})$ denotes a (random) training example. Loosely speaking, this reduces the risk $R_{\mathcal{L}}(f) := \mathbb{E}_{X,Y\sim\mathbb{P}_{X,Y}}[\mathcal{L}(f(X), Y)]$, the quantity that measures generalization performance [85]. The authors take a Bayesian approach since PAC-Bayesian bounds have provided non-vacuous generalization bounds for neural networks, unlike other approaches [256]. In this context, the measure of generalization performance becomes $R_{\mathcal{L}}(\mathbb{Q}) = \mathbb{E}_{f\sim\mathbb{Q}}[R_{\mathcal{L}}(f)]$, where $\mathbb{Q}$ is a posterior distribution obtained after estimating a function f from a class $\mathcal{F}$ on which we have placed some prior $\mathbb{Q}_0$. In short, PAC-Bayesian bounds find an upper-bound for $R_{\mathcal{L}}(\mathbb{Q})$ in terms of the empirical risk, sample size, and the KL divergence $\mathrm{KL}(\mathbb{Q}||\mathbb{Q}_0)$ between the prior and posterior. This KL term can be seen as a measure of model complexity [255]. We prefer the risk upper bound for a model to be smaller, in which case the generalization performance may be superior.

In short, Theorem 7 in Lyle, Wilk, Kwiatkowska, *et al.* [9] shows that the generalization bound that comes from training any model with (full) Janossy pooling is tighter than that of standard empirical risk minimization. The intuition is that training towards invariance through averaging *compresses* the model, bringing down the $\mathrm{KL}(\mathbb{Q}||\mathbb{Q}_0)$ term and thus the generalization bound. The authors were even able to prove this for the case when the group $\mathbb{G}$ is large and sampling is required. In particular, they show that increasing the number of samples can lead to a reduction in the KL term (Proposition 8). This suggests that sampling more permutations in JP can be beneficial to generalization, even in the case when r is not the identity map. Finally, they show that this training scheme reduces the variance in gradient estimates.

However, we cannot unequivocally advise that increasing the number of permutations sampled at training time will improve performance. Given the complexity of stochastic optimization of neural networks, the interplay between theory and practical performance is still not completely understood [9]. A telling example is mini-batch size. Using larger mini-

batches reduces variance in the estimates of the gradient but it has been argued that *smaller* batches lead to faster convergence and better generalization performance [257]. Moreover, sampling more permutations at training time results in substantial computational overhead since derivatives must be computed for each. In our empirical work, we typically sample one permutation per epoch at training time.

It is also worth pointing out the connection between π-SGD and data augmentation. Note that data augmentation can refer to both *sampling* transformations of the training inputs and adding those transformations to the dataset. When a JP layer is applied directly to the input, as is the case for RPGNN, and we sample one training-time permutation, it is a form of data augmentation. When sampling multiple permutations in the context of JP, we average over the outputs before computing *one* loss, which is no longer data augmentation in the usual sense. Lyle, Wilk, Kwiatkowska, *et al.* [9] and Chen, Dobriban, and Lee [188] study the impact of data augmentation by studying the associated risk function, which is equivalent to Equation 4.10. Unfortunately, their results most meaningful only when full data augmentation can be computed – that is, all transformations are considered in the training data – so these analyses are generally not applicable to permutations. The results for Feature Averaging are more applicable to JP.

Finally, the authors conduct experiments to show that these two approaches can make the model more invariant during training. We show some experiments of our own in Section 4.4.

Convergence. We conclude this section by showing that training with π-SGD (Algorithms 1 and 2) converges under similar conditions as usual SGD. The following result extends the Stochastic Approximation Convergence theorem in [90]. The first condition states that the expected update is large enough in the direction of the true solution and the second indicates that the update noise is bounded [90].

Proposition 3 ***(π-SGD Convergence)*** *Consider π-SGD training (Algorithm 2). Let $\boldsymbol{\Theta}^*$ denote an optimum of the objective function for some JP model and denote $D(\boldsymbol{\Theta}) = \frac{1}{2}\|\boldsymbol{\Theta} - \boldsymbol{\Theta}^*\|^2$. Write the π-SGD random gradient (Equation 4.8) as $\mathbf{Z}(\boldsymbol{\Theta}, \boldsymbol{\pi}, \mathcal{B})$, to emphasize the parameters and random permutations, and denote the sequence of parameters by $\boldsymbol{\Theta}(t)$. Then, $\boldsymbol{\Theta}(t)$ converges to $\boldsymbol{\Theta}^*$ with probability 1 if (1) $-\nabla D(\boldsymbol{\Theta})^T \mathbb{E}[\mathbf{Z}(\boldsymbol{\Theta}, \boldsymbol{\pi}, \mathcal{B})] \geq K_1 D(\boldsymbol{\Theta})$ for some constant K_1, (2) $\mathbb{E}_t\left[\|\mathbf{Z}(\boldsymbol{\Theta}, \boldsymbol{\pi}, \mathcal{B})\|^2\right] \leq K_2(1 + D(\boldsymbol{\Theta}))$ for some constant K_2 and*

the expectation is over all data prior to step t, and (3) the learning rates $\eta(t)$ follow the Robbins-Monro conditions in Definition 14.

Proof 5 π-SGD training is a stochastic approximation algorithm of the form of Equation (3) in [90]. Thus, the result holds directly from the proof therein and is a consequence of the supermartingale convergence theorem [258]. ■

It is reassuring that π-SGD training has similar convergence properties to SGD, a widely used method. In practice, other variants such as Adam [91] can achieve better performance than SGD, and we may also train with Adam-style updates rather than SGD updates. The key step, sampling permutations to estimate a random gradient, is unchanged. Another important observation is that this convergence guarantee does not shed light on the *variance* in π-SGD training (cf. [9]). One simple strategy for reducing the variance in our prediction at inference time is to sample more permutations.

Inference Time. Recall that JP can be written as an expected value over permutations and π-SGD effectively trains by Monte Carlo estimates at each gradient step. At inference time, we can sample more permutations and average over the results to obtain a lower-variance estimate of $\mathbb{E}_{\pi\sim\text{Unif}(\mathbb{S}_{|\underline{\mathbf{X}}|})}[\vec{f}(\pi\cdot\underline{\mathbf{X}};\boldsymbol{\Theta})]$. That is, we sample permutations $\pi_1,\ldots,\pi_m$ uniformly at random from all permutations and compute

$$\hat{\vec{f}}(\underline{\mathbf{X}};\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\vec{f}(\pi_i\cdot\underline{\mathbf{X}};\boldsymbol{\Theta}). \tag{4.12}$$

At inference time, we do not need to compute gradients, so the process is less time and memory intensive.

π-SGD Theoretically Justifies Existing Methods. Previous works found that passing randomly-permuted sequences to an RNN could achieve surprisingly strong performance as a neighborhood aggregator in graph tasks [45], [55]. JP offers a theoretical explanation; this approach estimates a JP layer with π-SGD. Our experiments show that, using this insight, we can straightforwardly improve the performance of GraphSAGE [45].

Remark 7 *Since this analysis did not make use of the special structure of the permutation group, π-SGD is a very general approach. We could extend it to approximating JP over the*

actions of another finite group $\mathbb{G}$ on the input space, sampling from $\mathbb{G}$ instead of $\mathbb{S}_n$. This was indeed considered in later work [9].

Tractability with k-ary Dependencies

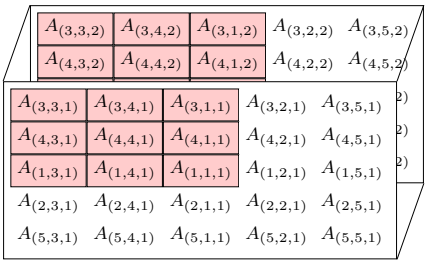
Next, we provide a different strategy that trades off model expressiveness and computational complexity in Janossy pooling. To simplify the sum over permutations in Equation 4.1, we impose the constraint that $\vec{f}$ depends only on the first k components of its input. This results in redundant permutations that can be ignored. Thus, if it is appropriate to assume that only k-ary dependencies in the input are relevant to the task at hand, we can use this strategy to substantially reduce computational cost. For example, to compute variance, only pairwise ($k = 2$) relationships are needed. We will show that DeepSets [26], which uses sum pooling, corresponds to a $k = 1$ approximation. It is not clear whether the k-ary approach generalizes beyond set and graph inputs, so we will focus on these for this section. First we formalize the operation of selecting the first k elements.

Definition 15 *Fix $k \in \mathbb{Z}_{\geq 1}$. For variable-length sets, we take the input space to be $\mathcal{X} = \mathscr{S}_{d_s} = \bigcup_{n \in \mathbb{Z}_{\geq 1}} \mathscr{S}_{n,d_s}$, for some feature dimension d_s. Recall that length-n sets are encoded as $n \times d_s$ matrices, $\mathscr{S}_{n,d_s} = \mathbb{R}^{n \times d_s}$. Define the function $\downarrow_k \colon \mathscr{S}_{d_s} \to \mathscr{S}_{k,d_s}$ by*

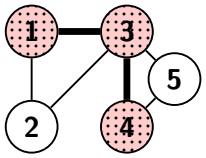
$$\downarrow_k(\boldsymbol{S}) = \begin{cases} \boldsymbol{S}_{1:k,:}, & |\boldsymbol{S}| \geq k \\ \boldsymbol{S} \between \mathbf{0}_{k-|\boldsymbol{S}|}, & |\boldsymbol{S}| < k \end{cases} \quad \forall \boldsymbol{S} \in \mathscr{S}_{d_s},$$

where we have used colon subsetting notation (Definition 7), $\between$ denotes column concatenation,[6] $|\boldsymbol{S}|$ denotes the length, and $\mathbf{0}_{k-|\boldsymbol{S}|}$ is a $(k - |\boldsymbol{S}|) \times d_s$ matrix of zeros. This zero-pads sequences of length less than k. For graphs, $\mathcal{X} = \mathscr{G}_{d_v,d_e} = \bigcup_{n \in \mathbb{Z}_{\geq 1}} \mathscr{G}_{n,d_v,d_e}$ and for a graph $\mathcal{G} \in \mathscr{G}_{d_v,d_e}$, $\downarrow_k(\mathcal{G}) = \downarrow_k(\mathbf{A}, \boldsymbol{F}) = (\mathbf{A}_{1:k,1:k,:}, \boldsymbol{F}_{1:k,:})$ if $|\mathcal{G}| \geq k$. If $|\mathcal{G}| < k$ we zero-pad the remaining elements of the adjacency tensor and vertex feature matrix.

[6] If A is $m \times p$ and B is $n \times p$, then $A \between B$ is $(m + n) \times p$.



Permuted adjacency tensor $\pi \cdot \mathbf{A}$ where π is the permutation such that $\pi \cdot (1,2,3,4,5)^{\mathrm{T}} = (3,4,1,2,5)^{\mathrm{T}}$. The result of $\downarrow_3(\pi \cdot \mathbf{A})$ is the $3 \times 3 \times 2$ shaded subtensor.



An illustration of $\downarrow_3(\pi \cdot \mathbf{A})$ using the permutation defined in the left panel. We select a k-sized induced subgraph with the vertices $\{3,4,1\}$, indicated by shaded vertices and thick edges.

Figure 4.4. Example of k-ary JP, or in particular the operation $\downarrow_3$ on a 5-vertex graph. The adjacency tensor $\mathbf{A}$ is $5 \times 5 \times 2$, since we assume $d_e = 2$ for illustration. We show a permutation of that tensor, $\pi \cdot \mathbf{A}$, and show $\downarrow_3 (\pi \cdot \mathbf{A})$. In k-ary approximations, we must compute $\downarrow_k(\pi \cdot \mathbf{A})$ for every non-redundant permutation (see text). For 3-ary JP, we iterate over all $5!/2!$ such permutations. For π-SGD with k-ary, we only need to sample one subgraph each iteration. Note that we would also need to compute $\downarrow_3(\boldsymbol{F})$ since graphs have both vertex and edge features, $\mathcal{G} = (\mathbf{A}, \boldsymbol{F})$, which is not shown.

For example, if $\underline{\mathbf{x}} = (1,2,3)^{\mathrm{T}} \in \mathbb{R}^{3\times 1}$ and $\pi \cdot \underline{\mathbf{x}} = (3,2,1)^{\mathrm{T}}$, then $\downarrow_2(\pi \cdot \underline{\mathbf{x}}) = (3,2)^{\mathrm{T}}$. For graphs, this amounts to selecting an induced subgraph of size k.[7] An example is shown in Figure 4.4. The graph $\mathcal{G}$ on the right is numbered by some "original" arbitrary vertex ordering. We assume there is one edge feature, in addition to connectivity information, so $d_e = 2$ (not shown). The tensor on the left shows $\pi \cdot \mathbf{A}$ for some permutation π. $\downarrow_3(\pi \cdot \mathbf{A})$ selects the shaded $3 \times 3 \times 2$ upper-left "corner" of this tensor, and corresponds to the shaded subgraph. Since our chosen permutation satisfies $\pi \cdot (1,2,3,4,5)^{\mathrm{T}} = (3,4,1,2,5)$, the subgraph contains vertices "1", "3", and "4" in the original ordering. Along with the vertices and edges shown, we also select the features of the edges between "1", "3", and "4", indicated by the farther slice of the adjacency tensor.

[7] A k-sized induced subgraph is formed by selecting k vertices and keeping all edges between them, see [259] for details.

To define k-ary JP, we suppose there is some (generally permutation-sensitive) function $\vec{f}$ over variable-size inputs and define a new (generally permutation-sensitive) function $\vec{f}_k$ by $\vec{f}_k(\underline{\mathbf{x}}; \boldsymbol{\Theta}) = \vec{f}(\downarrow_k(\underline{\mathbf{x}}); \boldsymbol{\Theta})$. In particular, k-ary JP is defined by

$$\overline{\vec{f}}_k(\underline{\mathbf{x}}; \boldsymbol{\Theta}) = \frac{1}{|\underline{\mathbf{x}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \vec{f}_k(\pi \cdot \underline{\mathbf{x}}; \boldsymbol{\Theta}) = \frac{1}{|\underline{\mathbf{x}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \vec{f}(\downarrow_k(\pi \cdot \underline{\mathbf{x}}); \boldsymbol{\Theta}). \tag{4.13}$$

The subscript k emphasizes that $\vec{f}_k$ only defines a computation over a size-k input and we will write $\overline{\vec{f}}_k$ as the Janossy function associated with $\vec{f}_k$. We may also say that $\overline{\vec{f}}_k$ is associated with $\vec{f}$, but it is implied that we take only the first k when computing $\vec{f}$. We call $\overline{\vec{f}}_k$ a k-ary approximation to Janossy pooling.[8] As written, *it may appear* that this does not reduce any computation, due to the sum over the full set of permutations. However, the implementation should ignore redundant permutations, and the next proposition shows that this results in a significant reduction in computation.

Proposition 4 *Computing the k-ary JP in Equation 4.13 only requires summing over $\frac{|\underline{\boldsymbol{x}}|!}{(|\underline{\boldsymbol{x}}|-k)!}$ distinct terms, thus saving computation when $k < |\underline{\boldsymbol{x}}|$.*

Proof 6 Fix $k \in \mathbb{Z}_{\geq 1}$ and let $\vec{f}$ denote any function. For an arbitrary $\underline{\mathbf{x}}$, let $\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}$ and let $Q_\pi = \{\tilde{\pi} \in \mathbb{S}_{|\underline{\mathbf{X}}|} : \downarrow_k(\pi \cdot \underline{\mathbf{x}}) = \downarrow_k(\tilde{\pi} \cdot \underline{\mathbf{x}})\}$. Notice that $|Q_\pi| = (|\underline{\mathbf{x}}| - k)!$ (the number of permutations of the remaining elements). For any π, we can always pick a representative element of Q_π, the equivalence class, and we let $\mathbb{S}_{|\underline{\mathbf{X}}|}[k]$ denote the set of all such representative permutations. Observe that $|\mathbb{S}_{|\underline{\mathbf{X}}|}[k]| = \frac{|\underline{\mathbf{X}}|!}{(\underline{\mathbf{X}}-k)!}$. Thus, k-ary JP becomes

$$\frac{1}{|\underline{\mathbf{x}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \vec{f}(\downarrow_k(\pi \cdot \underline{\mathbf{x}})) = \frac{(|\underline{\mathbf{x}}| - k)!}{|\underline{\mathbf{x}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}[k]} \vec{f}(\downarrow_k(\pi \cdot \underline{\mathbf{x}})),$$

a sum over only $\frac{|\underline{\mathbf{X}}|!}{(\underline{\mathbf{X}}-k)!}$ distinct elements. ■

For example, given prior belief that the task only requires modeling pairwise interactions, we can set $k = 2$ to reduce the number of terms to sum over from $|\underline{\mathbf{x}}|!$ to $|\underline{\mathbf{x}}|(|\underline{\mathbf{x}}| - 1)$. In fact, Santoro, Raposo, Barrett, *et al.* [51] take this approach in training a neural network to

[8] We use "approximation" loosely here, simply referring to a more tractable version of full JP.

model pairwise relationships among objects in a scene; their model can be seen as JP with 2-ary approximations. In practice, we can compute $\sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{x}}|}[k]} \vec{f}(\downarrow_k(\pi \cdot \underline{\mathbf{x}}))$ without explicitly permuting $\underline{\mathbf{x}}$. Instead, we can enumerate subsequences/subgraphs of size k and traverse the list to compute the summation. The practitioner can choose whichever approach is more efficient in any given problem.

Trade-offs of k-ary Approaches. Unsurprisingly, the computational savings of k-ary approximations come at the cost of reduced flexibility in the JP layer. Theorem 3 below shows that the class of functions that can be expressed by $(k+1)$-ary approximations is strictly larger than that of k-ary approximations. Note that this theorem does not involve universal approximation but simply the size of a function class. To emphasize this, we drop the parameters $\mathbf{\Theta}$. Additionally, this result refers purely to the pooling layer, not the full Janossy model $r \circ \overline{\overline{f}}_k$.

Theorem 3 *Suppose $\mathcal{X}$ is the space of graph or sequence inputs and $\mathcal{Y}$ is the target space. For any $k \in \mathbb{Z}_{\geq 1}$, define $\mathcal{F}_k = \{\sum_{\pi \in \mathbb{S}_{|\underline{\boldsymbol{x}}|}} \vec{f}(\downarrow_k(\pi \cdot \underline{\boldsymbol{x}}))\,|\,\vec{f} : \mathcal{X} \to \mathcal{Y} \text{ is arbitrary}\}$ to be the set of permutation-invariant functions defined on $\mathcal{X}$ that can be represented (exactly) by k-ary Janossy pooling. Then, $\mathcal{F}_k \subset \mathcal{F}_{k+1}$ if $|\mathcal{X}| > 1$. Furthermore, for graphs, there exist $\overline{\overline{f}}_{k+1} \in \mathcal{F}_{k+1} \setminus \mathcal{F}_k$ such that $\overline{\overline{f}}_{k+1}$ is non-constant on featureless graphs of the same size.*

Note that the assumption $|\mathcal{X}| > 1$ holds in all nontrivial tasks. For graphs, we are interested in functions that use the graph structure, not simply the vertex features, or equivalently those that are non-constant over featureless graphs of the same size. Otherwise, the problem devolves into a set problem. The idea of the proof for sequences is that no function of k-sized subsequences can capture the product of $k+1$ elements in general. For graphs, we use large CSL graphs in which we cannot determine from a k-sized induced subgraph whether it is taken from a CSL graph with skip links of length $k+1$ or k.

Proof 7 To avoid confusion, note that when we write $\overline{\overline{f}}_k$, for some fixed k, we are defining only one function, not a class of functions parameterized by k. The subscript indicates that it is a k-ary JP function.

(**Sets**) To consider sets, write $\mathcal{X} = \mathscr{S}_{d_s}$ for some d_s.

First, we need to show that $\overline{\overline{f}}_k \in \mathcal{F}_k \implies \overline{\overline{f}}_k \in \mathcal{F}_{k+1}$. Let $\overline{\overline{f}}_k$ be associated with $\vec{f}$, some function of variable-size inputs, such that $\overline{\overline{f}}_k$ is the result of pooling over $\vec{f}_k(\cdot) = \vec{f}(\downarrow_k(\cdot))$. Observe that $\downarrow_k \circ \downarrow_{k+1} = \downarrow_k$ by Definition 15. Now, define some other function $\vec{g}$ by $\vec{g}(\boldsymbol{S}) = \vec{f}_k(\boldsymbol{S})$, whence $\vec{g}_{k+1}(\boldsymbol{S}) = \vec{g}\Big(\downarrow_{k+1}(\boldsymbol{S})\Big) = \vec{f}_k(\downarrow_{k+1}(\boldsymbol{S})) = \vec{f}_k(\boldsymbol{S})$. Thus, for any input sequence $\boldsymbol{S}$,

$$\overline{\overline{f}}_k(\boldsymbol{S}) = \frac{1}{|\boldsymbol{S}|!}\sum_{\pi\in\mathbb{S}_{|\boldsymbol{S}|!}} \vec{f}_k(\pi\cdot\boldsymbol{S}) = \frac{1}{|\boldsymbol{S}|!}\sum_{\pi\in\mathbb{S}_{|\boldsymbol{S}|!}} \vec{g}_{k+1}(\pi\cdot\boldsymbol{S}) \in \mathcal{F}_{k+1}.$$

Second, we need to demonstrate $\exists\overline{\overline{f}}_{k+1} \in \mathcal{F}_{k+1}$ such that $\forall\overline{\overline{g}}_k \in \mathcal{F}_k$, $\overline{\overline{g}}_k \neq \overline{\overline{f}}_{k+1}$. For $\boldsymbol{S} \in \mathscr{S}_{d_s}$ (encoded as a matrix, or equivalently a sequence of vectors), we can define $\vec{f}(\boldsymbol{S}) = \prod_i \boldsymbol{S}_{i,1}$, the product of the first coordinate of the vectors in $\boldsymbol{S}$. Let $\overline{\overline{f}}_{k+1}$ be the $(k+1)$-ary JP function associated with $\vec{f}$. We now prove that, for all functions $\vec{g}$ on $\mathscr{S}_{d_s}$, we have $\overline{\overline{g}}_k \neq \overline{\overline{f}}_{k+1}$ for all k-ary approximations $\overline{\overline{g}}_k$ associated with $\vec{g}$. It suffices to prove this for sets of length $k+1$, since functions must agree on every input to be equivalent. We will proceed by showing that, for any $\vec{g}$ and length-$(k+1)$ sequence $\boldsymbol{S}$, the quotient $\overline{\overline{g}}_k/\overline{\overline{f}}_{k+1}$ cannot be identically equal to 1.

By definition, since $|\boldsymbol{S}| = k+1$, and the product function is permutation-invariant,

$$\frac{\overline{\overline{g}}_k(\boldsymbol{S})}{\overline{\overline{f}}_{k+1}(\boldsymbol{S})} = \frac{\frac{1}{(k+1)!}\sum_{\pi\in\mathbb{S}_{k+1}} \vec{g}_k(\pi\cdot\boldsymbol{S})}{\frac{1}{(k+1)!}\sum_{\pi\in\mathbb{S}_{k+1}}\prod_i \boldsymbol{S}_{i,1}} = \frac{\sum_{\pi\in\mathbb{S}_{k+1}} \vec{g}_k(\pi\cdot\boldsymbol{S})}{(k+1)!\prod_i \boldsymbol{S}_{i,1}}.$$

Next, again using $|\boldsymbol{S}| = k+1$, we rewrite $\vec{g}_k(\pi\cdot\boldsymbol{S}) = \vec{g}((\pi\cdot\boldsymbol{S})_{-(k+1),:})$ where $\boldsymbol{S}_{-i,:}$ denotes the sequence with index i removed. We will use this to find another expression for the summation. Observe that $\{(\pi\cdot\boldsymbol{S})_{-(k+1),:} : \pi \in \mathbb{S}_{k+1}\} = \bigcup_{i=1}^{k+1}\bigcup_{\tilde{\pi}\in\mathbb{S}_k}(\tilde{\pi}\cdot(\boldsymbol{S}_{-i,:}))$. In other words, we remove element i in the (implicitly ordered) sequence and permute the remaining elements. Thus,

$$\sum_{\pi\in\mathbb{S}_{k+1}} \vec{g}_k(\pi\cdot\boldsymbol{S}) = \sum_{\pi\in\mathbb{S}_{k+1}} \vec{g}((\pi\cdot\boldsymbol{S})_{-(k+1),:}) = \sum_{j=1}^{k+1}\sum_{\tilde{\pi}\in\mathbb{S}_k} \vec{g}(\tilde{\pi}\cdot(\boldsymbol{S}_{-j,:}))$$

and so

$$\frac{\sum_{\pi\in\mathbb{S}_{k+1}} \vec{g}_k(\pi\cdot\boldsymbol{S})}{(k+1)!\prod_i \boldsymbol{S}_{i,1}} = \frac{\sum_{j=1}^{k+1}\sum_{\tilde{\pi}\in\mathbb{S}_k} \vec{g}(\tilde{\pi}\cdot(\boldsymbol{S}_{-j,:}))}{(k+1)!\prod_i \boldsymbol{S}_{i,1}} = \frac{1}{(k+1)!}\sum_{j=1}^{k+1}\frac{1}{\boldsymbol{S}_{j,1}}\underbrace{\sum_{\tilde{\pi}\in\mathbb{S}_k}\frac{\vec{g}(\tilde{\pi}\cdot\boldsymbol{S}_{-j,:})}{\prod_{i\neq j}\boldsymbol{S}_{i,1}}}_{\text{denote by }\overline{\overline{a}}(\boldsymbol{S}_{-j,:})}.$$

Therefore, altogether, we have shown

$$\frac{\overline{\overline{g}}_k(\boldsymbol{S})}{\overline{\overline{f}}_{k+1}(\boldsymbol{S})} = \frac{1}{(k+1)!}\sum_{j=1}^{k+1}\frac{\overline{\overline{a}}(\boldsymbol{S}_{-j,:})}{\boldsymbol{S}_{j,1}}. \tag{4.14}$$

Finally, suppose for contradiction that $\overline{\overline{g}}_k/\overline{\overline{f}}_{k+1}\equiv 1$. Then, $\frac{1}{(k+1)!}\sum_{j=1}^{k+1}\frac{\overline{\overline{a}}(\boldsymbol{S}_{-j,:})}{\boldsymbol{S}_{j,1}} = 1$ and so

$$\overline{\overline{a}}(\boldsymbol{S}_{-1,:}) = \boldsymbol{S}_{1,1}\Big((k+1)! - \sum_{j=2}^{k+1}\frac{\overline{\overline{a}}(\boldsymbol{S}_{-j,:})}{\boldsymbol{S}_{j,1}}\Big).$$

Thus, we see that $\overline{\overline{a}}(\boldsymbol{S}_{-1,:})$, a function that by definition ignores $\boldsymbol{S}_{1,:}$ is a function of $\boldsymbol{S}_{1,1}$. This cannot be true in general for $|\mathcal{X}|>1$, a contradiction that completes the proof.

(**Graphs**) To consider graphs, write $\mathcal{X}=\mathscr{G}_{d_v,d_e}$ for some d_v, d_e. Proving $\mathcal{F}_k$ is a subset of $\mathcal{F}_{k+1}$ is similar to the above, with $\downarrow_k$ now selecting induced subgraphs rather than subsequences. We need to show that there is a $(k+1)$-ary JP function, not constant on featureless graphs of the same size, that cannot be represented by a k-ary graph function, for $k\in\mathbb{Z}_{\geq 1}$. The case for $k=1$ is trivial, since a 1-ary graph function cannot select any pairs of vertices, so let $k\geq 2$.

We will consider CSL graphs (Definition 12 and Figure 4.2). Let $n_k > 2(k+1)^2$ be a prime integer.[9] Since n_k is prime, it is co-prime with k and $k+1$, and furthermore $n_k - 1 > k+1 > k$. Thus, we can construct CSL graphs $\mathcal{G}_{\text{CSL}}(n_k,k)$ and $\mathcal{G}_{\text{CSL}}(n_k,k+1)$, and it suffices to demonstrate that a $(k+1)$-ary graph function exists that can distinguish $\mathcal{G}_{\text{CSL}}(n_k,k)$ from $\mathcal{G}_{\text{CSL}}(n_k,k+1)$ while no k-ary function can.

First, observe that Definition 12 effectively defines an ordering for CSL graphs. It is easy to see that we can identify whether a graph is in this ordering – as opposed to some other permutation – by inspecting the adjacency matrix. In short, in this ordering, there

[9]The choice of n_k facilitates intuition. The idea is that the CSL graph is so large that many hops along edges are required before it can wrap around.

is a 1 in each entry just above and just below the main diagonal. For all graphs $\mathcal{G}$ and permutations π, define $\vec{f}_{k+1}(\pi \cdot \mathcal{G}) = \vec{f}_{k+1}(\pi \cdot \boldsymbol{A}, \mathbf{1}) = \vec{g}_{k+1}\Big(\pi \cdot \boldsymbol{A}\Big)\mathbb{I}(\pi \cdot \boldsymbol{A})$, where $\mathbb{I}$ is the indicator function that takes 1 if $\pi \cdot \boldsymbol{A}$ is in this canonical ordering and 0 otherwise, and $\vec{g}_{k+1}(\pi \cdot \boldsymbol{A}) = \sum_{i=1}^{k+1}\sum_{j=1}^{k+1}(\downarrow_{k+1}(\pi \cdot \boldsymbol{A}))_{i,j}$ is twice the number of edges in the k-size induced subgraph. Thus, the associated Janossy function $\overline{\vec{f}}_{k+1}$ counts (twice) the number of edges in the subgraph induced by vertices $1, 2, \ldots, k+1$ of the graph in canonical order. Letting $\mathcal{G}_{k+1}^{\text{canon}} := \mathcal{G}_{\text{CSL}}(n_k, k+1)$ be the graph in canonical ordering and similarly for $\mathcal{G}_k^{\text{canon}}$, we see $\vec{g}_{k+1}\Big(\mathcal{G}_{k+1}^{\text{canon}}\Big) < \vec{g}_{k+1}\Big(\mathcal{G}_k^{\text{canon}}\Big)$ since the skip connection of length $k+1$ skips "outside" the subgraph. Finally, the Janossy function associated with $\vec{f}_{k+1}$ is

$$
\begin{aligned}
\overline{\vec{f}}_{k+1}(\mathcal{G}_{\text{CSL}}(n_k, k+1)) &= \frac{1}{n_k!}\sum_{\mathbb{S}_{n_k}} \vec{f}_{k+1}\Big(\pi \cdot \mathcal{G}_{\text{CSL}}(n_k, k+1)\Big) \\
&= \frac{1}{n_k!}\vec{g}_{k+1}\Big(\mathcal{G}_{k+1}^{\text{canon}}\Big) \\
&< \frac{1}{n_k!}\vec{g}_{k+1}\Big(\mathcal{G}_{k}^{\text{canon}}\Big) \\
&= \overline{\vec{f}}_{k+1}(\mathcal{G}_{\text{CSL}}(n_k, k)).
\end{aligned}
$$

Thus, $\overline{\vec{f}}_{k+1}$ takes different values on these two graphs.

Second, we will show that no k-ary $\overline{\vec{f}}_k \in \mathcal{F}_k$ can distinguish $\mathcal{G}_{\text{CSL}}(n_k, k+1)$ and $\mathcal{G}_{\text{CSL}}(n_k, k)$. In particular, denote by $\mathscr{L}_k = \Big\{\!\!\Big\{ \downarrow_k(\mathcal{G}_{\text{CSL}}(n_k, k)) \Big\}\!\!\Big\}$ the multiset of induced subgraphs of size k in $\mathcal{G}_{\text{CSL}}(n_k, k)$ and $\mathscr{L}_{k+1} = \Big\{\!\!\Big\{ \downarrow_k \mathcal{G}_{\text{CSL}}(n_k, k+1) \Big\}\!\!\Big\}$ be the k-sized induced subgraphs in $\mathcal{G}_{\text{CSL}}(n_k, k+1)$. If we can show that $\mathscr{L}_k$ and $\mathscr{L}_{k+1}$ are "equivalent", then there is no way for a function that models k-sized subgraphs of each to distinguish the two graphs. Formally, we will construct a bijection $\gamma : \mathscr{L}_k \to \mathscr{L}_{k+1}$ such that, for every $\mathcal{H} \in \mathscr{L}_k$, $\gamma(\mathcal{H}) \in \mathscr{L}_{k+1}$ is isomorphic to $\mathcal{H}$, and if such a bijection exists, we will write $\mathscr{L}_{k+1} \simeq \mathscr{L}_k$. What follows is a sketch since the idea is clear but would require much notation.

We show that all induced subgraphs in $\mathscr{L}_k$ and $\mathscr{L}_{k+1}$ are forests[10] with maximum degree of four. The maximum degree is four since CSL graphs are four-regular by definition. To see they are acyclic, consider what is required to form a cycle in k-sized induced subgraphs of CSL graphs. Call the two types of edges in Definition 12 1-step and skip-link edges. We

[10] A forest is a disjoint union of trees. A tree is acyclic.

could create a cycle by wrapping around the circle on 1-step and skip-link edges, but this is not possible due to the large value of n_k. Next, we could form "local" cycles: without loss of generality, we could traverse clockwise and then return on a skip link. But, with skip links of size k or $k+1$, at least $(k+1)$-sized subgraphs would be required.

Without cycles as k-sized induced subgraphs, the induced subgraphs of these CSL graphs become identical. In particular, any tree in $\mathscr{L}_k$ must be created by following either 1-step or k-step links in $\mathcal{G}_{\text{CSL}}(n_k, k)$, but any such tree can be created in $\mathcal{G}_{\text{CSL}}(n_k, k+1)$. The other direction is true, so it follows that $\mathscr{L}_{k+1} \simeq \mathscr{L}_k$. Since the multisets of k-sized induced subgraphs are indistinguishable, no k-ary JP function can distinguish the graphs. ■

Thus, we see that k is an important hyperparameter, trading off between expressiveness and computational cost.

Implications for Latent Space Pooling. One of our motivations for JP was that DeepSets fails to capture higher-order interactions in the input. We now prove Lemma 1, that DeepSets is a special case of JP, 1-ary JP.

Proof 8 (**Lemma 1**) We prove that any DeepSets model can be written as a 1-ary JP model. Recall from Equation 3.3 that, for neural networks r and g with parameters $\mathbf{\Theta}_r$ and $\mathbf{\Theta}_g$ respectively, the DeepSets model computes $r\Big(\big(\sum_{i=1}^{|\boldsymbol{S}|} g\big(\boldsymbol{S}_{i,:}; \mathbf{\Theta}_g\big)\big); \mathbf{\Theta}_r\Big)$, for any set $\boldsymbol{S}$. A JP model is defined by a readout r_{JP} and a function $\vec{f}$ with parameters $\mathbf{\Theta}_{\text{JP}}$ and $\mathbf{\Theta}_f$ respectively. So, define $\vec{f}(\boldsymbol{S}; \mathbf{\Theta}_f) = |\boldsymbol{S}| g\Big(\downarrow_1(\boldsymbol{S}); \mathbf{\Theta}_g\Big)$ for all sets $\boldsymbol{S}$ and $r_{\text{JP}}(\boldsymbol{h}; \mathbf{\Theta}_{\text{JP}}) = r(\boldsymbol{h}; \mathbf{\Theta}_r)$ for all (hidden) vectors $\boldsymbol{h} \in \mathbb{R}^{d_h}$. Then, by Equation 4.3 and the same reduction of redundant terms used in the proof of Proposition 4, the JP prediction is

$$
\begin{aligned}
y(\boldsymbol{S}; \mathbf{\Theta}_f, \mathbf{\Theta}_{\text{JP}}) &= r_{\text{JP}}\Big(\frac{1}{|\boldsymbol{S}|!} \sum_{\pi \in \mathbb{S}_{|\boldsymbol{S}|}} \vec{f}(\pi \cdot \boldsymbol{S}; \mathbf{\Theta}_f); \mathbf{\Theta}_{\text{JP}}\Big) = r\Big(\frac{|\boldsymbol{S}|}{|\boldsymbol{S}|} \sum_{i=1}^{|\boldsymbol{S}|} g\big(\boldsymbol{S}_{i,:}; \mathbf{\Theta}_g\big); \mathbf{\Theta}_r\Big) \\
&= r\Big(\sum_{i=1}^{|\boldsymbol{S}|} g\big(\boldsymbol{S}_{i,:}; \mathbf{\Theta}_g\big); \mathbf{\Theta}_r\Big).
\end{aligned}
$$

Observing that this is a 1-ary JP model completes the proof. ■

Remark 8 *Theorem 3 and Lemma 1 imply that the pooling layer of DeepSets is less expressive than k-ary JP with $k > 1$. This formalizes our intuition that processing one input at a*

time is less expressive than processing pairs, triplets, and so on. Yet, since DeepSets has universal approximation guarantees, we conclude that the responsibility of modeling high-order relationships is pushed to r. This offers an alternative explanation for the fact that DeepSets requires highly complex functions to achieve universal guarantees over set inputs, sometimes making it difficult to learn in practice [62].

Unification. Let us highlight that existing approaches in the literature can be considered k-ary JP, such as the latent space and higher-order methods discussed in Section 3.2.3. DeepSets is 1-ary JP while SetTransformer and Relation Network use 2-ary JP layers. Theorem 3 offers theoretical insights and a new perspective for specifying permutation-invariant models. Choosing between DeepSets and SetTransformer, for example, amounts to a choice of k and $\vec{f}$. In particular, the modeler should carefully consider any prior knowledge about the order of interrelations (i.e., value of k) and the availability of computational resources when when specifying a permutation-invariant model.

Combining k-ary and π-SGD

While k-ary approximation reduces computation from $|\underline{\mathbf{x}}|!$ to $|\underline{\mathbf{x}}|!/(|\underline{\mathbf{x}}|-k)!$, this can still be intractably large even for inputs of moderate size. Therefore, we can combine π-SGD with k-ary approximations, which amounts to a scheme of selecting random k-sized subsets or induced subgraphs from $\underline{\mathbf{x}}$ before passing to $\vec{f}$. An example is shown in Figure 4.4. However, uniformly sampling would not result in unbiased estimation of graph characteristics. Exploring the connections between our approach and efficient unbiased samplers [260], [261] is an avenue for future work.

Canonical and Poly-canonical Orderings

We saw in Section 3.2.1 that ordering the data is a popular approach to constructing permutation-invariant models of graphs or sets. In the JP framework, this is equivalent to defining a new function $\vec{f}_{\text{canonical}} = \vec{f} \circ \text{ORDER}$ where $\text{ORDER}(\underline{\mathbf{x}}) = \text{ORDER}(\pi \cdot \underline{\mathbf{x}})$ is a "simple" permutation-invariant function and $\vec{f}$ is any function on $\mathcal{X}$. In this case, JP reduces to $\overline{\vec{f}}_{\text{canonical}}(\underline{\mathbf{x}}) = \frac{1}{|\underline{\mathbf{X}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \vec{f}_{\text{canonical}}(\pi \cdot \boldsymbol{x}) = \vec{f}\Big(\text{ORDER}(\underline{\mathbf{x}})\Big)$, eliminating the sum

over permutations. Equivalently, we can define this approximation as enforcing the constraint $\vec{f}(\underline{\mathbf{x}}) = \mathbf{0}$ whenever $\underline{\mathbf{x}} \neq \text{ORDER}(\underline{\mathbf{x}})$. Note that this is not as computationally intensive as original JP when ORDER is a *simple* function such as SORT. If we have prior knowledge about a "good" ordering, then this is a viable strategy for permutation-invariant modeling. However, we discussed its limitations in Section 3.2.1 and believe that the flexibility of π-SGD and k-ary approximations make them more generally applicable solutions.

We also saw that an ordering can be *learned* from the training data, and this is another viable approximation strategy if learning can be performed efficiently. However, we saw that this approach may not lead to strong generalization performance and is philosophically different than invariant training.

Poly-canonical Orderings. Rather than defining an ordering that maps all of $\text{orbit}(\underline{\mathbf{x}}) = \{g \cdot \underline{\mathbf{x}} : g \in \mathbb{G}\}$ to a singleton, we can define a *poly*-canonical ordering that yields more than one, but fewer than $|\underline{\mathbf{x}}|!$, possible outputs. For instance, consider a graph $\mathcal{G}$ with vertex set $\mathcal{V} = \{1, 2, \ldots, |\mathcal{G}|\}$. For all permutations π, $\pi \cdot \mathcal{G}$ can be sorted by depth-first search or breadth-first search starting from $\pi(1)$ [262]. A DFS defines a bijection $\text{DFS} : \mathcal{V} \to \mathcal{V}$, i.e., a permutation, and we can define $\Pi_{\text{dfs}}(\mathcal{G}) = \{\text{DFS}(\pi \cdot \mathcal{G}) : \pi \in \mathbb{S}_{|\mathcal{G}|}\}$. Typically, $|\Pi_{\text{dfs}}(\mathcal{G})| < |\mathcal{G}|!$, so pooling over permutations $\pi \in \Pi_{\text{dfs}}$ reduces computation. As an example, consider these two isomorphic graphs ①-②-③ and ①-③-② . The numbers illustrate orderings, or permutations,[11] π_1 and π_2. A DFS would start at the vertex labeled 1 and result in ①-②-③ for each. However, a DFS ordering of ②-①-③ is still ②-①-③ and therefore different from the previous examples. Hence, the DFS *reduces* the number of distinct permutations to sum over, but not to just one.

Enumerating all DFS or BFS orderings can be expensive, but this becomes computationally feasible when used in conjunction with our other tractability approaches. For instance, using π-SGD, k-ary approximations, and DFS ordering is equivalent to selecting a random vertex and running DFS until k vertices have been selected. This has the advantage of sampling *connected* subgraphs, whereas simply combining π-SGD and k-ary can sample *disconnected* subgraphs, which may be less meaningful. In a sparse graph, selecting only

[11] Recall from Chapter 3 that encoding a graph requires defining an ordering on the vertex set, which is effectively a permutation, without loss of generality.

connected induced subgraphs can dramatically reduce the number of subgraphs considered compared to naive k-ary with π-SGD.

More General Invariances. Canonical orientations have a natural extension to other group invariances beyond permutations. For instance, in a rotation-invariant image classification task, we could pre-rotate an image into one or a few useful orientations. As another example, we can view any preprocessing of image brightness or contrast as a canonicalization [160]. Whether the canonical form is meaningful will depend on the task and model at hand, and it still may be preferable to use other strategies such as an architecture that is transformation-invariant by design or an analogue of π-SGD.

Synthesis of Approximation Schemes

Our approximations are not mutually exclusive and can be combined to form a vast array of strategies for enforcing invariance or encouraging approximate invariances. On the one hand, k-ary and poly-canonical orderings provide spectrums that trade off prior knowledge and computation. Canonical orderings may succeed if the ordering is correlated to the task at hand but can fail otherwise. k-ary approximations correspond to an assumption about the order of relationships (pairwise, ternary, etc.) that are important to solving the task. On the other hand, π-SGD is broadly applicable and does not make an explicit assumption about the task other than that (approximate) invariances are important.

In terms of computation, canonical orderings collapse the number of permutations to sum over and can lead to significant time savings. π-SGD also reduces the number of permutations, but unlike training on data in fixed canonical orientations, requires permuting the input at every training epoch. While k-ary approximations also reduce the number of terms to pool over, computation may still be infeasible if $|\underline{\mathbf{x}}|$ is large, in which case we can pair it with π-SGD training.

4.2 Probabilistic Motivations

Our work has strong connections to probability and exchangeable distributions. We motivated JP through the lens of invariances in neural networks, but probabilistic insights

offer an alternative perspective. We begin with a review of the concepts and then discuss connections to JP.

4.2.1 Review of Infinite and Finite Exchangeability

A fundamental result underpinning Bayesian statistics is de Finetti's theorem, which characterizes the distribution of exchangeable random variables [175], [263]. We say that a sequence of random variables is *exchangeable* when the joint probability distribution is invariant to permutations, a weaker condition than independence. Consider an infinite sequence of exchangeable random variables or a finite sequence whose distribution arises as a marginalization of an infinitely long exchangeable sequence [175], [264], [265]. When this holds, the data distribution $\mathbb{P}(X_1, X_2, \ldots)$ can be decomposed as

$$\mathbb{P}(X_1, X_2, \ldots) = \int_{\theta \in \Theta} \prod_{i=1}^{\infty} \mathbb{P}(X_i \mid \theta) \mathbb{P}(\theta) d\theta, \tag{4.15}$$

where Θ is the parameter space.[12] One interpretation is that this theorem justifies the use of a prior distribution. For our purposes, it implies a mixture distribution over conditionally *independent* random variables given θ [263], [265]. Moreover, it imposes strong structural requirements: the probability of any subsequence must equal the marginalized probability of the original sequence (i.e., *projectivity*).

Finite exchangeability drops this projectivity requirement [264]. To be concrete, consider the *representational form* of such distributions [266], [267]. Letting $X_1, X_2, \ldots, X_m$ be m binary random variables, their probability distribution can be represented by the generation process: (1) randomly select an integer $s \in \{1, \ldots, m\}$, (2) put m balls in an urn where exactly s are labeled "success", and (3) randomly draw balls from that urn one at a time. Observe that if an urn has s successes, and the first s draws are all successes, next cannot be. In other words, the draws are not i.i.d. In general, finite exchangeable distributions cannot be simplified beyond some form involving a non-i.i.d. distribution $\mathbb{P}^m_{\text{exch}}(X_1, \ldots, X_m)$ [84]. Thus,

[12] This decomposition is written in a simplified form, as in [175], which is satisfactory for our discussion here. A more general form can be found in [265]. We follow convention in letting Θ denote the parameter space. In this discussion, it does not represent a matrix.

whereas infinite exchangeability implies a mixture over conditionally *independent* random variables, finitely exchangeability implies a mixture over *dependent* random variables [264].

A special permutation-invariant distribution arises in point processes, which include distributions over points in space. Consider a generating process where both the *number* of points and their *location* are random. This arises, for instance, in Lajos Janossy's study of particle showers [83], [84], and the distribution is described by a *Janossy measure*. We will introduce a simple example to make the connection apparent. Following Vo, Dam, Phung, *et al.* [268], if we suppose $\mathcal{X}$ is a countable input space, then, the likelihood function of a point pattern is

$$f(X_1, \ldots, X_K) = \mathbb{P}_c(K) \sum_{\pi} \mathbb{P}(X_{\pi(1)}, \ldots, X_{\pi(K)} \mid K),$$

where $\mathbb{P}_c(K)$ is the probability of there being K points total, π are permutation functions, and $\mathbb{P}(x_{\pi(1)}, \ldots, x_{\pi(K)} \mid K)$ gives the joint probability of the points. Noting the similarity of this expression to Equation 4.1, we call our approach *Janossy pooling*.

Turning to graphs, there is an analogue of de Finetti's theorem for exchangeable 2-arrays, of which graphs are a special case. Observed graphs are considered to be induced subgraphs of an infinite random 2-array, and the distribution is characterized by the Aldous-Hoover theorem [265], [269], [270]. There are two notions of exchangeability for graphs. We have focused on *joint* exchangeability: the distribution of the array is invariant to applying the *same* permutation to the rows and columns. A *separately* exchangeable distribution is invariant to applying *different* permutations to the rows and columns. These ideas helped formulate the definition of JP and our extension to bipartite graphs in Section 4.3.2.

4.2.2 Exchangeability and Neural Networks

Zaheer, Kottur, Ravanbakhsh, *et al.* [26] note the similarity between DeepSets and the decomposition in Equation 4.15. Indeed, the pooling operation $\sum_i g(\boldsymbol{S}_{i,:})$ can be viewed as (proportional to) a log likelihood of conditionally i.i.d. random variables, where $g(\boldsymbol{S}_{i,:})$ is the log probability. Thus, DeepSets arises from an assumption of infinite exchangeability, and may be misspecified. This connection provides another perspective into the limitation

that DeepSets struggles to capture higher-order relationships. In contrast, JP captures relationships among random variables in a set and does not assume conditional independence, more like the *finitely* exchangeable distributions. To draw further parallels, finite Gibbs models [271] restrict the structure of $\mathbb{P}_{\text{exch}}$ to fixed-order dependencies, similar to k-ary Janossy.

Bloem-Reddy and Teh [148] develop a link between invariant deterministic functions and exchangeable distributions to inform neural network design by leveraging the idea of *noise outsourcing*. That is, under some fairly general conditions, the conditional distribution between any random variables X and Y, $\mathbb{P}_{Y|X}$, has a decomposition into a deterministic function of X and independent noise. Let $\mathbb{G}$ be a compact group, which includes the symmetric group of permutations. They show that a $\mathbb{G}$-invariant distribution can be represented by noise outsourcing with a deterministic neural network if there exists a *maximal-invariant* function of the data. By definition, such functions are constant on orbits $\text{orbit}(x) = \{G \cdot x : G \in \mathbb{G}\}$ and assign different values to distinct orbits. Providing a unique representation for every orbit is precisely the goal of many JP models and forms the idea behind our proofs of their expressive power. It is interesting that maximal invariants can in theory always be computed for the discrete case, but may not be trivial to construct for continuous groups of transformations. This represents an interesting challenge for any future explorations of JP for continuous transformations.

4.3 Extensions

In this section, we briefly outline extensions of JP. These open interesting avenues for future exploration and further demonstrate the flexibility of JP as a framework, although we have not explored them at length. Moreover, just as JP provides a theoretical explanation of existing approaches for supervised learning on graphs and sets, a theoretical development of these ideas could further our understanding of other methods.

4.3.1 Equivariance

As defined in Equation 4.1, JP models *invariant* functions and maps possibly variable-size inputs to fixed-length vector or scalar outputs. However, JP can be extended to learning *equivariant* representations (see Definition 5). That is, it can map inputs to an output of the same size in an equivariant way.

For simplicity and brevity, we will define equivariant JP for fixed-length sequences, but it is straightforward to generalize the definition to graphs and variable-length inputs. Let $\vec{f} : \mathbb{R}^{n \times d_s} \to \mathbb{R}^{n \times d_h}$ be a permutation-sensitive function whose input and output are both length-n sequences. Then, we can define equivariant JP by

$$\vec{\vec{f}}(\underline{\mathbf{x}}) = \frac{1}{|\underline{\mathbf{x}}|!} \sum_{\pi \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \pi^{-1} \cdot \vec{f}(\pi \cdot \underline{\mathbf{x}}), \quad \underline{\mathbf{x}} \in \mathbb{R}^{n \times d_s},$$

where $\pi^{-1} \in \mathbb{S}_{|\underline{\mathbf{X}}|}$ denotes the inverse element of π in $\mathbb{S}_{|\underline{\mathbf{X}}|}$. Notice that $\mathbb{S}_n$ acts on both $\mathbb{R}^{n \times d_s}$ and $\mathbb{R}^{n \times d_h}$ by permuting the rows but not the columns (see Equation 2.3). Such a pooling layer could be useful for mapping every element in a sequence to a latent space without losing the sequence structure or the ordering presented. To define this for graphs, we simply let $\underline{\mathbf{x}}$ denote a graph and use the appropriate action of the symmetric group on graphs (Definition 9). Equivariant JP could provide another approach to learning vertex-level representations.

4.3.2 Separate Janossy Pooling

Motivated by the distinct notions of *separate* exchangeability and *joint* exchangeability [265], we can also extend Janossy pooling to *bipartite* graphs. A bipartite graph is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{V}$ is partitioned into nonempty sets $\mathcal{V}_1$ and $\mathcal{V}_2$ with the property that all edges are between exactly one vertex in $\mathcal{V}_1$ and one in $\mathcal{V}_2$ [259]. An example is shown in Figure A.1 of the appendix. Bipartite graphs are often used to represent relationships between distinct entities, such as users ($\mathcal{V}_1$), movies ($\mathcal{V}_2$), and the ratings that users assign to movies (edges) [265]. As before, we must *encode* such graphs by defining an ordering over

$\mathcal{V}_1$ and $\mathcal{V}_2$ (see Figure 3.1 and Section 3.1.2), and thus an invariant/equivariant model is typically desired.

For simplicity, consider a bipartite graph $\mathcal{G}$ without vertex or edge features. Rather than encoding $\mathcal{G}$ with an adjacency matrix, we construct a matrix $\boldsymbol{M} \in \{0,1\}^{|\mathcal{V}_1|\times|\mathcal{V}_2|}$ whose rows represent entities in $\mathcal{V}_1$, columns represent $\mathcal{V}_2$, and whose binary entries indicate whether an edge is present. Then, to extend JP, we sum over two sets of permutations, $\mathbb{S}_{|\mathcal{V}_1|}$ and $\mathbb{S}_{|\mathcal{V}_2|}$, applying permutations $\pi_1 \in \mathbb{S}_{|\mathcal{V}_1|}$ to rows and $\pi_2 \in \mathbb{S}_{|\mathcal{V}_2|}$ to columns. Extensions to graphs with features and to multipartite graphs are possible but omitted for brevity. Since bipartite graphs typically represent different types of entities, learned vertex-level representations may be more useful than graph-wide representations. Hence, we could combine equivariant JP with separate JP to learn vertex-level representations in bipartite graphs.

4.4 Experiments

In this section, we empirically demonstrate the stated benefits and validate the theoretical properties of JP. Five broad questions that we answer are: (1) Do we find that JP can better model complex relationships in a set, compared to DeepSets? (2) Does RPGNN improve the expressiveness of MPGNNs? (3) Do experiments bear out the predicted benefits of increasing k and the number of sampled permutations? (4) How do different ordering schemes perform? (5) Does π-SGD training learn an approximately invariant model? Meanwhile, by illustrating a diverse set of tasks and datasets, we showcase the flexibility of the JP framework.

Since different tasks are brought to bear on the same questions, we will first lay out the datasets. Then, we will proceed through different explorations and discuss the results. Implementation details such as hyperparameter tuning and loss functions are provided in the appendix.

4.4.1 Datasets

We use datasets of both sets and graphs. The first two are simple synthetic datasets which abstract away complexities that are not central to our investigation. That is, they are

Table 4.1. Arithmetic Tasks. Length n and support size M for (multi)sets of integers sampled uniformly at random with replacement from $\{0, 1, \ldots, M-1\}$.

	sum	range	variance	unique sum	unique count
n	5	5	10	10	10
M	100	100	100	10	10

designed to be what John Lafferty called "fruit fly" experiments in a presentation at Purdue. The remaining are real-world datasets that are popular in the literature.

Integer Arithmetic Datasets

This dataset extends the experiments in [26]. We sample (multi)sets of length n, uniformly at random and with replacement, from the set of integers $\{0, 1, \ldots, M-1\}$, where integers n and M depend on the task and are shown in Table 4.1. The goal is to compare JP to DeepSets (a universal approximator) on tasks that require modeling *relationships* between elements, and we predict DeepSets to struggle. In contrast, π-SGD training will allow us to model the full sequence (e.g., with an LSTM) to capture higher-order relationships, while promoting permutation invariance for the sake of generalization. We consider *summation* (unary relationships) for comparison, *population variance* (binary relationships), *unique sum, unique count,* and *range* (higher-order relationships).

Population variance is defined by $\mathrm{Var}(\boldsymbol{x}) = \frac{1}{n}\sum_i (x_i - \bar{x})^2 = \frac{1}{2n^2}\sum_{i,j}(x_i - x_j)^2$ for a sequence $\boldsymbol{x}$ of integers and involves 2-ary relationships. *Unique count* is the number of unique elements in the input and *unique sum* is the sum thereof. The *range* is defined by $\max(\boldsymbol{x}) - \min(\boldsymbol{x})$.

The values of n and M, shown in Table 4.1, are loosely based on the choices of [26], and are varied to control the difficulty of the task. We keep the sequences relatively small to enable experiments with *exact* k-ary training (i.e., k-ary without π-SGD). A large value of M for the sum, range, and variance tasks prevents the problem from becoming too easy, as these functions can center around some value that can be predicted without learning the correct function. Unique sum and unique count are already difficult, so we kept M at 10, per [26].

Following existing literature, we treat all tasks except for variance as binary classification: either the prediction is correct or incorrect. For variance, we consider a regression problem with RMSE as the metric. The training data consists of 100,000 randomly sampled sequences.

Circulant Skip Link Graphs

We create a dataset of CSL graphs (Definition 12 and Figure 4.2). Recall that $\mathcal{G}_{\text{CSL}}(n, a)$ and $\mathcal{G}_{\text{CSL}}(n, a')$ are not isomorphic unless $a \equiv \pm a' \mod n$ but MPGNNs (Equation 3.1) are incapable of yielding distinct predictions for the two. Thus we define a graph classification task: for a fixed graph size $n = 41$, we create CSL graphs with different skip lengths a and predict the length $a \in \{2, 3, 4, 5, 6, 9, 11, 12, 13, 16\}$. These skip lengths produce graphs that cannot even be distinguished by the more sophisticated vf2 testing algorithm [272], and 41 is the smallest number of vertices such that 10 such skip lengths can be produced. This is tantamount to classifying graphs into an isomorphism class. Note, we are not tackling the graph isomorphism problem since our method is an approximation. To generate multiple graphs in a class, we create a "canonical" CSL graph for each a, then apply random permutations. We sample 15 graphs in each class.

In this fruit fly experiment, no additional features were added, so $d_v = d_e = 1$. Thus, graphs are encoded by pairs $(\boldsymbol{A}, \mathbf{1})$ containing an adjacency matrix and a constant vertex feature.

Molecules

Learning on molecules, encoded as graphs (Figure 3.2), is a popular task for benchmarking the effectiveness of graph neural networks. Not only does modeling molecules have important applications, but molecular datasets are also more challenging than other available benchmarks [273]. We chose three datasets from the MoleculeNet project [41], [274]: MUV, HIV, and Tox21 [275]–[277].

The **HIV** dataset contains molecules and whether or not they inhibit HIV, resulting in a binary classification problem. The **Tox21** dataset is associated with the *Toxicology in the 21st Century* initiative to advance the science around biochemical pathways. It measures

Table 4.2. Molecular Graph Data. The datasets come from MoleculeNet and DeepChem [41], [274]. Each train/val/test split is roughly 80%, 10%, and 10% of the total number of graphs, respectively. The number of vertex and edge features depend on the model and thus the experiment.

Data Set	Total Number of Graphs	Number of Targets
HIV	41,127	1
MUV	93,087	17
Tox21	7,831	12

molecular toxicity on 12 targets such as *aromatase inhibition* and *aryl hydrocarbon activation.* Finally, the **MUV** dataset is a carefully curated dataset that facilitates virtual screening analyses. It contains information on whether compounds are active/inactive for 17 targets, including various kinase and protease inhibition. Note that Tox21 and MUV are multiple-target problems. All targets are binary across all tasks.

All vertices (atoms) and edges (bonds) in all graphs (molecules) are associated with feature vectors. Atom features include one-hot encodings of the atom type, hybridization (SP, SP2), and aromaticity. The bond features include a one-hot encoding of bond type and stereochemistry in the *E-Z* convention [278]. The dimension of these feature vectors depends on the model, and differs from experiment to experiment. In short, we follow the recommendations of DeepChem. For each dataset, the data are split randomly into training, validation, and testing sets containing roughly 80%, 10%, and 10% of the total graphs respectively. A basic summary is shown in Table 4.2.

For experiments on molecular datasets, we will use the MPGNN of Duvenaud, Maclaurin, Iparraguirre, *et al.* [40]. This is a highly influential method for predicting molecular properties and is implemented in the DeepChem library [274]. The method, detailed in Algorithm 5 of the appendix, takes advantage of constraints that chemistry places on graphs. In short, it uses different aggregation functions depending on the degree of the vertex, since vertex degree is bounded in a small set by bonding laws.

Table 4.3. Real-World Vertex Classification Datasets

Characteristic	Cora	Pubmed	PPI
Number of Vertices	2,708	19,717	Total 56,944, Average 2,373[a]
Average Degree	3.898	4.496	28.8[a]
Number of Vertex Features	1,433	500	50
Number of Classes	7	3	121[b]
Number of Training Vertices	1,208[c]	18,217[c]	44,906[d]
Number of Validation Vertices	500	500	6,514[d]
Number of Test Vertices	1,000	1,000	5,524[d]

[a] The PPI dataset comprises several graphs, so the quantities marked with an "a" represent an average across graphs.
[b] The PPI task has 121 binary targets (it is multi-task).
[c] The validation and test sets were split evenly among the remaining vertices.
[d] All of the training vertices come from 20 graphs while the testing vertices come from graphs not seen during training.

Vertex Classification Datasets: PPI and Citation

We will use JP as an aggregator in MPGNNs for *vertex classification/regression*. That is, we define a model with the recursions described in Equation 3.1, using JP as a neighborhood aggregator. However, rather than ultimately aggregating the hidden vertex representations $\boldsymbol{H}_{1,:}^{(L)}, \boldsymbol{H}_{2,:}^{(L)}, \ldots, \boldsymbol{H}_{|\mathcal{G}|,:}^{(L)}$ into a graph-wide prediction, we use each to predict properties of vertices $1, 2, \ldots, |\mathcal{G}|$.

We consider the three undirected graph datasets used by Hamilton, Ying, and Leskovec [45]: Cora, Pubmed [279], and a protein-protein interaction (PPI) graph [46], [280], [281].[13] The first two are citation graphs where vertices represent papers, edges represent citations, and vertex features are bag-of-words representations of the document text. The task is to classify the paper topic. The PPI dataset is a collection of several graphs each representing a different human tissue; vertices represent proteins, edges represent protein interactions, features include genetic and immunological characteristics, and the task is to classify protein roles. We now discuss each dataset in more detail, and summary statistics are provided in Table 4.3.

[13] It is much more common to say protein-protein interaction *network*, but we write *graph* for consistency and to avoid confusion.

Protein-Protein Interaction Graphs Understanding protein function is critical for developing therapies and advancing our understanding of biology. Modeling protein-protein interaction (PPI) graphs [46] is an important tool in this endeavor. In such graphs, vertices represent proteins and the edges represent interactions. Interactions are measured by lab experiments such as yeast two-hybrid screens [282].

We will use the PPI dataset created by Hamilton, Ying, and Leskovec [45]. The authors obtained PPI graphs from the 2015 update of the BioGRID database [280], a collection of expert-curated protein interactions from different studies. Inspired by [46] the authors selected several PPI graphs, each corresponding to different tissues. Their motivation was to evaluate whether their GraphSAGE model trained for vertex classification on one set of graphs could generalize to unseen graphs. Specifically, there are 24 graphs, with 20 used for training, two for validation, and two for test. These particular graphs (tissues) were selected as they have at least 15,000 edges. Altogether, there are 56,944 vertices (proteins). The authors obtained vertex feature and target information from the Molecular Signature Database, an oncology of protein information [281]. For each vertex, there are 50 features representing properties such as immunological genetic information. These particular 50 attributes were chosen since they are expressed relatively frequently in the proteins collected. Still, most feature vectors are sparse. Every vertex is associated with a 121-dimensional binary target vector. Each binary value indicates whether or not a protein has some specific function.[14] A binary cross-entropy loss is applied to each target. Additional information on these graphs can be found in Table 4.3.

Cora graph The Cora citation dataset represents 2,708 papers from seven categories such as Neural Networks, Reinforcement Learning, and Genetic Algorithms. It is encoded as a graph whose vertices represent papers, edges represent citations, and vertex features capture information about the words in the document. While citations are inherently directed, the graph is encoded as an undirected graph. The vertex features are 1,433-dimensional binary vectors where a 1 indicates presence of word and 0 indicates absence; 1,433 is the number of unique words in all papers after performing stemming and stop-word removal. The task is to classify the paper topic, and cross-entropy loss is used.

[14] Unfortunately, more detail about the features and targets were not provided by the authors.

Table 4.4. Modeling Higher-Order Relationships on Arithmetic Tasks. Mean (standard deviation) performance, measured in RMSE for the variance task and accuracy (A) for the remaining arithmetic tasks, averaged across 15 runs. We compare JP models trained with π-SGD on the full sequence to 1-ary JP. DeepSets [26], a special case of JP, is shown in `typewriter font`. "u sum" denotes unique sum, similarly for "u count", and "Aff." denotes an affine layer. We study k-ary approximations and the number of sampled permutations in Section 4.4.4.

Model	r	k	samples	var (RMSE)	sum (A)	range (A)	u sum (A)	u count (A)
MLP	Aff.	1	–	119.05(1.29)	1.00(0.00)	0.04(0.00)	0.07(0.00)	0.36(0.01)
`MLP`	`MLP`	`1`	–	1.95(0.24)	1.00(0.00)	0.97(0.01)	1.00(0.00)	1.00(0.00)
GRU	Aff.	$\lvert\underline{\mathbf{x}}\rvert$	1	1.43(0.23)	0.99(0.01)	0.98(0.00)	1.00(0.00)	1.00(0.00)
GRU	Aff.	$\lvert\underline{\mathbf{x}}\rvert$	20	1.20(0.23)	0.99(0.00)	0.99(0.00)	1.00(0.00)	1.00(0.00)
GRU	MLP	$\lvert\underline{\mathbf{x}}\rvert$	1	0.42(0.62)	0.99(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
GRU	MLP	$\lvert\underline{\mathbf{x}}\rvert$	20	0.40(0.37)	0.99(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)

Pubmed The Pubmed dataset is another citations graph consisting of 19,717 papers on the topic of diabetes. There are 44,338 citation links, which are treated as undirected. Each paper is classified into three topics: Experimental Diabetes, Type 1 Diabetes, or Type 2 Diabetes. The vertex features are the TFIDF scores [283], an indicator of word importance, for each of the 500 words in the corpus.

4.4.2 Modeling Higher-Order Relationships

As a first step, we explore modeling higher-order relationships in a sequence. One of our key observations is that DeepSets is equivalent to 1-ary Janossy pooling, so its modeling capacity relies on a complex readout function r. In contrast, we expect pooling over a permutation-sensitive function $\vec{f}$, such as an RNN, whose input is the entire sequence, to attain better performance in general. To efficiently estimate a JP model with an RNN, we will train with π-SGD, sampling one permutation at each training epoch. We will study k-ary approximations more closely in Section 4.4.4 using several different datasets.

In particular, we use the integer arithmetic dataset of Section 4.4.1. These tasks require modeling varying levels of interactions within the set. We define $\vec{f}$ as a GRU (an RNN) [94]; given a sequence input $\underline{\mathbf{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, the GRU model returns a sequence $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n)$,

and we take $\boldsymbol{h}_n$.[15] We will approximate the Janossy model with π-SGD, sampling one and 20 permutations at inference time (inference time is studied more closely in Section 4.4.4). Our baseline is a 1-ary JP model where $\vec{f}_1$ is an MLP. For both full-sequence and 1-ary JP, we leverage two readout functions r: affine and MLP. When r and $\vec{f}_1$ are both MLPs, the model is equivalent to the DeepSets model [26] (see Lemma 1 and its proof). For a fixed r, we choose hyperparameters that keep the number of learnable parameters in the model similar. Many architectural choices were made to parallel the experiments in [26], and more details are provided in the appendix. We quantify variability by running the experiment 15 times.

Table 4.4 bears out our theoretical expectations. With a simple affine readout function r, the 1-ary model performs poorly for several tasks (e.g., unique sum and variance). However, when r is an MLP (DeepSets), performance improves dramatically. This demonstrates reliance on complex outer layers to achieve adequate performance. Yet, although DeepSets is a universal approximator, it does not outperform the Janossy models on the variance task. Overall, the JP models achieve much better performance regardless of the choice of readout, though a more complex r does improve performance on the variance task. Additionally, sampling more permutations at inference time improves performance. Finally, it is interesting that all but the simplest models are able to achieve near-perfect performance for several of the tasks.

4.4.3 Expressiveness of MPGNNs and RPGNNs

Next, we empirically demonstrate another key observation: RPGNNs are more powerful than MPGNNs (Theorem 2). We saw that MPGNNs cannot distinguish graphs such as the CSL graphs, but by adding permutation-sensitive one-hot IDs and training with JP, RPGNNs can.

CSL task. We begin with the CSL synthetic dataset. We choose GIN, a theoretically most powerful MPGNN [56], as the baseline, and in particular the variant with additional parameters ϑ (see Section 3.2.2). By definition, RPGIN is a JP model that defines $\vec{f}$ by

[15] To be clear, we are following the notation defined at https://pytorch.org/docs/stable/generated/torch.nn.GRU.html (accessed March 2021).

Table 4.5. Mean (standard deviation) accuracy across five folds on the CSL task.

GIN	RPGIN
0.1(0.0)	1.0(0.0)

adding IDs to the vertex features then passing the augmented graph to GIN (Equation 4.5). We use π-SGD to estimate JP. To evaluate performance at inference time, we sample 20 permutations and average the output, per Equation 4.12. To tune hyperparameters, we swept over the number of GNN recursions, width of the aggregator MLP, dimension of the one-hot ID, batch normalization, and dropout on an independent set of CSL graphs.

We evaluate both models with five-fold cross validation to estimate the generalization error. The classes are balanced in each. Table 4.5 bears out the theoretical predictions; GIN achieves only random accuracy – 10% on this 10-class classification task. In contrast, the RPGIN achieves strong performance, verifying the theory that it is more expressive and demonstrating that it can be trained effectively with π-SGD training.

Molecular Task. Next, we evaluate the performance of RPGNN on the real-world molecular dataset. As a baseline GNN, we chose that of Duvenaud, Maclaurin, Iparraguirre, *et al.* [40], discussed in Section 4.4.1 and Algorithm 5. Again, we append permutation-sensitive IDs to the features to define the RPDuvenaud et al. model. We train with π-SGD and 20 inference-time permutations. In terms of decisions about hyperparameters, data splits, and performance metrics, we follow those of [41], [274], which were based on existing chemoinformatic literature. We provide more details in the appendix.

The results shown in Table 4.6 suggest that RPDuvenaud et al. can improve performance over the baseline but does not impose a risk of hurting performance. Note that canonical orderings approaches are also shown in this table so that they can be compared with RPDuvenaud et al. in our discussion in Section 4.4.5.

Conclusions The CSL task showed that the RPGNN model can dramatically improve over the baseline for highly symmetric graphs when no features are available. For the molecular tasks, rich features are available, which help to break symmetries even in the baseline GNN. For these, the results of RPDuvenaud et al. were promising on the HIV task and achieved similar performance to the baseline on the others. Putting these two experiments

Table 4.6. Performance on Molecular Tasks. We show mean (standard deviation) ROC-AUC across multiple random data splits. The baseline MPGNN is from Duvenaud, Maclaurin, Iparraguirre, *et al.* [40] and described in Algorithm 5. Models under the JP framework are RPDuvenaud et al. and two based on poly-canonical orderings, described in Sections 4.4.3 and 4.4.5.

Model	HIV (AUC)	MUV (AUC)	Tox21 (AUC)
RPDuvenaud et al.	0.832 (0.013)	0.794 (0.025)	0.799 (0.006)
Duvenaud et al.	0.812 (0.014)	0.798 (0.025)	0.794 (0.010)
CNN-DFS	0.542 (0.004)	0.601 (0.042)	0.597 (0.006)
RNN-DFS	0.627 (0.007)	0.648 (0.014)	0.748 (0.055)

together, we suggest that practitioners train with RPGNN in place of their favorite GNN as it may improve generalization performance. The extent of improvement will depend significantly on the nature of the task.

4.4.4 Impact of k and Number of Sampled Permutations: GraphSAGE

Now we investigate the impact of k and π-SGD in approximating a JP model. We expect that increasing the number of sampled permutations at inference time for models trained with π-SGD will drive down variance and lead to better generalization performance. We also expect that increasing k in k-ary approximation makes the pooling layer more expressive (Theorem 3). The effect of k can be studied both with and without π-SGD. Additionally, by casting GraphSAGE [45] as a k-ary JP model with π-SGD, we leverage the insights from our theory to provide a simple modification that improves its performance.

Vertex classification. JP can be used as a neighborhood aggregator in a vertex classification task. Indeed, the GraphSAGE-LSTM [45] model uses an RNN over a set of randomly sampled features in vertex neighborhoods as an aggregator, making it a k-ary π-SGD JP aggregator. The model uses two layers of recursion, hence two approximate JP layers. However, the authors originally used only one permutation sample at inference time. Therefore, the vertex classification dataset of [45] serves the double purpose of a task for studying the effect of increasing k and the number of permutations sampled, as well as whether sampling more permutations at inference time can improve GraphSAGE. Regarding the architecture and evaluation, we mostly followed [45]. Note that our choice of the micro-averaged F1 score

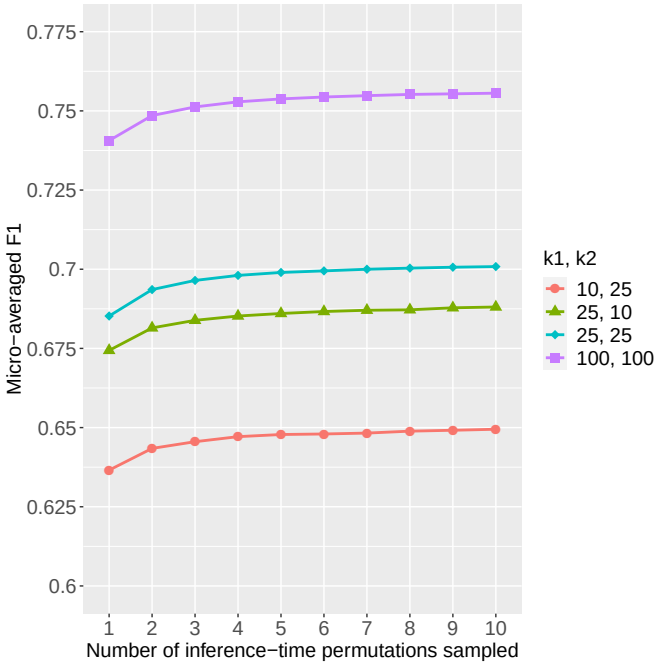


Figure 4.5. Impact of increasing k and the number of inference permutations in the GraphSAGE-LSTM model on the PPI dataset. The GraphSAGE-LSTM model uses a JP neighborhood aggregator with π-SGD and k-ary training. The setting k_1, k_2 refers to the number of permutations sampled in recursions 1 and 2, respectively.

as the metric for the PPI dataset also follows from that paper. Other details are provided in the appendix.

Figure 4.5 shows the impact of increasing the permutations and k at inference time using the PPI dataset. Since there are two aggregation steps, we can try different values of k in different layers. We see that increasing k systematically improves performance. Sampling more permutations also increases performance, but diminishing returns set in relatively quickly. Still, using paired tests – t and Wilcoxon signed rank – we find that test performance with seven sampled permutations versus one permutation is significant with $p < 10^{-3}$ over 12 replicates. It is exciting that the generalization of GraphSAGE can be improved by the straightforward modification of sampling more permutations at inference time.

For the other two citations datasets, we found that all models performed similarly, including naive mean pooling. This points to an easy task, and in retrospect it stands to reason that the topic of a paper can be adequately predicted by computing the average represen-

tations of papers in the neighborhood (papers with citation links). This may not require modeling relationships between neighboring papers. The research community has since come to the conclusion that these citation graphs cannot be used to effectively benchmark among graph models of varying degrees of complexity and sophistication [8]. The results, which are not very insightful, are shown in the appendix.

Arithmetic Tasks. We return to the "fruit fly" arithmetic task. On these tasks, we have some insight into the order of relationships (e.g., variance is 2-ary). Moreover, we made the sequences small so that we can evaluate the performance of **exact** k-ary inference. Our architecture for $\vec{f}_k$ performs the following computations: map each integer to a higher-dimensional space, concatenate each of the hidden vectors, and pass to an MLP. We perform this for all k-sized subsequences, pool the results, and pass the vector through a readout function r. Altogether, this constitutes a k-ary JP model. Note that we are not training with π-SGD.

Table 4.7 shows the results for two levels of complexity in the readout function. For a simple readout r, increasing k improves performance on several tasks, including range, unique sum, and unique count. Increasing k does not improve performance on the sum task (1-ary). On the variance task, which we know to be 2-ary, setting $k = 2$ achieves the best performance. We are slightly surprised to find that setting $k = 3$ hurts performance. This points to a challenge with optimizing the 3-ary model. Inspecting the results when r is an MLP reveals no obvious benefit to increasing k in the preceding JP layer. The most logical explanation is that the optimization problem is more challenging.

While these results somewhat support our theory when r is a simple affine layer, they also point to difficulty in training k-ary models. This observation, and the strong performance observed with π-SGD training on a model that uses the full sequence (Table 4.4), suggest the latter may be a more robust training approach. Simultaneously, exploring the optimization is an interesting avenue for future work.

Finally, we can study the impact of increasing the number of permutations when training with π-SGD. From Table 4.4 we observe a modest benefit when increasing the number of inference-time permutations from 1 to 20.

Table 4.7. k-ary Approximations and the Arithmetic Tasks. Mean (standard deviation) performance, measured in RMSE for the variance task and accuracy (A) for the remaining arithmetic tasks, averaged across 15 runs. We compare k-ary JP models trained *exactly* with different values of k. DeepSets [26], a special case of JP, is shown in `typewriter font`.

Model	k	r	var (RMSE)	sum (A)	range (A)	uniq. sum (A)	uniq. count (A)
MLP	1	Affine	119.05(1.29)	1.00(0.00)	0.04(0.00)	0.07(0.00)	0.36(0.01)
MLP	2	Affine	4.37(0.50)	0.99(0.00)	0.09(0.00)	0.17(0.00)	0.74(0.03)
MLP	3	Affine	8.99(0.99)	0.99(0.00)	0.21(0.00)	0.44(0.02)	0.89(0.04)
MLP	1	MLP	1.95(0.24)	1.00(0.00)	0.97(0.01)	1.00(0.00)	1.00(0.00)
MLP	2	MLP	3.49(0.48)	1.00(0.00)	0.97(0.01)	1.00(0.00)	1.00(0.00)
MLP	3	MLP	6.90(0.47)	0.93(0.02)	0.93(0.02)	1.00(0.00)	1.00(0.00)

4.4.5 Exploring Different $\vec{f}$ Architectures and Canonical Orderings

In this section, we investigate two other proposed schemes in the JP framework. First, we explore CNN and RNN architectures for $\vec{f}$ on graph tasks. Equation 4.7 describes the CNN model. The RNN $\vec{f}$ reinterprets the adjacency tensor as a sequence. In brief, for a vertex v in a graph $\mathcal{G}$, we pass the sequence $(\mathbf{A}_{v,1,:}, \mathbf{A}_{v,2,:}, \ldots, \mathbf{A}_{v,|\mathcal{G}|,:})$ to an LSTM, and the vertex features to an MLP, to obtain a hidden representation for the vertex. All vertex-level representations form another sequence that we pass to another LSTM model. The details are provided in the appendix. Arguably, these methods rely on the vertex and edge features more so than the previous approaches. Since the molecules dataset contains rich feature information, we evaluate these architectures on those.

Arguably, it is more sensible to run an RNN on a connected subgraph. Thus, we run BFS and DFS from a randomly-selected vertex to collect a k-sized *connected* induced subgraph. In other words, we take a k-ary π-SGD approach with poly-canonical orderings (Section 4.1.2).

The results are shown in Table 4.6. Unfortunately, we see that these approaches tend to perform poorly. The only exception is the RNN-DFS, which approaches the performance of the GNNs on the Tox21 dataset, but still appears inferior. We cannot conclude that CNNs and RNNs are systematically worse approaches, but that further exploration is required to understand if and under which circumstances these models could perform well.

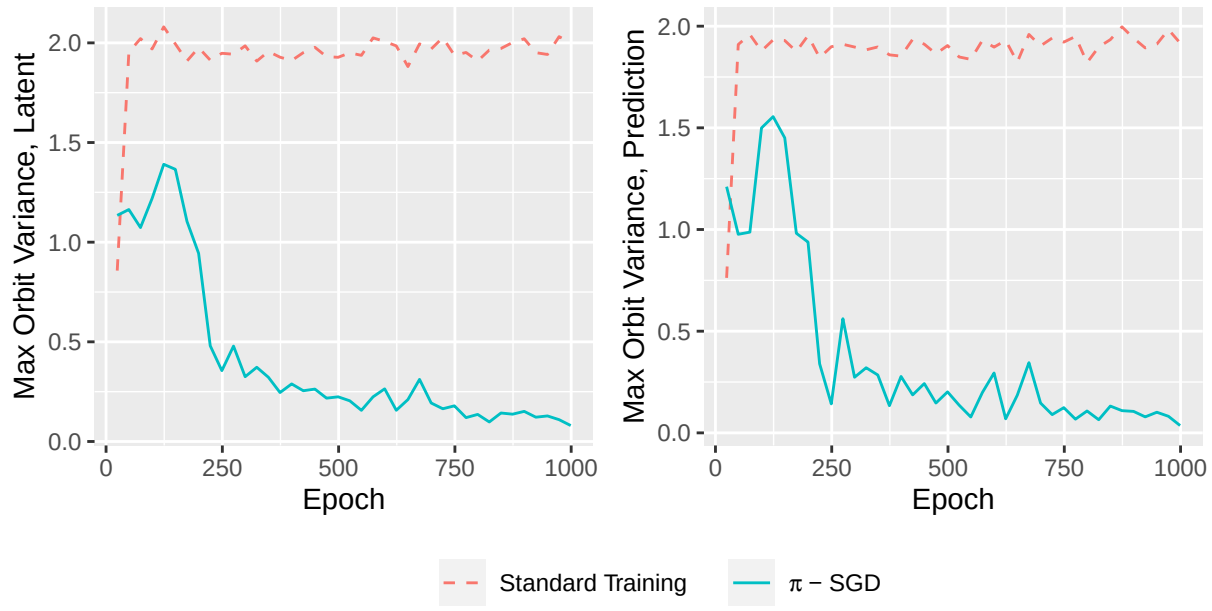


Figure 4.6. π-SGD Promotes Invariance. We show the permutation-sensitivity, measured by the variance over permutations, of RPGIN throughout optimization with π-SGD and "standard" training.

4.4.6 π-SGD Training Learns an Approximately Invariant Model

We have seen that π-SGD training of permutation-sensitive models can lead to strong generalization performance. However, we have not verified that π-SGD training learns an approximately invariant model. Formally, assuming that the output of $\vec{f}$ is scaled to have unit norm,[16] consider whether π-SGD training reduces the quantity

$$\sum_{\pi_1 \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \sum_{\pi_2 \in \mathbb{S}_{|\underline{\mathbf{X}}|}} \left\| \vec{f}(\pi_1 \cdot \underline{\mathbf{x}}; \hat{\mathbf{\Theta}}_f) - \vec{f}(\pi_2 \cdot \underline{\mathbf{x}}; \hat{\mathbf{\Theta}}_f) \right\|_2^2, \tag{4.16}$$

where $\hat{\mathbf{\Theta}}_f$ denotes the estimated parameters from π-SGD training. Note that Equation 4.16 is related to the sum of the diagonals of the empirical variance-covariance matrix. To investigate this, we compare training an RPGIN architecture with π-SGD to standard training on the CSL data. We train to convergence in both cases. Throughout training,

[16]The output of $\vec{f}$ is always a vector in this section. For scalar outputs, it is not sensible to scale to unit norm.

we estimate Equation 4.16 with 100 pairs of sampled permutations on a set of graphs that were not seen during training. Since there are several graphs in the validation dataset, we compute Equation 4.16 for each and report the maximum (the results for the mean are similar). Furthermore, we quantify the permutation-sensitivity of both the *latent graph representation* and the (pre-softmax) graph-wide *prediction*. Figure 4.6 shows these results for one round of training. We repeated this experiment several times and found that the results were similar. The results suggest that training with π-SGD reduces the permutation-sensitivity of the RPGIN function.

4.5 Impact in the Literature

Insights from the Janossy pooling framework have been leveraged in other theoretical and applied works. Of particular note is the idea of appending IDs to improve MPGNNs (i.e., RPGNNs), which has since been applied by numerous authors. Sato, Yamada, and Kashima [187] show that RPGNNs with π-SGD training can approximately learn algorithmic solvers on graph-theoretic problems. Chen, Villar, Chen, *et al.* [78] propose a *local RP* that operates over small egonets of a graph. RPGNNs also inspired a new method that uses Laplacian Eigenmaps to break symmetries in graphs [8]. Related to the expressiveness of GNNs, many authors have used the CSL task as a benchmark [8], [78]–[80]. A second key contribution of ours, highlighting limitations of DeepSets, was built upon in [62]. Indeed, Janossy pooling was suggested as a flexible approach for learning permutation-invariant aggregation functions in graph neural networks by the textbook [132]. Looking at the framework overall, JP was leveraged in [27] to learn set-of-set representations. A last example to mention is that the concept of poly-canonical orderings of graphs was used in [284] to learn an invariant graph generation model.

On the applied front, the ideas of JP have been recognized in works on calcium imaging [285] and diagnostic predictions from Electronic Health Records [286].

5. REGULARIZING TOWARDS INVARIANCE

There are two potential drawbacks of the methods we have discussed so far. First, as a general principle, invariant training can degrade generalization performance if the true function of the data is not invariant. Second, in spite of the theoretical and empirical evidence that π-SGD training is effective, it can also lead to challenges in optimization. We hypothesize that *regularizing a model towards invariance* is a sensible approach to addressing both challenges. First, training any model with a permutation-sensitivity penalty allows the extent of enforced invariance to be *data driven*; if a model trained with strong regularization towards invariance cannot fit the data, it suggests that an invariant model is inappropriate. To trade off between the generalization benefits of invariant models and the corresponding reduction in flexibility, we will select the model trained with the largest regularization strength that does not damage train/validation performance unacceptably. Typically, this model will not be exactly permutation-invariant, but we demonstrate that it is usually *less* variable over permutations of the input. Second, some of the challenges with π-SGD optimization may arise from the variance of $\vec{f}$ over transformations, and it stands to reason that regularization could stabilize π-SGD. Overall, such an approach provides a framework encompassing both (approximately) permutation-invariant models and permutation-sensitive ones.

In this chapter, we propose Birkhoff regularization (BReg), which defines a penalty term that can be added to the optimization objective. This BReg term penalizes the extent to which a function varies along tangent vectors in the direction of permutations. These tangent vectors are created with doubly-stochastic matrices, giving rise to the name of our method; the Birkhoff polytope is the set of doubly-stochastic matrices. In Section 5.1, we elaborate on the aforementioned motivations for introducing regularized training with BReg. In Section 5.2, we introduce the BReg penalty and discuss training details in Section 5.3. Then, we gain insights into BReg via an optimization perspective in Section 5.4 and finally empirically validate the method with experiments.

5.1 Motivating BReg

In this section, we further discuss the motivations for a regularization that penalizes model sensitivity to permutations.

5.1.1 Appropriateness of Invariance

Restricting the function class to invariant models comes at the cost of flexibility. While invariant models can generalize better when the true data distribution is (at least approximately) invariant to some transformation, they incur bias when the assumption is inappropriate. For example, the function $f(\boldsymbol{s}) = f(s_1, s_2, \ldots, s_n) = \max(s_1, s_2)$ for $\boldsymbol{s} \in \mathbb{R}^n$ is difficult to model with a permutation-invariant function. In practice, it is not always clear what *types* and *magnitudes* of invariances are appropriate. This was observed by Li, Hu, Wang, *et al.* [287], who propose a method to search for the *most appropriate invariances* for a task so that the data can be augmented accordingly. In Mouli and Ribeiro [179], the authors propose CGReg, a training scheme that preserves all invariances except those that are inconsistent with the training data. Their experiments demonstrate cases where incorrectly enforced invariances result in failure. Some authors are concerned with learning the *subset* of transformations the model should be invariant to. Benton, Finzi, Izmailov, *et al.* [176] propose to *learn* the correct magnitude of invariance (e.g., a subset $[0, 2\pi)$ of rotation angles) end-to-end using the reparameterization trick [202] and construct synthetic tasks where fully rotation-invariant steerable CNNs [288] will fail. Cohen-Karlik, Ben David, and Globerson [177] point out that the *half-range* function $\mathrm{HR}((x_1, \ldots, x_n)) = \max\{x_1, \ldots, x_{\lfloor n/2 \rfloor}\} - \min\{x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n\}$, is not invariant to *all* permutations [177]. In this section, we are most interested in the former case, where there is uncertainty about whether invariance to permutations is helpful for a task.

This challenge is quite relevant to modeling graph data. Consider time-evolving graphs such as the preferential attachment model with fitness [82]. In this model, new vertices are added to the graph at different time steps. The probability that new vertices form an edge with existing vertices is increasing in their degree and level of *fitness*. Thus, in the observed graph, a vertex with high degree can be either one that entered the graph early

or one with high fitness. Knowledge of the temporal ordering can help to identify whether high-vertex degrees are "fitter" or "earlier". Fitness and preferential attachment have been used to describe many phenomena including citations networks and protein interactions [82], [289]. As another example, the ordering of vertices in representing brain graphs can depend on brain atlases [113], [200], [290]. If nearby brain regions are close together in the indexing, then the ordering has scientific meaning. Using a model that is permutation-invariant could underperform when there is meaning in the ordering.

It may appear that our present argument – that ordering may carry useful information in some tasks – contradicts our previous account of the importance of permutation-invariance in graph and set data. Previously, we saw that *some ordering must be assigned* in order to encode a graph or set on the computer (e.g., Figure 3.1). There is not a unique ordering, and certainly when one is chosen arbitrarily, a permutation-invariant model is strongly recommended. However, part of our contribution is to present an argument in the relevant literature[1] that the ordering present in the data *may* be meaningful, and we will propose a new method on the basis of this argument. In other words, while an arbitrary encoding is *sometimes used* to encode these data, it does not imply that there is *never* a meaningful ordering. Moreover, we emphasize that one cannot always *ask an expert* whether the ordering is relevant. Often, datasets arise from a curation process that may involve several scholars, possibly working independently, and sometimes by aggregating different data sources [291]. A data-driven approach would make an important contribution to the literature.

To address this challenge, one might consider *learning the best ordering,* and we discussed some methods above. However, if order matters, standard training of a permutation-sensitive model may be a better solution. In that case, the model can adapt to optimally use the information in the ordering, supervised by the target variables. Moreover, many models that learn an ordering must still hand-code task knowledge into the architecture [5]. We believe that a permutation-sensitivity penalty is more adaptive.

Remark 9 *In this section, it is arguably more appropriate to say "graph-like" and "set-like" data than it is to say graphs and sets. Technically, stating that the order of elements may*

[1]The importance of ordering is not mainstream in the literature on neural networks for graphs and sets, but we do not argue that this is the case in other disciplines.

carry meaning contradicts the definition of a set. Nonetheless, our goal is to train models whose inputs are encoded as sets or graphs, as discussed in Chapter 3, with a regularization scheme designed to let the data decide whether the order matters through the choice of a hyperparameter.

5.1.2 π-SGD Variance

Various scholars have pointed out that training with π-SGD can prove challenging. Pabbaraju and Jain [292] construct synthetic tasks where π-SGD performs poorly and show that it leads to models with large orbit variance (Equation 4.16) when the input is a long sequence. Cohen-Karlik, Ben David, and Globerson [177] report that π-SGD failed to converge in their experiments. We speculate that these observations may arise when π-SGD training alone is insufficient for variance reduction. Figure 5.1 provides a glimpse into the challenges by plotting training loss against epoch. The left plot corresponds to training an RPGNN to solve the CSL task on very large graphs. We see that there is a large spike in the loss after apparent convergence, which creates challenges in model training and selection. On the right, we compare standard training to π-SGD in predicting the variance of fixed-length sequences. In this case, $\vec{f}$ is an MLP. Models trained with π-SGD stabilize at a large loss value far from zero. Since the target is indeed permutation-invariant, and the function is within the class of functions that JP can express, this points to an issue of variance. Adding a permutation-sensitivity penalty could mitigate these difficulties.

5.1.3 Additional Related Work

A few methods have been proposed related to regularization and invariances. SIRE [177] regularizes recurrent models towards permutation-invariance for set-like inputs specifically. Yang, Wang, and Heinze-Deml [293] propose defenses against worst-case transformations to improve model robustness, but their method can only enforce invariance to a subset of the group of transformations, such as small rotations. This is not suitable for permutation-invariance (invariance to the action of the symmetric group). In contrast, our approach is applicable to both graph and set data, and is compatible with a wide array of different

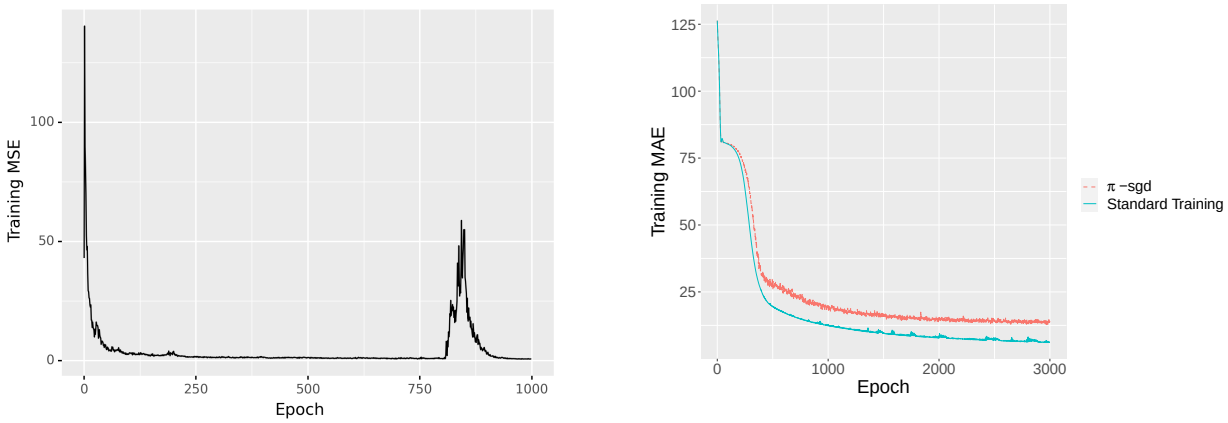


Figure 5.1. Challenges of π-SGD Training. We study loss as a function of epoch. Left: training RPGNN on a CSL task (Section 4.4.1) with large graphs. Right: training an MLP to predict the variance of the input.

models. In principle, BReg is equally applicable to sequence models like Transformer and Recurrent Neural Networks, as well as permutation-sensitive graph models such as RPGNNs. To the best of our knowledge, we are the first to leverage the Birkhoff-von Neumann theorem to propose a regularizer for permutation-invariance for set- and graph-like data.

To train models with the BReg penalty, we draw inspiration from [179], where the authors propose to encourage invariance to any transformations that do not contradict the data. This allows models to inherit the generalization (and extrapolation) benefits of invariant training but limits the bias. Similarly, we will let the permutation-invariant regularization strength grow until the performance degrades unacceptably. If even small regularization results in a significant deterioration, we can use the original model. In contrast to [179], this work delves deeply into graph- and set-like data, providing a host of specific strategies, as well as deeper insights into that literature.

5.2 Birkhoff Regularization Penalty

To promote permutation-invariance, we will add a permutation-sensitivity penalty to the optimization objective, analogously to the parameter norm penalties in LASSO and ridge regression [85], [244]. In particular, we will define a nonnegative function $R : \mathcal{X} \times \mathcal{F} \to \mathbb{R}_{\geq 0}$ where $\mathcal{X}$ is a input space of set- or graph-like data and $\mathcal{F} = \{\vec{f}(\cdot; \boldsymbol{\Theta})\}_{\boldsymbol{\Theta}}$ is the function

Table 5.1. Variants and Examples of BReg. (Left): An overview of regularization strategies existing under the BReg framework. We can (1) compute **tangent vectors** of model components or inputs; (2) **measure (and penalize) variability** in the predicted or latent quantities; (3) use TangentProp (TP) [15] or Finite Differences (FD) as the **style** of regularization. To define the regularization, we make a choice from each row. (Right): Examples. In the first row, we compute tangent vectors in input space, measure variability in the predicted quantity, and use a TangentProp penalty. As shown, more details on this case can be found in Definition 18 and Algorithm 3.

BReg Variants	
Tangent vector of:	Input, Model
Measure variation in:	Prediction, Latent
Penalty term style:	TP, FD

Examples of BReg Described At Length			
Tan Vec	Measure Var	Style	Reference
Input	Pred	TP	Def 18, Alg 3
Input	Pred	FD	Eq 5.2
Model	Pred	FD	Fig 5.4

class to search over.[2] We sum this penalty with the task loss in the objective function, and control its importance with a strength parameter $\lambda \geq 0$. Note that the task loss component could either be that of a Janossy model estimated with π-SGD (Equation 4.10) or the typical empirical risk for any permutation-sensitive model.

To compute the BReg penalty, we approximate the extent to which a function varies in the *direction* of permutations. Broadly speaking, we achieve this with an approach like TangentProp (Equation 2.4) or by finite differences. Tangent vectors are computed via multiplications with a doubly-stochastic matrix, which we will make precise below. In many cases, it is helpful to compute tangent vectors along directions in the input space (perturbations to the data) or the model (perturbations to model components). For example, we will see that regularization can be applied to the positional encodings of Transformer models. All told, BReg is highly flexible and can be implemented in many ways to better suit the model architecture, modeling goal, or task. Table 5.1 provides an overview and the terminology that we will use.

In the next section, we will simultaneously introduce the mathematical tools and one type of BReg to make the idea concrete. Then, we turn to a general discussion of the different forms of BReg, their individual merits, and some specific architectures. In this chapter, the

[2] As in Remark 5, we can take $\mathcal{X}$ to be a space of latent representations from prior layers.

norm $\|\cdot\|$ denotes the ℓ_2 norm if the input is a vector, the Frobenius norm for matrices, and its natural generalization for higher-order arrays (i.e, tensors). This is the norm used in the definition of TangentProp [15].

5.2.1 TangentProp Perspective and Mathematical Tools for BReg

We saw in Equation 2.4 that TangentProp (TP) penalizes the extent to which a model varies along tangent vectors of some transformation of the input, thereby training a model towards invariance. We will use this to motivate BReg.

A Closer Look at TangentProp. Simard, Victorri, LeCun, *et al.* [15] argue that, while invariances could be learned directly from infinite data and computation, this may not happen in practice. Accordingly, they propose TP, a regularization towards invariance. Relevant to our goal, the authors specifically mention that data augmentation – which describes π-SGD with one training-time sample – is an inefficient strategy for training towards invariance. We hypothesize that TP could be more efficient at reducing variance over permutations. Moreover, summing over multiple tangent vectors in Equation 2.4 can enforce invariance to linear combinations of transformations. In our case, this implies that the regularization can promote invariance in the span of multiple permutations, which could more efficiently explore the space of permutations than sampling alone. As another justification, TP-BReg can be seen as an efficient proxy for a minimization problem over the Birkhoff polytope. In brief, we will construct tangent vectors via multiplication with doubly-stochastic matrices, which also arise when relaxing a discrete optimization problem over permutations to their convex hull. We will explore this more in Section 5.4.

To the best of our knowledge, this is the first work to bridge the gap between TP and regularizing towards permutation-invariance in graphs/sets and to derive connections between regularization and set models. Like TP, training with the BReg penalty will not result in a fully permutation-invariant model, but can *reduce* the permutation sensitivity.

TangentProp for Permutation-Invariance and the Birkhoff Polytope. To use TP regularization, we must approximate tangent vectors. We begin by reviewing the scheme for enforcing rotation-invariance when modeling images. Let $\mathbf{I}_0$ denote an original image and $\mathbf{I}_\varepsilon$ denote the image rotated by a small angle $\varepsilon > 0$. Then, the tangent vector approximation

is $(\mathbf{I}_\varepsilon - \mathbf{I}_0)/\varepsilon$. Multiplying the Jacobian of the model with respect to the input by this tangent vector and taking the norm gives the penalty in Equation 2.4.

To introduce the idea for regularizing permutation sensitivity, let $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$ denote a sequence of n real numbers, and let $\boldsymbol{P}$ denote any $n \times n$ permutation matrix except the identity $\boldsymbol{I}_n$ (see Definition 6). Recall that $\boldsymbol{P}\boldsymbol{x}$ denotes a permutation of $\boldsymbol{x}$. The relevant analogue of rotating an image is a step along the line towards a permutation given by $s(c) = (1-c)\boldsymbol{I}\boldsymbol{x} + c\boldsymbol{P}\boldsymbol{x} = ((1-c)\boldsymbol{I} + c\boldsymbol{P})\boldsymbol{x}$, $c \in [0,1]$. Thus, a tangent vector pointing from $\boldsymbol{x}$ to its permutation $\boldsymbol{P}\boldsymbol{x}$ will be of the form

$$\frac{((1-c')\boldsymbol{I} + c'\boldsymbol{P})\boldsymbol{x} - \boldsymbol{x}}{c'} = \frac{\boldsymbol{D}\boldsymbol{x} - \boldsymbol{x}}{c'}, \tag{5.1}$$

for some $c' \in [0,1]$, where $\boldsymbol{D} := ((1-c')\boldsymbol{I} + c'\boldsymbol{P})$ is a *convex combination* of the identity and permutation matrices. We will see that $\boldsymbol{D}$ is a doubly-stochastic (DSt)[3] matrix, which forms the backbone of our approach.

Definition 16 *For any $n \in \mathbb{Z}_{\geq 1}$, a matrix $\boldsymbol{D} \in [0,1]^{n\times n}$ is* doubly-stochastic *if $\sum_{i=1}^n D_{i,j} = \sum_{j=1}^n D_{i,j} = 1$ for all $j \in \{1,\dots,n\}$ and all $i \in \{1,\dots,n\}$ [294].*

Observe that permutation matrices, including the identity matrix, are doubly-stochastic. Next, we will characterize the set of all DSt matrices.

Definition 17 *Let $\mathcal{A} = \{\boldsymbol{A}^{(1)},\dots,\boldsymbol{A}^{(m)}\} \subset \mathbb{R}^{n\times p}$ be a finite set of matrices. Then the convex hull of $\mathcal{A}$ is given by $\operatorname{conv}(\mathcal{A}) = \{\sum_i c^{(i)}\boldsymbol{A}^{(i)} : c^{(i)} \geq 0 \forall i,\ \text{and}\ \sum_i c^{(i)} = 1\}$.*

The following result shows formally that DSt matrices arise as convex combinations of permutations matrices, a fact that is often utilized in convex relaxations of minimization problems over permutations [200], [295]–[297].

Theorem 4 ***(Birkhoff-von Neumann Theorem** [298], [299]) For any $n \in \mathbb{Z}_{\geq 1}$, let $\mathcal{P}_n$ denote the set of all $n \times n$ permutation matrices. Then, $\operatorname{conv}(\mathcal{P}_n) = \{\boldsymbol{D} \in [0,1]^{n\times n} :$ $\boldsymbol{D}$ is doubly-stochastic$\}$.*

[3]We refrain from abbreviating doubly-stochastic as DS to avoid confusion with DeepSets.

$\mathcal{B}_n := \text{conv}(\mathcal{P}_n)$ is called the *Birkhoff polytope*. So, tangent vectors, which in this case arise as convex combinations of permutation matrices, are defined by DSt matrices. It can be shown that the Birkhoff polytope is overparameterized in the sense that fewer than $(n-1)^2+1$ terms (i.e., permutation matrices) in the convex combination are needed to represent any doubly-stochastic matrix [298]. This suggests that invariance along the directions of *all* permutations can be enforced with *far fewer* than $n!$ vectors. In other words, these $O(n^2)$ permutations suffice to constrain the model to achieve the desired invariance. However, in most cases, it will be too computationally expensive to pass $O(n^2)$ permutations of the input through the model at each optimization step.

The Permutohedron. Before generalizing to any set or graph, let us introduce the *permutohedron,* which facilitates visualization. Letting $\mathcal{P}_n$ denote the set of $n \times n$ permutation matrices, the set $\mathcal{PH}_n := \text{conv}(\{\boldsymbol{P}(1, 2, \ldots, n)^\mathrm{T} : \boldsymbol{P} \in \mathcal{P}_n\})$[4] is called the *permutohedron* [295], [300].[5] Examples are shown in Figure 5.2. We will view $(1, \ldots, n)$ as the indices of a set or graph input. Permutations of the indices correspond to vertices of the permutohedron, denoted $\mathcal{V}(\mathcal{PH}_n)$, and multiplication by a general DSt matrix maps to any point on $\mathcal{PH}_n$. In fact, the permutohedron is a projection of the Birkhoff polytope from $\mathbb{R}^{n \times n}$ to $\mathbb{R}^n$ [295]. These facts will facilitate our construction of tangent vectors in the direction of permutations.

Going beyond permutations of $(1, \ldots, n)^\mathrm{T}$, the *generalized permutohedron* is the convex hull of permutations of *any* sequence of real numbers. Therefore, the tangent vector in Equation 5.1 lies on the generalized permutohedron of $\boldsymbol{x}$, which we will denote $\mathcal{PH}(\boldsymbol{x})$. This is useful when visualizing permutations of sequences and is especially helpful for the discussion in Section 5.4.

Generalizing Tangent Vectors. Equation 5.1 hints that we can define tangent vectors for any graph or set input with doubly-stochastic matrices. Now, just as an angle parameterizes a rotation matrix, we parameterize doubly-stochastic matrices with a single hyperparameter $\varepsilon > 0$. For now, we write $\boldsymbol{D}(\varepsilon, n)$ as any DSt matrix such that $\|\boldsymbol{D}(\varepsilon, n) - \boldsymbol{I}_n\| = \varepsilon$,[6] for

[4] Following convention, vectors are assumed to be column vectors, hence we transpose here.
[5] The permutohedron is also spelled permutahedron, and some authors define it with 0-indexing: $(0, \ldots, n-1)$.
[6] Recall that $\|\cdot\|$ denotes the ℓ_2, Frobenius norm, or its natural generalization to tensors in this chapter, which is also the norm used in TangentProp [15].

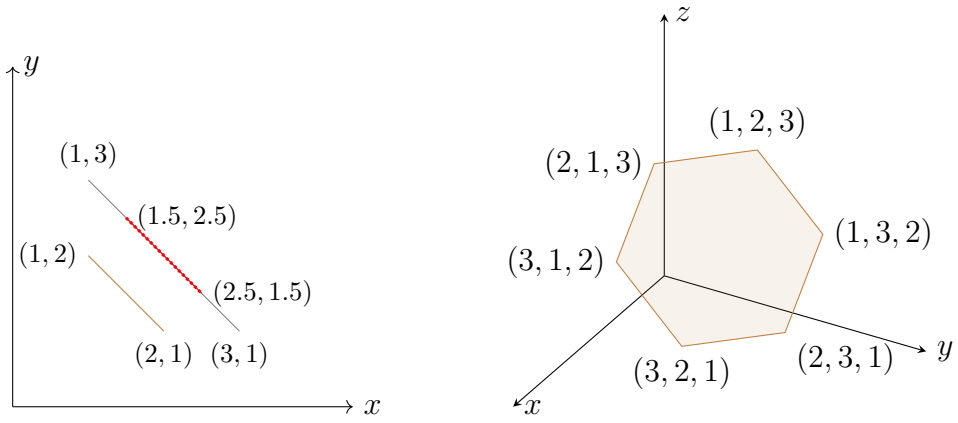


Figure 5.2. Examples of Permutohedra. For $n \in \mathbb{Z}_{\geq 1}$, the *standard permutohedron* $\mathcal{PH}_n$ is defined as the convex hull of all permutations of $(1, 2, \ldots, n)$. For any $\boldsymbol{x} \in \mathbb{R}^n$, the *generalized permutohedron* $\mathcal{PH}(\boldsymbol{x})$ is the convex hull of permutations of $\boldsymbol{x}$. Left: examples for $n = 2$. The *standard permutohedron* is the line segment connecting $(1, 2)$ to $(2, 1)$. Two generalized permutohedra, $\mathcal{PH}\big((1, 3)\big)$ and $\mathcal{PH}\big((1.5, 2.5)\big)$ are also shown. Notice that these permutohedra overlap since $1 + 3 = 1.5 + 2.5 = 4$. Right: the standard permutohedron $\mathcal{PH}_3$.

any $\varepsilon > 0$ and $n \in \mathbb{Z}_{n\geq 2}$.[7] The hyperparameter ε controls the distance between the identity matrix and a doubly-stochastic matrix inside the Birkhoff polytope, but is not equivalent to the length of the tangent vector. In particular, $\|\boldsymbol{D}(\varepsilon, n)\boldsymbol{x} - \boldsymbol{x}\|$ depends on the value of $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$. For instance, if $\boldsymbol{x} = (1, \ldots, 1)^{\mathrm{T}}$, then $\|\boldsymbol{D}\boldsymbol{x} - \boldsymbol{x}\| = \|\boldsymbol{0}\| = 0$, for all DSt $\boldsymbol{D}$, by definition. Analogously, the angle between vectors does not depend on their magnitudes. Details on creating $\boldsymbol{D}$ and setting the hyperparameter ε are provided in subsequent sections.

The next definition formalizes the tangent vectors, and an example is shown in Figure 5.3. Effectively, we already know how to permute a set or graph with permutation matrices $\boldsymbol{P}$, and we simply replace $\boldsymbol{P}$ with any DSt matrix $\boldsymbol{D}$.

Definition 18 ***(Tangent Vectors for BReg)*** *For any n, let $\mathscr{S}_{n,d_s} = \mathbb{R}^{n\times d_s}$ be the set of length-n sequences of d_s-dimensional vectors, $\varepsilon \in \mathbb{R}_{>0}$, and $\boldsymbol{D}(\varepsilon, n)$ denote any $n \times n$*

[7] We suppose $n \geq 2$ since the case $n = 1$ is not of practical interest and can result in uninteresting corner cases.

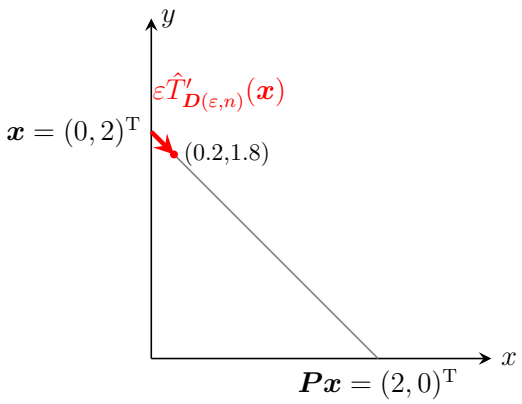


Figure 5.3. Example Tangent Vector for BReg. We visualize $\hat{T}'_{\boldsymbol{D}(\varepsilon,n)}$ for a sequence of scalars. The input sequence is $\boldsymbol{x} = (0,2)^{\mathrm{T}} \in \mathbb{R}^{2\times 1}$ and we let $\boldsymbol{P}$ be the only $n \times n$ non-identity permutation matrix for $n = 2$. If we let $\varepsilon = 0.2$, then the doubly-stochastic matrix $\boldsymbol{D} = \begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{pmatrix}$ satisfies $\|\boldsymbol{D} - \boldsymbol{I}_2\| = \varepsilon$ where $\boldsymbol{I}_2$ is the identity matrix. The product $T_{\boldsymbol{D}}(\boldsymbol{x}) = \boldsymbol{D}\boldsymbol{x} = (0.2, 1.8)^{\mathrm{T}}$ sends $(0,2)^{\mathrm{T}}$ towards the vector $(2,0)^{\mathrm{T}}$. The tangent vector is $\varepsilon\hat{T}'_{\boldsymbol{D}}(\boldsymbol{x}) = (0.2, 1.8)^{\mathrm{T}} - (0,2)^{\mathrm{T}} = (0.2, -0.2)^{\mathrm{T}}$.

doubly-stochastic matrix such that $\|\boldsymbol{D}(\varepsilon, n) - \boldsymbol{I}_n\| = \varepsilon$. *We can send length-n sequences in the direction of permutations with a function* $T_{\boldsymbol{D}(\varepsilon,n)} : \mathscr{S}_{n,d_s} \to \mathscr{S}_{n,d_s}$ *defined by*

$$T_{\boldsymbol{D}(\varepsilon,n)}(\boldsymbol{S}) = \boldsymbol{D}(\varepsilon, n)\boldsymbol{S}, \forall \boldsymbol{S} \in \mathscr{S}_{n,d_s}.$$

For graphs, we can define a similar operation. First, without edge features, we only have the adjacency matrix, and we can define

$$T_{\boldsymbol{D}(\varepsilon,n)}(\mathcal{G}) = T_{\boldsymbol{D}(\varepsilon,n)}(\boldsymbol{A}, \boldsymbol{F}) = (\boldsymbol{D}(\varepsilon, n)\boldsymbol{A}\boldsymbol{D}(\varepsilon, n)^T, \boldsymbol{D}\boldsymbol{F})$$

for all $\mathcal{G} = (\boldsymbol{A}, \boldsymbol{F}) \in \mathscr{G}_{n,d_v,d_e}$, $d_v \geq 1$ *and* $d_e = 1$. *When edge features are present* ($d_e > 1$), *we have an adjacency tensor* $\mathbf{A}$, *and we compute* $\boldsymbol{D}(\varepsilon, n)\mathbf{A}_{:,:,j}\boldsymbol{D}(\varepsilon, n)^T$ *for all* $j = 1, \ldots, d_e$. *We will call* $T_{\boldsymbol{D}(\varepsilon,n)}$ *a* soft permutation *and have deliberately overloaded it so that we can generalize to graph- or set-like inputs. Letting* $\underline{\boldsymbol{x}}$ *denote an arbitrary set or graph and* vec *the graph or set vectorization operation defined in the appendix, write*

$$\hat{T}'_{\boldsymbol{D}(\varepsilon,n)}(\underline{\boldsymbol{x}}) = \frac{\text{vec}(T_{\boldsymbol{D}(\varepsilon,n)}(\underline{\boldsymbol{x}})) - \text{vec}(\underline{\boldsymbol{x}})}{\varepsilon},$$

as the tangent vector approximation of a permutation transformation associated with doubly-stochastic matrix $\boldsymbol{D}$.

Remark 10 *Note that this definition applies to modeling variable-sized inputs for any* $n \geq 2$. *Our schemes for creating* $\boldsymbol{D}(\varepsilon, n)$ *are general and have similar geometric interpretations regardless of* n. *Moreover, we have used* vec *only for the sake of being mathematically precise. In practice, we can make use of broadcasting in computational libraries such as PyTorch [301], and leave the inputs as matrices/tensors.*

Notice that $T_{\boldsymbol{I}}((1, \ldots, n)^{\mathrm{T}}) = (1, \ldots, n)^{\mathrm{T}}$, so $T_{\boldsymbol{I}}$ leaves the indices unchanged. For permutation matrices $\boldsymbol{P}$, $T_{\boldsymbol{P}}((1, 2, \ldots, n)^{\mathrm{T}})$ affects a permutation, or a mapping of indices to another vertex on the permutohedron. In general, $T_{\boldsymbol{D}}$ performs the action of the symmetric group in the special case that $\boldsymbol{D}$ is a permutation matrix. An example of the TP-BReg penalty is shown in Algorithm 3 (see also Table 5.1).

Algorithm 3 Computing the TP-BReg Penalty

Input 1: Either sequence $\boldsymbol{S}$ or graph $\mathcal{G} = (\boldsymbol{A}, \boldsymbol{F})$
2: Hyperparameters $\varepsilon > 0$, regularization strength $\lambda > 0$
3: Model $\vec{f}$

$n \leftarrow$ size of input
Create doubly-stochastic $\boldsymbol{D}$ s.t. $\|\boldsymbol{D} - \boldsymbol{I}_n\| = \varepsilon$ ▷ See Section 5.2.3
if input is a sequence $\boldsymbol{S}$ **then**
    $\boldsymbol{S}_{\text{perturbed}} \leftarrow \boldsymbol{D}\boldsymbol{S}$ ▷ Perturb input, i.e., compute $T_{\boldsymbol{D}(\varepsilon,n)}$
    $\boldsymbol{v} \leftarrow \frac{1}{\varepsilon}\text{vec}(\boldsymbol{S}_{\text{perturbed}} - \boldsymbol{S})$ ▷ Compute tangent vector, i.e., $\hat{T}'_{\boldsymbol{D}(\varepsilon,n)}$
end if
if input is a graph $(\boldsymbol{A}, \boldsymbol{F})$ **then**
    $\boldsymbol{A}_{\text{per}} \leftarrow \boldsymbol{D}\boldsymbol{A}\boldsymbol{D}^{\text{T}}$ ▷ Perturb adjacency matrix, i.e., part of $T_{\boldsymbol{D}(\varepsilon,n)}$
    $\boldsymbol{F}_{\text{per}} \leftarrow \boldsymbol{D}\boldsymbol{F}$ ▷ Perturb vertex features, i.e., part of $T_{\boldsymbol{D}(\varepsilon,n)}$
    $\boldsymbol{v} \leftarrow \frac{1}{\varepsilon}\Big(\text{vec}(\boldsymbol{A}_{\text{per}}, \boldsymbol{F}_{\text{per}}) - \text{vec}(\boldsymbol{A}, \boldsymbol{F})\Big)$ ▷ Compute tangent vector $\hat{T}'_{\boldsymbol{D}(\varepsilon,n)}$
end if
$\boldsymbol{J} \leftarrow$ Compute Jacobian of $\vec{f}$ w.r.t (vectorized) input ▷ See Section 5.2.4
Return $\lambda\|\boldsymbol{J}\boldsymbol{v}\|^2$

Computing the TP-BReg penalty $R_{\text{TP}}(\underline{\mathbf{x}}, \vec{f}_{\boldsymbol{\Theta}})$, for the case that we (1) compute tangent vectors in the input space, (2) measure variation in the predicted quantity, and (3) use TP-style penalization (see Table 5.1). We have made a few simplifying assumptions for clarity. We can sum over multiple tangent vectors (see Equation 2.4) with a simple modification. When edge features are present, we compute $\boldsymbol{D}\boldsymbol{A}_{:,:,j}\boldsymbol{D}^{\text{T}}$ for all $j = 1, \ldots, d_e$. The vec function to vectorize inputs is used for mathematical clarity, but it is not required in practice (see Remark 10).

5.2.2 General BReg

Now we introduce several varieties of BReg, providing detail to Table 5.1. The following are different categories of BReg strategies, and the schemes we choose from each category can be used in any combination.

Space of Tangent Vector

We are not restricted to defining transformations $T_{\boldsymbol{D}}$ over the input. Rather, we can penalize sensitivity to transformations $T_{\boldsymbol{D}}$ applied to *parameters of the model*. As an example, consider the Transformer model [236] for sequences. Aside from a *positional encoding* layer, Transformer uses permutation-equivariant layers [60]. When used with the trainable

positional encoding of [302], which are permutation-equivariant for some values of the parameters, Transformers form a family of models on a spectrum of permutation sensitivity. Precisely, the learnable positional encoding layer is defined by $\vec{f}_1(\boldsymbol{S};\boldsymbol{\xi}) = \boldsymbol{S} + \boldsymbol{\xi}$, where $\boldsymbol{\xi} \in \mathbb{R}^{n \times d_s}$ is a matrix of trainable parameters and $\boldsymbol{S}$ is a sequence input. In words, we associate each index $i \in \{1, \ldots, n\}$ of the sequence with a vector of parameters $\boldsymbol{\xi}_{i,:}$ and add it to the corresponding vector $\boldsymbol{S}_{i,:}$. This helps to leverage information in the ordering of elements in a sequence. Yet, if the columns of $\boldsymbol{\xi}$ are constant, the layer is permutation-equivariant. Thus, to penalize permutation sensitivity, we can compute transformations $T_{\boldsymbol{D}(\varepsilon,n)}(\boldsymbol{\xi}) = \boldsymbol{D}\boldsymbol{\xi}$, tangent vectors $\hat{T}'_{\boldsymbol{D}(\varepsilon,n)}(\boldsymbol{\xi}) = \frac{1}{\varepsilon}\Big(\text{vec}(T_{\boldsymbol{D}(\varepsilon,n)}(\boldsymbol{\xi})) - \text{vec}(\boldsymbol{\xi})\Big)$, and use a TangentProp approach. In this way, we can simultaneously learn the correct amount of permutation invariance, in an interpretable fashion, and take advantage of the strong expressive power of Transformers. An example is shown in Figure 5.4, which also uses the Finite Differences BReg penalty described below.

Measuring Variation in the Latent or Prediction

Our discussion in the previous section suggested regularizing the permutation-sensitivity of the entire neural network $\vec{f} = \vec{f}_L \circ \vec{f}_{L-1} \circ \cdots \circ \vec{f}_1$. However, another common strategy in learning invariant models is to enforce the latent vector returned by the penultimate layer $\vec{f}_{L-1} \circ \cdots \circ \vec{f}_1$ to be invariant to permutations of the input [9], [149]. One reason for doing so is related to transfer learning; given a powerful invariant latent representation, we would only need to fine tune the last layer to transfer to a new task [24]. Thus, we may want to penalize "permutation sensitivity in the latent". The approach is similar penalizing permutation-sensitivity in the prediction. Tangent vectors are computed as before, but the Jacobian matrix is now defined as $(\partial \boldsymbol{h}_i / \partial \underline{\mathbf{x}}_j)_{ij}$ where $\boldsymbol{h} = \vec{f}_{L-1} \circ \cdots \circ \vec{f}_1(\underline{\mathbf{x}})$.

Arbitrary Scaling We encounter a difficulty when measuring variation in the latent; the penalty can be reduced by changes in model parameters that do not achieve the desired effect. For example, if we redefine $\vec{f}_L \circ \vec{f}_{L-1} \circ \vec{f}_{L-2} \circ \cdots \circ \vec{f}_1$ to $\vec{g}_L \circ \vec{g}_{L-1} \circ \vec{f}_{L-2} \cdots \circ \vec{f}_1$ where $\vec{g}_{L-1}(\boldsymbol{h}) = \frac{1}{\alpha}\vec{f}_{L-1}(\boldsymbol{h})$ and $\vec{g}_L(\boldsymbol{h}) = \vec{f}_L(\alpha \boldsymbol{h})$, we can reduce the penalty by making α large, since the penalty is computed on the output of layer $L-1$. To prevent this, we

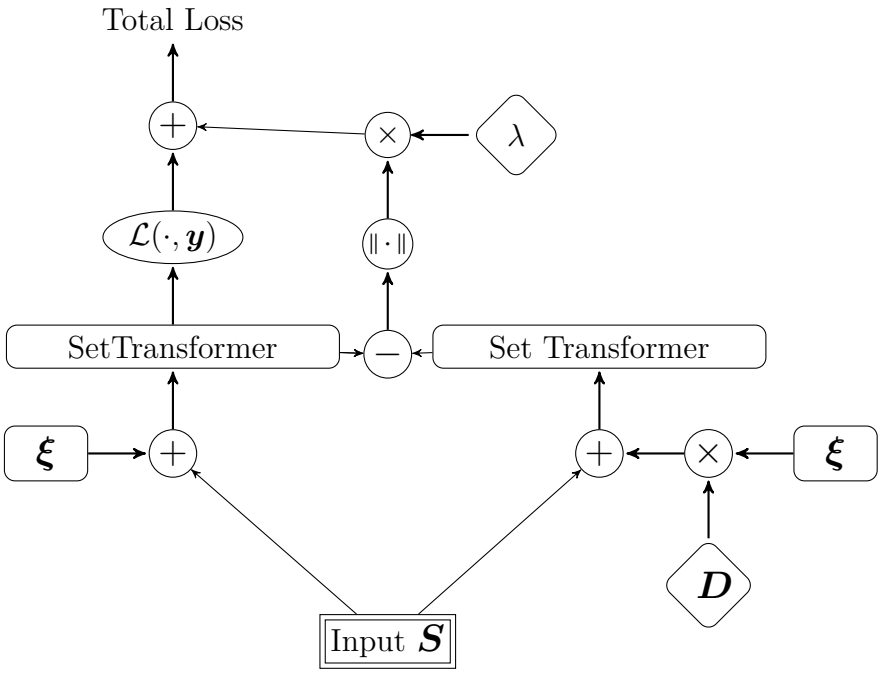


Figure 5.4. An example regularizing a permutation-sensitive Transformer-like model with BReg. The flavor of regularization, in the terms of Table 5.1, are: (1) compute a tangent vector on model parameters, (2) measure variation in the predicted quantity, and (3) use the Finite Differences style of penalty. The left chain describes the typical model architecture. A learnable positional encoding $\boldsymbol{\xi}$ is added to the input $\boldsymbol{S}$ and passed to SetTransformer. On the right, we see regularization computations. We multiply the positional encoding matrix with a doubly-stochastic matrix, created with a scheme from Section 5.2.3, and pass the result to SetTransformer. The norm of the difference is multiplied with a regularization strength and added to the task loss. Rectangles with rounded edges indicate operations with trainable parameters, diamonds indicate hyperparameters, and ellipses are mathematical operations.

propose to add random noise to the latent value. That is, if $\boldsymbol{h}^{(L-1)}$ is the hidden vector of the penultimate layer, then in the last layer, we compute

$$\vec{f}_L(\boldsymbol{h}^{(L-1)} + \boldsymbol{\zeta}),$$

where $\boldsymbol{\zeta} \sim N(0, \sigma^2 \boldsymbol{I})$ is of the same dimension as $\boldsymbol{h}^{(L-1)}$. When noise is added, the optimization should pressure the previous layers $\vec{f}_{L-1} \circ \cdots \circ \vec{f}_1$ to output values with larger norm, outside of the noise range. Otherwise, all the signal is lost in the noise. We do not have a formal proof of this but demonstrate the phenomenon empirically in the experiments. The

benefit of this strategy is that it only adds zero-mean noise to the penalty. In contrast, a naive solution such as scaling the latent vector to have a constant norm could interfere with the BReg penalty, which is designed to measure the norm difference in the latent values.

Finite Differences

As discussed, BReg involves computing a TP-style penalty. In practice, we find that TP-BReg is expensive in both time and memory. Backpropagating through the Jacobian requires two passes through the network in frameworks that use reverse-mode differentiation like PyTorch (see Section 5.2.4). An alternative, but similarly motivated approach is to directly use Finite Differences (FD) of the form

$$R_{\text{FD}}\left(\underline{\mathbf{x}}, \vec{f}\right) = \frac{\| \vec{f}\left(\underline{\mathbf{x}}\right) - \vec{f}\left(T_{\boldsymbol{D}(\varepsilon,\underline{\mathbf{X}})}(\underline{\mathbf{x}})\right)\|}{\varepsilon} = \frac{\| \vec{f}\left(T_{\boldsymbol{I}_{|\underline{\mathbf{X}}|}}(\underline{\mathbf{x}})\right) - \vec{f}\left(T_{\boldsymbol{D}(\varepsilon,|\underline{\mathbf{X}}|)}(\underline{\mathbf{x}})\right)\|}{\varepsilon}, \tag{5.2}$$

for any $\underline{\mathbf{x}} \in \mathcal{X}$ and $\vec{f} \in \mathcal{F}$, where $\mathcal{F}$ is parameterized by trainable parameters. This approach can be extended to measuring variation in the latent and tangent vectors of model components. An example is shown with Transformers in Figure 5.4.

5.2.3 Choice of Doubly-Stochastic Matrices

To compute meaningful tangent vectors, we have supposed that doubly stochastic matrices are distance ε from $\boldsymbol{I}_n$. There are two approaches we can take to ensure this. The first, which we use in our experiments, involves specifying ε and computing an appropriate $\boldsymbol{D}$. An alternative approach which could provide more flexibility is to sample a DSt matrix and then compute the appropriate ε. We can mix these approaches to create multiple tangent vectors at each epoch, and sum over the penalties corresponding to each (see Equation 2.4).

Prespecified ε

We define algorithms to construct DSt matrices $\boldsymbol{D}$ such that $\|\boldsymbol{D}(\varepsilon, n) - \boldsymbol{I}_n\| = \varepsilon$. These have a dual interpretation in geometry and our regularization. The geometric interpretations are independent of the choice of $n \in \mathbb{Z}_{\geq 2}$ and regularization can be applied to problems with

variable-length inputs. The strategies below leverage the fact that the convex combination of DSt matrices is DSt.

Center Steps. For all $n \in \mathbb{Z}_{\geq 2}$, the centroid of the polytope $\mathcal{B}_n$ is $\boldsymbol{C}_n := \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^{\mathrm{T}}$, where $\mathbf{1}_n$ is a vector containing n ones [298]. We define $\boldsymbol{D}_{\text{center}}(\varepsilon, n) = (1-c(\varepsilon, n))\boldsymbol{I}_n + c(\varepsilon, n)\frac{1}{n}\mathbf{1}_n\mathbf{1}_n^{\mathrm{T}}$ where $c(\varepsilon, n) = \frac{\varepsilon}{\sqrt{n-1}}$ and $\varepsilon \in (0, 1)$. The definition of $c(\varepsilon, n)$ ensures that $\|\boldsymbol{D}_{\text{center}}(\varepsilon, n) - \boldsymbol{I}_n\| = \varepsilon$ (proven in the appendix) and the condition $\varepsilon \in (0, 1)$ ensures $c(\varepsilon) \in (0, 1)$ for all $n \geq 2$. We call this the *center-step* scheme.

Notice that, for a sequence $\boldsymbol{S}$ of length n, the multiplication $T_{\boldsymbol{C}_n}(\boldsymbol{S}) = \boldsymbol{C}_n\boldsymbol{S}$ computes the average of the vectors in $\boldsymbol{S}$. If $\boldsymbol{A}$ is an adjacency matrix of a graph, then $\boldsymbol{C}_n\boldsymbol{A}\boldsymbol{C}_n^{\mathrm{T}} = \frac{1}{n^2}\sum_{i,j} A_{i,j}\mathbf{1}\mathbf{1}^{\mathrm{T}}$. This matrix can be viewed as the adjacency matrix of a completely connected weighted graph, where all edges have the same weight, and the weight depends on the original edge density. Thus, $\hat{T}'_{\boldsymbol{D}_{\text{center}}(\varepsilon,n)}$ computes tangent vectors that point to the average of the input. In some sense, a BReg penalty defined with center-step accounts for all permutations at once.

Random Segment Steps. We can also create an $n \times n$ DSt matrix by the following scheme: (1) sample i, j uniformly without replacement from $\{1, \ldots, n\}$, (2) swap rows i, j of the identity matrix $\boldsymbol{I}_n$ to form matrix $\boldsymbol{Q}_n$, (3) define $\boldsymbol{D}_{\text{segment}}(\varepsilon, n) = (1 - c(\varepsilon))\boldsymbol{I}_n + c(\varepsilon)\boldsymbol{Q}_n$ where $c(\varepsilon) = \varepsilon/2$ for $\varepsilon \in (0, 2]$. Again, the definition of $c(\varepsilon)$ guarantees $\|\boldsymbol{D}_{\text{segment}}(\varepsilon, n) - \boldsymbol{I}_n\| = \varepsilon$ and the range for ε guarantees a valid convex combination. Instead of swapping two rows, we could sample any random permutation matrix and perform a convex combination. However, in this case, we would need to compute the appropriate form of $c(\varepsilon, n)$ such that $\|\boldsymbol{D} - \boldsymbol{I}\| = \varepsilon$ and ensure that the sampled matrix is not the identity. In our experiments, we just swap two rows for simplicity.

For the geometric intuition, let $v \in \mathcal{V}(\mathcal{PH})$ be a vertex of the permutohedron obtained by swapping two elements of $(1, \ldots, n)^{\mathrm{T}} \in \mathcal{V}(\mathcal{PH})$. A line segment can be drawn between these two vertices hence we call this the *random-segment* scheme and write $\boldsymbol{D}_{\text{segment}}$.

In the context of regularization, this corresponds to enforcing invariance to random transpositions. Invariance to all such transpositions implies invariance to all permutations. Together with the fact that TP can enforce invariances to linear combinations of the tangent vectors penalized, we hypothesize that this is an efficient way to penalize sensitivity to permu-

tations. This somewhat resembles π-SGD training, and the two approaches are not mutually exclusive. We can train with π-SGD and regularize to tame orbit variance. However, this regularization places an explicit penalty on permutation sensitivity.

Computation. In practice, there are more efficient strategies than computing explicit convex combinations. Instead, we can analytically derive the result. For instance, if i and j are the rows to swap when computing $\boldsymbol{D}_{\text{segment}}$, we can directly set $D_{i,j} = D_{j,i} = c(\varepsilon)$, $D_{i,i} = D_{j,j} = 1 - c(\varepsilon)$, and let all other rows besides i, j have 1 on the diagonal and 0 on the off-diagonal.

Sampling Doubly-Stochastic Matrices

We can construct valid tangent vectors by sampling any doubly-stochastic matrix $\boldsymbol{D}$ and then computing $\varepsilon = \|\boldsymbol{D} - \boldsymbol{I}\|$.

Linderman, Mena, Cooper, *et al.* [200] proposes a *stick-breaking* approach for sampling doubly-stochastic matrices from the Birkhoff polytope. This distribution includes a temperature parameter that controls the concentration around exact permutation matrices (the vertices of the permutohedron), which is arguably most natural. Alternatively, we can create any random permutation matrix $\boldsymbol{P}$ (except the identity matrix) and take a convex combination with the identity matrix, as with the center-step and random-segment schemes. On a final note, we could leverage the popular Sinkhorn-Knopp algorithm [303]. This algorithm is designed to map matrices of positive elements to the Birkhoff polytope by iteratively normalizing the rows and columns to sum to one. It is known to converge, but may not result in an exact DSt matrix in infinite iterations. Nonetheless, we could sample a matrix of positive elements with any sampler, then apply Sinkhorn-Knopp for several iterations. Note that these are not computations we need to differentiate as we are not optimizing over DSt matrices, in contrast with several works in the literature.

5.2.4 Backprop through the Jacobian

TangentProp requires computing and differentiating a Jacobian-vector product (JVP), i.e., a vector multiplied on the left by the Jacobian (see Equation 2.4). While automatic

differentiation typically liberates the researcher from calculating derivatives, this particular computation is not straightforward. In particular, frameworks like PyTorch [301] often use *reverse-mode* differentiation [304], which does not give direct or efficient access to the Jacobian or to the Jacobian-vector product. Rather, these are optimized for computing vector-Jacobian product (VJP). To compute the JVP efficiently, we use the trick developed in [305], which only requires backpropagating twice through the network. In contrast, the computation required to extract the Jacobian directly grows with the size of the input in current PyTorch implementations.

In brief, we can compute (and backprop through) the Jacobian-vector product with PyTorch by computing two vector-Jacobian products. Let f be a model and $\boldsymbol{v}$ a vector of appropriate dimension, and we desire the JVP $f'(\boldsymbol{x})\boldsymbol{v}$, where $f'(\boldsymbol{x})$ denotes the Jacobian of the model f with respect to input $\boldsymbol{x}$ (see Equation 2.4). Then, letting $\boldsymbol{u}$ denote a dummy vector of the same dimension as $f(\boldsymbol{x})$, we first compute the vector-Jacobian product $\boldsymbol{u}^{\mathrm{T}} f'(\boldsymbol{x})$. The second vector-Jacobian product multiplies $\boldsymbol{v}$ by the Jacobian of $\boldsymbol{u}^{\mathrm{T}} f'(\boldsymbol{x})$, viewed as a function of $\boldsymbol{u}$. A short mathematical justification is provided in [305], and we provide additional details in the appendix.

5.3 Training with BReg

So far, we have given a mathematical description of R, the BReg penalty function. Optimizing a model regularized with BReg is similar to training models that use other regularization schemes like the ℓ_2 weight penalty. However, there are some nuances to discuss.

Choosing a Regularization Strength. The goal of BReg is to train a permutation-invariant model whenever invariance does not contradict the data. Increasing the regularization strength λ used during training should result in a corresponding reduction in permutation sensitivity of the estimated model. We demonstrate this in our experiments. As a result, we anticipate that regularized models will often have worse performance on the data observable during training, since we are limiting the flexibility of the model. However, the goal is to reap benefits in generalization or extrapolation, especially if the test-set data are ordered differently than what is observed during training. Accordingly, we should choose the model trained with the strongest acceptable regularization, which we call the

Maximum Acceptable Strength (MAS), for test-time predictions. To select λ_{MAS}, we inspect the validation-set performance of models trained with varying strengths. Numerically, we can follow [179] and choose the strength resulting in a model whose accuracy is within five percentage points of the largest validation accuracy. The practitioner can also decide from a plot, for instance choosing a strength beyond which performance starts to degrade rapidly (see Section 5.5). In our experiments, we found it helpful to search over a logarithmic scale including zero to identify an appropriate range, and one can subsequently sweep over a finer set of values if necessary. For instance, we chose $\lambda \in \{0, 2^{-3}, 2^{-2}, \ldots, 2^4\}$.

To emphasize the importance of choosing a MAS λ, consider the task of predicting the maximum element in a sequence. If all sequences in the training data are sorted in ascending order, then a permutation-sensitive model can fit the data by learning a function that simply reports the last sequence element. However, this would fail if test-time sequences have a different ordering. Naively choosing the regularization strength with best performance at training time could result in selecting the smallest regularization strength, corresponding to the fully permutation-sensitive model, and failing at test time. Meanwhile, shuffling the sequences (i.e., data augmentation or indeed π-SGD training) to solve this problem is not the only suitable approach. Doing so encodes an assumption of invariance, which may contradict the task. Our experiments further explore these ideas.

Selecting ε. The hyperparameter ε appears in the definition of tangent vectors. In the TangentProp framework, it should be infinitesimally small, but in practice this could lead to numerical stability challenges. We experimented with tuning ε and found that the optimal value does indeed vary with the task, model, and regularization scheme (TP versus FD and so on). We typically tuned ε in the range $[0.0001, 0.1]$.

Alternatively, setting ε to be "large" results in regularization approaches with a different interpretation. In particular, when ε is large enough, our schemes for constructing doubly-stochastic matrices can result in strict *permutation* matrices other than the identity. For instance, consider the naive penalty scheme of sampling a random permutation π from the subset of permutations that swap two elements and computing $\| \vec{f}(\underline{\mathbf{x}}; \boldsymbol{\Theta}) - \vec{f}(\pi \cdot \underline{\mathbf{x}}; \boldsymbol{\Theta}) \|$. This is equivalent to using Random Segment doubly-stochastic matrices with $\varepsilon = 2$ (see Section 5.2.3). Thus, such approaches are part of our framework. A direction for future work is

to provide concrete theoretical guidance on the choice of ε, and we explore a few different choices in the experiments.

Computational Cost. The regularization penalty adds computational overhead to the optimization procedure. We find that regularized models must be trained for more epochs to converge. Intuitively, minimization must balance competing goals of reducing the penalty R and fitting the training data. However, we argue that the additional computational cost is justified in pursuit of models that have better generalization performance and a data-driven scheme for deciding the appropriate amount of invariance for a given task. As a solution, we typically do not stop training until the validation-set performance has stopped decreasing for (say) 100 epochs. Another interesting direction for future work would be exploring regularization strength schedulers, their relationships with learning rate schedulers, and the impact on convergence.

Selecting a BReg Variant. We have proposed numerous variants of BReg. To choose among them, one can begin by deciding whether the chosen model for the task admits a natural regularization with BReg, such as the positional encoding of Transformer-like models. Next, if computational resources or training time are limited, the FD approach offers a faster alternative to TP, although TP achieves better performance in some of our experiments. Broadly speaking, the question of which approach to use is an empirical one that we begin to address in our experiments section.

5.4 Optimization Perspective and Connections to Other Work

Fundamentally, the goal of BReg is to reduce some quantity over all permutations, so we draw inspiration from optimization literature on *permutation problems*. These are NP-Hard discrete optimization problems and are approached with a variety of relaxations and heuristics [306]–[308]. Relaxations involve solving a minimization over the Birkhoff polytope $\mathcal{B}_n$ – a convex set – and converting the solution back to as permutation [200], [295]–[297].

We will connect optimization problems over $\mathcal{B}_n$ to BReg. For simplicity of notation and visualization, assume for now that the input is a sequence of scalars, $\mathcal{X} = \mathbb{R}^{n\times 1}$. So, regularization is posed as reducing the fluctuations in $\vec{f}$ over the convex hull of permutations

of its input. That is, letting $\mathcal{PH}(\boldsymbol{x})$ denote the generalized permutohedron of $\boldsymbol{x}$, we could define the penalty as

$$R(\boldsymbol{x}, \vec{f}) = \mathbb{E}_{\boldsymbol{x}^\dagger \sim \mu(\mathcal{PH}(\boldsymbol{x}))}\Big[\big|\mathrm{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger)\big|\Big], \tag{5.3}$$

where μ is any probability measure on $\mathcal{PH}(\boldsymbol{x})$, $\mathrm{comp}_{\mathcal{PH}(\boldsymbol{x})}(\cdot)$ is the *scalar projection* onto the permutohedron, and $|\cdot|$ is the absolute value. For vectors $\boldsymbol{v}$ and $\boldsymbol{u}$, the scalar projection $\mathrm{comp}_{\boldsymbol{u}}\boldsymbol{v}$ gives the magnitude of $\boldsymbol{v}$ in the direction of $\boldsymbol{u}$ and is given by $\mathrm{comp}_{\boldsymbol{u}}\boldsymbol{v} = \cos(\omega(\boldsymbol{v}, \boldsymbol{u}))\|\boldsymbol{v}\|$, where $\omega(\boldsymbol{v}, \boldsymbol{u})$ is the angle between $\boldsymbol{v}$ and $\boldsymbol{u}$ [309]. In our case, the absolute value of $\mathrm{comp}_{\mathcal{PH}(\boldsymbol{x})}$ measures the length of a vector in the direction of the permutohedron. Without the projection, we would penalize fluctuations in $\vec{f}$ along directions that do not point to permutations. The probability measure μ can be chosen to induce a continuous distribution that concentrates near permutations (vertices of permutohedra), a discrete distribution that only puts mass on permutations, and so on.

Notice that Equation 5.3 can be written

$$\mathbb{E}_{\boldsymbol{D} \sim \tilde{\mu}(\mathcal{B}_n)}\Big[\big|\mathrm{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{D}\boldsymbol{x}} \vec{f}(\boldsymbol{D}\boldsymbol{x})\big|\Big],$$

where the expectation now runs over doubly-stochastic matrices $\boldsymbol{D}$ in the Birkhoff polytope $\mathcal{B}_n$, and $\tilde{\mu}$ is a probability measure over $\mathcal{B}_n$. As we will see in Section 5.4.3, the projections are difficult to implement efficiently in practice. Hence, we will not dwell on a formal description, other than noting that TP and FD BReg can be viewed as tractable proxies. Instead, we will study this theoretically to glean new insights into BReg, including the FD and TP variants.

5.4.1 Linear Model Example: Not a Shrinkage Penalty

It is instructive to study the linear model defined by $\vec{f}(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ for any $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$ and fixed $\boldsymbol{w} \in \mathbb{R}^{n\times 1}$. The gradient is $\nabla_{\boldsymbol{x}} \vec{f}(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}$, for any $\boldsymbol{x}$, so if our penalty was not defined in terms of projected gradients, it would effectively reduce to $\mathbb{E}_{\boldsymbol{x}^\dagger \sim \mu(\mathcal{PH}(\boldsymbol{x}))}[\|\boldsymbol{w}\|] = \|\boldsymbol{w}\|$.[8] Hence, the penalty would only be minimized by the zero vector, which is a stronger

[8] Without the projection, the absolute value would be a norm.

condition than needed. For permutation invariance, it suffices to set $\boldsymbol{w} = w\mathbf{1}$, a vector where every weight has the same value $w \in \mathbb{R}$. We will see that this is the condition enforced by Equation 5.3.

To compute the penalty, we must compute $\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger; \boldsymbol{w})$ for all $\boldsymbol{x}^\dagger \in \mathcal{PH}(\boldsymbol{x})$. First, we observe that $\mathcal{PH}(\boldsymbol{x})$ lies on a hyperplane: if $s = \sum_i x_i$, then $\sum_i x_i^\dagger = s$ for all $\boldsymbol{x}^\dagger \in \mathcal{PH}(\boldsymbol{x})$. Hence, we describe the permutohedron by its normal vector, which we can read off of the coefficients of the hyperplane equation as $\mathbf{1} = (1, \ldots, 1)^{\mathrm{T}}$ [309]. Since vectors are normally written to be unit-norm, we write the normal vector of the permutohedron as $\boldsymbol{z} = \frac{1}{\sqrt{n}}\mathbf{1}$.

Next we compute $\cos\Big(\omega(\nabla_{\boldsymbol{x}} \vec{f}(\boldsymbol{x}; \boldsymbol{w}), \boldsymbol{v})\Big) = \cos(\omega(\boldsymbol{w}, \boldsymbol{v}))$ where $\omega(\boldsymbol{w}, \boldsymbol{v})$ is the angle between $\boldsymbol{w}$ and any vector $\boldsymbol{v}$ that points in a direction along $\mathcal{PH}(\boldsymbol{x})$. First observe that the angle with the normal vector is $\omega(\boldsymbol{w}, \boldsymbol{z}) = \cos^{-1}\Big(\frac{\boldsymbol{w}^{\mathrm{T}}\boldsymbol{z}}{\|\boldsymbol{w}\|}\Big)$. Moreover, we can suppose without loss of generality that the angle between $\boldsymbol{w}$ and $\boldsymbol{v}$ is not obtuse, since we only need the absolute value of the scalar projection. Thus, with standard trigonometry, we can derive

$$\cos\Big(\omega(\boldsymbol{w}, \boldsymbol{v})\Big) = \sqrt{1 - \frac{1}{\|\boldsymbol{w}\|^2}\Big(\boldsymbol{z}^{\mathrm{T}}\boldsymbol{w}\Big)^2}.$$

Hence, after some computation, for any $\boldsymbol{x}^\dagger \in \mathcal{PH}(\boldsymbol{x})$,

$$\begin{aligned}
|\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger; \boldsymbol{w})| &= |\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \boldsymbol{w}| \\
&= \|\boldsymbol{w}\| |\cos\Big(\omega(\boldsymbol{w}, \boldsymbol{v})\Big)| \\
&= \|\boldsymbol{w}\| \sqrt{1 - \frac{1}{\|\boldsymbol{w}\|^2}\Big(\boldsymbol{z}^{\mathrm{T}}\boldsymbol{w}\Big)^2} \\
&= \sqrt{\sum_{i=1}^{n} w_i^2 - n\bar{w}^2},
\end{aligned}$$

where $\bar{w} := \frac{1}{n}\sum_{i=1}^{n} w_i$. Notice that this is proportional to the (population) standard deviation of $\boldsymbol{w}$. Since this holds true for any $\boldsymbol{x}^\dagger$, we have proven the following proposition.

Proposition 5 *When $\vec{f}(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x}$, the penalty defined in Equation 5.3 becomes*

$$R(\boldsymbol{x}, \vec{f}) = \mathbb{E}_{\boldsymbol{x}^\dagger \sim \mu(\mathcal{PH}(\boldsymbol{x}))}\Big[|\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger)|\Big] = \sqrt{\sum_{i=1}^{n} w_i^2 - n\bar{\boldsymbol{w}}^2} = \text{sd}(\boldsymbol{w})\sqrt{n-1},$$

for any $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$. *That is, the penalty is proportional to the standard deviation of the regression coefficients. Therefore,* $R(\boldsymbol{x}, \vec{f}) = 0$ *if* $\boldsymbol{w} = w\mathbf{1}$, $w \in \mathbb{R}$ *and* $R(\boldsymbol{x}, \vec{f}) > 0$ *otherwise.*

Remark 11 *This example has shown that the penalty enforced by the alternative regularization defined in Equation 5.3 has an intuitive interpretation in the case of a linear model. The penalty is only minimized when the coefficients are all the same, rendering the model permutation-invariant. Furthermore, we highlighted that we cannot simply compute the gradient of* $\vec{f}$*, but rather the component of its projection along the direction of permutohedra.*

5.4.2 Connections to Existing Methods

In this section, we will see that sum pooling arises as the only model that can achieve zero penalty in general using the alternative regularization in Equation 5.3. Thus, when the strength λ tends to infinity, sum pooling is the only function that can achieve finite penalty and thus a finite cost.

Before stating the theorem, we note that Figure 5.2 gives a visual proof. Vectors whose elements sum to the same value have overlapping permutohedra. Therefore, $\vec{f}$ must be constant on all inputs with the same sum in order for the expectation to be zero.

Theorem 5 ***(Sum pooling)*** *Suppose* μ *puts full support on the input space (thus full support on all permutohedra) and let* $\vec{f} \in \mathcal{F}$ *denote an architecture parameterized by a vector of parameters. Then, if* R *is defined by Equation 5.3,*

$$R(\boldsymbol{x}, \vec{f}) = 0 \iff \vec{f}(\boldsymbol{x}) = w\sum_i x_i + C \quad \forall \boldsymbol{x} = (x_1, \ldots, x_n)^T \in \mathcal{X},$$

for some $w \in \mathbb{R}$ *and constant* C.

Proof 9 ($\Longleftarrow$) When $\vec{f}(\boldsymbol{x}) = w\sum_i x_i + C$, $\nabla_{\boldsymbol{x}} \vec{f}(\boldsymbol{x}) = w\mathbf{1}$ for any $\boldsymbol{x} \in \mathcal{X}$. We saw that $\mathbf{1}$ is orthogonal to vectors along the direction of $\mathcal{PH}(\boldsymbol{x})$. Therefore, for all $\boldsymbol{x}^\dagger \in \mathcal{PH}(\boldsymbol{x})$, $\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger) = 0$. Thus,

$$R(\boldsymbol{x}, \vec{f}) = \mathbb{E}_{\boldsymbol{x}^\dagger \sim \mu(\mathcal{PH}(\boldsymbol{x}))}\Big[\big|\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger)\big|\Big] = \mathbb{E}_{\boldsymbol{x}^\dagger \sim \mu(\mathcal{PH}(\boldsymbol{x}))}[0] = 0.$$

($\implies$) Let $\boldsymbol{x} \in \mathcal{X}$ be arbitrary. If $R(\boldsymbol{x}, \vec{f}) = 0$, then we must have $|\text{comp}_{\mathcal{PH}(\boldsymbol{x})} \nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger)| = 0$ for all $\boldsymbol{x}^\dagger \in \mathcal{PH}(\boldsymbol{x})$, since μ puts full support on $\mathcal{X}$. This implies that $\nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger)$ is orthogonal to vectors lying in the direction of the permutohedron. Since normal vectors are unique, up to scaling, $\nabla_{\boldsymbol{x}^\dagger} \vec{f}(\boldsymbol{x}^\dagger) = w\mathbf{1}$ for some w. It follows that $\vec{f}(\boldsymbol{x}) = w \sum_i x_i + C$. ■

The key idea behind this proof is that summation of a vector can be written as the inner product with $\mathbf{1}$, and $\mathbf{1}$ is orthogonal to the permutohedron. With this idea, the result can be extended to regularization for variable-sized sequences of *vectors*.

This result implies that not all permutation-invariant functions have zero penalty, and thus Equation 5.3 imposes a much stronger condition than permutation invariance. Moreover, since Theorem 5 can be extended to sequences of vectors, it offers new insights into sum pooling methods like DeepSets. Suppose we penalize permutation sensitivity over *hidden* vectors of a model. If the regularization strength is infinite, then sum pooling is the only function that can have finite cost. That is, sum pooling arises as the only model that can satisfy the stronger restrictions of Equation 5.3. This is the second time sum pooling emerged as a model satisfying very strong structural assumptions, the first being an assumption of infinite exchangeability (see Section 4.2).

5.4.3 Projections and BReg

Equation 5.3 is very computationally expensive. While the expectation can be sampled, the projection is challenging to handle in the general case for variable-size sequences and graphs. In particular, the general case requires projections onto the Birkhoff polytope. Doing so efficiently is an active area of research and uses sophisticated mathematical machinery. For instance, Li, Sun, and Toh [310] use a semismooth Newton method on the dual and Jiang, Liu, and Wen [311] use a dual gradient method with Barzilai-Borwein step sizes (see also [308], [312]). Birdal and Simsekli [298] use Riemannian geometry in an optimization over the Birkhoff polytope specifically to avoid these projections. Since we would need to backpropagate through this projection at every epoch, it is infeasible.

Fortunately, our TP/FD BReg approaches can be seen as efficient proxies for Equation 5.3. Taking the FD case as an example, $\frac{1}{\varepsilon}\| \vec{f}\left(\underline{\mathbf{x}}\right) - \vec{f}\left(T_{\boldsymbol{D}(\varepsilon, \underline{\mathbf{X}})}(\underline{\mathbf{x}})\right)\|$ measures the change

in $\vec{f}$ in the direction of the Birkhoff polytope, which was exactly the goal of computing the length of the projected gradient. Moreover, computing tangent vectors starting at $\underline{\mathbf{x}}$ corresponds to defining a measure μ that puts probability only on the vertices of $\mathcal{B}_n$. Hence, this perspective offers another perspective and justification of the BReg penalty.

5.5 Experiments

Next, we will investigate the performance and behavior of BReg. First, we construct an idealized pair of tasks to illustrate that the appropriateness of permutation-invariance is uncertain and evaluate whether BReg training can adapt. Moreover, we demonstrate empirically that training with BReg reduces permutation sensitivity. Second, we explore whether BReg can improve the variance of π-SGD training on a graph classification task, proposed in [245], on which π-SGD training of an RPGNN underperformed.

5.5.1 BReg Training in Permutation-Sensitive and Permutation-Invariant Tasks

In this section, we explore the key hypothesis that it may not be obvious a-priori whether a permutation-sensitive or permutation-invariant model will achieve stronger test-time performance on any given task. We conjecture that training with BReg and selecting the Maximum Acceptable Strength λ_{MAS} provides a data-driven solution. If λ_{MAS} is large, then we ultimately use a model trained with strong regularization – a more invariant model – for test-time predictions.

Accordingly, we propose a pair of synthetic tasks. One is permutation-invariant (PI) and the other is permutation-sensitive (PS). In both, we will compare: (1) a PI model, (2) a PS model with standard (unregularized) training, (3) a PS model trained with BReg.

All models will be based on the SetTransformer, which is PI by design [60]. To introduce a natural spectrum of permutation sensitivity, we incorporate a trainable positional encoding layer with weights $\boldsymbol{\xi}$ as discussed above [302]. When the model has such a layer, we will call it a Transformer model for brevity. This in a slight abuse of terminology as it is not identical to the original Transformer [236], which uses a larger architecture and extra components such as dropout layers [313].

Remark 12 *The permutation-sensitivity of a Transformer can be measured by the sum of the variances down every column in the positional encoding* $\boldsymbol{\xi}$*. If all columns in* $\boldsymbol{\xi}$ *have zero variance, then the model will be permutation-invariant. We will write Positional Encoding Variance as PEV and use it as a simple and efficient metric for quantifying permutation sensitivity.*

We consider several BReg strategies for regularizing the Transformer (see Table 5.1) to empirically explore their relative merits. The first follows the architecture shown in Figure 5.4, a FD approach where $\boldsymbol{D}$ is multiplied by the positional encodings, and the second is similar but replaces the FD approach with TP. Similarly, we also train with the TP and FD variants when we multiply $\boldsymbol{D}$ by the *input*. For each of these, we train by constructing doubly-stochastic matrices using both the *random-segment* and *center-step* schemes (Section 5.2.3). There are eight approaches in total. In light of this, we carefully evaluate on several independently sampled test sets – not seen during training – to avoid being misled by multiple testing.

Permutation-Invariant Task: Predict the Maximum

Taking inspiration from the SetTransformer paper, we predict the maximum of a sequence of integers. In particular, input sequences have length 10 and are sampled uniformly at random from $\{0, 1, \ldots, 98\}$.

For PI tasks, we expect that a PS model *can* fit the training data. Indeed, MLPs are PS and usually *over*fit. The real challenge for PS models is in achieving adequate test-set performance. To test this, we will order the sequences in training (and validation) in ascending order but apply different orderings in the test data. Specifically, we *randomly shuffle* and apply a *descending sort*. The PS model should fail on the test set, but PI models should not be distracted by the ordering and perform well on test. We hope to see: (1) training with BReg causes the model to become more invariant; (2) the model trained with BReg using a maximum acceptable strength λ_{MAS} generalizes better than the PS model; (3) the model trained with λ_{MAS} achieves similar performance compared to the

PI SetTransformer. Also, we are interested in the behavior of the various regularization strategies.

We proceed as follows: we (1) tune hyperparameters, including identifying λ_{MAS}, on the *validation data alone*; (2) select a model trained with BReg and λ_{MAS}, and compare its performance with the SetTransformer and Transformer (which are PI and PS models, respectively) on test datasets; (3) visualize the effect of regularization on the test datasets and on model variance. Note that step (2) emulates a real modeling scenario whereas (3) is for the purpose of exploring BReg more thoroughly.

Our model architecture closely follows that provided by the authors of SetTransformer, and we add a trainable positional encoding to form a Transformer [302]. We performed a hyperparameter search over learning rate and batch size with cross validation. For the regularized models, we swept over $\varepsilon \in \{0.01, 0.1\}$ and regularization strengths $\lambda \in \{0, 2^{-3}, 2^{-2}, \ldots, 2^{4}\}$, and the aforementioned BReg variants. All cross-validation folds have about 32,000 observations. Following [60], we evaluate with ℓ_1 loss (Mean Absolute Error). Further details are provided in the appendix.

Model Selection and λ_{MAS}. We found that the BReg variant of taking tangent vectors on the positional encoding and a value of $\varepsilon = 0.01$ performed better on the validation data. Our next step is to choose λ_{MAS} before evaluating models on the test set. To do so, we visualize performance as a function of λ on the *validation data*, which has the *same sort-order* as the training data. We will pick the largest λ before performance drops off. The left side of Figure 5.5, shows that the validation performance does not start to deteriorate substantially for TangentProp models even with a regularization strength of 16. In contrast, the Finite Differences models deteriorate around 2^1. Since the models trained with TangentProp appear more stable, we will choose models trained with TangentProp and strength $\lambda = 16$. We will show the performance of both center-step and random-segment matrices for comparison. Note that, in this stage, we are *not* looking at the test-set data! We must select hyperparameters based on the validation set alone.

Measuring Performance. In Table 5.2, we compare the performance of the two baselines to the Transformer trained with the aforementioned regularization strategy. We see that the Transformer is able to perform well on the validation data but fails when the ordering

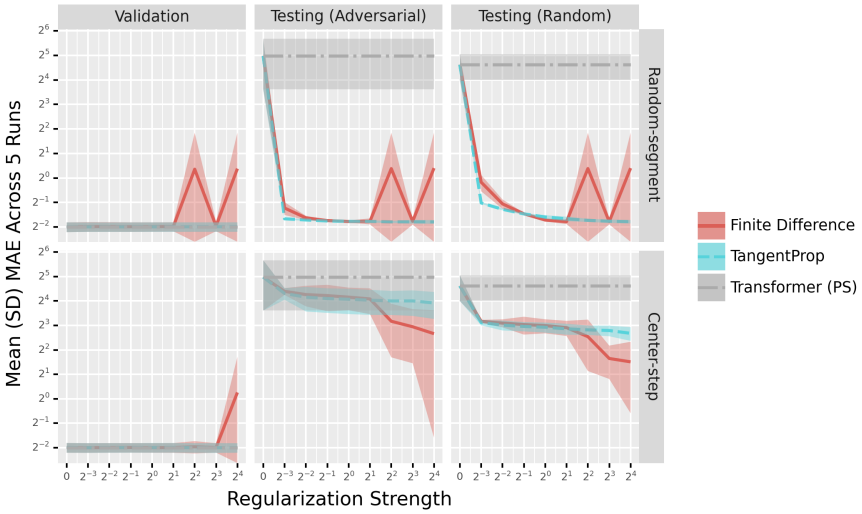


Figure 5.5. Performance vs. λ on Permutation-Invariant Max Task. Performance is shown in Mean Absolute Error on three types of datasets, averaging over five independent samples of each. The sequences in the validation data have the same ordering as the training data: ascending sort. The sequences in the two test datasets are respectively sorted in descending order and randomly shuffled. The permutation-sensitive Transformer model achieves low MAE on the training data but a much larger loss on the test datasets. In contrast, the regularized Transformer models generally perform much better on the test data.

changes in the test set. In contrast, the Transformer trained with BReg and the random-segment scheme achieves similar performance to the SetTransformer, which is permutation-invariant by design. BReg with center-step still outperforms the unregularized Transformer but is worse than the random-segment scheme. Finally, we see that regularization substantially decreases the permutation-sensitivity of the model, as measured by PEV (see Remark 12).

Further Exploration of Regularization Strength. Strictly for the purposes of exploration, we can inspect the test-set performance of all the regularized models in Figure 5.5. In general, we see that stronger regularization corresponds to increasingly better test-set performance. This is encouraging as the task is specifically designed such that only PI models can perform well on the test data. This suggests that BReg is correctly regularizing towards permutation-invariance. Further evidence is shown in Figure 5.6, where we see that

Table 5.2. Performance on Permutation-Invariant Max Task. Mean (SD) Mean Absolute Error as well as model permutation sensitivity. Sequences in the validation datasets have the same sort-order as the training data (ascending sort), while the two test datasets contain sequences with different orderings. The first two rows show a PI and PS model, respectively. The bottom two rows correspond to Transformers trained with BReg using the random-segment and center-step schemes. The permutation sensitivity is measured by PEV (Remark 12).

Model	Val (ASort) MAE	Test (DSort) MAE	Test (Shuffle) MAE	Perm Sens
SetTransformer	0.253(0.033)	0.288(0.013)	0.288(0.013)	N/A
Transformer	0.251(0.034)	31.530(19.321)	24.585(8.582)	0.014(0.006)
BReg, $\lambda=16$, RS	0.252(0.033)	0.289(0.015)	0.291(0.014)	0.000(0.000)
BReg, $\lambda=16$, CS	0.250(0.033)	15.136(5.513)	6.397(1.233)	0.001(0.000)

PEV, our measure of permutation sensitivity, tends to decrease as the regularization strength increases.

Permutation-Sensitive Task: "First Large"

Using the same dataset of randomly sampled integers, we propose to learn the classifier

$$f(\boldsymbol{S}) = \begin{cases} 1 & S_1 > 49 \\ 0 & \text{otherwise} \end{cases},$$

for any sequence $\boldsymbol{S} = (S_1, \ldots, S_n)^{\mathrm{T}}$, where 49 is the median of the support $\{0, 1 \ldots, 98\}$. This function only depends on the first element, in some ordering, and is therefore PS. Unlike the previous task, we do not need to apply different orderings to the sequences to illustrate our ideas for this task.

For PS tasks, the true function lies outside the class of PI models. Thus, we expect the SetTransformer (a PI model) to perform poorly on both the validation and test data whereas the Transformer should be able to achieve strong performance. For this experiment, we are expecting: (1) that stronger regularization causes the model to become more permutation-invariant, and thus worse at modeling this PS function, and (2) that the techniques for selecting a maximum acceptable strength λ point to a small value.

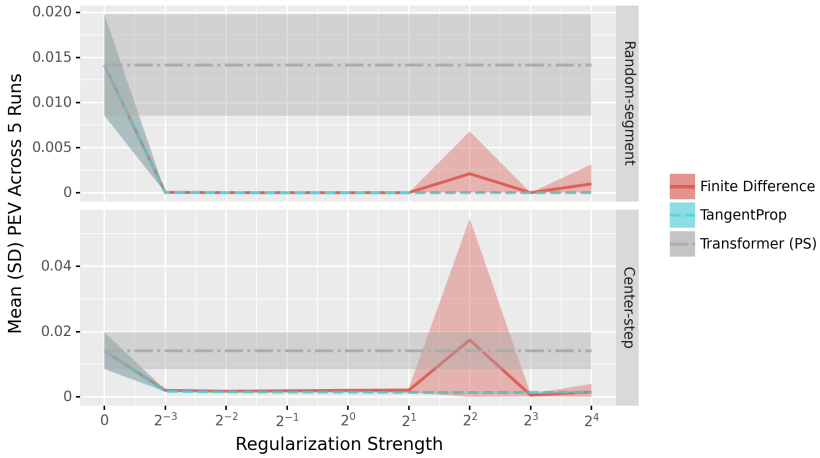


Figure 5.6. Permutation Sensitivity vs. λ on Permutation-Invariant Max Task. Permutation sensitivity measured by Positional Encoding Variance (see Remark 12), as a function of regularization strength. The variance of the fully permutation-sensitive Transformer model is shown for reference.

The hyperparameters and other training details are similar to those used in the max task (additional details can be found in the appendix), with one difference being that fewer epochs were needed for convergence. We measure performance in accuracy since this is a binary classification task, and the target is relatively balanced among training, validation, and test datasets.

Effect of Regularization. Figure 5.7 shows the performance as a function of regularization strength on the validation and test datasets. The results are similar on both datasets since the sequences are sampled from the same distribution. As expected, the performance degrades as regularization strength increases, regardless of the choice of regularization strategy. The performance of the regularized models seems to converge to that of SetTransformer – a permutation-invariant model – as the regularization strength increases. Figure 5.8 shows the impact of regularization strength on permutation sensitivity. When DSt matrices are created via the random-segment scheme, stronger regularization leads to a corresponding reduction in permutation-sensitivity, as expected, for both TP and FD variants. Indeed, when large regularization is in effect, the variance is significantly smaller than that of the unregularized and fully permutation-sensitive Transformer. However, such a trend does not appear for the center-step scheme of creating doubly-stochastic matrices. In fact, as regu-

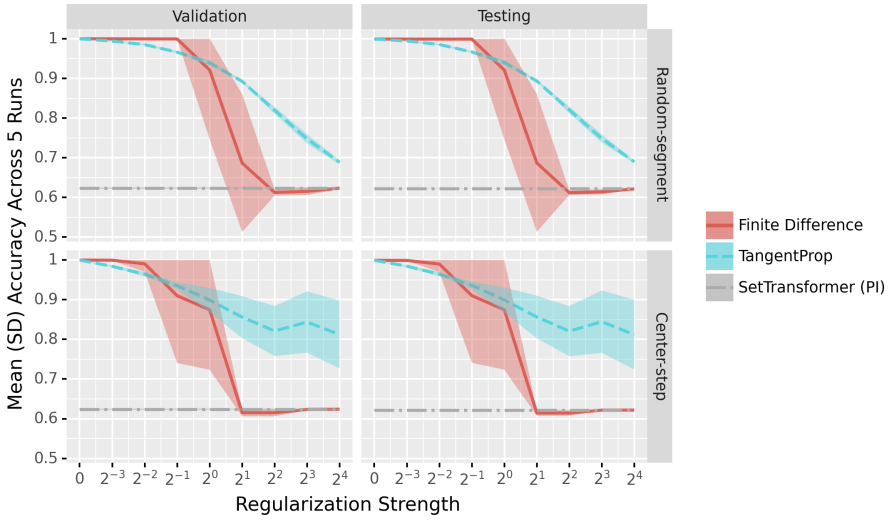


Figure 5.7. Performance vs. λ on Permutation-Sensitive "First Large" Task. We show accuracy on validation and testing datasets, averaged over five independent samples of each. We trained Transformers with BReg variants TP/FD and created doubly-stochastic matrices via the random-segment and center-step schemes. The performance of the fully permutation-invariant SetTransformer – which is expected to fail on this task – is shown for reference. The plots are similar as the validation and testing datasets come from the same distribution.

larization increases, permutation sensitivity seems to increase for these models. Notice that the vertical axes are very different since the PEV is large for center-step.

λ_{MAS} and Test-set Performance In practice, we would need to select the maximum acceptable regularization strength before selecting a model and making predictions on test data. We query the model whose validation-set performance is closest to 95% accuracy, which is within five percentage points of unregularized model (see Section 5.3). This turns out to be the model trained with $\lambda = 0.25$, a TP-style penalty where $\boldsymbol{D}$ is multiplied by the positional encoding and constructed with the random-segment scheme.

Table 5.3 displays performance results numerically. We see that the PI model (SetTransformer) performs poorly whereas the PS model (Transformer) achieves near-perfect accuracy. All models perform similarly on the validation and test data. More interestingly, we see that the PEV of the model trained with regularization is substantially smaller than that of the Transformer, pointing towards a reduction in permutation sensitivity (see Remark 12).

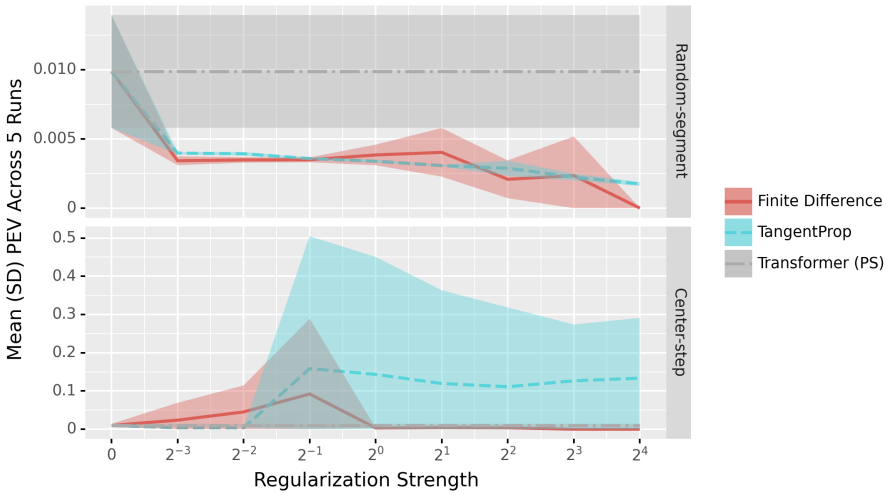


Figure 5.8. Permutation Sensitivity vs. λ on Permutation-Sensitive "First Large" Task. Permutation sensitivity is measured by Positional Encoding Variance (see Remark 12). The variance of the fully permutation-sensitive Transformer model is shown for reference.

Table 5.3. Performance on Permutation-Sensitive "First Large" Task. We show mean (SD) accuracy, as well as model permutation sensitivity. The training and validation datasets are sampled from the same distribution. The first two rows show a PI and PS model, respectively. The bottom row corresponds to a Transformer model trained with BReg and $\lambda_{\text{MAS}} = 0.25$. The permutation sensitivity is measured by PEV (Remark 12).

Model	Validation Accuracy	Test Accuracy	Perm Sens
SetTransformer	0.624 (0.002)	0.622 (0.000)	N/A
Transformer	1.000 (0.000)	0.999 (0.000)	0.010 (0.004)
BReg, $\lambda = 0.25$	0.964 (0.006)	0.964 (0.006)	0.004 (0.000)

Summary

Overall, regularization performed as expected. On the permutation-invariant task, training with BReg helps performance when the out-of-training datasets have sequences ordered differently than in training. In contrast, the permutation-sensitive Transformer model was distracted by the ordering. On the permutation-sensitive task, BReg correctly adapted, helping us observe that strong regularization is not appropriate for the task. In both cases, regularization successfully reduced model permutation sensitivity, as measured by PEV.

Additionally, we performed experiments with $\varepsilon = 2$ in the Random Segment scheme, which recovers exact permutation matrices (see Section 5.3). The overall patterns of the results were similar.

5.5.2 Variance Reduction and Latent Regularization

In this section, we explore the variance-reduction of BReg in the context of π-SGD. First, we explore the strategy of adding latent noise, discussed in Section 5.2.2, with a simple fruit-fly task. The goal is to check whether adding random noise strategy can increase the value of latent layers, preventing them from becoming arbitrarily small in a regularization context. Then, we apply BReg regularization to π-SGD training of an RPGNN on the cycle-detection graph classification task from [245].

Adding Random Noise to Latent Quantities

For simplicity, we estimate a linear model $f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$, where $\boldsymbol{w} \in \mathbb{R}^{10}$ is sampled from a normal distribution. We train an MLP (Equation 2.1) with $L = 2$ and $d_h = 16$. We add random noise, so that $\boldsymbol{H}^{(1)} + \boldsymbol{\zeta}$ is the input to the last layer, where $\boldsymbol{\zeta}$ is sampled from a standard normal distribution. The results are shown in Figure 5.9, comparing to a model where we do not add random noise. We can see that training with noise causes the latent to increase, but apparently without bound. Our goal is to stabilize the scale of the latent – without it becoming too large or too small – so we add an additional penalty term to the objective, $\gamma\|\boldsymbol{H}^{(1)}\|$. Training with this additional term led the loss to stabilize to a value

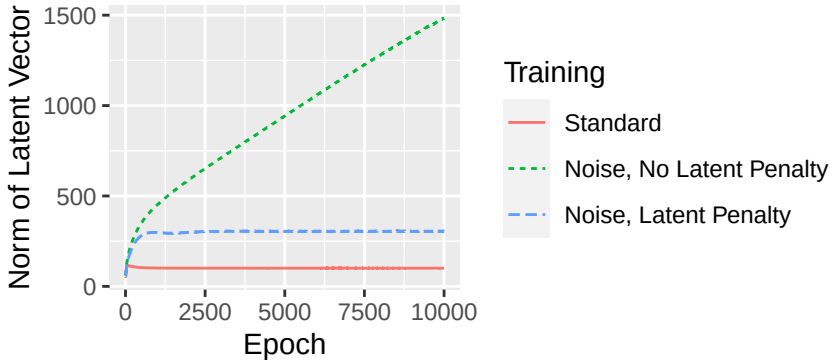


Figure 5.9. Toy Experiment: Adding Noise to Latent Vector. We show the norm of the hidden layer (*before* noise is added, if applicable) as a function of epoch.

that is larger than the standard model. Next, we experiment with these strategies in context of BReg.

Variance Reduction

We have hypothesized that it can be difficult to train RPGNNs with π-SGD (especially in large graphs) due to large variance. We explore whether BReg can ameliorate the variance and improve performance.

Vignac, Loukas, and Frossard [245] propose a cycle-detection task on larger graphs on which RPGNNs perform poorly. Specifically, the task is binary classification, to identify whether the graph contains a cycle of length 8. The dataset contains synthetically generated graphs that are very sparse and contain graphs that do or do not have cycles. We take up this task to explore whether we can improve the performance. After tuning hyperparameters, we train the following hierarchy of models: (1) a baseline RPGNN, (2) an RPGNN with BReg, (3) a regularized model that adds noise to the latent graph representation (before an affine layer), (4) a model with noise and an additional penalty $\|\boldsymbol{H}\|$. We also propose a fifth model, a regularized model with a penalty for $\|\boldsymbol{H}\|$ but no noise. We test two variants of BReg that measure permutation-sensitivity in the hidden graph representation (BReg-penalize-latent) and in the prediction (BReg-penalize-pred). We expect that the fourth model – adding noise and a small norm penalty – is necessary for successful regularization when training

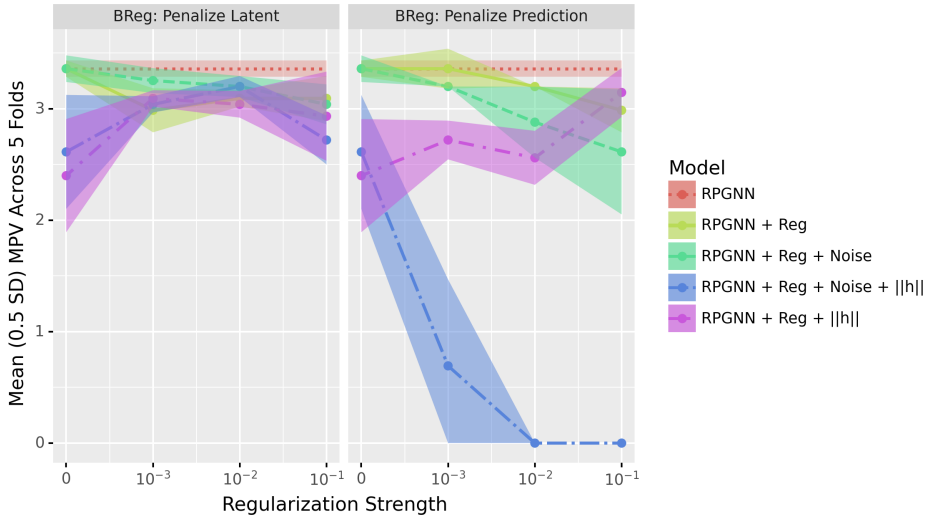


Figure 5.10. Variance over Permutations in Regularized RPGNN Training. That is, Maximum Prediction Variance (across sampled permutations) of RPGNN models trained with π-SGD and a variety of BReg schemes. Ribbons show the mean $\pm$ 0.5 standard deviations for clarity.

with BReg-penalize-latent. We will call this the noise-and-norm regularization strategy. We quantify permutation sensitivity with the Maximum Prediction Variance over permutations (MPV), and report results over five cross-validation folds. That is, we estimate Equation 4.16 for all graphs in the validation data and report the maximum.

Results in Figure 5.10 shows the MPV, after training, as a function of regularization strength. The left corresponds to BReg-penalize-latent. We see that most models slightly reduce variance compared to the unregularized RPGNN baseline. Surprisingly, the variance does not reveal a downward trend as a function of regularization strength. The benefit of the noise-based regularization strategies is not apparent. Interestingly, if we look at the model with BReg-penalize-pred, we find the trends we expected for BReg-penalize-latent training. The noise-and-norm strategy leads to by far the smallest variance and the variance decreases as a function of strength.

Next we study the validation-set accuracy in Figure 5.11. Again, we do not find convincing evidence that the regularization strategies improve performance in the case of BReg-penalize-latent. However, the noise-and-norm strategy, which effectively reduced variance in

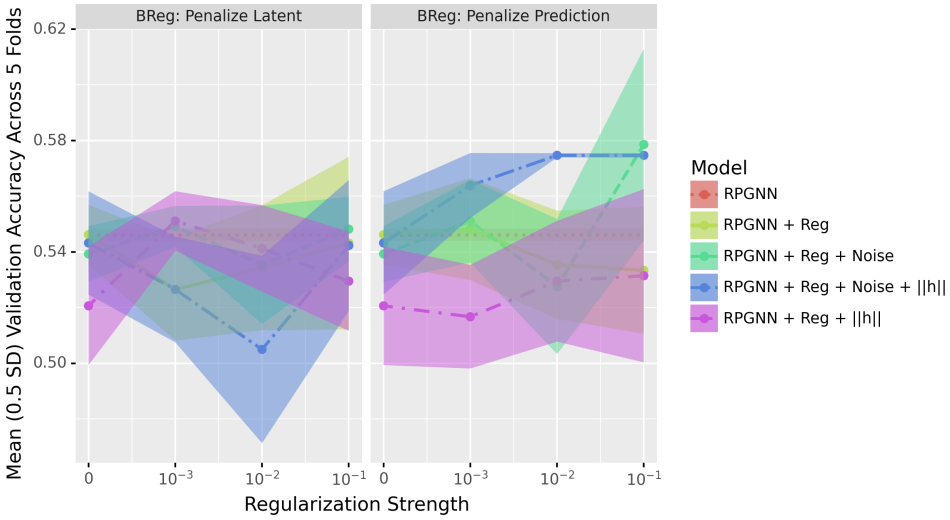


Figure 5.11. Performance of Regularized RPGNN Training in Cycle Detection. Accuracy of RPGNN models trained with π-SGD and a variety of BReg schemes. Ribbons show the mean $\pm$ 0.5 standard deviations for clarity.

BReg-penalize-pred, also results in a noticeable increase in accuracy over the baseline when regularization is sufficiently large.

In summary, we did not find that random noise benefited the strategy BReg-penalize-latent, but it did confer a small benefit for BReg-penalize-pred. However, the performance is still not strong (less than about 60% on a binary classification task). Generally speaking, we have yet to find convincing evidence that BReg can improve π-SGD training. We conclude that there is a stronger case for BReg regarding our first hypothesis, that the extent of invariance is not always clear.

6. SUMMARY AND DIRECTIONS FOR FUTURE WORK

Summary. In this work, we made two contributions to building neural network models that respect invariances in the data, which is known to improve their generalization and extrapolation capabilities. Set and graph data are widespread in practice, and models thereof typically respect *permutation-invariance*.

First, we saw Janossy pooling, a framework that provides novel approaches to invariant training, especially in the context of sets and graphs. In JP, we define invariant functions $\overline{\overline{f}}$ as an average over a transformation-sensitive function $\vec{f}$. Modeling $\vec{f}$ with a flexible network such as our proposed RPGNN for graphs or an RNN provably and empirically leads to a more expressive model. Since averaging over permutations is computationally intractable, we proposed three general approximation schemes. First, in π-SGD, we sample permutations π of the inputs at each epoch but otherwise proceed as we would in standard SGD training. We saw that π-SGD minimizes an upper bound of the original loss, tightens generalization bounds by compressing the model, and converges under similar conditions to standard SGD. π-SGD does not make explicit assumptions about the task and proved to be a successful approximation scheme in many synthetic and real-world experiments. Second, with k-ary approximations, we redefine $\vec{f}$ to take only the first k elements of its input, rendering many permutations redundant. Averaging over only the nonredundant permutations reduces the computational complexity. Unlike π-SGD, k-ary approximations make an assumption about the degree of relationships that are important in a task (e.g., variance is a $k = 2$ task). Increasing k strictly enlarges the set of functions that a pooling layer can express, which reveals a limitation of methods that correspond to 1-ary JP approximations like DeepSets. Third, in poly-canonical orderings, we pass an input through some ordering algorithm (like sort or depth-first search) that collapses the number of possible permutations. Like k-ary, this approach also makes an assumption about the data and in particular which orderings will be useful for the task. These three general approximations can be used together, leading to strategies such as running a DFS from a random starting vertex to select k vertices from a graph (π-SGD, k-ary, and poly-canonical orderings). In parallel, we saw that a wide array of existing methods can be viewed as approximations to JP, which offered insights

and simple but effective modifications that improve their performance. Experiments with JP showed that our theoretical predictions are realized and that it can increase performance on real-world tasks. Of particular note is that our CSL task (Section 4.4.1) was adopted by numerous scholars.

Next, we observed that it may not be clear a-priori whether enforcing permutation-invariance is appropriate in a given task. For example, while graph and set models are typically permutation-invariant due to the arbitrary ordering of such data, it does not imply the ordering never carries information in the data-generating process. Accordingly, we propose regularization towards invariance to let the data make this decision. We choose the model trained with the largest acceptable strength, identified from the validation data, for test-set predictions. Specifically, we propose Birkhoff regularization, which uses doubly-stochastic matrices to define tangent vectors in the direction of permutations. Given these vectors, we compute a penalty inspired by TangentProp, which we also saw can be viewed as a proxy for a minimization problem over permutations. We outlined several variations of this approach. In particular, we may be interested in enforcing permutation-invariance in *hidden* vectors, but this may encounter a scaling problem that we propose to fix by adding random noise to the hidden vectors. From a theoretical perspective, we saw that infinite regularization results in a sum pooling model like DeepSets. In fact, we also saw that DeepSets arises from the strong assumption of infinite exchangeability. In experiments, we showed that Transformers trained with BReg can indeed perform well when the appropriateness of invariance is not clear. When using regularization to reduce variance in π-SGD training, we had surprising findings. Although we thought that adding random noise was only relevant for obtaining invariant latent vectors – as opposed to invariant predictions – it seemed more effective for the latter.

Future work. JP is a new framework, and there are several interesting avenues for future work. First, while JP can naturally be extended beyond permutation-invariance, we did not explore these directions experimentally or theoretically in detail. Perhaps there are $\vec{f}$ functions that, when pooled over, lead to benefits in tasks beyond graphs and sets. Also, k-ary approximations are somewhat tailored for such inputs, and there may be interesting approximations for others. Next, we observed some unanticipated behavior in the optimiza-

tion. For example, when the readout r is not a simple affine layer, we saw that increasing the value of k did not improve performance as expected (Table 4.7). Given that larger k can only make the model more expressive, the most natural conclusion is that there are difficulties in optimization. Turning to Birkhoff regularization, there is an opportunity to expand on the experiments, especially on real-world datasets in which invariance is not clear a-priori. Protein-Protein-Interaction graphs are an interesting direction for this, as the vertex ordering arguably carries meaning [289]. For sequence tasks, sentiment analysis may not depend strongly on the ordering, but simply the presence, of keywords anywhere in the sentence. Methodologically, one could consider *schedulers* that increase or decrease the regularization strength throughout training. This may help the model converge faster or to better optima, but requires a careful empirical and theoretical study. Finally, we did not investigate *sampling* doubly-stochastic matrices with an approach such as stick-breaking [200], as opposed to constructing them with convex combinations. Such an approach could add flexibility in the directions of tangent vectors.

REFERENCES

[1] N. Segol and Y. Lipman, "On universal equivariant set networks," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=HkxTwkrKDB.

[2] K. Zhang and J. S. Bloom, "Classification of periodic variable stars with novel cyclic-permutation invariant neural networks," *arXiv preprint arXiv:2011.01243*, 2020.

[3] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, and L. Zdeborová, "Machine learning and the physical sciences," *Reviews of Modern Physics*, vol. 91, no. 4, p. 045 002, 2019.

[4] N. Dehmamy, A.-L. Barabási, and R. Yu, "Understanding the representation power of graph neural networks in learning graph topology," *arXiv preprint arXiv:1907.05008*, 2019.

[5] A. Grover, E. Wang, A. Zweig, and S. Ermon, "Stochastic optimization of sorting networks via continuous relaxations," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1eSS3CcKX.

[6] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman, "Provably powerful graph networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019, pp. 2156–2167. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bb04af0f7ecaee4aae62035497da1387-Paper.pdf.

[7] K. Xu, J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka, "What can neural networks reason about?" In *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rJxbJeHFPS.

[8] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *arXiv preprint arXiv:2003.00982*, 2020.

[9] C. Lyle, M. van der Wilk, M. Kwiatkowska, Y. Gal, and B. Bloem-Reddy, *On the benefits of invariance in neural networks*, 2020. arXiv: 2005.00178 [cs.LG].

[10] T. N. Sainath, R. J. Weiss, A. Senior, K. W. Wilson, and O. Vinyals, "Learning the speech front-end with raw waveform cldnns," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[11] D. Yu and J. Li, "Recent progresses in deep learning based acoustic models," *IEEE/-CAA Journal of automatica sinica*, vol. 4, no. 3, pp. 396–409, 2017.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[13] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proceedings of 2010 IEEE international symposium on circuits and systems*, IEEE, 2010, pp. 253–256.

[14] J. Dauparas and F. Fuchs. (2021). Alphafold 2 and equivariance, [Online]. Available: https://fabianfuchsml.github.io/alphafold2/ (visited on 02/10/2021).

[15] P. Simard, B. Victorri, Y. LeCun, and J. S. Denker, "Tangent prop-a formalism for specifying selected invariances in an adaptive network," in *NIPS*, Citeseer, vol. 91, 1991, pp. 895–903.

[16] D. A. Van Dyk and X.-L. Meng, "The art of data augmentation," *Journal of Computational and Graphical Statistics*, vol. 10, no. 1, pp. 1–50, 2001.

[17] A. Fawzi, H. Samulowitz, D. Turaga, and P. Frossard, "Adaptive data augmentation for image classification," in *2016 IEEE international conference on image processing (ICIP)*, Ieee, 2016, pp. 3688–3692.

[18] J. Shawe-Taylor, "Symmetries and discriminability in feedforward network architectures," *IEEE Transactions on Neural Networks*, vol. 4, no. 5, pp. 816–826, 1993.

[19] S. Ravanbakhsh, J. Schneider, and B. Poczos, "Equivariance through parameter-sharing," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 2892–2901.

[20] H. Gholamalinezhad and H. Khosravi, "Pooling methods in deep neural networks, a review," *arXiv preprint arXiv:2009.07485*, 2020.

[21] T. S. Cohen, M. Geiger, J. Köhler, and M. Welling, "Spherical cnns," *arXiv preprint arXiv:1801.10130*, 2018.

[22] S. Graham, D. Epstein, and N. Rajpoot, "Dense steerable filter cnns for exploiting rotational symmetry in histology images," *IEEE Transactions on Medical Imaging*, vol. 39, no. 12, pp. 4124–4136, 2020. DOI: 10.1109/TMI.2020.3013246.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[25] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 652–660.

[26] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, "Deep Sets," in *NIPS*, 2017.

[27] C. Meng, J. Yang, B. Ribeiro, and J. Neville, "Hats: A hierarchical sequence-attention framework for inductive set-of-sets embeddings," KDD, 2019.

[28] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 5, p. 146, 2019.

[29] O. Maron and T. Lozano-Pérez, "A framework for multiple-instance learning," in *Advances in neural information processing systems*, 1998, pp. 570–576.

[30] Z.-H. Zhou, Y.-Y. Sun, and Y.-F. Li, "Multi-instance learning by treating instances as non-iid samples," in *Proceedings of the 26th annual international conference on machine learning*, ACM, 2009, pp. 1249–1256.

[31] D. Shen, G. Wu, and H.-I. Suk, "Deep learning in medical image analysis," *Annual review of biomedical engineering*, vol. 19, pp. 221–248, 2017.

[32] S. Kalra, M. Adnan, G. Taylor, and H. Tizhoosh, "Learning permutation invariant representations using memory networks," *arXiv preprint arXiv:1911.07984*, 2019.

[33] Z. Wang, J. Poon, S. Sun, and S. Poon, "Attention-based multi-instance neural network for medical diagnosis from incomplete and low quality data," in *2019 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8851846.

[34] M. Ilse, J. Tomczak, and M. Welling, "Attention-based deep multiple instance learning," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 2127–2136. [Online]. Available: http://proceedings.mlr.press/v80/ilse18a.html.

[35] G. Shi, W. Hönig, Y. Yue, and S.-J. Chung, "Neural-swarm: Decentralized close-proximity multirotor control using learned interactions," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 3241–3247.

[36] Y. Guo, T. Ge, and F. Wei, "Fact-aware sentence split and rephrase with permutation invariant training," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 7855–7862.

[37] H. Kim, A. Mnih, J. Schwarz, M. Garnelo, A. Eslami, D. Rosenbaum, O. Vinyals, and Y. W. Teh, "Attentive neural processes," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=SkE6PjC9KX.

[38] J. Lee, Y. Lee, and Y. W. Teh, "Deep amortized clustering," *arXiv preprint arXiv:1909.13433*, 2019.

[39] A. Richard and J. Gall, "A bag-of-words equivalent recurrent neural network for action recognition," *Computer Vision and Image Understanding*, vol. 156, pp. 79–91, 2017.

[40] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional Networks on Graphs for Learning Molecular Fingerprints," in *NIPS*, 2015.

[41] Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. Pande, "Moleculenet: A benchmark for molecular machine learning," *Chemical science*, vol. 9, no. 2, pp. 513–530, 2018.

[42] J. Klicpera, J. Groß, and S. Günnemann, "Directional message passing for molecular graphs," *arXiv preprint arXiv:2003.03123*, 2020.

[43] H. Altae-Tran, B. Ramsundar, A. S. Pappu, and V. Pande, "Low data drug discovery with one-shot learning," *ACS central science*, vol. 3, no. 4, pp. 283–293, 2017.

[44] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, International Convention Centre, Sydney, Australia: PMLR, 2017, pp. 1263–1272. [Online]. Available: http://proceedings.mlr.press/v70/gilmer17a.html.

[45] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *NIPS*, Jun. 2017. arXiv: 1706.02216. [Online]. Available: http://arxiv.org/abs/1706.02216.

[46] M. Zitnik and J. Leskovec, "Predicting multicellular function through multi-layer tissue networks," *Bioinformatics*, vol. 33, no. 14, pp. i190–i198, 2017.

[47] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*, Springer, 2018, pp. 593–607.

[48] C. Shang, Y. Tang, J. Huang, J. Bi, X. He, and B. Zhou, "End-to-end structure-aware convolutional networks for knowledge base completion," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 3060–3067.

[49] F. B. Fuchs, A. R. Kosiorek, L. Sun, O. P. Jones, and I. Posner, "End-to-end recurrent multi-object tracking and trajectory prediction with relational reasoning," *arXiv preprint arXiv:1907.12887*, 2019.

[50] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[51] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," in *Advances in neural information processing systems*, 2017, pp. 4967–4976.

[52] M. Zitnik, M. Agrawal, and J. Leskovec, "Modeling polypharmacy side effects with graph convolutional networks," *Bioinformatics*, vol. 34, no. 13, pp. i457–i466, 2018.

[53] C. Li and D. Goldwasser, "Encoding social information with graph convolutional networks forpolitical perspective detection in news media," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2594–2604.

[54] R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro, "Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=BJluy2RcFm.

[55] J. Moore and J. Neville, "Deep collective inference.," in *AAAI*, 2017, pp. 2364–2372.

[56] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" In *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km.

[57] C. Bueno and A. G. Hylton, *Limitations for learning from point clouds*, 2020. [Online]. Available: https://openreview.net/forum?id=r1x63grFvH.

[58] A. Zweig and J. Bruna, "A functional perspective on learning symmetric functions with neural networks," *arXiv preprint arXiv:2008.06952*, 2020.

[59] T. Pevny and V. Kovarik, *Approximation capability of neural networks on sets of probability measures and tree-structured data*, 2019. [Online]. Available: https://openreview.net/forum?id=HklJV3A9Ym.

[60] J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh, "Set transformer: A framework for attention-based permutation-invariant neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, Long Beach, California, USA: PMLR, Jun. 2019, pp. 3744–3753. [Online]. Available: http://proceedings.mlr.press/v97/lee19d.html.

[61] B. Yang, S. Wang, A. Markham, and N. Trigoni, "Robust attentional aggregation of deep feature sets for multi-view 3d reconstruction," *International Journal of Computer Vision*, Aug. 2019, ISSN: 1573-1405. DOI: 10.1007/s11263-019-01217-w. [Online]. Available: https://doi.org/10.1007/s11263-019-01217-w.

[62] E. Wagstaff, F. Fuchs, M. Engelcke, I. Posner, and M. A. Osborne, "On the limitations of representing functions on sets," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, Long Beach, California, USA: PMLR, Jun. 2019, pp. 6487–6494. [Online]. Available: http://proceedings.mlr.press/v97/wagstaff19a.html.

[63] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," Sep. 2016. arXiv: 1609.02907. [Online]. Available: http://arxiv.org/abs/1609.02907.

[64] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ.

[65] J. Klicpera, A. Bojchevski, and S. Gunnemann, "Predict then propagate: Graph neural networks meet personalized pagerank," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gL-2A9Ym.

[66] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, "Graph neural networks with convolutional arma filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021. DOI: 10.1109/TPAMI.2021.3054830.

[67] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[68] D. Yarotsky, "Universal approximations of invariant maps by neural networks," *arXiv preprint arXiv:1804.10306*, 2018.

[69] D.-X. Zhou, "Universality of deep convolutional neural networks," *Applied and computational harmonic analysis*, vol. 48, no. 2, pp. 787–794, 2020.

[70] W. Kumagai and A. Sannai, "Universal approximation theorem for equivariant maps by group cnns," *arXiv preprint arXiv:2012.13882*, 2020.

[71] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[72] H. Siegelmann and E. Sontag, "On the computational power of neural nets," *Journal of Computer and System Sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[73] A. M. Schafer and H.-G. Zimmermann, "Recurrent neural networks are universal approximators.," *International Journal of Neural Systems*, vol. 17, no. 4, pp. 253–263, 2007.

[74] R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro, "Relational pooling for graph representations," in *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[75] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International conference on machine learning*, PMLR, 2016, pp. 2014–2023.

[76] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen, "Pointcnn: Convolution on x-transformed points," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/f5f8590cd58a54e94377e6ae2eded4d9-Paper.pdf.

[77] O. Vinyals, S. Bengio, and M. Kudlur, "Order Matters: Sequence to Sequence for Sets," *ICLR*, 2016.

[78] Z. Chen, S. Villar, L. Chen, and J. Bruna, "On the equivalence between graph isomorphism testing and function approximation with gnns," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/71ee911dd06428a96c143a0b135041a4-Paper.pdf.

[79] J. Toenshoff, M. Ritzert, H. Wolf, and M. Grohe, *Graph learning with 1d convolutions on random walks*, 2021. arXiv: 2102.08786 [cs.LG].

[80] Z. Zhang, P. Cui, J. Pei, X. Wang, and W. Zhu, *Eigen-gnn: A graph structure preserving plug-in for gnns*, 2020. arXiv: 2006.04330 [cs.LG].

[81] R. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro, "Relational pooling for graph representations," in *International Conference on Machine Learning*, PMLR, 2019, pp. 4663–4673.

[82] G. Bianconi and A.-L. Barabási, "Competition and multiscaling in evolving networks," *EPL (Europhysics Letters)*, vol. 54, no. 4, p. 436, 2001.

[83] L. Jánossy, "On the absorption of a nucleon cascade," in *Proceedings of the Royal Irish Academy. Section A: Mathematical and Physical Sciences*, JSTOR, vol. 53, 1950, pp. 181–188.

[84] D. J. Daley and D. Vere-Jones, *An introduction to the theory of point processes: volume II: general theory and structure*. Springer Science & Business Media, 2007.

[85] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning: Prediction, inference and data mining," *Springer-Verlag, New York*, 2009.

[86] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.

[87] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[88] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[89] L. Younes, "On the convergence of markovian stochastic algorithms with rapidly decreasing ergodicity rates," *Stochastics: An International Journal of Probability and Stochastic Processes*, vol. 65, no. 3-4, pp. 177–228, 1999.

[90] A. Yuille, "The Convergence of Contrastive Divergences," in *NIPS*, 2004.

[91] D. P. Kingma and J. L. Ba, "ADAM: A Method for Stochastic Optimization," *International Conference on Learning Representations, ICLR*, 2015.

[92] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[93] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[94] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *EMNLP*, 2014.

[95] P. Domingos, "Every model learned by gradient descent is approximately a kernel machine," *arXiv preprint arXiv:2012.00152*, 2020.

[96] A. Segato, A. Marzullo, F. Calimeri, and E. De Momi, "Artificial intelligence for brain diseases: A systematic review," *APL bioengineering*, vol. 4, no. 4, p. 041 503, 2020.

[97] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning based text classification: A comprehensive review," *arXiv preprint arXiv:2004.03705*, 2020.

[98] M. Yang, Y. Lin, F. Lv, S. Zhu, K. Yu, M. Dikmen, L. Cao, and T. S. Huang, "Videos semantic indexing using image classification.," in *TRECVID*, Citeseer, 2010.

[99] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the seventh IEEE international conference on computer vision*, Ieee, vol. 2, 1999, pp. 1150–1157.

[100] N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh, "Deep learning vs. traditional computer vision," in *Science and Information Conference*, Springer, 2019, pp. 128–144.

[101] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016.

[102] F.-F. Li, A. Karpathy, and J. Johnson, "Cs231n: Convolutional neural networks for visual recognition 2016," [Online]. Available: http://cs231n.stanford.edu/.

[103] M. Hardt and B. Recht, *Patterns, predictions, and actions: A story about machine learning*. https://mlstory.org, 2021. arXiv: 2102.05242 [cs.LG].

[104] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," in *Thirty-First International Conference on Machine Learning Workshop*, Lille, France, 2015.

[105] S. Siddiqui, I. Malik, F. Shafait, A. Mian, M. Shortis, and E. Harvey, "Automatic fish species classification in underwater videos: Exploiting pretrained deep neural network models to compensate for limited labelled data," *ICES Journal of Marine Science*, vol. 75, May 2017. DOI: 10.1093/icesjms/fsx109.

[106] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, Springer, 2014, pp. 818–833.

[107] J. Ugander, L. Backstrom, and J. Kleinberg, "Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 1307–1318.

[108] T. Guo and X. Zhu, "Understanding the roles of sub-graph features for graph classification: An empirical study perspective," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 817–822.

[109] S. Bonner, J. Brennan, G. Theodoropoulos, I. Kureshi, and A. S. McGough, "Deep topology classification: A new approach for massive graph classification," in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, 2016, pp. 3290–3297.

[110] J. P. Canning, E. E. Ingram, S. Nowak-Wolff, A. M. Ortiz, N. K. Ahmed, R. A. Rossi, K. R. Schmitt, and S. Soundarajan, "Predicting graph categories from structural properties," 2018.

[111] D. Rogers and M. Hahn, "Extended-connectivity fingerprints," *Journal of chemical information and modeling*, vol. 50, no. 5, pp. 742–754, 2010.

[112] T.-A. Song, S. Chowdhury, F. Yang, H. Jacobs, G. Fakhri, Q. Li, K. Johnson, and J. Dutta, "Graph convolutional neural networks for alzheimer's disease classification," vol. 2019, Apr. 2019, pp. 414–417. DOI: 10.1109/ISBI.2019.8759531.

[113] M. De Domenico, S. Sasai, and A. Arenas, "Mapping multiplex hubs in human functional brain networks," *Frontiers in neuroscience*, vol. 10, p. 326, 2016.

[114] N. M. Kriege, F. D. Johansson, and C. Morris, "A survey on graph kernels," *Applied Network Science*, vol. 5, no. 1, pp. 1–42, 2020.

[115] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels.," *Journal of Machine Learning Research*, vol. 12, no. 9, 2011.

[116] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 2010.

[117] W. Ye, O. Askarisichani, A. Jones, and A. Singh, "Deepmap: Learning deep representations for graph classification," *arXiv preprint arXiv:2004.02131*, 2020.

[118] M. Belkin, S. Ma, and S. Mandal, "To understand deep learning we need to understand kernel learning," in *International Conference on Machine Learning*, PMLR, 2018, pp. 541–549.

[119] M. M. Bronstein and I. Kokkinos, "Scale-invariant heat kernel signatures for non-rigid shape recognition," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE, 2010, pp. 1704–1711.

[120] M. Aubry, U. Schlickewei, and D. Cremers, "The wave kernel signature: A quantum mechanical approach to shape analysis," in *2011 IEEE international conference on computer vision workshops (ICCV workshops)*, IEEE, 2011, pp. 1626–1633.

[121] Y. Fang, J. Xie, G. Dai, M. Wang, F. Zhu, T. Xu, and E. Wong, "3d deep shape descriptor," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 2319–2328.

[122] X. Wang, Y. Yan, P. Tang, X. Bai, and W. Liu, "Revisiting multiple instance neural networks," *Pattern Recognition*, vol. 74, pp. 15–24, 2018.

[123] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez, "Solving the multiple instance problem with axis-parallel rectangles," *Artificial intelligence*, vol. 89, no. 1-2, pp. 31–71, 1997.

[124] K. Hornik, M. Stinchcombe, and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural networks*, vol. 3, no. 5, pp. 551–560, 1990.

[125] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[126] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

[127] R. Reed and R. J. MarksII, *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999, Chapter 8.

[128] G. B. Orr and K.-R. Müller, *Neural networks: tricks of the trade*. Springer, 2003, pp. 9–48.

[129] E. W. Weisstein, *Symmetry. From MathWorld—A Wolfram Web Resource*. [Online]. Available: https://mathworld.wolfram.com/Symmetry.html.

[130] Y. Shmaliy, *Continuous-time systems*. Springer Science & Business Media, 2007, Chapter 4.

[131] B. G. Osgood, *Lectures on the Fourier transform and its applications*. American Mathematical Soc., 2019, vol. 33, See Chapters 3 and 8.

[132] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159,

[133] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[134] J. F. Henriques and A. Vedaldi, "Warped convolutions: Efficient invariance to spatial transformations," in *International Conference on Machine Learning*, PMLR, 2017, pp. 1461–1469.

[135] T. Cohen and M. Welling, "Group equivariant convolutional networks," in *International conference on machine learning*, PMLR, 2016, pp. 2990–2999.

[136] A. Kanazawa, A. Sharma, and D. Jacobs, *Locally scale-invariant convolutional neural networks*, 2014. arXiv: 1412.5104 [cs.CV].

[137] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, contour and grouping in computer vision*, Springer, 1999, pp. 319–345.

[138] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[139] M. Shaban, Z. Ogur, A. Mahmoud, A. Switala, A. Shalaby, H. Abu Khalifeh, M. Ghazal, L. Fraiwan, G. Giridharan, H. Sandhu, *et al.*, "A convolutional neural network for the screening and staging of diabetic retinopathy," *Plos one*, vol. 15, no. 6, e0233514, 2020.

[140] A. S. Lundervold and A. Lundervold, "An overview of deep learning in medical imaging focusing on mri," *Zeitschrift für Medizinische Physik*, vol. 29, no. 2, pp. 102–127, 2019.

[141] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[142] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, IEEE, 2015, pp. 4580–4584.

[143] Y. Aytar, C. Vondrick, and A. Torralba, "Soundnet: Learning sound representations from unlabeled video," *arXiv preprint arXiv:1610.09001*, 2016.

[144] A. Hyvärinen, J. Hurri, and P. O. Hoyer, *Natural image statistics: A probabilistic approach to early computational vision.* Springer Science & Business Media, 2009, vol. 39.

[145] I. R. Kondor, *Group theoretical methods in machine learning*. Columbia University New York, 2008, vol. 2.

[146] E. D. Bloch, *Proofs and fundamentals: a first course in abstract mathematics*. Springer Science & Business Media, 2011.

[147] P. A. Grillet, *Abstract algebra*. Springer Science & Business Media, 2007, vol. 242.

[148] B. Bloem-Reddy and. Teh, "Probabilistic symmetries and invariant neural networks," *Journal of Machine Learning Research*, vol. 21, no. 90, pp. 1–61, 2020. [Online]. Available: http://jmlr.org/papers/v21/19-322.html.

[149] H. Maron, E. Fetaya, N. Segol, and Y. Lipman, "On the universality of invariant networks," in *ICML*, 2019, pp. 4363–4371. [Online]. Available: http://proceedings.mlr.press/v97/maron19a.html.

[150] A. Bart and B. Clair, *Math and the art of mc escher*, https://mathstat.slu.edu/escher/index.php/Math_and_the_Art_of_M._C._Escher, Image file Symmetry-group-square.png.

[151] K. Conrad, *Group actions*, https://kconrad.math.uconn.edu/blurbs/grouptheory/gpaction.pdf.

[152] B. Ribeiro, *Tensor decomposition*, Lecture Notes for CS573 Data Mining, Fall 2016.

[153] E. E. Papalexakis, C. Faloutsos, T. M. Mitchell, P. P. Talukdar, N. D. Sidiropoulos, and B. Murphy, "Turbo-smt: Accelerating coupled sparse matrix-tensor factorizations by 200x," in *Proceedings of the 2014 SIAM International Conference on Data Mining*, SIAM, 2014, pp. 118–126.

[154] J. Hartford, D. Graham, K. Leyton-Brown, and S. Ravanbakhsh, "Deep models of interactions across sets," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, Stockholmsmässan, Stockholm Sweden: PMLR, Jul. 2018, pp. 1909–1918. [Online]. Available: http://proceedings.mlr.press/v80/hartford18a.html.

[155] E. W. Weisstein, *Permutation matrix. From MathWorld—A Wolfram Web Resource.* [Online]. Available: https://mathworld.wolfram.com/PermutationMatrix.html.

[156] M. L. Minsky and S. Papert, "Perceptrons, expanded ed," *MIT Press, Cambridge, MA*, vol. 15, p. 767 776, 1988.

[157] H. Maron, H. Ben-Hamu, N. Shamir, and Y. Lipman, "Invariant and equivariant graph networks," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=Syx72jC9tm.

[158] Y. Gong, L. Wang, R. Guo, and S. Lazebnik, "Multi-scale orderless pooling of deep convolutional activation features," in *European conference on computer vision*, Springer, 2014, pp. 392–407.

[159] H. Jégou, F. Perronnin, M. Douze, J. Sánchez, P. Pérez, and C. Schmid, "Aggregating local image descriptors into compact codes," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 9, pp. 1704–1716, 2011.

[160] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," *arXiv preprint arXiv:1607.08022*, 2016.

[161] T. Lindeberg, "Provably scale-covariant continuous hierarchical networks based on scale-normalized differential expressions coupled in cascade," *Journal of Mathematical Imaging and Vision*, vol. 62, no. 1, pp. 120–148, 2020.

[162] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[163] M. Beer, J. Urenda, O. Kosheleva, and V. Kreinovich, "Why spiking neural networks are efficient: A theorem," in *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Springer, 2020, pp. 59–69.

[164] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, vol. 111, pp. 47–63, 2019.

[165] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[166] D. Laptev, N. Savinov, J. M. Buhmann, and M. Pollefeys, "Ti-pooling: Transformation-invariant pooling for feature learning in convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 289–297.

[167] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=Sy8gdB9xx.

[168] R. Gilmore, *Lie Groups, Lie Algebras and some of their Applications*. Wiley, New York, 1974.

[169] R. Gens and P. M. Domingos, "Deep symmetry networks," *Advances in neural information processing systems*, vol. 27, pp. 2537–2545, 2014.

[170] F. Anselmi, J. Z. Leibo, L. Rosasco, J. Mutch, A. Tacchetti, and T. Poggio, "Unsupervised learning of invariant representations with low sample complexity: The magic of sensory cortex or a new framework for machine learning?," 2014.

[171] J. Shawetaylor, "Sample sizes for threshold networks with equivalences," *Information and Computation*, vol. 118, no. 1, pp. 65–72, 1995.

[172] Y. S. Abu-Mostafa, "Hints and the vc dimension," *Neural Computation*, vol. 5, no. 2, pp. 278–288, 1993.

[173] N. Vapnik Vladimir, *The nature of statistical learning theory (information science and statistics)*, 1999.

[174] P. L. Bartlett, N. Harvey, C. Liaw, and A. Mehrabian, "Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks," *Journal of Machine Learning Research*, vol. 20, no. 63, pp. 1–17, 2019. [Online]. Available: http://jmlr.org/papers/v20/17-612.html.

[175] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin, *Bayesian data analysis*. CRC press, 2013.

[176] G. Benton, M. Finzi, P. Izmailov, and A. G. Wilson, "Learning invariances in neural networks from training data," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 17 605–17 616. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/cc8090c4d2791cdd9cd2cb3c24296190-Paper.pdf.

[177] E. Cohen-Karlik, A. Ben David, and A. Globerson, "Regularizing towards permutation invariance in recurrent models," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 18 364–18 374. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/d58f36f7679f85784d8b010ff248f898-Paper.pdf.

[178] M. van der Wilk, M. Bauer, S. John, and J. Hensman, "Learning invariances using the marginal likelihood," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/d465f14a648b3d0a1faa6f447e526c60-Paper.pdf.

[179] S. C. Mouli and B. Ribeiro, "Neural network extrapolations with g-invariances from a single environment," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=7t1FcJUWhi3.

[180] F. Anselmi, G. Evangelopoulos, L. Rosasco, and T. Poggio, "Symmetry-adapted representation learning," *Pattern Recognition*, vol. 86, pp. 201–208, 2019, ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2018.07.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320318302620.

[181] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[182] M. Varma and D. Ray, "Learning the discriminative power-invariance trade-off," in *2007 IEEE 11th International Conference on Computer Vision*, IEEE, 2007, pp. 1–8.

[183] N. Keriven and G. Peyré, "Universal invariant and equivariant graph neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019, pp. 7092–7101. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/ea9268cb43f55d1d12380fb6ea5bf572-Paper.pdf.

[184] W. Kumagai and A. Sannai, *Universal approximation theorem for equivariant maps by group {cnn}s*, 2021. [Online]. Available: https://openreview.net/forum?id=7TBP8k7TLFA.

[185] A. Sannai, Y. Takai, and M. Cordonnier, *Universal approximations of permutation invariant/equivariant functions by deep neural networks*, 2020. [Online]. Available: https://openreview.net/forum?id=HkeZQJBKDB.

[186] N. Dym and H. Maron, "On the universality of rotation equivariant point cloud networks," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=6NFBvWlRXaG.

[187] R. Sato, M. Yamada, and H. Kashima, "Random features strengthen graph neural networks," in *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM*, 2021.

[188] S. Chen, E. Dobriban, and J. H. Lee, "A group-theoretic framework for data augmentation," *Journal of Machine Learning Research*, vol. 21, no. 245, pp. 1–71, 2020.

[189] T. Lee and S. Soatto, "Video-based descriptors for object recognition," *Image and Vision Computing*, vol. 29, no. 10, pp. 639–652, 2011.

[190] M. Arjovsky, L. Bottou, I. Gulrajani, and D. Lopez-Paz, "Invariant risk minimization," *arXiv preprint arXiv:1907.02893*, 2019.

[191] E. Creager, J. Jacobsen, and R. Zemel, "Exchanging lessons between algorithmic fairness and domain generalization," *arXiv preprint arXiv:2010.07249*, 2020.

[192] B. Sanchez-Lengeling and A. Aspuru-Guzik, "Inverse molecular design using machine learning: Generative models for matter engineering," *Science*, vol. 361, no. 6400, pp. 360–365, 2018.

[193] S. Kearnes, K. McCloskey, M. Berndl, V. Pande, and P. Riley, "Molecular graph convolutions: Moving beyond fingerprints," *Journal of computer-aided molecular design*, vol. 30, no. 8, pp. 595–608, 2016.

[194] T. Chen, S. Bian, and Y. Sun, "Graph feature networks," in *Proceedings of the ICLR-2019 Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[195] G. Montavon, K. Hansen, S. Fazli, M. Rupp, F. Biegler, A. Ziehe, A. Tkatchenko, A. V. Lilienfeld, and K.-R. Müller, "Learning invariant representations of molecules for atomization energy prediction," in *Advances in Neural Information Processing Systems*, 2012, pp. 440–448.

[196] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th international conference on World Wide Web*, Citation Key: jeh2003scaling bibtex[organization=Acm], 2003, pp. 271–279.

[197] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank citation ranking: Bringing order to the web.* 1999.

[198] G. Mena, D. Belanger, S. Linderman, and J. Snoek, "Learning latent permutations with gumbel-sinkhorn networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Byt3oJ-0W.

[199] Y. Zhang, J. Hare, and A. Prügel-Bennett, "Learning representations of sets through optimized permutations," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=HJMCcjAcYX.

[200] S. W. Linderman, G. E. Mena, H. Cooper, L. Paninski, and J. P. Cunningham, "Reparameterizing the birkhoff polytope for variational permutation inference," *arXiv preprint arXiv:1710.09508*, 2017.

[201] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[202] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[203] P. de Haan, T. S. Cohen, and M. Welling, "Natural graph networks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 3636–3646. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/2517756c5a9be6ac007fe9bb7fb92611-Paper.pdf.

[204] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[205] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[206] D. Grattarola and C. Alippi, "Graph neural networks in tensorflow and keras with spektral," in *Proceedings of the ICML-2020 workshop on Graph Representation Learning and Beyond*, 2020.

[207] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation Learning on Graphs with Jumping Knowledge Networks," in *ICML*, 2018. [Online]. Available: http://arxiv.org/abs/1806.03536.

[208] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks," *Proc. Thirty-Third AAAI Conf. Artif. Intell.*, 2019.

[209] V. Arvind, J. Köbler, G. Rattan, and O. Verbitsky, "Graph isomorphism, color refinement, and compactness," *computational complexity*, vol. 26, no. 3, pp. 627–685, 2017.

[210] B. Weisfeiler and A. Lehman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.

[211] M. Fürer, "On the combinatorial power of the Weisfeiler-Lehman algorithm," in *International Conference on Algorithms and Complexity*, Springer, 2017, pp. 260–271.

[212] G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein, *Improving graph neural network expressivity via subgraph isomorphism counting*, 2021. [Online]. Available: https://openreview.net/forum?id=LTOKSFnQDWF.

[213] N. Keriven and G. Peyré, "Universal invariant and equivariant graph neural networks," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 7090–7099. [Online]. Available: http://papers.nips.cc/paper/8931-universal-invariant-and-equivariant-graph-neural-networks.pdf.

[214] Y. Zhou, H. Zheng, and X. Huang, *Graph neural networks: Taxonomy, advances and trends*, 2021. arXiv: 2012.08752 [cs.LG].

[215] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021. DOI: 10.1109/TNNLS.2020.2978386.

[216] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[217] X. Miao, N. M. Gürel, W. Zhang, Z. Han, B. Li, W. Min, X. Rao, H. Ren, Y. Shan, Y. Shao, Y. Wang, F. Wu, H. Xue, Y. Yang, Z. Zhang, Y. Zhao, S. Zhang, Y. Wang, B. Cui, and C. Zhang, *Degnn: Characterizing and improving graph neural networks with graph decomposition*, 2020. arXiv: 1910.04499 [cs.LG].

[218] K. Oono and T. Suzuki, "Graph neural networks exponentially lose expressive power for node classification," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=S1ldO2EFPr.

[219] J. Zhang and L. Meng, "Gresnet: Graph residual network for reviving deep gnns from suspended animation," *arXiv preprint arXiv:1909.05729*, 2019.

[220] J. Zhang, H. Zhang, C. Xia, and L. Sun, "Graph-bert: Only attention is needed for learning graph representations," *arXiv preprint arXiv:2001.05140*, 2020.

[221] Q. Liu, M. Nickel, and D. Kiela, "Hyperbolic graph neural networks," *arXiv preprint arXiv:1910.12892*, 2019.

[222] I. Chami, R. Ying, C. Ré, and J. Leskovec, "Hyperbolic graph convolutional neural networks," *Advances in neural information processing systems*, vol. 32, p. 4869, 2019.

[223] A. Porter and M. Wootters, *On greedy approaches to hierarchical aggregation*, 2021. arXiv: 2102.01730 [cs.DS].

[224] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, "Redundancy-free computation for graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 997–1005.

[225] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rytstxWAW.

[226] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *International conference on machine learning*, PMLR, 2019, pp. 6861–6871.

[227] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric Deep Learning: Going beyond Euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, Jul. 2017, ISSN: 1053-5888. DOI: 10.1109/MSP.2017.2693418. [Online]. Available: http://ieeexplore.ieee.org/document/7974879/.

[228] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, 2016, pp. 3844–3852.

[229] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory," *Applied and Computational Harmonic Analysis*, vol. 30, no. 2, pp. 129–150, 2011.

[230] C. Colas, T. Karch, N. Lair, J.-M. Dussoux, C. Moulin-Frier, P. Dominey, and P.-Y. Oudeyer, "Language as a cognitive tool to imagine goals in curiosity driven exploration," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 3761–3774. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/274e6fcf4a583de4a81c6376f17673e7-Paper.pdf.

[231] G. Pellegrini, A. Tibo, P. Frasconi, A. Passerini, and M. Jaeger, *Learning aggregation functions*, 2020. arXiv: 2012.08482 [cs.LG].

[232] J. Ramon and L. De Raedt, "Multi instance neural networks," in *Proceedings of the ICML-2000 workshop on attribute-value and relational learning*, 2000, pp. 53–60.

[233] O. Z. Kraus, J. L. Ba, and B. J. Frey, "Classifying and segmenting microscopy images with deep multiple instance learning," *Bioinformatics*, vol. 32, no. 12, pp. i52–i59, 2016.

[234] T. Le and Y. Duan, "Pointgrid: A deep network for 3d shape understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9204–9214.

[235] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/d8bf84be3800d12f74d8b05e9b89836f-Paper.pdf.

[236] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[237] H. Zhao, L. Jiang, J. Jia, P. Torr, and V. Koltun, *Point transformer*, 2020. arXiv: 2012.09164 [cs.CV].

[238] M.-H. Guo, J.-X. Cai, Z.-N. Liu, T.-J. Mu, R. R. Martin, and S.-M. Hu, *Pct: Point cloud transformer*, 2020. arXiv: 2012.09688 [cs.CV].

[239] M. Billik and G.-C. Rota, "On reynolds operators in finite-dimensional algebras," *Journal of Mathematics and Mechanics*, vol. 9, no. 6, pp. 927–932, 1960.

[240] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković, "Principal neighbourhood aggregation for graph nets," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 13 260–13 271. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/99cad265a1768cc2dd013f0e740300ae-Paper.pdf.

[241] J.-Y. Cai, M. Fürer, and N. Immerman, "An optimal lower bound on the number of variables for graph identification," *Combinatorica*, vol. 12, no. 4, pp. 389–410, 1992.

[242] V. Vilfred, "On circulant graphs," in *Graph Theory and its Applications*, R. Balakrishnan, G. Sethuraman, and R. J. Wilson, Eds., Narosa Publishing House, 2004, pp. 34–36.

[243] R. Sato, *A survey on the expressive power of graph neural networks*, 2020. arXiv: 2003.04078 [cs.LG].

[244] M. H. Kutner, C. J. Nachtsheim, J. Neter, W. Li, *et al.*, *Applied linear statistical models*. McGraw-Hill Irwin Boston, 2005, vol. 5.

[245] C. Vignac, A. Loukas, and P. Frossard, "Building powerful and equivariant graph neural networks with structural message-passing," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 14 143–14 155. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/a32d7eeaae19821fd9ce317f3ce952a7-Paper.pdf.

[246] R. Abboud, I. I. Ceylan, M. Grohe, and T. Lukasiewicz, *The surprising power of graph neural networks with random node initialization*, 2021. [Online]. Available: https://openreview.net/forum?id=L7Irrt5sMQa.

[247] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. Reutter, and J. P. Silva, "The logical expressiveness of graph neural networks," in *International Conference on Learning Representations*, 2019.

[248] G. Dasoulas, L. Dos Santos, K. Scaman, and A. Virmaux, "Coloring graph neural networks for node disambiguation," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, Ed., Main track, International Joint Conferences on Artificial Intelligence Organization, Jul. 2020, pp. 2126–2132. DOI: 10.24963/ijcai.2020/294. [Online]. Available: https://doi.org/10.24963/ijcai.2020/294.

[249] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[250] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://www.aclweb.org/anthology/N19-1423.

[251] Y. Sun, S. Wang, Y. Li, S. Feng, X. Chen, H. Zhang, X. Tian, D. Zhu, H. Tian, and H. Wu, "ERNIE: enhanced representation through knowledge integration," *CoRR*, vol. abs/1904.09223, 2019. arXiv: 1904.09223. [Online]. Available: http://arxiv.org/abs/1904.09223.

[252] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[253] C. R. Qi, H. Su, M. Nießner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view cnns for object classification on 3d data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 5648–5656.

[254] L. Bottou and Y. LeCun, "Large scale online learning," *Advances in neural information processing systems*, vol. 16, pp. 217–224, 2004.

[255] B. Guedj, "A primer on pac-bayesian learning," *arXiv preprint arXiv:1901.05353*, 2019.

[256] G. K. Dziugaite and D. M. Roy, "Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data," *arXiv preprint arXiv:1703.11008*, 2017.

[257] D. Masters and C. Luschi, *Revisiting small batch training for deep neural networks*, 2018. arXiv: 1804.07612 [cs.LG].

[258] G. Grimmett and D. Stirzaker, *Probability and random processes*. Oxford university press, 2001.

[259] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, NJ, 1996, vol. 2.

[260] C. H. Teixeira, L. Cotta, B. Ribeiro, and W. Meira Jr, "Graph pattern mining and learning through user-defined relations (extended version)," *arXiv preprint arXiv:1809.05241*, 2018.

[261] C. H. Teixeira, M. Kakodkar, V. Dias, W. Meira Jr, and B. Ribeiro, "Sequential stratified regeneration: Mcmc for large state spaces with an application to subgraph counting estimation," *arXiv preprint arXiv:2012.03879*, 2020.

[262] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[263] B. De Finetti, "La prévision: Ses lois logiques, ses sources subjectives," in *Annales de l'institut Henri Poincaré*, [Translated into Enlish: H. E. Kyburg and H.E. Smokler, eds. Studies in Subjective Probability. *Krieger* 53-118, 1980], vol. 7, 1937, pp. 1–68.

[264] P. Diaconis, "Finite forms of de finetti's theorem on exchangeability," *Synthese*, vol. 36, no. 2, pp. 271–281, 1977.

[265] P. Orbanz and D. M. Roy, "Bayesian models of graphs, arrays and other exchangeable random structures," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 2, pp. 437–461, 2015.

[266] P. Diaconis and D. Freedman, "De finetti's generalizations of exchangeability," *Studies in inductive logic and probability*, vol. 2, pp. 233–249, 1980.

[267] M. L. Easton, "Chapter 8: Finite de finetti style theorems," in *Group invariance in applications in statistics*, ser. Regional Conference Series in Probability and Statistics. Haywood CA and Alexandria VA: Institute of Mathematical Statistics and American Statistical Association, 1989, vol. Volume 1, pp. 108–120. [Online]. Available: https://projecteuclid.org/euclid.cbms/1462061038.

[268] B.-N. Vo, N. Dam, D. Phung, Q. N. Tran, and B.-T. Vo, "Model-based learning for point pattern data," *Pattern Recognition*, vol. 84, pp. 136–151, 2018.

[269] D. J. Aldous, "Representations for partially exchangeable arrays of random variables," *Journal of Multivariate Analysis*, vol. 11, no. 4, pp. 581–598, 1981.

[270] D. N. Hoover, "Relations on probability spaces and arrays of random variables.," Institute of Advanced Study, Princeton", Tech. Rep., 1979.

[271] J. Moller and R. P. Waagepetersen, *Statistical inference and simulation for spatial point processes*. Chapman and Hall/CRC, 2003.

[272] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, 2001, pp. 149–159.

[273] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," *arXiv preprint arXiv:1811.05868*, 2018.

[274] B. Ramsundar, P. Eastman, K. Leswing, P. Walters, and V. Pande, *Deep Learning for the Life Sciences*. O'Reilly Media, 2019, https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837.

[275] S. G. Rohrer and K. Baumann, "Maximum unbiased validation (muv) data sets for virtual screening based on pubchem bioactivity data," *Journal of chemical information and modeling*, vol. 49, no. 2, pp. 169–184, 2009.

[276] A. Mayr, G. Klambauer, T. Unterthiner, and S. Hochreiter, "Deeptox: Toxicity prediction using deep learning," *Frontiers in Environmental Science*, vol. 3, p. 80, 2016.

[277] R. Huang, M. Xia, D.-T. Nguyen, T. Zhao, S. Sakamuru, J. Zhao, S. A. Shahane, A. Rossoshek, and A. Simeonov, "Tox21challenge to build predictive models of nuclear receptor and stress response pathways as mediated by exposure to environmental chemicals and drugs," *Frontiers in Environmental Science*, vol. 3, p. 85, 2016.

[278] R. Panico, W. Powell, and J.-C. Richer, *A guide to IUPAC Nomenclature of Organic Compounds*. Blackwell Scientific Publications, Oxford, 1993, vol. 2, Recommendation 7.1.2.

[279] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, p. 93, 2008.

[280] A. Chatr-Aryamontri, B.-J. Breitkreutz, R. Oughtred, L. Boucher, S. Heinicke, D. Chen, C. Stark, A. Breitkreutz, N. Kolas, L. O'Donnell, *et al.*, "The biogrid interaction database: 2015 update," *Nucleic acids research*, vol. 43, no. D1, pp. D470–D478, 2015.

[281] A. Subramanian, P. Tamayo, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander, *et al.*, "Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles," *Proceedings of the National Academy of Sciences*, vol. 102, no. 43, pp. 15 545–15 550, 2005.

[282] R. Oughtred, J. Rust, C. Chang, B.-J. Breitkreutz, C. Stark, A. Willems, L. Boucher, G. Leung, N. Kolas, F. Zhang, *et al.*, "The biogrid database: A comprehensive biomedical resource of curated protein, genetic, and chemical interactions," *Protein Science*, vol. 30, no. 1, pp. 187–200, 2021.

[283] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[284] R. Liao, Y. Li, Y. Song, S. Wang, W. Hamilton, D. K. Duvenaud, R. Urtasun, and R. Zemel, "Efficient graph generation with graph recurrent attention networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/d0921d442ee91b896ad95059d13df618-Paper.pdf.

[285] E. Pnevmatikakis, *Invariant and equivariant neural networks*, https://indico.flatironinstitute.org/event/52/attachments/81/122/main.pdf, 2019.

[286] T. Zhang, M. Chen, and A. A. Bui, "Diagnostic prediction with sequence-of-sets representation learning for clinical events," in *International Conference on Artificial Intelligence in Medicine*, Springer, 2020, pp. 348–358.

[287] Y. Li, G. Hu, Y. Wang, T. Hospedales, N. M. Robertson, and Y. Yang, "Differentiable automatic data augmentation," in *European Conference on Computer Vision*, Springer, 2020, pp. 580–595.

[288] T. Cohen and M. Welling, "Steerable cnns," in *International Conference on Learning Representations*, 2017.

[289] G. Caldarelli, A. Capocci, P. De Los Rios, and M. A. Munoz, "Scale-free networks from varying vertex intrinsic fitness," *Physical review letters*, vol. 89, no. 25, p. 258 702, 2002.

[290] O. Sporns, *Networks of the Brain*. MIT press, 2010.

[291] B. Bevilacqua, Y. Zhou, and B. Ribeiro, *Size-invariant graph representations for graph classification extrapolations*, 2021. arXiv: 2103.05045 [cs.LG].

[292] C. Pabbaraju and P. Jain, *Learning functions over sets via permutation adversarial networks*, 2020. arXiv: 1907.05638 [cs.LG].

[293] F. Yang, Z. Wang, and C. Heinze-Deml, "Invariance-inducing regularization using worst-case transformations suffices to boost accuracy and spatial robustness," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/1d01bd2e16f57892f0954902899f0692-Paper.pdf.

[294] E. W. Weisstein, *Doubly stochastic matrix. From MathWorld—A Wolfram Web Resource*. [Online]. Available: https://mathworld.wolfram.com/DoublyStochasticMatrix.html.

[295] C. H. Lim and S. Wright, "Beyond the birkhoff polytope: Convex relaxations for vector permutation problems," in *Advances in neural information processing systems*, 2014, pp. 2168–2176.

[296] R. E. Burkard, "Quadratic assignment problems," *Handbook of combinatorial optimization*, pp. 2741–2814, 2013.

[297] J. T. Vogelstein, J. M. Conroy, V. Lyzinski, L. J. Podrazik, S. G. Kratzer, E. T. Harley, D. E. Fishkind, R. J. Vogelstein, and C. E. Priebe, "Fast approximate quadratic programming for graph matching," *PLOS one*, vol. 10, no. 4, e0121002, 2015.

[298] T. Birdal and U. Simsekli, "Probabilistic permutation synchronization using the riemannian structure of the birkhoff polytope," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 105–11 116.

[299] G. Birkhoff, "Tres observaciones sobre el algebra lineal," *Univ. Nac. Tucuman, Ser. A*, vol. 5, pp. 147–154, 1946.

[300] A. Postnikov, "Permutohedra, associahedra, and beyond," *International Mathematics Research Notices*, vol. 2009, no. 6, pp. 1026–1106, 2009.

[301] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[302] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *International Conference on Machine Learning*, PMLR, 2017, pp. 1243–1252.

[303] R. Sinkhorn and P. Knopp, "Concerning nonnegative matrices and doubly stochastic matrices," *Pacific Journal of Mathematics*, vol. 21, no. 2, pp. 343–348, 1967.

[304] B. Speelpenning, "Compiling fast partial derivatives of functions given by algorithms," Illinois Univ., Urbana (USA). Dept. of Computer Science, Tech. Rep., 1980.

[305] J. Townsend. (2017). A new trick for calculating jacobian vector products, [Online]. Available: https://j-towns.github.io/2017/06/12/A-new-trick.html (visited on 03/15/2021).

[306] J. E. Atkins, E. G. Boman, and B. Hendrickson, "A spectral algorithm for seriation and the consecutive ones problem," *SIAM Journal on Computing*, vol. 28, no. 1, pp. 297–310, 1998.

[307] Y. Aflalo, A. Bronstein, and R. Kimmel, "On convex relaxation of graph isomorphism," *Proceedings of the National Academy of Sciences*, vol. 112, no. 10, pp. 2942–2947, 2015.

[308] F. Fogel, R. Jenatton, F. Bach, and A. d'Aspremont, "Convex relaxations for permutation problems," in *Advances in Neural Information Processing Systems*, 2013, pp. 1016–1024.

[309] J. Stewart, D. K. Clegg, and S. Watson, *Calculus: early transcendentals*. Cengage Learning, 2020.

[310] X. Li, D. Sun, and K.-C. Toh, "On the efficient computation of a generalized jacobian of the projector over the birkhoff polytope," *Mathematical Programming*, vol. 179, no. 1-2, pp. 419–446, 2020.

[311] B. Jiang, Y.-F. Liu, and Z. Wen, "L_p-norm regularization algorithms for optimization over permutation matrices," *SIAM Journal on Optimization*, vol. 26, no. 4, pp. 2284–2313, 2016.

[312] F. Wang, P. Li, and A. C. Konig, "Learning a bi-stochastic data similarity matrix," in *2010 IEEE International Conference on Data Mining*, IEEE, 2010, pp. 551–560.

[313] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[314] G. M. (https://tex.stackexchange.com/users/3954/gonzalo-medina), *Bipartite graphs*, TeX Stack Exchange, https://tex.stackexchange.com/questions/15088/bipartite-graphs (accessed March 5, 2021).

[315] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

A. APPENDIX

In the appendix, we provide additional details.

A.1 Technical Details and Examples

This section provides additional examples and definitions.

A.1.1 Permutation Matrices

An example showing that multiplication by a permutation matrix applies the action of the symmetric group (i.e., a permutation). Put

$$\boldsymbol{P} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \boldsymbol{X} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix}$$

as examples of a permutation matrix and a matrix. Then left-multiplication yields

$$\boldsymbol{PX} = \begin{pmatrix} x_{31} & x_{32} \\ x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}.$$

A.1.2 Vec Operation

Vectorizing a sequence or graph is useful in our theory, and may also be used in practice to render these inputs suitable for input to an MLP model (with padding). There is already a precise mathematical definition of the vec operation on matrices, but it should be generalized and modified to our encoding of graphs.

Graphs

For simplicity, assume graphs are a fixed size n. It is easy to extend this operation to variable-size graphs. Let $\mathcal{G} = (\mathbf{A}, \boldsymbol{F})$ be a graph where $\mathbf{A}$ is $n \times n \times d_e$ and $\boldsymbol{F}$ is $n \times d_v$.

The function vec : $\mathbb{R}^{n\times n\times d_e}\times\mathbb{R}^{n\times d_v}\to\mathbb{R}^{n^3 d_v d_e}$ converts a graph to a vector by generalizing the traditional vec operation. One exception is that vec is often defined by running down the rows first then the columns. Instead, we run down the latter dimensions first so that features for a given vertex are contiguous. For example, consider a graph with three vertices, one edge attribute at each edge, and two vertex attributes at each vertex. The connectivity structure and edge attributes are represented by the $3\times 3\times 2$ adjacency tensor $\mathbf{A}$,

$$\mathbf{A} = \begin{array}{ccc} A_{(1,1,2)} & A_{(1,2,2)} & A_{(1,3,2)} \\ A_{(2,1,2)} & A_{(2,2,2)} & A_{(2,3,2)} \\ A_{(3,1,2)} & A_{(3,2,2)} & A_{(3,3,2)} \end{array}$$
$$\begin{array}{ccc} A_{(1,1,1)} & A_{(1,2,1)} & A_{(1,3,1)} \\ A_{(2,1,1)} & A_{(2,2,1)} & A_{(2,3,1)} \\ A_{(3,1,1)} & A_{(3,2,1)} & A_{(3,3,1)} \end{array}$$

and vertex attributes are represented in a matrix

$$\boldsymbol{F} = \begin{pmatrix} \boldsymbol{F}_{1,1} & \boldsymbol{F}_{1,2} \\ \boldsymbol{F}_{2,1} & \boldsymbol{F}_{2,2} \\ \boldsymbol{F}_{3,1} & \boldsymbol{F}_{3,2} \end{pmatrix}.$$

A simple vec operation is shown below. The modeler is free to make modifications such as applying an MLP to the vertex attributes before concatenating with the edge attributes. Representing $\mathcal{G}$ by $\mathbf{A}$ and $\boldsymbol{F}$,

$$\begin{aligned}\text{vec}(\mathcal{G}) =\Big(&A_{(1,1,1)}, A_{(1,1,2)}, A_{(1,2,1)}, A_{(1,2,2)}, A_{(1,3,1)}, A_{(1,3,2)}, \boldsymbol{F}_{1,1}, \boldsymbol{F}_{1,2}, A_{(2,1,1)}, A_{(2,1,2)}, \\ &A_{(2,2,1)}, A_{(2,2,2)}, A_{(2,3,1)}, A_{(2,3,2)}, \boldsymbol{F}_{2,1}, \boldsymbol{F}_{2,2}, A_{(3,1,1)}, A_{(3,1,2)}, A_{(3,2,1)}, A_{(3,2,2)}, \\ &A_{(3,3,1)}, A_{(3,3,2)}, \boldsymbol{F}_{3,1}, \boldsymbol{F}_{3,2}\Big).\end{aligned}$$

Starting with the first vertex, each edge attribute (including the edge indicator) is listed, then the vertex attributes are added before doing the same with subsequent vertices. The vectorization method for k-ary type models is similar, except that we apply vec on induced subgraphs of size k.

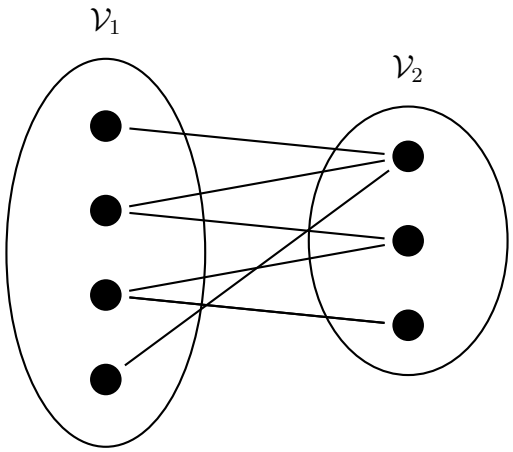


Figure A.1. Bipartite Graph. Such graphs represent two distinct groups of entities. Figure adapted from [314].

Sequences

We represent sequences as a matrix $\boldsymbol{S} \in \mathbb{R}^{n \times d_s}$,

$$\boldsymbol{S} = \begin{pmatrix} S_{11} & \cdots & S_{1n} \\ \vdots & \ddots & \vdots \\ S_{n1} & \cdots & S_{nn} \end{pmatrix}.$$

Consistent with our definition of vec for graphs, we traverse the row dimension first. This makes vector elements contiguous in the vectorized output. That is,

$$\text{vec}(\boldsymbol{S}) = (S_{11}, S_{12}, \ldots, S_{1n}, S_{21}, S_{22}, \ldots S_{2n}, \ldots S_{n1}, S_{n2}, \ldots S_{nn})^{\text{T}}.$$

A.1.3 Bipartite Graphs

An example of a bipartite graph is shown in Figure A.1.

A.1.4 Additional Details for Backpropagation through the Jacobian

Implementing the Jacobian-vector product in a way that admits backpropropagation can be done in a few lines of Python code, using PyTorch, but there are several nuances that

are lost in generic pseudocode. For instance, due to broadcasting and other details, we do not actually need to transpose the tensors. Accordingly, the Python code we use is shown in Algorithm 4.

Algorithm 4 PyTorch Code for Jacobian-vector Product

```
def get_jvp(y, x, vec):
  """
  Construct a differentiable Jacobian-vector product for a function
  Trick from: https://j-towns.github.io/2017/06/12/A-new-trick.html

  :param y: output of f
  :param x: input of f
  :param vec: vector in the Jacobian-vector product
  """
  # Arbitrary auxillary variable
  u = torch.zeros_like(y, requires_grad=True)
  # vector-jacobian product: model Jacobian times auxillary variable
  # (create_graph=True so we can backprop through this operation)
  ujp = torch.autograd.grad(y, x, grad_outputs=u, create_graph=True)[0]
  # Second backprop gives jacobian-vector product
  jvp = torch.autograd.grad(ujp, u,
                            grad_outputs=vec,
                            create_graph=True)[0]
  return jvp
```

A.2 MPGNN for Molecular GNN

We make use of the Message Passing Graph Neural Network proposed in [40]. This algorithm exploits special properties of molecules. Duvenaud, Maclaurin, Iparraguirre, *et al.* [40] refer to the procedure as finding a learnable *fingerprint* of a molecule, which would be called a learned representation in the machine learning community. The details are provided in Algorithm 5.

A.3 More Experimental Setup

Code from our publications are on GitHub. For [54], https://github.com/PurdueMINDS/JanossyPooling, and for [74], https://github.com/PurdueMINDS/RelationalPooling.

Algorithm 5 Duvenaud et al. MPGNN for Molecules

Input: Graph $\mathcal{G} = (\mathbf{A}, \boldsymbol{F})$, Number of Layers, L, dimension of latent d_h
Randomly initialized learnable parameters:
$\boldsymbol{\Theta}_1^{(1)}, \ldots, \boldsymbol{\Theta}_5^{(1)}, \ldots, \boldsymbol{\Theta}_5^{(L)}$, degree-specific aggregation parameters,
$\boldsymbol{\Xi}^{(1)}, \ldots, \boldsymbol{\Xi}^{(L)}$, aggregation parameters,
$\boldsymbol{\Lambda}^{(1)}, \ldots, \boldsymbol{\Lambda}^{(L)}$, layer-specific readout function parameters,
$\boldsymbol{\gamma}^{(1)}, \ldots, \boldsymbol{\gamma}^{(L)}$, bias (intercept/translation) for aggregation,
$\boldsymbol{\beta}^{(1)}, \ldots, \boldsymbol{\beta}^{(L)}$, bias (intercept/translation) for readout.

Let N be the number of vertices
$\boldsymbol{H}^{(0)} \leftarrow \boldsymbol{F}$ ▷ Write the vertex features as $\boldsymbol{H}^{(0)}$
for $l \leftarrow 1, \ldots, L$ **do** ▷ Message passing loop
  for $v \leftarrow 1, \ldots, N$ **do** ▷ Loop over vertices (for loop for clarity)
    $\boldsymbol{z} \leftarrow \sum_{u \in \mathcal{N}(v)} \boldsymbol{H}_{u,:}^{(l-1)} \bowtie \mathbf{A}_{v,u,:}$ ▷ Sum neighboring features
    $d \leftarrow |\mathcal{N}(v)|$ ▷ The degree of v
    $\boldsymbol{H}_{v,:}^{(t)} \leftarrow \psi\Big(\boldsymbol{H}_{v,:}^{(l-1)}\boldsymbol{\Xi}^{(l)} + \boldsymbol{z}\boldsymbol{\Theta}_d^{(l)} + \boldsymbol{\gamma}^{(l)}\Big)$ ▷ Learnable aggregation
  end for
end for
for $l \leftarrow 0, \ldots, L$ **do** ▷ Readout for each layer
  Initialize matrix $\boldsymbol{R}$ with N rows
  for $v \leftarrow 1, \ldots, N$ **do**
    $\boldsymbol{R}_{v,:} \leftarrow \text{softmax}\Big(\boldsymbol{H}_{v,:}^{(t)}\boldsymbol{\Lambda}^{(t)} + \boldsymbol{\beta}^{(l)}\Big)$ ▷ Updates not in message passing
  end for
  $\boldsymbol{f}^{(l)} \leftarrow \text{colSums}(\boldsymbol{R})$ ▷ Molecular 'fingerprint' at layer
end for
$\boldsymbol{f} \leftarrow \text{colSums}(\boldsymbol{R})$ ▷ Learned 'fingerprint' of entire molecule.

Algorithm proposed by Duvenaud, Maclaurin, Iparraguirre, *et al.* [40] to learn a molecular "fingerprint" with MPGNNs. In the relevant chemical applications, the degree of a vertex is in {1, 2, 3, 4, 5}, hence the use of 5 $\boldsymbol{\Theta}$ matrices at each layer. All vectors shown are row vectors here. Clearly, this exposition favors clarity rather than efficiency.

We presented the experiments by topic of inquiry (expressiveness, impact of k, and so on). Thus, the same dataset appeared in different sections. Here, since our models are tuned based on the dataset, we group model details by dataset.

A.3.1 Integer Sequences

Most of our implementation and hyperparameter choices are based on [26] as DeepSets is an important baseline. Full-sequence Janossy pooling uses a GRU as $\vec{f}$ with 80 hidden units, chosen to be consistent with [26]. The MLPs related to $\vec{f}$ have 30 neurons whereas the MLPs in r have 100 neurons. The readout r is either a learnable affine function or an MLP with one hidden layer, with 100 units. All activations are tanh.

For all models, we map the sequence elements to higher-dimensional space with a non-learnable dictionary encoding. For k-ary models with $k \in \{2, 3\}$, the output of this function is $\lfloor 100/k \rfloor$ to keep the total number of parameters consistent. We concatenate then pass the output into an MLP.

We optimize with Adam [91] with a tuned learning rate, searching over $\{0.01, 0.001, 0.0001, 0.00001\}$. We train for 1,000 epochs when the readout is affine and 2000 epochs otherwise. Training was performed on GeForce GTX 1080 Ti GPUs.

A.3.2 CSL Graphs

Our GIN architecture uses eight layers of recursion, where every $\text{MLP}^{(l)}$ has 32 neurons in the hidden layers. Each layer outputs a vector $\boldsymbol{h}_v^{(l)}$ osf dimension 16 for each vertex v. The graph embedding is mapped to the output through a final affine layer. We use the variant of GIN with additional scalar parameters $\vartheta^{(l)}$. For RPGNN, we add one-hot IDs to the vertices. The results we show were stable across a wide set of architectures, as GIN is theoretically unable to distinguish the CSL graphs. We train RPGNN with π-SGD. For inference, we average the score over 20 random permutations. We trained for 800 epochs using Adam [91] and a learning rate of 0.01. We performed 5-fold cross validation with classes balanced in all folds.

A.3.3 Molecules

Here we provide additional details on the molecular experiments.

MPGNN and RPGNN For the models based on Duvenaud et al., we extend the architecture provided from DeepChem and the MoleculeNet project [41], [274]. Following them, the learning rate was set to 0.003, we trained with mini-batches of size 96, and used the Adagrad optimizer [315]. Models were trained for 100 epochs. Training was performed on 48 CPUs using the inherent multithreading of DeepChem.

The loss function in DeepChem is weighted since MUV and Tox21 are multi-task problems. Misclassification is weighted differently depending on the target. The overall performance metric is the mean of the AUC across all tasks. One difference is that the DeepChem recommends either metrics PRC-AUC or ROC-AUC and splits "random" or "scaffold" depending on the dataset under consideration. Since ROC-AUC and random splits were the most commonly used among the three datasets we chose, we decided – before training any models – to use random splits and ROC-AUC for every dataset for simplicity. We also note that the authors of MoleculeNet report ROC-AUC scores on all three datasets. Regarding the sizes of the train/validation/test splits, we used the default values provided by DeepChem.

CNN and RNN. We explore $k = 20$ -ary RP with $\vec{f}$ as a CNN, learned with π-SGD. At each forward step, we run a DFS from a different randomly-selected vertex to obtain a $20 \times 20 \times 14$ subtensor of $\mathbf{A}$ there are 14 edge features, which we feed through two iterations of convolution, ReLU, and max pool to obtain a representation $\boldsymbol{h}_{\mathbf{A}}$ of $\mathbf{A}$. The corresponding vertex attributes are fed through an MLP and concatenated with $\boldsymbol{h}_{\mathbf{A}}$ to obtain a representation $\boldsymbol{h}_{\mathcal{G}}$ of the graph which in turn is fed through an MLP to obtain the predicted class. Zero padding was used to account for the variable-size molecules. Twenty initial vertices for the DFS (i.e., random permutations) were sampled at inference time.

We also consider RP with an RNN as $\vec{f}$ learned with π-SGD, starting with a DFS to yield a $|\mathcal{G}| \times |\mathcal{G}| \times 14$ subtensor. For $\vec{f}$, we treat the edge features of a given vertex as a sequence: for vertex v, we apply an LSTM to the sequence $(\mathbf{A}_{v,1,:}, \mathbf{A}_{v,2,:}, \ldots, \mathbf{A}_{v,|\mathcal{G}|,:})$ and extract the long-term state. We also take the vertex attributes and pass them through an

Table A.1. Vertex Classification Results. Performance (Micro-F1 score) using Janossy pooling with k-ary dependencies and π-SGD training in a graph neural network – GraphSAGE – with 20 permutations sampled at test time. k_1 and k_2 are the number of neighbors sampled at aggregations one and two, respectively. Standard deviations over 30 runs for Cora/Pubmed and 4 runs for PPI (a much larger graph) are shown in parentheses.

$\vec{f}$	method	k_1	k_2	Cora	Pubmed	PPI
LSTM	π-SGD	3	3	0.860 (.009)	0.889 (0.01)	0.538 (.005)
LSTM	π-SGD	5	5	–	–	0.579 (.015)
LSTM	π-SGD	10	25	–	–	0.650 (.013)
LSTM	π-SGD	25	10	–	–	0.689 (.062)
LSTM	π-SGD	25	25	–	–	0.702 (.044)
LSTM	π-SGD	$\lvert\mathcal{G}\rvert$	$\lvert\mathcal{G}\rvert$	–	–	0.757 (.040)
Identity (mean-pool)	exact	1	1	0.860 (.008)	0.881 (.011)	0.767 (.013)

MLP. The long term state and output of the MLP are concatenated, ultimately forming a representation for every vertex (and its neighborhood) which we view as a second sequence. We apply a second LSTM and again extract the long term state, which can be denoted $\boldsymbol{h}_{\mathcal{G}}$, the embedding of the graph. Last, $\boldsymbol{h}_{\mathcal{G}}$ is forwarded through an MLP yielding a class prediction. Twenty starting vertices (i.e., permutations) were sampled at inference time. Variability was quantified with 5 random train/val/test splits for both neural network based models. To model the RNN, we use an LSTM with 100 neurons.

We used the Adam optimizer [91], training all models with mini-batches of size 96 and 50 epochs, again following DeepChem. We performed a hyperparameter line search over the learning rate in $\{0.003, 0.001, 0.01, 0.03, 0.1, 0.3\}$.

A.3.4 Vertex Classification

Table A.1 shows the vertex classification results. All models perform similarly on Cora and Pubmed, likely due to the ease of the task.

Our implementation follows the PyTorch code associated with [45], available at https://github.com/williamleif/graphsage-simple/. That repo did not include an LSTM aggregator, so we implemented our own following the TensorFlow implementation of Graph-

SAGE, and describe it here. At the beginning of every forward pass, each vertex v is associated with a vertex attribute. For every vertex in a batch, we sample k_1 neighbors uniformly at random and then pass through an LSTM. From the LSTM, we take the hidden state associated with the last element in the input sequence. This hidden state is passed through an affine layer to yield a vector of dimension $d_h/2$, where d_h is the user-specified latent representation dimension. The vertex's own attribute is also fed forward through an affine layer with $d_h/2$ output neurons. At this point, for each vertex, we have two representation vectors of size $d_h/2$ representing the vertex v and its neighbor multiset, which we concatenate to form an representation of size d_h. This describes one layer, and it is repeated a second time with a distinct set of learnable weights for the affine and LSTM layers, sampling k_2 vertices from each neighborhood and using the representation of the first layer as features. After each message passing step, we may optionally apply a ReLU activation and/or embedding normalization, and we follow the decisions shown in the GraphSAGE code. After both layers, we apply a final affine layer to obtain the score, followed by a softmax (Cora, Pubmed) or sigmoid (PPI). The loss function is cross-entropy for Cora and Pubmed, and binary cross-entropy for PPI.

The number of trainable parameters in each model is independent of k_1 and k_2 by the design of LSTMs (the same is true for the mean-pooling aggregator). The only variation in the number of weights is in the dimensions of the input and output features, which differ by dataset.

Optimization is performed with the Adam optimizer. The training routine for the smaller graphs (Cora, Pubmed) is not guaranteed to see the entire training data, in contrast with the scheme applied to the larger PPI graph (following the author's code). For Cora and Pubmed, we form 100 minibatches by randomly sampling subsets of 256 vertices from the training dataset (with replacement). With PPI, we perform 10 full epochs: at each epoch, the training data is shuffled, partitioned into minibatches of size 512, and we pass over each. In either case, the weights are updated after computing the gradient of the loss on each minibatch.

The hyperparameters were set by following[45]; no hyperparameter optimization was performed. For every dataset, the embedding dimension was set to 256 at both message-

passing layers. For Pubmed and PPI, the learning rate is set at 0.01 while for Cora it is set at 0.005. Finally, evaluation is with F1 score per [45].

At test time, we load the weights obtained from training, perform 20 forward passes, and average the predicted probabilities (i.e., softmax output). We choose the class that maximizes the averaged probabilities.

A.3.5 BReg in PS and PI Tasks

We used the SetTransformer [60] implementation from the authors, available at https://github.com/juho-lee/set_transformer. Our max regression task was inspired by theirs, and we use the model that they used for max regression. This is a so-called "small" Set-Transformer. Our "Transformer" model has a trainable positional encoding matrix $\boldsymbol{\xi}$. We initialized it to have the same mean and standard deviation as the FairSEQ implementation of the models in [302] https://github.com/pytorch/fairseq.

There is a difference in our training. The authors sample new data at every batch. We followed a more traditional setup. However, to guarantee that the training data was rich enough, we made sure there were about 32,000 training samples in every training fold.

We evaluated hyperparameters with 3-fold cross validation. We swept over learning rates in $\{0.0001, 0.001, 0.01\}$, and compare full-batch to mini-batch size of 128. Ultimately, we found that full-batch training was most effective for the baselines. The optimal learning rate was 0.0001 for max prediction and 0.001 for "first large". We trained models with at least 10,000 epochs and did not stop training until validation-set performance stopped decreasing for 300 consecutive epochs. The regularized models typically require more epochs in order to converge, so we set the minimum epochs to 20,000 epochs. Please note that all models were trained to convergence, so the comparison is still fair. For regularization, we tuned $\varepsilon \in \{0.01, 0.1\}$.

A.4 Proofs

Here we give more proof details.

A.4.1 Doubly-Stochastic Matrices

For any $n \in \mathbb{Z}_{\geq 2}$, let $\boldsymbol{D} := \boldsymbol{D}_{\text{center}}(\varepsilon, n) = (1 - c(\varepsilon, n))\boldsymbol{I} + c(\varepsilon, n)\frac{1}{n}\mathbf{1}\mathbf{1}^{\mathrm{T}}$ and $\boldsymbol{I} := \boldsymbol{I}_n$. Then,

$$\begin{aligned}
\|\boldsymbol{D} - \boldsymbol{I}\|^2 &= \|(1 - c(\varepsilon, n))\boldsymbol{I} + c(\varepsilon, n)\frac{1}{n}\mathbf{1}\mathbf{1}^{\mathrm{T}} - \boldsymbol{I}\|^2 \\
&= \| - c(\varepsilon, n)\boldsymbol{I} + c(\varepsilon, n)\frac{1}{n}\mathbf{1}\mathbf{1}^{\mathrm{T}})\|^2 \\
&= \Big(c(\varepsilon, n)\Big)^2\Big(n(\frac{n-1}{n})^2 - n(n-1)\frac{1}{n^2}\Big) \\
&= \Big(c(\varepsilon, n)\Big)^2\Big(n-1\Big)
\end{aligned}$$

Next, using the fact that $c(\varepsilon, n) = \frac{\varepsilon}{\sqrt{n-1}}$, this becomes $(\frac{\varepsilon}{\sqrt{n-1}})^2(n-1) = \varepsilon^2$. Hence, $\|\boldsymbol{D} - \boldsymbol{I}\| = \varepsilon$.